FlipBack: Automatic Targeted Protection Against Silent Data Corruption

Xiang Ni†, Laxmikant V. Kale⋆

†IBM T.J. Watson Research Center, Yorktown Heights, New York 10598
⋆Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801
E-mail: †xiang.ni@ibm.com, ⋆kale@illinois.edu

***Abstract*— The decreasing size of transistors has been critical to the increase in capacity of supercomputers. The smaller the transistors are, less energy is required to flip a bit, and thus silent data corruptions (SDCs) become more common. In this paper, we present FlipBack, an automatic software-based approach that protects applications from SDCs. FlipBack provides targeted protection for different types of data and calculations based on their characteristics. It leverages compile-time analysis and program markup to identify data critical for control flow and enables selective replication of computation by the runtime system. We evaluate FlipBack with various HPC mini-applications that capture the behavior of real scientific applications and show that FlipBack is able to protect applications from most silent data corruptions with only $6-20\%$ performance degradation.**

***Index Terms*—Software reliability, Fault detection, Computer errors, Redundancy**

I. Introduction

Reliability is one of the most important characteristics expected from computer systems. Effectively utilizing hardware resources is difficult if faults are frequently encountered. In high performance computing (HPC), where the system comprises hundreds of thousands of components, even making sure that all components are functional at all the times is a daunting task. This is because although the mean time to failure (MTTF) of individual components is high, the aggregate MTTF of the full system is low due to the large number of system components. As a result, hardware vendors and software designers have made a significant effort to enable a smooth user experience even in the presence of fail-stop failures [18].

Besides fail-stop failures, the major deterrent to achieving reliability is presence of soft errors. A soft error typically results from transient faults caused by electronic noise or high-energy particle strikes. For example, silent data corruption (SDC) may occur due to transient bit-flips [21]. SDCs are becoming more prevalent in HPC as lower power chips with smaller feature sizes are being developed. Research has shown that there exists a strong correlation between an increase in soft error rate and a decrease in device sizes and operating voltages [10]. Even recent and current systems face a modest number of soft errors. For example, ASC Q system at Los Alamos National Laboratory experienced on average 26.1 CPU failures induced by cosmic rays per week [20]. On Jaguar, double-bit errors were observed once every 24 hours in Jaguar's 360TB memory [1].

In HPC, redundancy and duplication are being proposed and explored to tolerate SDCs [9], [12], [13], [22]. With these approaches, correctness is guaranteed by repeating application execution using methods such as instruction-level duplication and replica creation. However, due to duplication, the maximum efficiency achievable in most of these approaches in comparison to the base version is 50%. This paper aims at addressing this limitation of duplication-based approaches by exploiting datatype-specific methods for addressing SDCs.

Our work is motivated by the following observation made in [11]: an application execution on a system is reliable if it satisfies two conditions – 1) it computes the data correctly, i.e. the execution performs all the work in the right order and 2) it computes the correct data, i.e. when the expected work is performed, no errors are made. For example, results of branch instructions determine the program flow and hence any miscalculation in computing the result of a branch instruction leads to the first condition not being fulfilled. Data corruptions in any other part of the execution prevent the second condition from being satisfied.

In a typical HPC application, different types of data may be critical for carrying out these two steps and for determining their correctness. Additionally, different methods may be more effective and efficient for protecting different types of data. We explore these possibilities and show that significant reliability and performance improvements can be obtained by using datatype-specific methods.

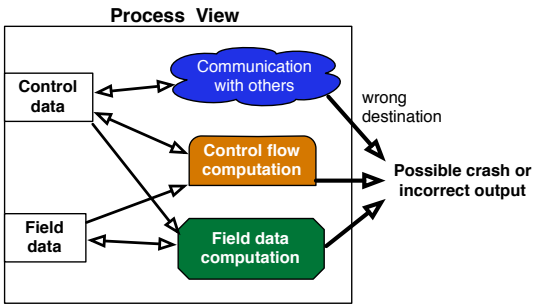

Fig. 1: Different types of data and their effect on reliability.

†This work was done when Xiang Ni was at University of Illinois.
SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

Broadly speaking, as shown in Fig. 1, we divide the data of an application into two categories: *control* data and *field* data. Control data, as its name suggests, is used to determine the flow of a program, e.g. an iteration counter, number of message receives, or rank of neighboring process. A corruption in control data can make the program perform incorrect computation, communicate with wrong processes, or even crash. Hence, SDCs in them should be detected and corrected as soon as possible. On the other hand, field data is used in most of the computation work and typically leads to incorrect output. Application-specific methods for ensuring reliability of field data may be sufficient and faster.

Building upon the distinction between the control data and the field data, we have developed a framework, called *FlipBack*, that provides automated targeted protection against SDC. The main contributions of our work that are described in detail in the rest of this paper are:

- We present the design of a novel framework that automatically adjusts the protection mechanism based on the data and calculation types (§ III, § IV, § V).
- We demonstrate the role of compiler in automatic identification of data critical for control flow and of runtime system in selectively replicating computation to improve application reliability (§ III, § IV).
- We evaluate FLIPBACK with two proxy applications and show that 80% to 100% soft error coverage can be provided by the framework with additional 6% to 20% performance overhead.

II. BACKGROUND AND RELATED WORK

The key ideas on which FLIPBACK is based are broadly applicable. However, as a proof of concept, we have implemented FLIPBACK in the CHARM++ runtime system [2]. The reasons for choosing CHARM++ as our test bed are explained at the end of Section II-A.

A. CHARM++

CHARM++ is a general purpose parallel programming framework based on the concept of overdecomposed units created by programmers [2]. Control flow in a CHARM++ program is written in terms of these units and the user-level communication is also directed at these units. Under the hood, a runtime system (RTS) manages the placement of the units on processes and redirects communication based on it. The execution order of the units is based on availability of messages for them and is guided by the RTS. Several large applications take advantage of the CHARM++ methodology and scale to very large HPC systems [2], [24], [15], [19]. Here, we briefly describe a few key concepts related to CHARM++ that are relevant to this paper.

Chares: C++ objects are used to represent work and data units in CHARM++ and are referred to as chares. For different types of work and data units, different C++ classes, i.e. chares, should be defined by the programmers. The decomposition granularity is typically chosen such that the number of chares is much larger than the number of cores (e.g. 8×). Fig. 2 (left) shows an example in which from an application's point of view, two types of work units exist with five objects in total (named *A1, A2, A3, B1, B2*). These C++ objects remain inactive till messages arrive for them and are then scheduled for execution by the RTS.

