FlipIt: An LLVM Based Fault Injector for HPC

Jon Calhoun, Luke Olson, and Marc Snir

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{jccalho2,luko,snir}@illinois.edu

Abstract. High performance computing (HPC) is increasingly subjected to faulty computations. The frequency of silent data corruptions (SDCs) in particular is expected to increase in emerging machines requiring HPC applications to handle SDCs. In this paper we, propose a robust fault injector structured through an LLVM compiler pass that allows simulation of SDCs in various applications. Although fault injection locations are enumerated at compile time, their activation is purely at runtime and based on a user-provided fault distribution. The robustness of our fault injector is in the ability to augment the runtime injection logic on a per application basis. This allows tighter control on the spacial, temporal, and probability of injected faults. The usability, scalability, and robustness of our fault injection is demonstrated with injecting faults into an algebraic multigird solver.

1 Introduction

Driven by a need to solve ever larger problems, high performance computing (HPC) has become a fundamental part of scientific investigation and discovery. This dependence is evident in the push for increased performance of supercomputers over the past few decades. The petascale barrier was broken almost six years ago , and while the exascale barrier is expected to be broken within the next decade, it is not expected to be met without overcoming a host of challenges [10] [3]. One key challenge facing HPC as we march toward exascale is the need to deal with faults. Faults afflicting HPC systems are classified as either hard or soft, and are the cause of errors in the system. Hard faults are faults that are reproducible — e.g. the inability to communicate with a node that is offline. Soft faults are faults where activation is not systematically reproducible [1] — e.g. a bit-flip caused by a charged particle.

Traditionally failures due to hard faults are handled by checkpoint-restart [9]. Issues with scalability [14] [15] are prompting the development of hierarchical approaches [2], while alternatives to checkpoint-restart focus on replication [7]. Although these solutions provide safeguards against faults present in the system, they provide little if any insight about the application's ability to handle faults.

Soft errors are common on DRAM chips, and all DRAMs in modern HPC include error correcting codes (ECC). The addition of ECC and more advanced features such as chipkill dramatically reduce the errors in DRAM [17]. Processors are more difficult to protect, but recent designs add protection to memories, data

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 547–558, 2014.

paths, and register files. Even so, with increased core count and continued use of commodity parts, soft errors are likely to be common encounters in emerging architectures [3].

Consequently, we are motivated to determine the impact of soft errors on HPC applications and the effectiveness of resiliency schemes to safeguard against them. Because soft errors are rare and do not normally exhibit during testing, their manifestation must be simulated by a fault injector.

This paper makes the following contributions:

- The development of a robust LLVM based fault injector that targets HPC applications.
- An overview in its utility as a general purpose fault injection framework.
- A demonstration of its usability, scalability, robustness on production level HPC code.

The remainder of this paper is structured as follows. In the next section, we discuss related background in the area of fault injection. Section 3 details the design, use, and adaptation to individual applications. Results from its usability, scalability, and robustness are shown in Section 4.

2 Background

There are many forms of fault injectors. The more accurate the fault injector the closer to physical hardware the faults are injected. At the lowest level, injections come in two forms real and simulated.

In real injections, the hardware is bombarded with a concentration of neutrons. While this method is highly accurate, it has significant drawbacks mainly its cost and availability to a limited number of researchers, which limits its applicability to HPC.

Simulated injections comprise many techniques at both the hardware and software level. At the hardware level, gate-accurate models are constructed, and fault injection occurs systematically with gate level granularity. With gate-accurate simulations, execution of a full application is possible, but a large scale machine is prohibitive due to execution overhead. A fault injection framework that operates with low overhead on current hardware is needed to inject faults in HPC applications.

Many fault injectors have been created that operate in real time and on real hardware. DOCTOR [8] injects faults into memory, the CPU, and network communications by using time-outs and traps to overwrite memory locations and modify the binary. XCEPTION [4] is an exception handler that injects faults when triggered by accesses to specific memory addresses, and simulates stuck-at-zero, stuck-at-one, and bit-blips. NFTAPE [18] uses a driver based fault injection scheme to inject fault inside the user or kernel space, but requires OS modification.

In compiler based fault injection, hard errors are simulated by adding extra instructions that always inject in the same location. To address the static nature of these injections, the injection is made dynamic by addition of code that corrupts data at runtime based on programmatic and environmental factors.

