BLOCKWATCH: Leveraging Similarity in Parallel Programs for Error Detection

Jiesheng Wei, Karthik Pattabiraman
Department of Electrical and Computer Engineering
The University of British Columbia, Canada

***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 study, 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 seven SPLASH-2 benchmarks to measure its performance overhead and error detection coverage. We find that BLOCKWATCH incurs an average overhead of 16% across all programs, and provides an average SDC coverage of 97% for faults in the control data.

Keywords: Parallel programs, Control-data, SPMD, Static Analysis, Runtime checks

I. INTRODUCTION

The continued scaling of Silicon devices has exacerbated their unreliability and error-proneness. In the near future, microprocessors will experience significantly higher rates of hardware faults [1]. Processor faults have hitherto been masked from software through redundancy at the hardware level [2] (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. 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 [3]. 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 [4]. 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 [5], efficient error detection remains a challenge.

In this paper, we explore the use of similarity among tasks (i.e., threads) of a parallel program for runtime error detection. The similarity arises as a result of high-level programming models, such as Single Program Multiple Data (SPMD) paradigm. 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].

SPMD is the most commonly used style for parallel programming [6]. While there are many sources of similarity in an SPMD program, we focus on the similarity of control-data (i.e. the data that is used to make branch and loop decisions), to detect faults that corrupt the control-data. 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 can lead disproportionately to Silent Data Corruptions (SDCs)[2] [7], (2) SPMD programs exhibit substantial similarity in the control-data (Section V), and (3) no software technique other than duplication can protect this class of program data.

Duplication, or running two copies of a program and comparing their outputs, has been used to detect errors in sequential programs [8]. 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. Second, duplication requires twice the amount of hardware resources, and hence reduces the resources available for the actual program, thus leading to significant slowdowns [9].

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 [10], [11]. 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,

[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. SDCs are often the most difficult to detect among all failure types.

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. 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.

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

1) Identify generic code patterns that characterize control-data similarity in parallel programs.
2) Develop compiler techniques to statically extract the control-data similarity patterns, and instrument the program with runtime checks corresponding to the patterns.
3) Build a scalable, lock-free monitor for dynamically executing the runtime checks inserted by the compiler.
4) Evaluate BLOCKWATCH on seven SPLASH-2 benchmark programs [12]. 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 16% 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[3].

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).

The rest of this paper is organized as follows: Section II discusses the BLOCKWATCH approach with an example, while Section III details its implementation. Section IV introduces the experimental setup, and Section V presents the evaluation. Section VI quantitatively compares BLOCKWATCH to software-based duplication. Finally, Section VII surveys related work and Section VIII concludes the paper.

II. APPROACH

This section describes the high-level approach of BLOCKWATCH. Section II-A presents the fault model for BLOCKWATCH, while Section II-B lists the assumptions we make about the parallel program. Section II-C uses an example parallel program to illustrate the kinds of similarity considered by BLOCKWATCH. Section II-D illustrates the runtime checks introduced by BLOCKWATCH on the example program.

[3]We measure coverage as fraction of errors that do not lead to SDCs.

A. 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 datapath, 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 faults are relatively 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 paper.

B. 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 a 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.

C. Control-data Similarity in Parallel Programs

We use Figure 1 to illustrate the presence of similarity in the control-data of a parallel program. In Figure 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 18 - 21 in Figure 1. It then executes four branches labeled 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.

We now illustrate the control-data similarity among the program's threads in Figure 1 for each of the four branches in the *slave()* function. The generic code patterns that result in the similarity are shown in Table I. 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 I.
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 I.

TABLE I
BRANCH CONDITION SIMILARITY CATEGORY DEFINITION

Similarity Category	Static characteristics of control data	Branch runtime behaviour similarity
shared	All operands of the instruction are shared variables among threads, such as global variables and constants	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 are in the same group and take the same decision. For example, if the condition comparison statement is an equality comparison between thread ID and shared variables, no more than 1 thread goes through the 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 variable should take the same decision.
none	Local variables that cannot be statically inferred to be similar across threads	No known similarity in branch runtime behaviour among the threads.

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 I.
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 I.

Thus, the control-data for each of the four branches above belongs to a different similarity category according to Table I. 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 II-D.

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 III-A).

D. Runtime Checking

In the previous section, we saw how to statically identify the similarity of the control data used in the branches in Figure 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 is likely to deviate from the statically inferred behaviour. BLOCKWATCH detects the deviation and raises an exception.

```
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
10   for(i = 0; i < nprocs; i++)
11     gp[id].num = rand();
12   for (i = 0; i < nprocs; i++)
13     pthread_create((void *)slave);
14 }
15
16 void slave() {
17   int private, procid;
18   pthread_mutex_lock();
19   //procid is the thread id
20   procid = id++;
21   pthread_mutex_unlock();
22
23   //Branch 1: threadID
24   if (procid == 0) {
25     ...
26   }
27   ...
28   //Branch 2: shared
29   for(i = 0; i <= im - 1; i = i + 1) {
30     ...
31   }
32   ...
33   //Branch 3: none
34   if (gp[procid].num > im - 1) {
35     private = 1;
36   }
37   else {
38     private = -1;
39   }
40   ...
41   //Branch 4: partial
42   if(private > 0) {
43     ...
44   }
45 }
```

Fig. 1. Listing of a 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 I. Note that the comments are not part of the parallel program.