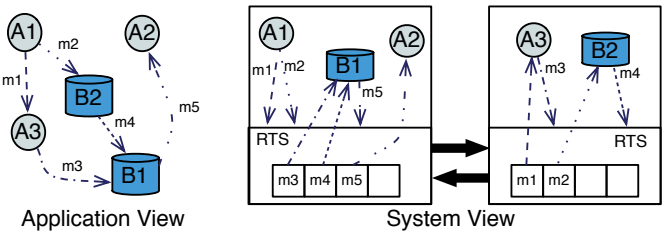

Fig. 2: Charm++ execution model. Applications define work units and their interactions. The runtime system manages execution and communication of these work units by mapping them onto processes.

Entry methods: Specially designated member functions, called entry methods, form the basic unit of execution of chares. They also provide the mechanism for communication defined in terms of destination chares and designated entry methods. For example, in Fig. 2 (left), when chare *A1* is executed, it invokes entry methods (*m1, m2*) on other chares (*A3, B2*). These possibly remote invocations lead to execution of these chares on the processes at which they reside.

CHARM++ RTS: The RTS is responsible for following key tasks in CHARM++ among others: 1) mapping and remapping chares to processes, 2) transforming communication defined by programmers among chares to communication among processes, 3) executing entry methods on chares. On the right side of Fig. 2, the corresponding RTS view of the application view (shown on left) is presented assuming only two processes. Chares *A1, A2, B1* are mapped to the first process, while chares *A3, B2* are on the second process. When chare *A1* invokes entry method *m1* on chare *A3*, the RTS records the request and returns the control to chare *A1*. After the execution of *A1's* entry method is completed, the RTS finds the process on which chare *A3* resides and forwards a message that requests execution of entry method *m1* on chare *A3* to that process. At the destination process, the message is enqueued into a message queue and eventually leads to execution of the member function *m1* on the chare *A3*.

We have chosen CHARM++ to implement a proof-of-concept of our ideas for two reasons. First, the notion of chares is highly suitable for containing faults. Since a chare only modifies the data that it owns, if the silent data corruptions can be detected and corrected within a chare, their propagation to the rest of the application data can be avoided. Second, since the runtime system manages what work is executed on which chares at what time, it provides opportunities to insert hooks for FLIPBACK to detect and correct SDCs.

B. Related work

Software instruction duplication: SWIFT [25] is a fully software-based instruction duplication technique that leverages unused instruction level parallelism to schedule duplicated computation. SWIFT provides good failure coverage except when bit flips occur between verification and actual use of data or when bit flips change the opcode of non-store instructions to store. However, it doubles the number of dynamic instructions and thus leads to significant cost in terms of performance and power. Shoestring [11] enhances SWIFT with symptom-based detection and only applies instruction duplication to the code segments that are more vulnerable to silent data corruptions. Compared to FLIPBACK which is a more generic approach based on the characteristics of HPC applications, Shoestring relies on hardware-specific knowledge. HAUBERK-NL [30] duplicates the non-loop computation in GPU programs and applies value-ranging check to protect the loop portions. IPAS [16] uses machine learning techniques to find the instructions that are more likely to cause silent data corruptions and only protect those instructions by duplication. However, it requires extra training time that can be substantially more costly than a compiler and runtime-based approach.

Redundancy techniques: Redundancy techniques have been deployed in both hardware and software. IBM S/360 [27] and HP NonStop systems [3] use large scale modular redundancy to provide fault tolerance. Watchdog [5] uses an extra processor to detect errors by monitoring the behavior of the main processor. ACR [22] enhances the traditional checkpointing scheme with replication to detect and correct both hard failures and silent data corruptions. RedMPI [13] compares messages received by the original and replicated MPI processors in order to detect silent data corruptions.

Anomaly based detection: Based on the characteristics of application data, different techniques have been proposed to compare the application data with expected values in order to detect silent data corruptions. Detection based on temporal and spatial similarity of application data has been proposed and explored in various ways. Yim et al. [29] computes a histogram of application data to detect outliers in conjunction with temporal and spatial similarity. Sheng et al. [7] have designed a detector that can select the best-fit prediction method during execution and automatically adjust the detection range according to the false-positive events observed. Bautista-Gomez et al. [4] use multivariate interpolation to detect and correct silent data corruptions in stencil applications. All these different anomaly based detection techniques can be easily plugged into FLIPBACK to improve its failure coverage.

Algorithm based fault tolerance: Specially designed algorithms can protect different computation kernels with low cost and high detection coverage. Examples of such algorithms are [26] for sparse linear algebra, [23] for iterative solvers and [8] for LU factorization. However, this approach is not applicable to every application and requires significant development time. In comparison, FLIPBACK is a generic approach that can be used for a wide variety of of HPC applications.

III. RUNTIME GUIDED REPLICATION

FLIPBACK is built upon the concept of datatype specific protection from silent data corruptions. It views the application data as a collection of control data and field data. For each of these data types, it deploys different mechanisms to protect them against silent data corruptions. In this section, we describe how *runtime guided replication* can protect the data and computation related to program control flow. In Section IV, we discuss the method that is used to protect transient data followed by Section V in which techniques to protect field data are introduced.

A. Motivating Examples

Fig. 3 shows two code snippets written in CHARM++ that represent common use cases for HPC programs. Fig. 3a consists of a stencil computation in which relaxation-updates are applied on grid-based data. The control flow of this program is as follows: at the beginning of every iteration (*beginNextIter*), each chare (C++ object) checks if the iteration count (*iterCount*) has reached the total iterations desired (*totalIter*) and if the program should exit. If not, each chare sends the boundary data to its neighbors to begin the next iteration. At this point, all chares wait to receive messages from their neighbors. When a message arrives, the ghost data received is processed (*receiveMessage*). When all the ghost messages are received, computation is triggered (*invokeComputation*) and the control is passed to the *beginNextIter* function.

In stencil computation described above, certain pieces of data used for calculation play an important role to ensure that the program flow is correct. For example, if the value of *msgCount* is modified due to bit flips, the program may hang. This is because some processors may now expect to receive more number of ghost messages than they should, and thus lead to a logical error. Similarly, if a bit flip happens during the computation of *iterCount*, the program may either execute fewer or more iterations than originally specified. Finally, if the calculation of the message destination is corrupted, the program may either compute using the wrong ghost data or may hang because of the incorrect delivery of messages. We refer to variables like *msgCount* and *iterCount* as *control variables* since their correctness affects the program's control flow significantly.

Although the molecular dynamics simulation program shown in Fig. 3b has a different control flow structure in comparison to stencil computation, we still observe that certain calculations are more vulnerable than others. For example, the calculation of the destination where each particle should migrate to (*id*) is essential for correctness. If corruption affects such calculations, particles may move to the wrong chare and soon silent data corruptions will propagate through the entire system.