Fault injection for MPI applications is often focused on *message* injection [7]. Yet other works consider soft errors that manifest in register modifications [5] or that arise in the memory image of the running application [13,12]. The approach in [12] allows user identification of stack and heap items to target for injection. This is similar to our approach where we allow the user to select functions that are faulty.

Since DRAMs have a higher level of protection from silent errors than processors thanks to being easier to protect by ECC and chipkill, our focus is on faults that arise in the processor. In the following section, we detail the design of our fault injector that simulates the presence of silent errors as the manifestation of bit-flips in register values.

Relax [6], an LLVM based fault injector, is similar to our approach; however, it is not publicly available, is designed from an old version of LLVM, and does not target HPC applications. KULFI [16] is a publicly available fault injector similar to Relax. Because KULFI is still currently maintained and is easily modifiable, its structure provides a basis for our fault injector. In particular, we utilize the framework of their compiler pass. We provide a fault injector that is more expansive, extensible, and provides more user control than KULFI.

3 Fault Injector

3.1 Overview

Our fault injector is structured as an LLVM compiler pass [11] and is based on KULFI [16]. Notable extensions have been added to increase its robustness and efficacy. Such extensions include:

- Support for complex pointer types.
- Ability to work with multiple source files simultaneously.
- User customized fault distribution and event logger.
- Support for a larger subset of the LLVM instructions.
- MPI rank aware.

We chose to use an LLVM compiler pass to simulate transient errors instead of randomly flipping bits inside the binary in order to provide more control over what section of the code is faulty and when faults arise. The compiler pass proceeds by iterating over all modules in a source file. Upon discovery a module marked for injection, the included instructions are surrounded by instructions that probabilistically inject a fault. Here we say a fault is a single bit-flip in a source operand or the result of an LLVM instruction. Figure 1 illustrates this transformation for a single `add` instruction in which the result is corrupted.

All subsequent use of the variable that is possibly corrupted is replaced with the value returned from the corrupt function. The locations where faults can occur are enumerated at compile time, but their activation occurs randomly at runtime based upon a user provided fault distribution.

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    ret i32 %add
}
```

(a) Original LLVM IR.

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    %data = sext i32 %add i64
    %tmp = call i32 @crptInt(i32 0, i32 0,
                double 0.01, i32 2, i64 %data)
    %crptAdd = trunc i64 %tmp to i32
    ret i32 %crptAdd
}
```

(b) Transformed LLVM IR.

Fig. 1. Code Transformation to Inject Faults

3.2 Design

As shown in Figure 1, a function call to `crptInt` is used to determine if a bit should be flipped and performs the flip. Algorithm 1 provides generic logic for the injection. In every corrupting function, the argument list is the same except for the data type of the value being corrupted. Table 1 details the argument list of the corrupt functions from left to right. The byte where a bit-flip is to occur is either determined at compile time (0-7) or is calculated randomly at runtime, negative value. If the byte specified is outside the range of the data type, we use the modulus operator to wrap the bytes to the correct range. For both the dynamic and static byte selection, the bit that is flipped is determined at random. We fix a byte before selecting a bit to flip in order to provide the ability to look at bit flips in certain bit positions.

Algorithm 1. Generic corrupt logic

Input: *siteProb*: Probability that this site is faulty.
siteIndex: Unique index of this fault site.
data: Value eligible for corruption.
Result: Data unmodified(no injection), or data with a single bit-flip(injection).

```
1 if ¬ shouldInject(injectorOn, siteProb) then
2     return data;
3 else
4     bitPosition ← random bit position in targeted byte;
5     logInjection(siteIndex, bitPosition);
6     data_corrupt ← data ⊕ (0x1 ≪ bitPosition);
7     return data_corrupt;
```

In our basic model and the experiments in Section IV, we make the simplifying assumption that all LLVM instructions have an equal probability for a fault to be injected. Each LLVM instruction should have differing probabilities and is a function of the underline hardware. Therefore, if one knew this information for a processor *a priori*, the scaled probabilities should be incorporated into to the configuration file of the fault injector.