As an example, we use *branch 1* in Figure 1 to explain the runtime checks. As we show in Section II-C, *branch 1* belongs to category *threadID* according to the classification in Table I. 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.

III. Implementation

The implementation of BlockWatch 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 III-A. The second step is to compare the actual runtime behaviours' of the branches with the inferred behavior according to the branches' similarity categories using a runtime monitor, and is described in Section III-B.

A. 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 [13], 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 Section II, 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[4] (*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 II.

Propagation Rules: Before we present the overall algorithm, we first explain Table II. The rows of Table II 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.

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

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 II
Rules to infer instruction's similarity category from its current category and the operand's category

curr inst \ operand	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 II with an example. Assume that the current instruction's similarity category is *partial*. This corresponds to the fifth row in Table II. 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 II is when a local variable is assigned with a shared value in one outcome of an if-else branch but not assigned in another, or is assigned different values in both outcomes. 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 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 2, which is adapted from FFT in the SPLASH-2 Benchmark Suite [12].

In Figure 2, 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 local variable. However, in each invocation of the function, the local variable used in the branch condition is the same, namely *arg*.

```
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 }
```

Fig. 2. Example code of multiple runtime instances of the same branch

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 separately and ensure that we are comparing the values from each call site separately. Another possibility is to merge the values across the call sites, 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 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 instruction's similarity category in Figure 3. The algorithm iterates over all instructions in the program and updates the similarity category of each instruction by calling the *visit* function (lines 4 - 10) 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. *One exception is when the branch's category is* threadID*, where we divide it to two subcategories in the end*: first is branches whose operators are $=$ or $!=$, and branches of this category have no more than 1 thread goes one path while the remaining threads take the other path (see Table II); the remaining branches belong to the second subcategory (branches whose operators are $>$, $<$, etc.). For branches of this subcategory, 1 thread must take the its neighbour threads' branch runtime behaviour if the two neighbour threads take the same decision (e.g. if thread 2 and thread 4 takes the branch, thread 3 also has to take the branch), as the thread ID flows monotonically.

The *visitInst()* function 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()* with the current instruction's category as well as the category of the operand.

```
1  map categorymap;
2
3  main() {
4    bool changed = true;
5    while(changed) {
6      changed = false;
7      for (inst in program) {
8        changed = visitInst(inst) || changed;
9      }
10   }
11
12   for(branch in program) {
13     if (branch in categorymap) {
14       branchcategory =
15         categorymap[branch];
16     }
17     else {
18       branchcategory = "none";
19     }
20   }
21 }
22
23 bool visitInst (inst) {
24   Category category = NA;
25
26   for(op in operands) {
27     if (op is constant/global) {
28       category = lookupTable(
29         category, "share");
30     }
31     else if (op is thread id) {
32       category = lookupTable(
33         category, "threadID");
34     }
35     else if (op in categorymap) {
36       category = lookupTable(
37         category, categorymap[op]);
38     }
39     else{  // op is NA
40       return false;
41     }
42   }
43
44   Category old = categorymap[inst];
45   categorymap[inst] = category;
46   return (category != old);
47 }
```

Fig. 3. Pseudo-code to show the similarity category identification algorithm

The *lookupTable()* function uses Table II 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 II 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 with the example in Figure 2. Table III shows the similarity categories

TABLE III
EXAMPLE OF CATEGORY PROPAGATION ALGORITHM ON FIGURE 2

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

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) [5]. 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.

Optimizations: We perform two optimizations over the base algorithm in Figure 3 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 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.

B. Runtime Checking

This section details the implementation of a runtime monitor to check the statically inferred similar branches in Section III-A. The monitor is spawned as a separate thread in the program[6], 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 behaviours.
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.

[5]In SSA form, instructions and variables are synonymous with each other.
[6]BLOCKWATCH adds instrumentation to the program to spawn the monitor thread.

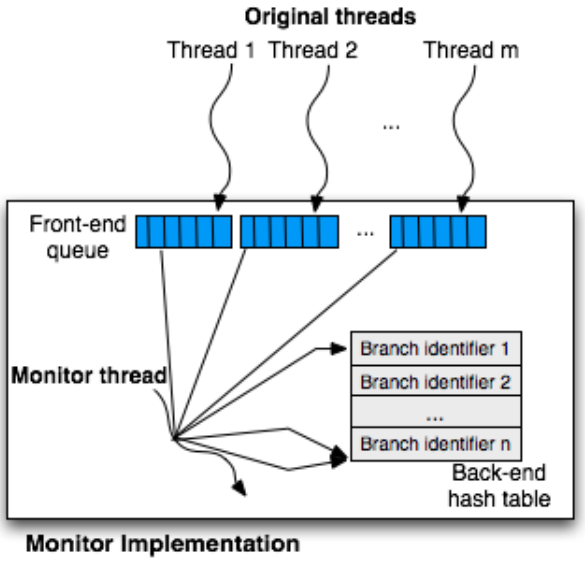

Fig. 4. Monitor architecture

Architecture: We achieve goals 1 and 3 through separate front-end queues 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 4.

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 III-A.
- 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 lock is required as the removal is done from the front of the queue while the insertion is

```
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 /*behaviour*/, loop_iter);
12       ...
13    }
14    else {
15       sendBranchAddr(4 /*static branch ID*/, procid,
16          NOTTAKEN /*behaviour*/, loop_iter);
17    }
18 }
```

Fig. 5. Example code to show the instrumented program

done at the back. Further the queues are of fixed length[7], 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, the same instance of a given branch across different threads will all 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.