What we learn from these code examples is that protection of the computation involving control variables and message destinations is necessary to ensure that the program executes the right computation. Hence, to protect these code regions from silent data corruptions, we *conduct recomputation* using

```
1  void Stencil::beginNextIter()
2  {
3    iterCount++;
4    if(iterCount >= totalIter){
5      mainProxy.done(); //program exits
6    }else{
7      for(int i = 0; i < totalDirections; i++)
8      {
9        ghostMsg * m1 = createGhostMsg(dirs[i]);
10       copy(m1->data, boundary[i]);
11       int sendTo = myIdx+dirs[i];
12       stencilProxy(sendTo).receiveMessage(m1);
13     }
14   }
15 }
16 void Stencil::receiveMessage(ghostMsg * m)
17 {
18   msgCount++;
19   processGhostMsg(m);
20   if(msgCount == numMsgExpected){
21     msgCount = 0;
22     thisProxy(self).invokeCompute();
23   }
24 }
25 void Stencil:invokeCompution()
26 {
27   //computation routine
28   for(int i = 0; i < size; ++i){
29     temperature[i] = ...
30   }
31   thisProxy(self).beginNextIter();
32 }
```

(a) Stencil computation

```
1  void Cell::compute()
2  {
3    for(each p in particles){
4      computeTotalForce(p);
5    }
6    cell(self).startMigration();
7  }
8  void Cell::startMigration()
9  {
10   for(each p in particles){
11     int id = findOwner(p);
12     if(id != myID){
13       cell(id).migrateParticl(p);
14     }
15     deletedParticles.push_back(p);
16   }
17 }
```

(b) Molecular dynamics simulation

Fig. 3: Code snippets from scientific applications.

the same inputs. The results thus obtained are compared with the control variables and message destinations obtained from the original computation. If there is a mismatch, we detect that silent data corruption has happened.

B. Compiler Slicing Pass

Re-executing all the computation is very expensive since it doubles the execution time of the program and potentially doubles the memory requirement as well. In order to reduce the recomputation time and memory overhead, FLIPBACK introduces a *slicing pass* into the standard compiler backend

Input:
f: the targeted function to perform slicing on
c: set of control variables
Output: slices: the program slice for recomputation

```
// search for slicing criterions
1  foreach Instruction I in f do
2      if Defs(I) ⊂ c or I sends messages then
3          criterions.push(I);
4      end
5  end
6  while !criterions.empty() do
7      I←criterions.top(); criterions.pop();
8      if !I.processed() then
9          slices.push(I);
           // data flow analysis
10         foreach Values I′ in Uses(I) do
11             foreach Instruction I″ in Defs(I′) do
12                 if I″ may lead to I then
13                     criterions.push(I″);
14                 end
15             end
16         end
           // control flow analysis
17         foreach BasicBlock B that may lead to I do
18             criterions.push(B.getTerminator());
19         end
20     end
21 end
```

Algorithm 1: Slicing pass to find recomputation region.

to limit the recomputation scope. A program slice is part of the program that affects the values computed at some points of interest. In the slicing pass, FLIPBACK first identifies the instructions that modify the value of either a control variable or variables that affect communication. We refer to those instructions as *critical* instructions. Next, FLIPBACK finds all other instructions that directly or indirectly affect the *critical* instructions. In this way, all the instructions whose execution can potentially affect the computation of control variables are discovered.

Having identified the program slice for control variables and message destinations, recomputation is performed only on those instructions. We have found that this reduces the overheads of recomputation significantly. Moreover, only control data and messages need to be duplicated for recomputation. Typically, the memory space used by control data and messages is much less than the other data used in scientific computation. Thus, the slicing pass not only reduces the computation time but also relieves memory pressure.

Algorithm 1 shows how the slicing pass works. First the slicing pass finds the slicing criterions: instructions that either assign values to control variables or sends messages (line $1-5$). Using the example shown in Fig. 3a, the code on lines 3, 5, 12, 18, 21, 22, and 31 will be selected as slicing criterions after this step. Next, in order to find all the instructions that

```
1  //mark control and field data
2  addControl(&msgCount);
3  addControl(&iterCount);
4  addField(&particles);
5  ------------------------------
6  //annotate important members in messages
7  class ghostMsg
8  {
9    control int direction;
10   double * data;
11 }
```

Fig. 4: Annotations of the control and field variables.

Input: **o**: the original full fledged chare
s: the shadow chare
// RTS receives a message *M* for **o**

```
1  checkpointControl(o);
2  checkpointControl(s);
3  restart←true;
4  while restart do
       // buffering outgoing messages
5      o.invoke(M); s.invoke(M);
6      if compareControl(o, s) and compareMsgs(o, s) then
7          restart←false;
8          sendMsgs(o); deleteMsgs(s);
9      end
10     else
11         restartControl(o); restartControl(s);
12     end
13 end
```

Algorithm 2: Workflow in RTS.

may affect the instructions in the slicing criterions, we conduct backward data flow analysis and control flow analysis. To better understand the process of data flow analysis, let us look at line 12 in Fig. 3a. According to Algorithm 1, first we need to find the values used in that instruction (line 10), which are *sendTo* and *m1*. The definitions of *sendTo* and *m1* are at line 11 and 9 in Fig. 3a. Since line 12 is in the execution path from either line 11 or 9 to the end of the function, those two new lines are added to the *criterions* and *slicing* set. Similarly, the definition of *i* in line 7 is also be included in the *slicing* set due to the data flow analysis. Control flow analysis can be illustrated using line 21 in Fig. 3a. Since branch instruction at line 20 acts as the terminator instructions of the basic blocks that lead to the execution of line 21, it is also included in the *slicing* set.

Note that although line 10 in Fig. 3a should be part of the slice according to Algorithm 1, we prune it since the content of the message depends on non-control variables that can be protected using other methods. One limitation of the presented slicing tool is that inter-procedure and pointer analysis is not supported but we plan to address that in the future.

C. Runtime Support

The runtime system (RTS) component of FLIPBACK has multiple roles. It is responsible for local checkpoint/recovery, recomputation and verification. When a chare is created, FLIPBACK automatically creates a corresponding "shadow" chare that is invisible to users for recomputation. Shadow chares only execute the program slice obtained from the compiler pass. Thus, another compiler pass is needed after the original slicing pass to prevent shadow chares from executing instructions that do not belong to the slice by adding conditional instructions. FLIPBACK allows users to mark control and field data using a simple API as shown in Fig. 4. The RTS creates shadow chares using default constructors and then copies the value of control variables from the original chare. As for the field data, RTS ensures that shadow chares and the original chares point to a common copy of the variables.

Algorithm 2 shows the interaction between the original and shadow chares in the runtime system. Broadly speaking, the execution model is transaction based at the level of entry method executions. Actions that affect other chares can only be committed once it is confirmed that there are no silent data corruptions. As shown in Algorithm 2, when a message *M* targeted at the original chare is received, the RTS first checkpoints the control variables in both the original chare and the shadow chare (line $1-2$). Next, the RTS invokes the entry method associated with *M* on the original chare. During the execution of the original chare, all the outgoing messages are buffered. Then, the RTS invokes the same entry method on the shadow chare and buffers all the outgoing messages as well. As mentioned earlier, when the entry method is invoked on the shadow chares, only the instructions that are part of the sliced set will be executed.