Advance pointer types, multiple levels of indirection, are not considered for injection in KULFI. Pointers are represented using a finite number of bits;

Table 1. Corrupt function's arguments (left to right)

Arg	Description
1	Unique fault site index.
2	Boolean: one injection per active rank.
3	Probability that instruction is faulty.
4	Byte location targeted for bit-flip.
5	Data to be corrupted by a bit-flip.

```
#include "/path/to/fault/lib/corrupt.h"
#include <mpi.h>
int main(int argc, char** argv) {
 MPI_Init(&argc, &argv);
 int id; int seed = 71;
 MPI_Comm_rank(MPI_COMM_WORLD, &id);
 FLIPIT_Init(id, argc, argv, seed);
 foo();
 FLIPIT_Finalize(NULL);
 MPI_Finalize();
}
```

Fig. 2. Source modifications to use fault injector

Table 2. User callable functions

Function name	Description
`FLIPIT_Init`	Initializes fault injector. Turns on injector.
`FLIPIT_Finalize`	Cleans up injector. Turns off injector.
`FLIPIT_SetInjector`	Zero: turns off injector. Non-zero: turns on injector.
`FLIPIT_SetFaultProbability`	Sets the probability function with a user defined function.
`FLIPIT_SetCustomLogger`	Sets user defined logging function. Called on all injections.

therefore, in our solution, we inject into pointers by first casting the variable to a 64-bit integer. Next, a call to corrupt this 64-bit integer is inserted, and the value returned from this call is cast back to the appropriate data type. As with all corruptions, all subsequent uses are replaced.

Since a function targeted for corruption may include additional function calls, corrupting these function calls is critical to properly modeling a fault function. There are two options to handle propper corruption, depending on the type of function. First, proper corruption may be issued by recompiling the source of the corrupting function. Second, if recomplation is not an option — e.g. due to unavailable source — then we scale the probability of injecting a fault into return value or argument of the `call` instruction. The probably is not specified until compile time, either by using the function's execution time or by the amount of hardware active during its execution.

In order to support HPC injections, our fault injector is aware of the processes' current MPI rank inside `MPI_COMM_WORLD`. This allows fault injections on a subset of ranks specified at runtime via command line arguments. Large scale machines have custom MPI distributions that are tuned for performance. Because a substantial portion of time in MPI applications is spent in MPI routines, we must consider MPI calls as faulty. The source code for the machine optimal MPI is not available. This implies that we must utilize the mechanism outlined above to modify probabilities of function call injections based upon the execution characteristics of the function.

3.3 Usability and Extensibility

Source Modification. Our fault injector is designed to require minimal modification to existing codes while at the same time providing a high degree of robustness and flexibility. Figure 2 shows the minimum required changes to `main` in order to use the fault injector. The call to `FLIPIT_Init` should dominate all usage of code compiled with our fault injector, and no such code should be executed after the call to `FLIPIT_Finalize`. We elect to have the user to insert calls to `FLIPIT_Init` because it is possible that we never see the file containing `MPI_Init`. To provide the user with a more fine grain control over fault injection, the functions in Table 2 are provided.

Compiling. The use of our fault injector requires little modification to current building practices. Once functions have been identified by the programmer, the source files containing the functions is recompiled using our compiler pass. Figure 3 shows this process with a change in the Makefile.

```
$(CC) -c bob.c
```

(a) Original Makefile line.

```
INJPASS = /path/to/compiler/pass
INJLIB = -L/path/to/injeciton/lib -lcorrupt.a
FIPARMS = -prob 0.01 -funcList "foo1 foo2"
HEADER =/path/to/bitcode/header
clang -g -emit-llvm bob.c -c -o bob.bc
llvm-link -o bob_c.bc bob.bc $(HEADER)
opt -load $(INJPASS) $(FIPARAMS) < bob_c.bc > bob_F.bc 2> bob.log
clang -c bob_F.bc -o bob.o
$(CC) [...] bob.o [...] $(INJLIB)
```

(b) Modified Makefile line.

Fig. 3. Compilation steps

Step 1 in the compilation process has the original source being compiled with clang into LLVM bitcode. We compile with `-g` to relate fault sites indexed by our injector to the source code lines. If this flag is omitted we can relate the injection to the location it the bit-code which doesn't always have a clear translation back to the source code due to a provided optimization level. The bitcode generated is transformed in Step 2 via our compiler pass to enable fault injections. Here we add code to inject faults into the functions `foo1` and `foo2` with a fault probability of 1×10^{-2}, each instruction has a 1 in 100 chance of producing an incorrect result. Table 3 details all possible arguments to our compiler pass and their default values. Step 3 compiles this transformed bit-code into object code. Finally we compile the application linking with a static library that contains the functions used by our injector. To simplify we provide a wrapper script that replaces the selected compiler in the Makefile.