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

We illustrate the instrumentation with an example. Figure 5 shows the instrumentation added for *branch 4* in Figure 1. Recall that this branch belongs to the *partial* category. The library calls are highlighted with boldface in Figure 5, 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: The hash table identifier 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).

With the hash table identifier information, we implement the hash table as shown in Figure 6. In Figure 6, the hash table is implemented as a 2-level table. In the first level, the function ID (added by instrumented code) and the static branch identifier is used to generate the key for the map. In the second level, the loop iteration number of all outer loops is used to generate the key for the hash table and the branch runtime behaviours across threads are the value of the hash table. The static branch identifier is put in the first level because different runtime instances of the same static branch share some information (e.g. similarity category) and all runtime instances can share the one copy of data if it is put in the first level. The function IDs of the call sites and the loop iteration numbers are put in two separate levels due to memory and performance overhead considerations (better utilize the memory in L1 table and improve hash table access performance of L2 table).

Performance Optimizations: For performance considerations, the monitor executes asynchronously and does not affect the program execution. For the original program, it only needs to send the branch runtime behaviours to the front-end queue. In order to further improve the performance, we do two main optimizations:

1) *Multi-core optimization:* since we need the loop iteration numbers and call site identifiers to generate runtime identifiers for the hash table, we maintain the information across threads in the same object. This can lead to false sharing[8]. According to the cache coherence protocol [14], 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, different threads have to re-fetch the data in the memory and this will lead to the increase of execution time. Since different cores in a processor have separate L1 and L2 caches, the multi-core optimization makes sure different threads' data occupy different L2 cache lines and hence reduce the false sharing.
2) *Multi-processor optimization:* Although data of different threads are put in different L2 cache lines and it reduces the false sharing, they may be put in the same L3 cache line in current multi-core processors. It won't affect the performance when the program runs on multi-core processors where L3 cache is shared among the cores in a processor. However, when the program runs on multi-processors, L3 cache is not shared among the processors and it will lead to false sharing in L3

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

[8]False sharing happens when one processor core attempts to periodically access data that will never be altered by another core, but that data shares a cache line with data that is altered, and the caching protocol may force the first core to reload the whole cache line despite a lack of logical necessity

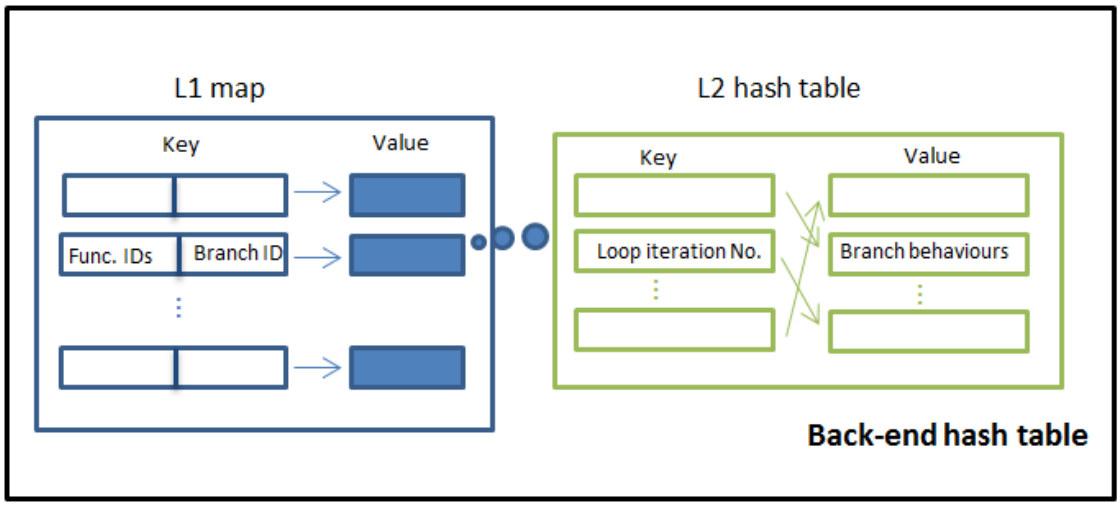

Fig. 6. Back-end hash table

cache line, which will cause the invalidation of L3 cache line when there is a write to the shared cache lines. Therefore, we use thread-local object to store data of different threads and ensure they are not put on the same L3 cache line, which further improves the performance on multi-processors.

IV. Experimental Setup

In this section, 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 fault coverage of BlockWatch.

Implementation Tools: We implement BlockWatch using the LLVM compiler infrastructure [15]. 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. 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 [16].

Benchmarks: We use seven programs in the SPLASH-2 Benchmark Suite [12] for evaluating BlockWatch. The SPLASH-2 Benchmark Suite has been extensively used for studies of shared memory parallel programs. 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 IV describes the characteristics of the seven 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.

TABLE IV
Characteristics of Benchmark Programs

Benchmark	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

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. For execution time with BlockWatch, we measure it under both multi-core optimization and multi-processor optimization cases separately. 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 at least to 64 threads [12].

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 for the program with the monitor). 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. We do

not have a machine with more than 32 cores, and hence we adopt this approach.

False Positives: 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.