In the end, control variables and buffered messages obtained from the original chare and the shadow chare are compared (line 6). If there is mismatch between the data from the two sources, the RTS rollbacks both the shadow chare and the original chare to the beginning of the entry method execution using the checkpoints of the control variables, and then repeats the executions. The buffered messages are not sent out until the control variables and buffered messages from the original and shadow chares match with each other.

To enable faster comparison of messages, the RTS also allows users to annotate important members in each message. As shown in Fig. 4, users can annotate members, such as the integer data *direction*, as a control variable. This results in creation of a customized function that is used by the RTS to only compare the *direction* from the two messages. This scheme is coherent with the slicing pass methodology where the shadow chares do not execute line 10 in Fig. 3a since it is not needed for comparison. Data corruptions in the *data* field of the *ghostMsg* are protected using techniques for field data described in a later section.

We find that the message driven execution model like CHARM++ greatly eases the process of local recovery by separating the control and computation flow into different entry

methods. As can be seen in Fig. 3a, actions before and after the computation routine *invokeComputation* are encapsulated in entry methods *receiveMessage* and *beginNextIter*. Hence, there is no need to checkpoint the field data *temperature* for the local recovery of computations in *receiveMessage* and *beginNextIter*. As for the entry method *invokeComputation*, we protect it using the techniques described in Section IV-V.

IV. Selective Instruction Duplication

Runtime guided replication is not adequate for dealing with certain categories of SDCs. In particular, since the comparison is made only after the entry method finishes execution, errors that are "felt" only during the execution are left unprotected. For example, it is difficult to protect loop variable *i* in the entry method *invokeCompution* shown in Fig. 3a using runtime guided replication since its lifetime ends before the completion of the entry method. However, bit flips in *i* can lead to severe consequences: the program may either crash due to out-of-bound access or cause SDCs because incorrect data is used. FlipBack uses *selective instruction duplication* to protect such transient calculations. More specifically, FlipBack selectively duplicates the integer arithmetic instructions in the code region that are not part of the sliced set to be replicated by shadow chares.

To enable duplication of integer arithmetic instructions, we perform another compiler pass in FlipBack, primarily to protect address calculations used for memory access. In this pass, we first identify all the instructions that calculate the addresses used in load and store instructions. Next, we find the duplication path for those instructions using *use-def* and *def-use* chains, and duplicate every instruction in this path. This step is similar to what has been shown in the previous work [25], [11]. Our current implementation limits duplication path to be within a basic block. Finally, comparison instructions are added at synchronization points, i.e. before store and branch instructions to check for SDCs.

```
1  ;label 0
2  %1 = add i, 1
3  %2 = add i, 1
4  %3 = icmp eq %1, %2
5  br %3, label %4, label %6
6  ;label 4
7  5 = add i, 1
8  br label %6
9  ;label 6
10 %7 = phi [%1, label %0], [%5, label %4]
```

Fig. 5: Illustration of the LLVM IR code after selective instruction duplication pass.

Our approach for instruction level duplication is different from the previous work [25], [11] in that FlipBack performs local recovery immediately if SDC is detected. Fig. 5 shows a simplified version of the LLVM IR code after the selective instruction duplication pass. Line 3 duplicates the original instruction in line 2. Line 4 compares the two execution paths. If there is mismatch, we perform recomputation again in line 7. At line 10, we decide which computation result should be used based on whether SDC has occurred or not. Afterwards, we replace all the future uses of instruction 1 with instruction 7. Note that the underlying assumption we make is that the probability of bit flips occurring at the same place in a short duration is very low.

V. Adaptive Protection for Field Data

FlipBack provides several detection routines to protect field data that are typically computed using floating point calculations against SDCs. In general, scientific HPC applications simulate phenomena that occur in the real world, such as particle motion, heat-propogation, climate changes, and fluid dynamics. As a result, the continuity of data found in nature is also observed in a correctly executing scientific program unless bit flips silently change its output. For example, Fig. 6 shows a heat map for data values arranged as a 2D grid in two applications that perform stencil-based property update and molecular dynamics based on Car-Parrinello method [15]. It is easy to see that gradual changes are observed in data values as a sweep is made through the grid.

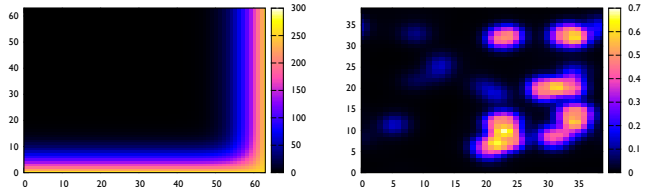

Fig. 6: Continuity in scientific data: gradual changes are observed in data values arranged as a 2D grid in a property-relaxation (left) and a molecular dynamics (right) application.

FlipBack leverages the aforementioned continuity in the data values and reports anomalies when such a continuity is not observed. Currently, we provide different detectors that exploit different types of data similarity as discussed below. At the beginning of each run, FlipBack tests the application data with all the detectors and selects the one that predicts values most accurately. We refer to this initial stage as the testing phase in the rest of this section.

Spatial data similarity: This detector predicts the data value at each point based on the value at its neighbor points. This predicted value is compared with the real value to check if the difference between the real value and the predicted value is within a user-defined range. If the difference is out of the range, FlipBack reports the anomaly to users. In a typical HPC application, data is arranged as a multi-dimensional structured or unstructured mesh. Depending on the application, data values may show higher similarity or better correlation along a dimension in comparison to other dimensions. Thus, in the testing phase, FlipBack first finds the best dimension to use for prediction and then predicts the data values along that dimension.

In order to combine the information from multiple dimensions for better prediction, FlipBack utilizes a method

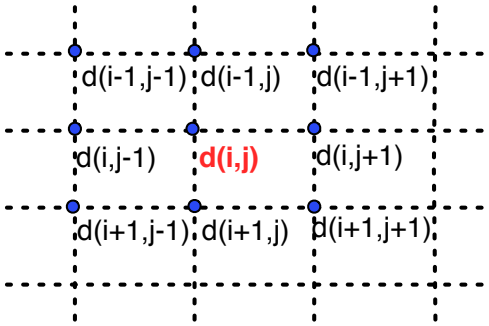

Fig. 7: Representation of data in 2D grid.

that explores the relationship between neighbors points in one dimension through an orthogonal dimension. In the data grid shown in Figure 7, to predict the value at point (i, j), FLIPBACK first studies the relationship of each j^{th} data point with its left $(j-1)$ and right $(j+1)$ neighbors in rows $i-1$ to $i+1$. More specifically, FLIPBACK first calculates r_{i-1} and r_{i+1} using the following formula:

$$r_{i-1} = \frac{d_{i-1,j-1} - d_{i-1,j}}{d_{i-1,j-1} - d_{i-1,j+1}},\ r_{i+1} = \frac{d_{i+1,j-1} - d_{i+1,j}}{d_{i+1,j-1} - d_{i+1,j+1}}$$

Then it uses r_{i-1} and r_{i+1} to interpolate the corresponding r_i in row i. Next, FLIPBACK predicts the value at point (i,j) as $p_{i,j} = d_{i,j-1} - (d_{i,j-1} - d_{i,j+1}) * r_i$. Finally, it compares $p_{i,j}$ with $d_{i,j}$ to detect if an anomaly has been found.