For MPI applications, `mpicc` is a wrapper around a native compiler. The compiler flags `-show` and `-showme` for MPICH and OpenMPI, respectfully, provides the exact compiler command used to compile the source file. This command is subsequently modified in accordance to Figure 3, or this information can be provided to our wrapper script.

Table 3. Compiler pass arguments

Argument	Type	Req.	Default	Description
config	string	No	FlipIt.config	Path to configuration file.
funcList	string	Yes	—	Functions to corrupt.
prob	double	Yes	1e-8	Default per instruction fault probability.
byte	int	No	-1	Byte to flip bit in (0-7). (-1 random).
singleInj	bool	No	true	One injection per rank.
ptr	bool	No	true	Allow bit-flips in pointers.
ctrl	bool	No	true	Allow bit-flips in control variables.
arith	bool	No	true	Allow bit-flips in arithmetic.

Programmer Control. The choice to use a static library for the corruption routines is influenced by three key points: 1) the need to compile multiple source files for a single executable; 2) the ability to limit overhead of the fault injector; and 3) to allow for application specific behavior such as the collection of user defined statistics. Straightforward use of KULFI is restricted to one source file, which limits use by requiring the programmer to place all functions of interest into a single source file, or by requiring multiple recompilations to cover all functions of interest, but sacrificing the ability to look at complex function interactions.

Our approach to fault injection increases the static and dynamic instruction count for the application, which leads to increased execution time. The overhead depends on the additional computation performed by the corrupted functions apart from injecting the fault. Extra instructions are attributed to collecting statistics and the granularity of the spacial and temporal locality of a fault. Algorithm 1 shows the outline for a generic, corrupt function, and Algorithm 2 shows a basic `shouldInject` function. This function allows for fine-grained application specific selectivity for fault injection, but requires recompilation of the static library when modified. A simple modification of Algorithm 2 allows for fault injection on certain MPI modifications of the conditional in line 2; this introspection includes a check for a faulty rank.

Algorithm 2. Basic `shouldInject` logic.

Input: *siteProb*: Probability that this site is faulty.
injectorOn: Boolean signifying if injector is on.
Result: Boolean indicating an injection.

```
1 P ← probability();
2 if injectorOn and siteProb > P then
3     return TRUE;
4 else
5     return FALSE;
```

Injection at the finest granularity in an MPI application has only one active fault site, capable of generating a bit-flip, on a single MPI rank. To remove the need for recompilation, command line arguments are provided and passed to `FLIPIT_Init` detailing which fault sites are active and which MPI ranks are candidates for injection. These command line arguments listed in Table 4.

Table 4. Command line arguments for fault injector

Argument	Description
`--numFaulty`	Number of faulty MPI ranks.
`--faulty`	List of faulty MPI ranks.
`--numFaultyLoc`	Number of active fault locations.
`--faultyLoc`	List of active fault locations.

The three classifications of injection types mentioned in Table 3 are *pointer*, *control*, and *arithmetic*. The classification *pointer* refers to all calculations directly related to use of a pointers (loads, stores, and address calculation), *control* refers to all calculations of branching and control flow (comparisons for branches and modification of loop control variables), *arithmetic* refers to pure mathematical operations. By default all of these are active, but each can be toggled to simulate different injection campaigns.

Analysis. As code is being compiled with our fault injector, a log file is generated that specifies all locations where faults can be injected. Each fault site is given a unique identifier and classified depending on which the fault is injected *pointer*, *control*, or *arithmetic*. In addition, a brief description of the fault site is listed to discern any ambiguities about the injection location along with the source line number if compiled with `-g`.

As the application is run, information about the faults being injected is logged per rank for later inspection. Two types of data are logged every time a fault is injected. The first kind is information about the faults themselves — i.e. the fault site numbers, bits flipped, and values from the fault distribution. This information is used in conjunction with the fault site log files to determine in what function and where in this function the fault is injected. The second type of information logged on each fault injection is accomplished through a user-defined function. This user defined function is set using `FLIPIT_SetCustomLogger` (Table 2).