Coverage Evaluation: We evaluate the error detection coverage of BLOCKWATCH through fault injection studies. Specially, we focus on detections of Silent Data Corruptions (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 [17]. 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. In the technical report, we also inject faults that only propagate to the flag register and condition data of the instrumented branches (branches that belong to *shared*, *threadID* or *partial*), and we also inject faults that propagate to any instructions in the program. 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 find how many faults are activated[9]. We calculate the coverage as the probability that an activated fault will not lead to an SDC [18]. 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/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.

V. RESULTS

In this section, we present the results of the evaluation of BLOCKWATCH. First, we 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 for the programs.

A. Similarity Category Statistics of Branches

We run the static analysis part of BLOCKWATCH on the seven SPLASH-2 programs. Table V shows the number of branches in each program that fall into the similarity categories in Table I, 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.

TABLE V
SIMILARITY CATEGORY STATISTICS OF THE BRANCHES IN 7 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%)

The results in Table V are as follows. In general, between 49% to 98% of the branches fall into the *shared*, *threadID* and *partial* categories. This means the BLOCKWATCH is able to

[9]An activated fault is one that is exercised in the system in some way. Over 75% of the injected faults are activated in our experiments.

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 III-A.

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.

B. Performance Overheads

Comparison of multi-core optimization and multi-processor optimization: Figure 7 shows the execution time of the seven SPLASH-2 programs with BLOCKWATCH for multi-core optimization and multi-processor optimization in 4-thread case. 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. It shows that the geometric mean of the performance overhead for multi-core optimization is 2.65X, and the multi-processor optimization brings the overhead down to 2.15X. The reason is that the computer used for evaluation is a multi-processor and L3 cache is not shared across processors. Multi-processor optimization reduces false sharing of BLOCKWATCH in L3 cache and hence decreases the performance overhead. Note that the performance results hereafter are data with multi-processor optimization.

Performance overhead: Figure 8 shows the execution times of the seven 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.

From Figure 8, the geometric mean of the performance overhead of BLOCKWATCH is 2.15X with 4 threads, and 1.16X with 32 threads. Thus *the performance overhead of* BLOCKWATCH *with 32 threads is only 16%*, and is lower than that for 4 threads (see below for why).

Scalability: We study the scalability of BLOCKWATCH by considering the variation of the geometric mean of the performance overheads (across all 7 programs) with the number of threads. The results are shown in Figure 9 as the number of threads is varied from 1 to 32.

In Figure 9, 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 schedules 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.

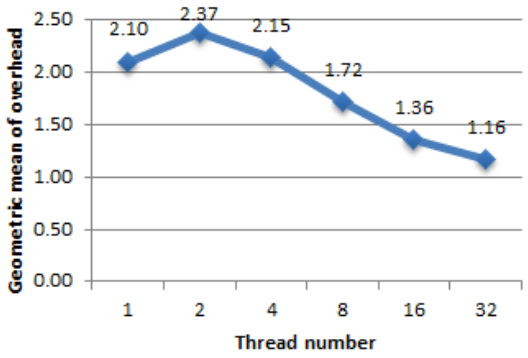

Fig. 9. Geometric mean of BLOCKWATCH overhead (baseline is program without BLOCKWATCH) Vs. number of threads

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 on BLOCKWATCH reduces and so does the overhead of BLOCKWATCH (upto 32 threads in Figure 9).

C. Error Detection Coverage

We study the coverage (for SDC) with BLOCKWATCH under five kinds of faults: branch-flip faults, branch-condition faults, branch-flip faults in instrumented branches, branch-condition faults in instrumented branches and regular instruction faults. The first type of fault is guaranteed to flip the branch but does not corrupt any program data directly. The second type of fault corrupts the branch's condition data but does not necessarily lead to branch flip. The third and the forth types of faults are similar to former two types, but they are injected only into the instrumented branches. The final type of faults that propagate to any instructions.

The results are shown in Figure 10 to Figure 14. Note that the $coverage$ of y axis in all five figures starts 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.

1) Coverage results for branch-flip faults: Figure 10 shows the $coverage$ with and without BLOCKWATCH for all programs under branch flip faults. Across the programs, the average $coverage_{original}$ is 83%, while average $coverage_{\text{BLOCKWATCH}}$ is 97% for the 4-thread program, and 98% for the 32-thread program. 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 six of the seven programs.