Temporal data similarity: This detector studies the temporal evolution of the application data. Assume that the current detection time step is t and data from previous two detection steps is stored (steps $t-k$ and $t-2k$ where detection is done every k time steps). First, FLIPBACK computes and records δ as the data difference from time step $t-2k$ to $t-k$. Next, FLIPBACK uses δ and d^{t-k} (the data value at time step $t-k$) to predict the data value at time step t. If the difference between the predicted value and the real value (d^t) is beyond a predefined range, FLIPBACK reports the anomaly to users.

Spatial & temporal data similarity: For certain applications, it is hard to predict the expected data values by use of either spatial or temporal locality only. In such scenarios, FLIPBACK attempts to use both spatially and temporally close data to detect anomalies.

In this method, for each data point d, the first step is to compute if spatial or temporal similarity exists. If neither spatial or temporal similarity exists, the method checks if there is a similarity between the temporal updates made to spatially neighboring data values, i.e. if the update to the value of d since the last time step is similar to the change in the value(s) of the neighboring point(s) of d. For example, in a 1D grid, this step checks if the difference $d^t(i) - d^{t-k}(i)$ is similar to either $d^t(i-1) - d^{t-k}(i-1)$ or $d^t(i+1) - d^{t-k}(i+1)$. Next, if needed, we check if there is a temporal similarity between the difference in the neighboring values: if the difference between the value of d and its neighbor is similar at time step t and $t-k$. If none of these steps find the data to be as expected, an anomaly is reported to the user.

VI. EVALUATION

We use LLFI [28], a fault injection tool based on LLVM [17], to inject bit flip induced soft errors. LLFI works at the level of LLVM intermediate representation code and injects failure into live registers. The original version of LLFI developed by Wei et al. is for injecting faults into sequential programs only. In order to use it for our experiments, we extend LLFI to work with parallel programs. The parallel version of LLFI randomly selects a processor and injects failure to it during the execution. In this way, we mimic the occurrence of a bit flip on arbitrary processors.

All experiments presented in this section were conducted on Catalyst, a cluster at Lawrence Livermore National Laboratory. It consists of Intel E5-2695 processors with two sockets per node and 12 cores per socket. We use two proxy applications to evaluate the failure coverage and performance of FLIPBACK: Miniaero and Particle-in-cell (PIC). Miniaero [14] is a proxy application from Sandia National Laboratory which solves the compressible Navier-Stokes equations using explicit RK4 method. Particle-in-cell (PIC) is a proxy application from the PRK benchmark suite [6], in which particles are distributed within a fixed grid of charges. In each time step, PIC calculates the impact of the Coulomb potential at neighboring grid points on the motion of every particle. We also use a micro-benchmark, Stencil3D, that performs 7-point stencil-based computation on a 3D-structured mesh to further evaluate FLIPBACK.

Both Miniaero and PIC provide verification routines that we use to determine if an execution generates correct output at the end of the run. For Stencil3d, output of a given run is compared with the output generated from the failure-free run. If the maximum difference between the two sets of outputs is less than 1%, we deem the execution to be correct.

Silent data corruptions in different parts of the programs manifest themselves in different ways, thus we categorize the failure injection experiments into four sets and study them one by one. The first set impacts communication routines in which message creation and reception occurs. The second set corresponds to the integer calculations in the computation routine. The third one relates to floating point calculations in the computation routine. The last set includes all the remaining code regions which we refer to as the *control* set since the computation in this set can easily affect the program flow. For every bit position that can be flipped, we run the proxy application 100 times, and in each run LLFI selects when and to which register a fault is injected. As a result, we perform $3,200$ to $6,400$ failure injection experiments in total for each set that contains either 32-bit data or 64-bit data.

A. Miniaero

Fig. 8 shows how Miniaero reacts to bit flips with and without the protection provided by FLIPBACK using the *3d-*

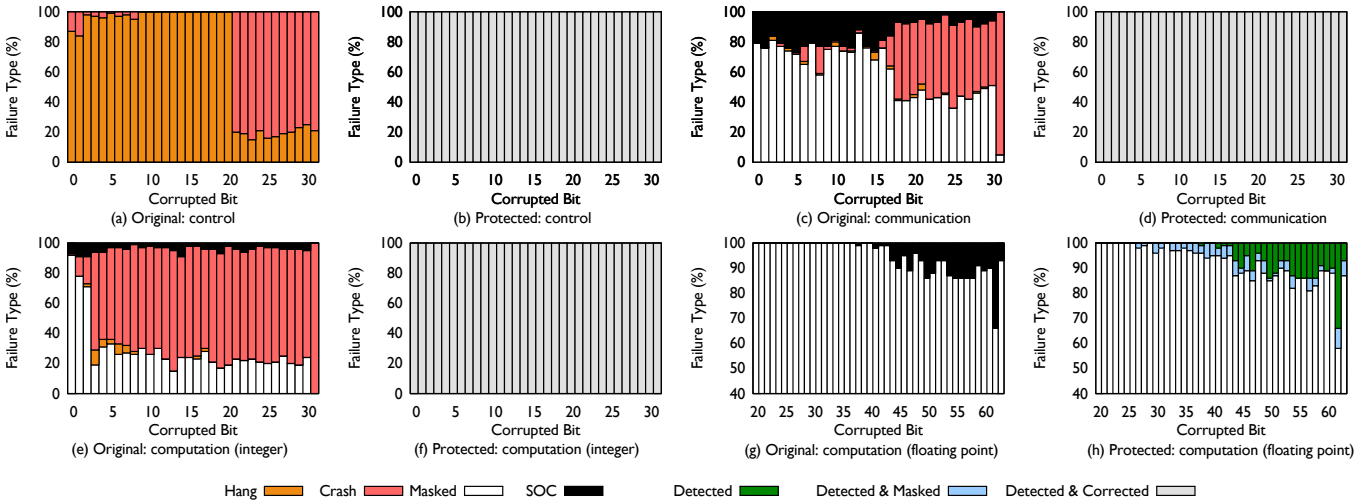

Fig. 8: Effect of bit flips on **Miniaero: 3d-sod dataset** with and without FLIPBACK protection.