Even if no faults are injected, some statistics are still collected. For every rank, a histogram is generated showing the frequency each fault site is looked at. To determine if the execution path of the application changes due to a fault, the histogram generated in the fault free case is compared with the histogram from an application run with faults. Any discrepancies in these histograms suggest differing paths of execution. Further insights are found by using the injection log and the fault site log along side the histograms to determine precisely what occurred due to the fault, as we see in the next section.

4 Experiments on Hypre

In order to show the scalability and flexibility of our fault injector, we compile sections of Hypre with our fault injector to look at SDCs that arise during the

solving of a linear system. To solve the linear system we use Algebraic Multigrid (AMG) with one iteration of Jacobi relaxation for smoothing. The problem is a 2D Laplacian with zero on the boundaries. Profiling Hypre allows us to determine the call stack inside `HYPRE_BoomerAMGSolve`. We select all functions in this call stack for injection.

4.1 Scalability

To characterize the scalability of our fault injector, we run weak scaling experiments on Blue Waters with 16 processes per node, Figure 4, where each data point is the average of three runs. In this figure, the injector is turned on, but we inject no faults due to the probability of injecting a fault being zero. The number of unknowns per process is kept at approximately 16,384 throughout all weak scaling runs. From Figure 4 we see that our modified Hypre's execution time grows roughly linearly on a log scale just as the unmodified Hypre. From this we can conclude that the weak scaling results demonstrate that our fault injector is scalable. To determine how much our fault injector adversely effects performance we look at how much it increases execution time, Table 5. Our fault injector increases the dynamic instruction count, which produces a corresponding increase computation time. Yet, the overhead introduced by our injection is reduced as the the number of processes increases since communication becomes the bottleneck.

Table 5. Increase in execution time due to fault injector

Processes	Increased Time	Processes	Increased Time
1	123.77x	128	44.41x
2	89.48x	256	38.06x
4	82.55x	512	37.89x
8	69.04x	1024	28.79x
16	62.58x	2048	26.59x
32	54.40x	4096	20.96x
64	48.05x	8192	17.21x

4.2 Selective Injection

To show the ease at which our fault injector can inject precise injections, we target the first element of the residual vector just before it is written to memory for injection. The fault occurs during the first cycle on the finest level as we are creating the residual vector before restriction. In order to have this precise injection, we need to first determine which fault site should be active. To determine the correct fault site index, we consult the fault site log. The index is passed to our fault injector via command line arguments along with the rank that is to experience the fault, rank 0. Because the residual is computed via a SpMV routine, that is also used during problem setup, two calls to `FLIPIT_SetInjector` are added, one to turn off the injector after initialization and the other to turn it on just before calculating the residual. Figure 5 shows what effect this injection

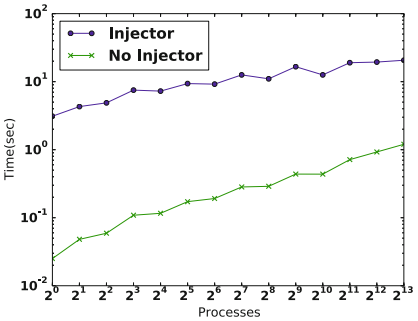

Fig. 4. Weak scaling of Hypre. Approximately 16,384 unknowns per process.

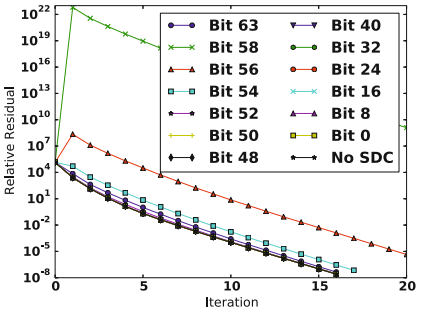

Fig. 5. Selective injection in residual calculation on rank 0. 8 processes with approximately 16,384 unknowns per process

has on the relative residual, the stopping criterion for AMG. The name of the trend indicates which bit is flipped in the 64-bit floating point number.

As we can see, a single SDC can either be masked by the application, or increase the number of iterations. Since this fault occurs in the mathematics of the problem, it wouldn't be detected until the application's results were analyzed. This suggests that work should be done to design SDC detectors to catch such SDCs early. In order to test the effectiveness of these SDC detectors, a fault injector such as the one presented here is required.