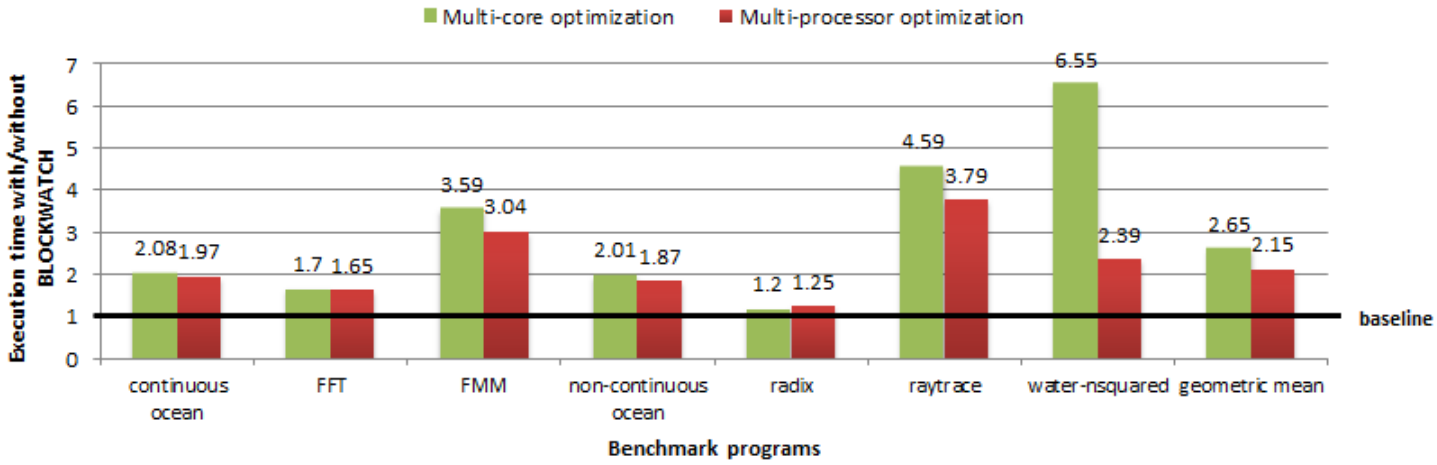

Fig. 7. Execution time of program with BLOCKWATCH/ execution time of program without BLOCKWATCH for multi-core optimization and multi-processor optimization in 4-thread case. Lower is better

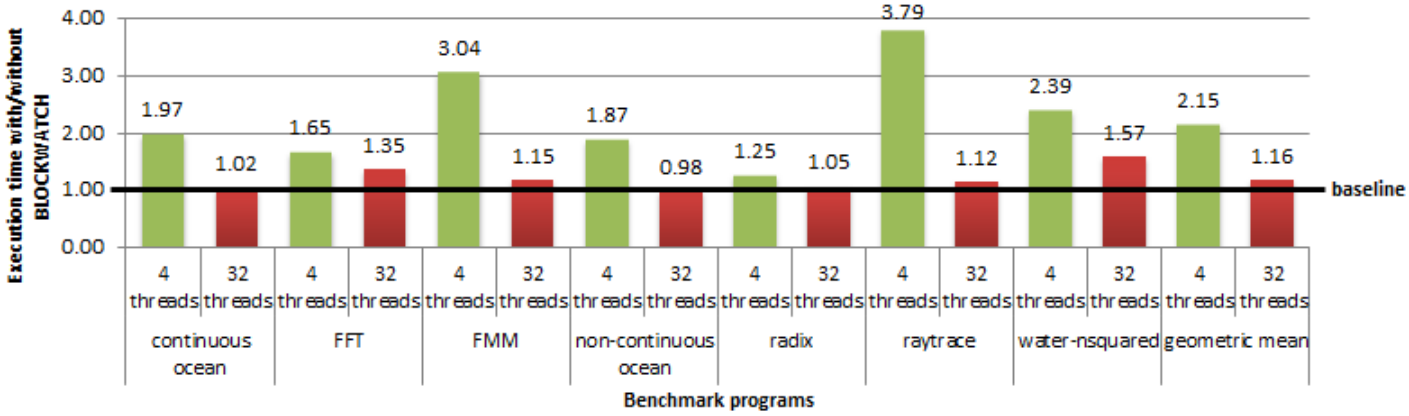

Fig. 8. Execution time of program with BLOCKWATCH/ execution time of program without BLOCKWATCH. Lower is better

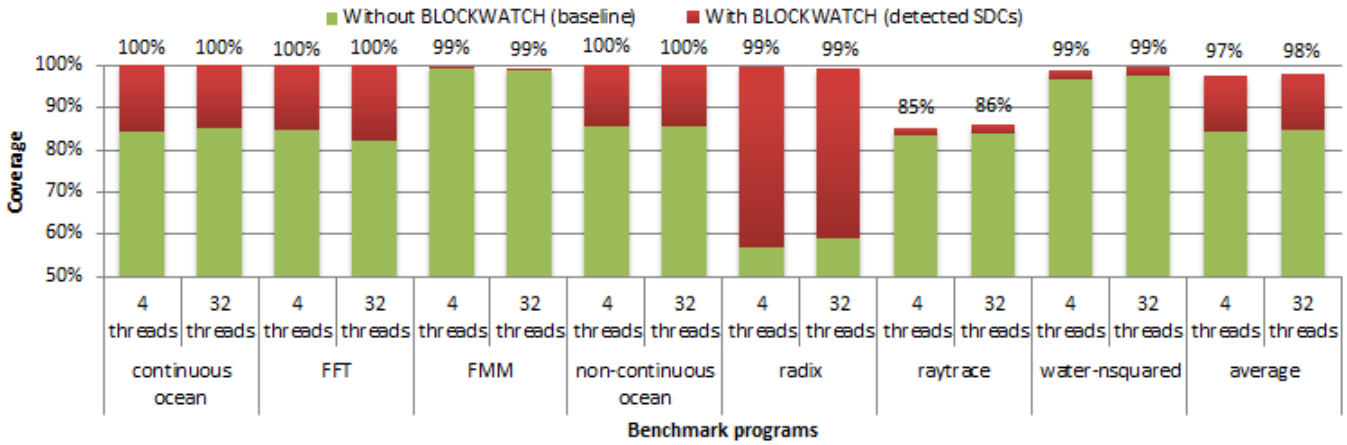

Fig. 10. $coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for branch-flip faults: The dark part is due to the detection provided by BLOCKWATCH. Higher is better.