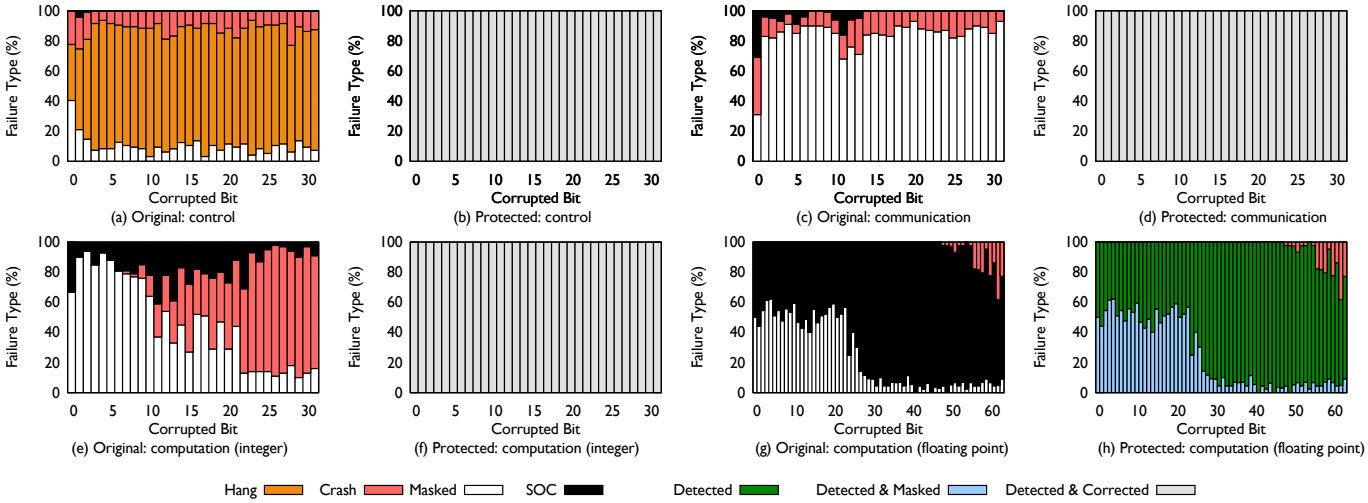

Fig. 9: Effect of bit flips on **Particle-in-cell** with and without FLIPBACK protection.

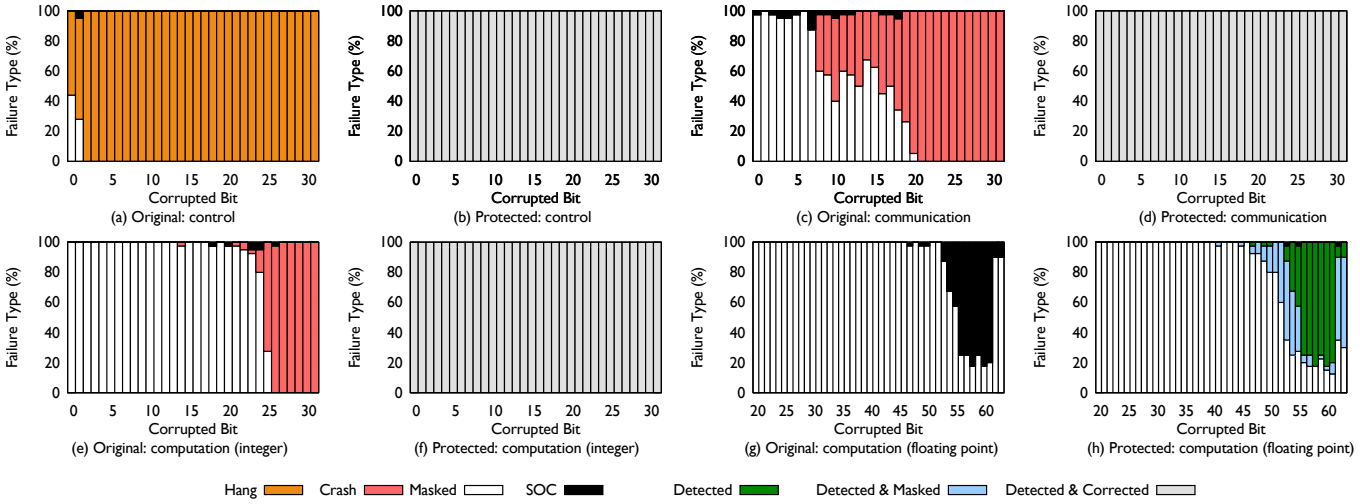

Fig. 10: Effect of bit flips on **Stencil3d** with and without FLIPBACK protection.

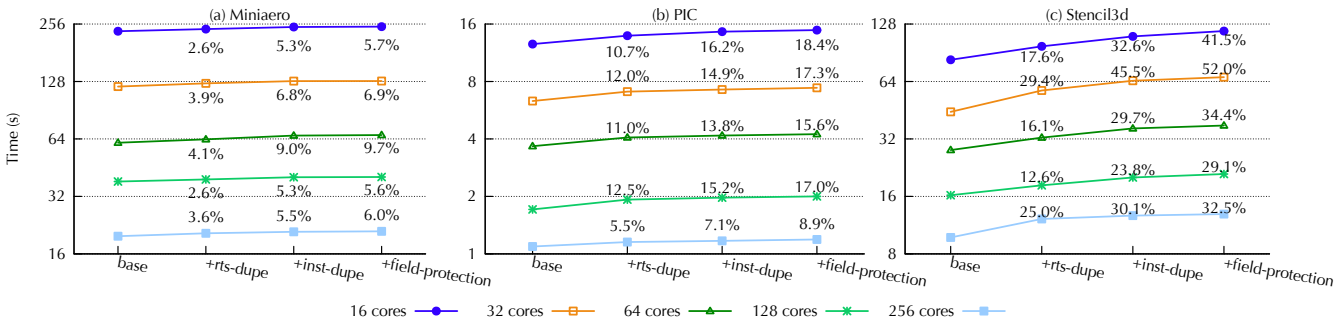

Fig. 11: Performance with different levels of protection: x-axis is functionality aggregated from left to right.

sod data set distributed with MiniAero. As can be seen in Fig. 8(a), the program hangs in more than 80% of the cases when failures are injected to the control data related to computation routines and if there is no soft error protection. The primary for the hang is the incoherent behavior of different processors caused by the bit flips. When bit flips happen in the higher bits, Miniaero crashes most of the time. This occurs when control data is used to calculate the array access indices and a out-of-bound access causes the crash. Fig. 8(b) shows the program behavior when the same bit flip pattern is injected into executions protected by FLIPBACK. It can be seen that with the runtime guided replication and selective instruction duplication in FLIPBACK, all the bit flips are captured as soon as they occur and automatic local recovery is performed. As a result, the program runs smoothly without any crash or hang.