We now look at the result of injection into certain instruction types as outlined in Table 3. For this we inject a single fault into the aforementioned problem on rank 3.

Our fault injector allows us to target different classifications of instructions, and depending upon what classifications are active, the effects on the application vary. In Table 6, the average of 1000 trials, we see injection into pointers has a corresponding increase in the percent of trials that crash. Likewise injection in the mathematics of AMG, or accessing the wrong data with corrupted pointers, increases the percent of trials that require a higher number of iterations required to converge. We see a small increase in the percent of trials that crash with control injections due taking incorrect paths and incorrect indexing. By the use of these classifications, unique injection campaigns can be created allowing the study of an applications susceptibility to certain types of errors and the effectiveness of detection schemes.

Table 6. Results of injecting into certain types

	Pointer	Control	Arithmetic	All
Crash	41	29	21	29
More V-Cycles	6	0	6	4
Same V-Cycles	53	71	73	67

5 Conclusion

As SDCs become more common in HPC, research needs conducted to investigate application resilience and the effectiveness of SDC detectors. This paper presents an LLVM based fault injector designed for HPC that can aid research in this area. Scalability of our fault injector is shown with weak scaling experiments with Hypre. Our fault injector's overhead diminishes as the application's communication begins to dominate computation. To support various application requirements, our fault injector is designed to be extensible. We provide the ability to turn injections on and off from inside the application and use custom probability distributions and logging information. Using the aforementioned features we inject a fault into Hypre at a specific location and time and show that it can significantly impact convergence.

Acknowledgments. This work is sponsored by the Air Force Office of Scientific Research under grant FA9550-12-1-0478. It is also supported by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
2. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: high performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 32:1–32:32. ACM, New York (2011)
3. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience. Int. J. High Perform. Comput. Appl. 23(4), 374–388 (2009)
4. Carreira, J., Madeira, H., Silva, J.G.: Xception: a technique for the experimental evaluation of dependability in modern computers. IEEE Transactions on Software Engineering 24(2), 36–125 (1998)
5. Casas, M., de Supinski, B.R., Bronevetsky, G., Schulz, M.: Fault resilience of the algebraic multi-grid solver. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 91–100. ACM, New York (2012)
6. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: An architectural framework for software recovery of hardware faults. In: Proceedings of the 37th International Symposium on Computer Architecture (ISCA) (2010)
7. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–78. IEEE Computer Society Press, Los Alamitos (2012)

8. Han, S., Rosenberg, H.A., Shin, K.G.: Doctor: An integrated software fault injection environment (1995)
9. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for linux clusters. Journal of Physics: Conference Series 46(1), 494 (2006)
10. Kogge, P.M., La Fratta, P., Vance, M.: [2010] facing the exascale energy wall. In: Proceedings of the 2010 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, IWIA 2010, pp. 51–58. IEEE Computer Society, Washington, DC (2010)
11. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO2004), Palo Alto, California (March 2004)
12. Li, D., Vetter, J.S., Yu, W.: Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 57:1–57:11. IEEE Computer Society Press, Los Alamitos (2012)
13. Lu, C.-d., Reed, D.A.: Assessing fault sensitivity in MPI applications. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC 2004, p. 37. IEEE Computer Society, Washington, DC (2004)
14. Riesen, R., Ferreira, K., Da Silva, D., Lemarinier, P., Arnold, D., Bridges, P.G.: Alleviating scalability issues of checkpointing protocols. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–18. IEEE Computer Society Press, Los Alamitos (2012)
15. Sato, K., Gamblin, T., Moody, A., de Supinski, B.R., Mohror, K., Maruyama, N.: Design and modeling of non-blocking checkpoint system. In: Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?, ATIP 2012, pp. 39:1–39:2. A*STAR Computational Resource Centre, Singapore (2012)
16. Sharma, V.C., Haran, A., Rakamarić, Z., Gopalakrishnan, G.: Towards formal approaches to system resilience. In: Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC (2013)
17. Sridharan, V., Liberty, D.: A study of DRAM failures in the field. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 76:1–76:11. IEEE Computer Society Press, Los Alamitos (2012)
18. Stott, D.T., Floering, B., Burke, D., Kalbarczyk, Z., Iyer, R.K.: NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings of the IEEE International Computer Performance and Dependability Symposium, pp. 91–100 (2000)