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, BLOCKWATCH uses the outer loops' iteration numbers to generate the hash table key for a branch (Section III-B). However, due to the 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 many loops deeper than six levels of nesting.

2) Coverage results for branch-condition faults: Figure 11 shows the results of $coverage$ of the seven programs both with and without BLOCKWATCH, when faults are injected into the branch's condition data. The results are similar to those in Figure 10. For example, when BLOCKWATCH is used, the coverage increases from 90% to 97% for both the 4-thread and 32-thread cases. However, the average $coverage_{original}$ value is 90%, which is much higher than the $coverage_{original}$ for branch-flip faults (average 83%). 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 SDCs.

3) Coverage results for branch-flip faults and branch-condition faults in instrumented branches: Figure 12 and Figure 13 show the results of $coverage$ of the seven programs both with and without BLOCKWATCH, when faults are injected into the instrumented branch's flag register and condition data. For continuous ocean, FFT and non-continuous ocean, The results are consistent with the corresponding results in Figure 12 and Figure 13. This is because we instrument almost all branches in the parallel section(98%). For the remaining 4 benchmarks, the $coverage$ for instrumented branches is higher than the corresponding $coverage$ for all branches.

4) Coverage results for regular instruction faults: Figure 14 shows the results of $coverage$ of the seven programs where faults are propagated to any regular instructions. The results show that BLOCKWATCH detects a small proportion of the SDCs. The reason is that the faults that propagate to the regular instructions may not corrupt the control-data and lead to branch flip, which is the errors that BLOCKWATCH is applicable to.

VI. DISCUSSION

In this section, we compare the error-detection coverage and performance overhead of BLOCKWATCH with that of software-based duplication.

Coverage: Duplication is a general technique that can protect programs from a large class of errors. However, we focus on control-data errors in this section as this is the focus of BLOCKWATCH. 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 6 of the 7 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, we can 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.

Performance: The average performance overhead of BLOCKWATCH is 115% for 4 threads and 16% for 32 threads. In contrast, software-based duplication incurs overheads of 200% to 300% for sequential programs [9]. Although this overhead can be reduced through the use of speculative optimizations, 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 through using determinism inducing techniques [19], [20]. 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.

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 subgroup of threads. This is an avenue for future work.

In this study, we use a sufficiently large fixed-size queue for each thread to buffer the branch runtime behaviours. it is not memory efficient and sometimes it is not easy to decide an appropriate queue size. To further optimize it, we can implement a queue array for each thread and dynamically allocate and deallocate a fixed-size queue when necessary. In

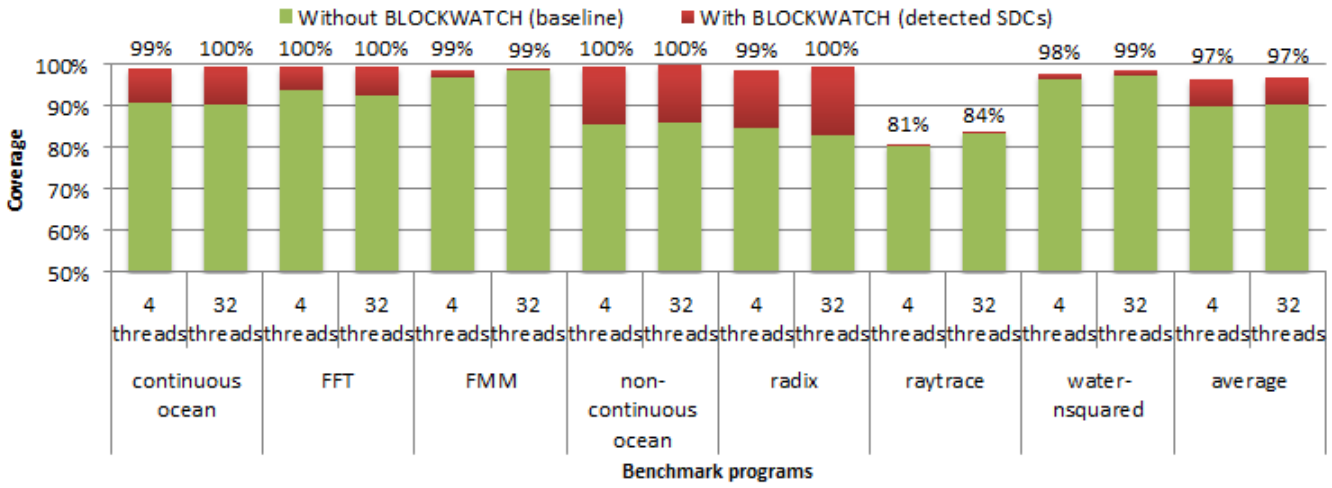

Fig. 11. $coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for branch-condition faults: The dark part is due to the detection provided by BLOCKWATCH. Higher is better