Fig. 8(c) shows the results of the runs in which bit flips are injected into the communication routines. The main work performed in the communication routines of Miniaero is sending and receiving of messages that contain the ghost region. For 10% to 15% cases in which bit flips happen in the communication routines, silent output corruptions (SOC) occur due to sending of wrong data to the neighbors. Silent output corruptions (SOC) means that the output obtained is incorrect. Also, when a large fraction of bit flips happen on higher bits, the execution crashes because of out-of-bound access to the data. Most of remaining executions are not affected by bit flips, i.e. the bit flip is masked. This can happen if the value of the corrupted data is similar to the value of the correct data, and thus the final result is still within the acceptable error tolerance. In Figure 8(d) with FLIPBACK, we are able to detect and correct all the bit flips in communication routines.

Fig. 8(e) shows the behavior of Miniaero when failures are injected into the integer calculations in computation routines. In majority of these cases, the program crashes due to out-of-bound accesses. Incorrect results are observed due to use of corrupted data for around 10% cases. As before, when used, FLIPBACK detects and corrects all such corruptions.

In Fig. 8(g,h), we inject bit flips to floating point calculations of the computation routines in Miniaero. In these plots, note that x-axis, which represents the position of corrupted bit, starts from 20 while y-axis starts from 40. The bit flip injections not shown in Fig. 8(g,h) are masked, i.e. all bit flips

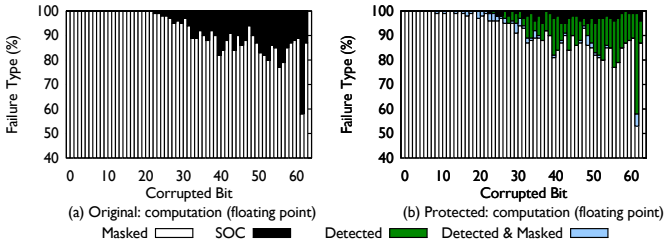

Fig. 12: Effects of bit flips on **miniaero: flat-plate dataset** with and without FLIPBACK protection.

inserted to bits in position 0 to 20 are masked. According to Fig. 8(g), the higher the position of bit that gets flipped, the higher the chances that silent output corruptions is observed. The most accurate detector for the 3d-sod data set found by FLIPBACK during the test phase is the *spatial similarity* detector using the y dimension. With that detector, FLIPBACK is able to detect all the anomalies that lead to incorrect output as shown in Fig. 8(h). However, in a few experiments, FLIPBACK detects anomaly but the final result satisfies the verification test shown as *detected & masked* in Fig. 8(h). This is because over time, the corrupted data slowly converges to a reasonable value using the uncorrupted data from the neighboring points in the data grid.

Although the *spatial similarity* detector works well with the 3d-sod data set, FLIPBACK fount it unsuitable for another data set distributed with Miniaero, the *flat-plate* data set. The best detector for the flat-plate data set is the one that combines both spatial and temporal similarity. Fig. 12 shows the effect of bit flips on the floating point calculations of the computation routines when Miniaero is executed with the flat-plate data set. It can be seen in Fig. 12(b) that FLIPBACK is able to detect more than 80% of the bit flips that lead to incorrect results.

B. Particle-in-cell

Fig. 9 shows the behavior of the Particle-in-cell proxy application when bit flips are inserted with and without protection provided by FLIPBACK. Fig. 9(a,c,e) shows that PIC's execution behavior is similar to Miniaero when FLIPBACK is not used. The only major deviation from the behavior shown by Miniaero is when bit flips are inserted to communication

routines. In this case, the bit flips are mostly masked. FLIPBACK is able to detect and correct all bit flips in the control and communication routines, and in the integer calculations performed in computation routine as shown in Fig. 9(b,d,f).

For the field data in PIC, the *temporal similarity* detector provides the best results. Fig. 9(g) shows that PIC is much more sensitive to bit flips in the field data in comparison to Miniaero: more than 70% bit flips lead to incorrect output and 2% of bit flips cause the program execution to crash. The crashes happen because in PIC, each particle uses its position to calculate the grid indices it should interact with. If the position data is corrupted, the calculation of the grid indices is wrong and thus out-of-bound accesses occur and cause crashes. FLIPBACK detects all the bit flips that lead to SOC as well as bit flips that temporarily cause anomalies but are eventually masked as the simulation continues.

C. Stencil3d

Fig. 10 compares how Stencil3d reacts to bit flips with and without using FLIPBACK. The program behavior is similar to the previous two application with two exceptions. First, bit flips in Stencil's communication routines are more likely to lead to crashes. Second, bit flips during integer calculations of the computation routine causes fewer crashes. Nonetheless, FLIPBACK is able to detect and correct all the bit flips in the control routine, communication routine and integer calculations part in computation routine as shown in Fig. 9(b,d,f).

The *spatial similarity* detector along both x and y dimensions works the best with the field data in Stencil3d. Fig. 10(g) shows that bit flips injected into higher bits of the floating point data are much more likely to induce incorrect results. Among all these bit flips that lead to incorrect results, FLIPBACK is unable to catch only 1.8% bit flips as shown in Fig. 10(h).

D. Performance

Fig. 11 shows the performance overhead of using FLIPBACK to protect Miniaero, PIC and Stencil3d from silent data corruptions. Both the proxy applications and the micro-benchmark are strong scaled from 16 cores to 256 cores in these experiments to evaluate FLIPBACK's performance. Since we are interested in studying the impact of each protection mechanism provided by FLIPBACK on the application performance, the presented result aggregates the overheads as we move from left to right and add different features. As can be seen from Fig. 11(a), for Miniaero proxy application, the overhead added by runtime guided replication ranges only from 2.6% to 4.1% as the application is scaled from running from 16 to 256 cores. Adding selective instruction duplication incurs an additional 2% to 4% overhead. The performance degradation caused by field data detector is negligible: less than 1%. Overall on 256 cores, FLIPBACK increases the execution time of Miniaero by 6% even when all the protection mechanisms enabled.

For PIC, runtime guided replication increases the execution time by around 10% when compared to the baseline case in which no protection mechanism is used as shown in Fig. 11(b). Selective instruction duplication introduces an additional $2-5\%$ overhead. Adding a detector to protect field data increases the total overhead of FLIPBACK to $9-18\%$. An interesting observation for PIC is that as the parallel efficiency of the program drops from 128 cores to 256 cores, the performance overhead introduced by FLIPBACK decreases. This is because FLIPBACK utilizes the idle time such as the time spent on communication to perform the replicated work.