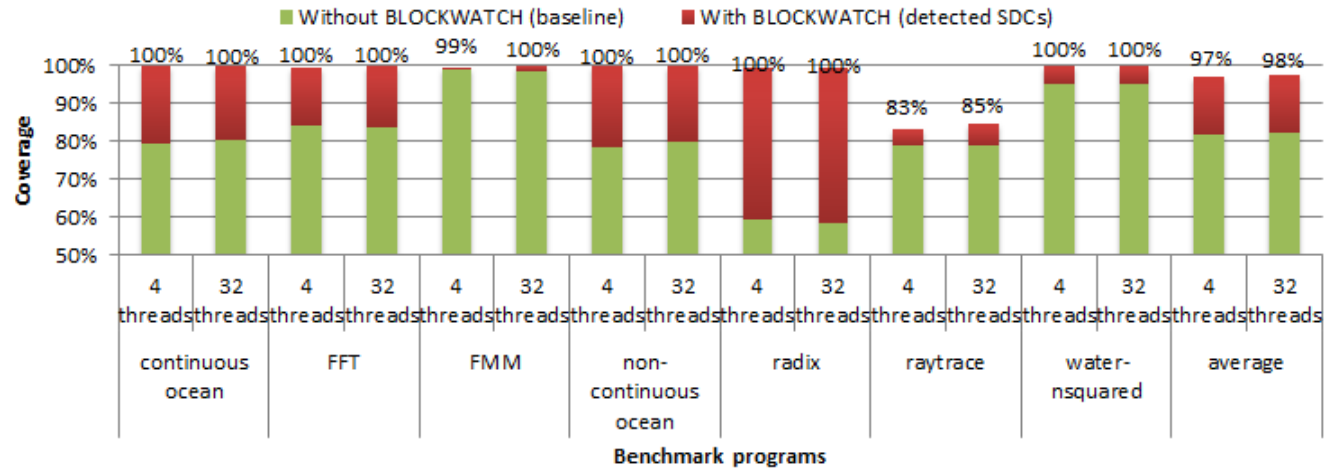

Fig. 12. $coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for branch-flip faults in instrumented branches: The dark part is due to the detection provided by BLOCKWATCH. Higher is better.

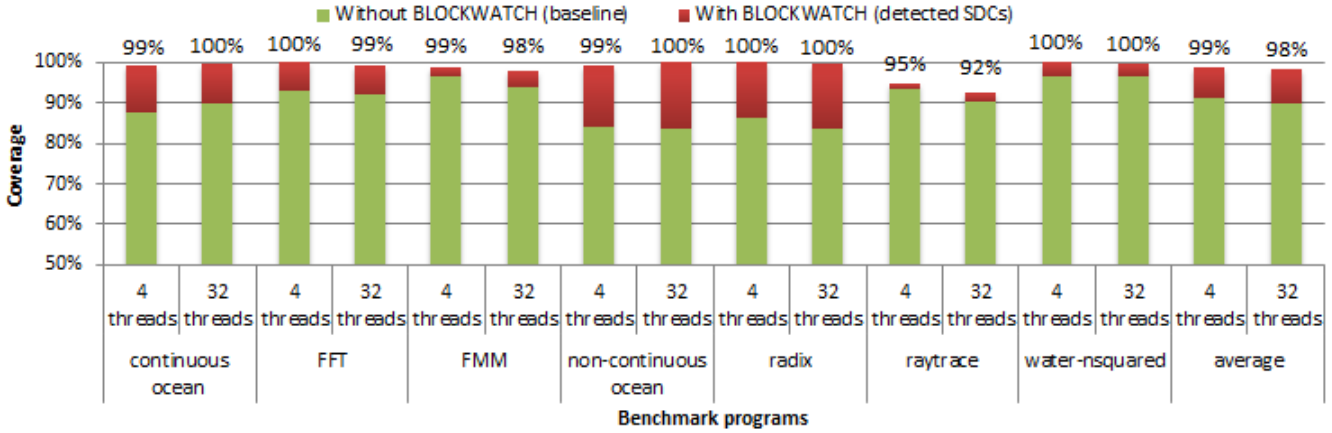

Fig. 13. $coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for branch-condition faults in instrumented branches: The dark part is due to the detection provided by BLOCKWATCH. Higher is better

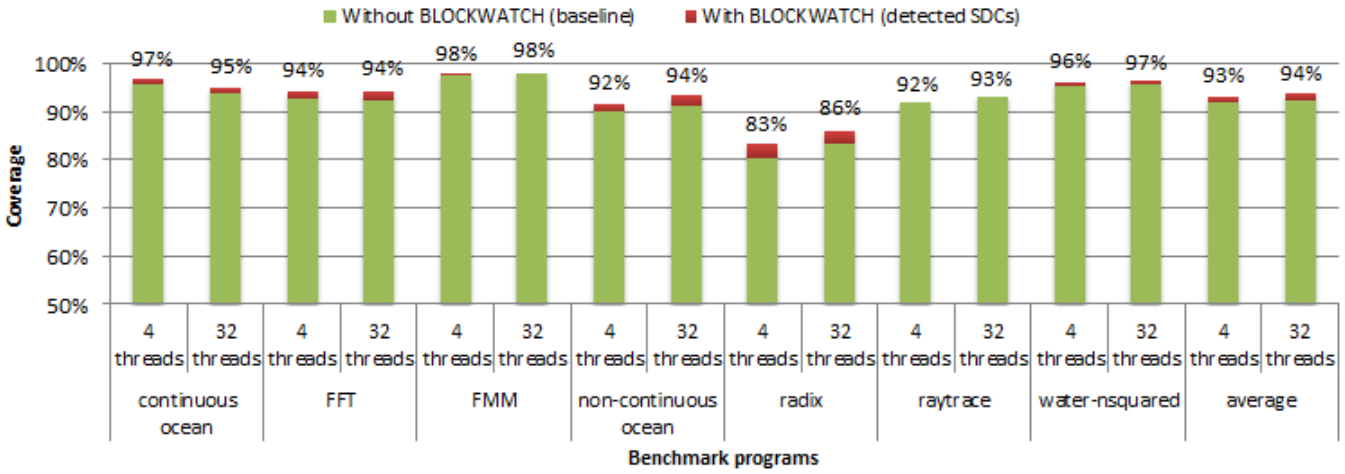

Fig. 14. $coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for regular instruction faults: The dark part is due to the detection provided by BLOCKWATCH. Higher is better

this way, we can dynamically change the front-end queue's size while guaranteeing lock-free architecture.