Fig. 11(c) shows the performance of Stencil3d when executed with FLIPBACK. Similar trends as the two proxy applications are observed for Stencil3d. However, the absolute amount of overhead due to FLIPBACK is higher for Stencil3d. The higher overhead is because Stencil3d is a synthetic micro-benchmark that performs much less floating point computation in comparison to the other applications. As a result, the ratio of the computation that is replicated to the total work is high.

VII. DISCUSSION

Comparison with traditional checkpoint/restart strategy: In the previous section, we showed that when bit flips that definitely cause programs to crash or hang occur, FLIPBACK can detect them and perform local recovery with 100% coverage. Hence, such failures are completely hidden from users. Although use of FLIPBACK results in an increase in the total execution time when there are no failures, the recovery overhead using FLIPBACK is almost minimal. It only needs to either re-execute an entry method or re-run a few instructions for recovery.

Alternatively, one may choose to use traditional checkpoint restart in which if the program crashes or does not make progress for a long period of time, the application state is rolled back to the last checkpoint and restarted from there. This scheme may work if the bit flips induced soft errors are rare, though it will be difficult to be certain if the results are correct. Nevertheless, as the soft error rate increases, the recovery overhead with the checkpoint restart strategy will be very high and prevent applications from achieving sustainable scalable performance. Moreover, as applications scale, the increase in checkpoint time will also reduce performance further.

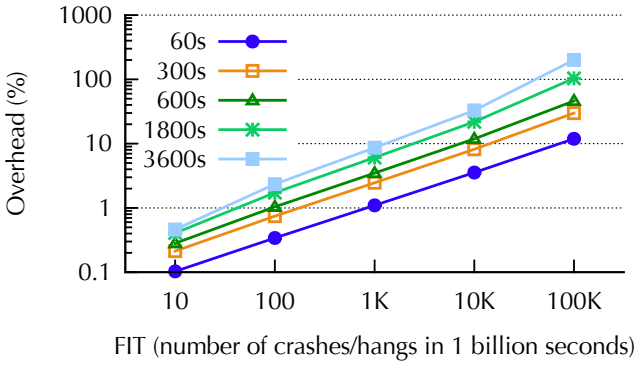

Fig. 13: Modeling the overhead of checkpoint/restart strategy for different crash/hang rates and time to checkpoint.

Figure 13 shows the expected overhead of using checkpoint/restart strategy for various soft-error induced crash/hang rates and time to checkpoint. It can be seen that the overhead increases linearly as the soft error rate increases. An increase in time to checkpoint also leads to a significant increase in the total checkpoint/restart overhead. With the worst failure rate (100K crashes/hangs in 1 billion seconds), even if one checkpoint takes only $300s$, the total overhead is almost 30%.

In contrast to the overhead of checkpoint restart, the overhead of using runtime guided replication and selective instruction duplication provided by FLIPBACK has been observed to be less than 10% for applications that we have tested, e.g. Miniaero and PIC. Besides, FLIPBACK can also protect applications from bit flips that cause incorrect results which is not possible with the checkpoint restart strategy. Moreover, FLIPBACK is a much more scalable solution. The overheads introduced by FLIPBACK do not scale with the number of cores; additionally, FLIPBACK leads to very low overhead during bit flip recovery that may lead to crashes or hangs.

Advantage of runtime guided replication: Both the techniques of runtime guided replication and selective instruction duplication are able to detect and correct bit flips that happen in the functional units and registers including bit flips in ALU, registers, branch instructions and instructions that calculate the addresses used in load/store instructions. In addition, runtime guided replication in FLIPBACK improves fault coverage in two ways over the software based instruction duplication approach.

First, runtime guided replication approach is able to detect SDCs that occur in memory by comparing the control variables and messages between the original and shadow chares. In contrast, with instruction duplication technique, incorrect values will be loaded from the memory for both the original and duplicated instruction. Second, store instructions is a single point-of-failure for the software based instruction duplication approach. Even if store instructions are duplicated, the second one will overwrite the previous store instruction. In contrast, for runtime guided replication, the values being stored are compared at the end of entry method between the original and shadow chares, and thus bit flips in store instruction can be detected.

VIII. CONCLUSION

As the rate of bit flips increase due to decreasing feature size and lower operating voltage, efficient methods to protect HPC applications from SDCs will be needed. As a step towards achieving this goal, this paper introduced FLIPBACK, a framework that self adapts the protection mechanism based on the data and computation characteristics. We have shown that combined use of compiler techniques and runtime system can significantly improve applications reliability by detecting and correcting SDCs. We evaluated FLIPBACK with two proxy applications and a micro-benchmark and showed that FLIPBACK can achieve high soft error coverage with only $6 - 20\%$ performance overhead for the proxy applications.

ACKNOWLEDGMENT

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. This research used computer time on Livermore Computing's high performance computing resources, provided under the M&IC Program.

REFERENCES

[1] How To Kill A Supercomputer. http://www.hpcwire.com/2016/02/24/how-to-kill-a-supercomputer-tips-from-an-expert, 2016.

[2] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, New York, NY, USA, 2014. ACM.

[3] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):87–96, 2004.

[4] L. Bautista-Gomez and F. Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 595–602. IEEE, 2015.

[5] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. A watchdog processor to detect data and control flow errors. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 144–148. IEEE, 2003.

[6] R. V. der Wijngaart, A. Kayi, J. Hammond, G. Jost, T. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson. Comparing runtime systems with exascale ambitions using the parallel research kernels. International Supercomputing Conference ISC, 2016.

[7] S. Di and F. Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications.

[8] P. Du, P. Luszczek, and J. Dongarra. Algorithm-based fault tolerance method for soft error resilience in high-performance linpack. In *IEEE Cluster*, 2011.

[9] C. Engelmann, H. H. Ong, and S. L. Scott. The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194. ACTA Press, Calgary, AB, Canada, Feb. 2009.

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Architectural support for programming languages and operating systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[11] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[12] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[13] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Supercomputing*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[15] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. Martyna, and L. Kale. Openatom: Scalable ab-initio molecular dynamics with diverse capabilities. In *International Supercomputing Conference*, ISC HPC '16 (to appear), june 2016.

[16] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson. Ipas: intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 227–238. ACM, 2016.

[17] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[18] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale. Using migratable objects to enhance fault tolerance schemes in supercomputers. In *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[19] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[20] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329 – 335, sept. 2005.

[21] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247. IEEE, 2005.

[22] X. Ni, E. Meneses, N. Jain, and L. V. Kale. ACR: Automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, Nov. 2013.

[23] F. Oboril, M. B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss. Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 144–153. IEEE, 2011.

[24] J. Phillips, G. Zheng, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

[25] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[26] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[27] L. Spainhower and T. A. Gregg. Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5.6):863–873, 1999.

[28] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 375–382. IEEE, 2014.

[29] K. S. Yim. Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 458–467. IEEE, 2014.

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