VII. Related Work

We classify related work on error detection into five broad categories. For brevity, we consider only software techniques, as BLOCKWATCH is software-based. Also, we discuss duplication in detail in Section VI, and do not consider it here.

Control-flow checking: Control-flow Checking (CFC) techniques such as ECCA [21], PECOS [22] and CFCSS [23] 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 behaviour. This is the class of errors that BLOCKWATCH detects.

Statistical techniques: AutomaDeD [10] 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. [11] 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 [24] 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 [25] 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 runtime hardware errors.

Static and Dynamic Analysis: Static analysis has been extensively used for verifying in parallel programs [26], [27]. In these cases, the goal is to find bugs in the program, rather than detect runtime errors arising in hardware. Pattabiraman et al. [28] 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 [29], [30], [31]. These techniques target only sequential programs, and hence do not consider similarity across threads . In recent work, Yim et. al. [32] 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 [33], [34]. In recent work, Sloan et al. [35] 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. While these techniques are efficient, they only protect programs of the specific type they target. In contrast, BLOCKWATCH targets general-purpose parallel programs.

VIII. CONCLUSION

This paper 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. Experimental results show that BLOCKWATCH increases the average SDC coverage across seven programs from 83% (90%) to 97% for branch-flip faults (branch-condition faults), while incurring only 16% overhead in the 32 thread case (on a 32 core machine). BLOCKWATCH is automated, incurs zero false-positives and can run on unmodified hardware and on existing programs.

REFERENCES

[1] S. Borkar and A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

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

[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proc. of the 44th Annual Design Automation Conference*, 2007, pp. 746–749.

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, 2002.

[6] F. Darema, "The SPMD model: Past, present and future," in *Proc. of the 8th Euro. PVM/MPI Users' Group Meeting*, 2001, p. 1.

[7] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. Chong, "Characterization of error-tolerant applications when protecting control data," in *IEEE Int'l Symposium on Workload Characterization*, 2006, pp. 142–149.

[8] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software implemented fault tolerance," in *Proc. of the Int'l Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[9] Y. Zhang, J. Lee, N. Johnson, and D. August, "DAFT: decoupled acyclic fault tolerance," in *Proc. of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 87–98.

[10] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz, "AutomaDeD: Automata-based debugging for dissimilar parallel tasks," in *DSN*, 2010, pp. 231–240.

[11] A. Mirgorodskiy, N. Maruyama, and B. Miller, "Problem diagnosis in large-scale computing environments," in *Proc. of ACM/IEEE Conference on Supercomputing*, 2006, pp. 88–100.

[12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, 1995, pp. 24–36.

[13] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[14] J. Archibald and J.-L. Baer, "Cache coherence protocols: evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, pp. 273–298, 1986.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

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

[17] V. Reddi, A. Settle, D. Connors, and R. Cohn, "PIN: a binary instrumentation tool for computer architecture research and education," in *Proc. of Workshop on Computer Architecture Education*, 2004, p. 22.

[18] R. Alexandersson and J. Karlsson, "Fault injection-based assessment of aspect-oriented implementation of fault tolerance," in *DSN*, 2011, pp. 303–314.

[19] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Loose synchronization of multithreaded replicas," in *Proc. of 21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 250–255.

[20] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *ACM SIGPLAN Notices*, vol. 44, no. 3, 2009, pp. 97–108.

[21] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.

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

[23] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[24] Q. Gao, F. Qin, and D. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *Proc. of ACM/IEEE Conference on Supercomputing*, 2007, p. 15.

[25] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in MPI libraries via message flow checking," in *SC*, 2010, pp. 1–11.

[26] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," *ACM PLDI*, vol. 41, no. 6, pp. 308–319, 2006.

[27] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for MPI programs," in *SC*, 2010, pp. 1–10.

[28] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-aware error detectors using static analysis," in *IEEE On-Line Testing Symposium, IOLTS 07.*, 2007, pp. 211–216.

[29] M. Hiller, A. Jhumka, and N. Suri, "On the placement of software mechanisms for detection of data errors," in *DSN*, 2002, pp. 135–144.

[30] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. of the 24th Int'l Conference on Software Engineering*, 2002, pp. 291–301.

[31] S. Sahoo, M. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *DSN*, 2008, pp. 70–79.

[32] K. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *IPDPS*, 2011, pp. 287–300.

[33] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518–528, 1984.

[34] J. Plank, Y. Kim, and J. Dongarra, "Algorithm-based diskless checkpointing for fault tolerant matrix operations," in *Fault-Tolerant Computing (FTCS)*, 1995, pp. 351–360.

[35] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *DSN*, 2010, pp. 161–170.