Towards Formal Approaches to System Resilience

Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamarić, Ganesh Gopalakrishnan
School of Computing, University of Utah, USA
Email: {vcsharma,haran,zvonimir,ganesh}@cs.utah.edu

***Abstract*—Technology scaling and techniques such as dynamic voltage/frequency scaling are predicted to increase the number of transient faults in future processors. Error detectors implemented in hardware are often energy inefficient, as they are "always on." While software-level error detection can augment hardware-level detectors, creating detectors in software that are highly effective remains a challenge. In this paper, we first present a new LLVM-level fault injector called KULFI that helps simulate faults occurring within CPU state elements in a versatile manner. Second, using KULFI, we study the behavior of a family of well-known and simple algorithms under error injection. (We choose a family of sorting algorithms for this study.) We then propose a promising way to interpret our empirical results using a formal model that builds on the idea of predicate state transition diagrams. After introducing the basic abstraction underlying our predicate transition diagrams, we draw connections to the level of resilience empirically observed during fault injection studies. Building on the observed connections, we develop a simple, and yet effective, predicate-abstraction-based fault detector. While in its initial stages, ours is believed to be the first study that offers a formal way to interpret and compare fault injection results obtained from algorithms from within one family. Given the absolutely unpredictable nature of what a fault can do to a computation in general, our approach may help designers choose amongst a class of algorithms one that behaves most resilient of all.**

I. Introduction

With the growing scale of systems and the level of integration of transistors within CPU and GPU cores, undetected bit-flips pose a serious challenge to our ability to rely upon computational results. Recent studies [1] show that it is unaffordable to employ hardware-only solutions to detect (and hopefully correct) hardware faults. A follow-on study [2] in fact expresses the need for software-based solutions to be used in tandem with hardware-based solutions. The reason hardware solutions are deficient is that they are "always on"—and hence power inefficient—whereas well-designed software level solutions have the advantage of becoming active only when faults occur (albeit with higher amounts of fault-handling latencies).[1] One may think of these software solutions as "asserts" placed within the code to trap errors.

It is clear, upon a brief reflection, that the synthesis of assertions capable of trapping faults is not so straightforward. In a sense, a piece of software subject to transient bit-flips during execution is akin to the same piece of software where one secretly sows one of a myriad of possible bugs! Given the daunting complexity of detecting even single logical bugs in (otherwise perfectly working) software, the problem of synthesizing effective *assert* statements that trap transient faults appears much harder. We believe that the use of formal methods can help the resilience research community make measurable progress in the face of this complexity. Already, several such formal methods have, or are being proposed in resilience research. The use of likely program invariants is suggested in previous work [3]. Other efforts go further, proposing formal operational semantics for a faulty Lambda calculus [4].

In this paper, we focus on ranking implementations of different algorithms that solve the same problem—a study that, to the best of our knowledge, is novel. While this approach may help us synthesize better error detectors, this connection is not the focus of this paper. However, we do believe that ranking algorithms according to resilience may eventually give us the necessary insights for synthesizing error detectors. Our choice of sorting algorithms is based on their relative familiarity, the fact that our research is in its early stages, and that there are many different sorting algorithms.

We offer three specific contributions in this paper:

1) We present a new LLVM-level fault injector called KULFI that helps simulate faults occurring within CPU state elements conveniently. We describe how the features of this (publicly released [5]) tool compare with existing LLVM-level fault injectors.
2) We run the sorting algorithms under controlled situations and observe their behavior empirically in terms of benign faults, segmentation faults, and silent data corruptions. We plot the faulty behaviors observed and provide evidence that our results are statistically significant.
3) We devise a novel, promising abstraction method for programs executing under faults. Our ideas build on the basic tenets of *predicate abstraction* [6]–[8], although we do apply abstractions that tend to reflect the degree of degradation of control flows during program execution. We show how our abstraction seems to explain the measured resilience results to some extent—at least enough to serve as a way to rank one sorting approach above another. In the end, we also develop a simple predicate-abstraction-based fault detector, and prove its effectiveness on our benchmarks.

Supported in part by NSF Award CCF 1255776, SRC Contract 2013-TJ-2426, and Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics). See http://super-scidac.org/.

[1] Interestingly, many of the faults themselves are caused by power-saving mechanisms such as dynamic voltage/frequency scaling [2].

II. Background and Related Work

Fault-tolerant computing has been very actively researched for decades, and forms the basis of many practical techniques in use, including redundant designs, voting schemes, and hardware-level error and correction schemes. There has also been extensive study to identify and ameliorate fault inducing mechanisms at the circuit level [9]. We do not intend to perform a survey of this vast area; rather, our attention is confined to the modern upsurge in resilience enhancing mechanisms based on the use of software-level assertions.

The manifestation of faults at the software level can be modeled by flips (changes) in bit-values of the computational state. Depending on the nature of these state changes, one can classify faults as follows.

Permanent Faults. Permanent faults are those that, once introduced into a state element, persist for the remainder of the computation, thus modeling permanent hardware failures.

Transient Faults. Transient faults are those that may disappear as well as reoccur during a computation. Since transient faults model rare events, such as alpha particle strikes or marginal circuit operation (often caused by noise), it is customary to study a given computation under a *single* transient fault occurrence.

When a fault occurs, the effect can be one of the following, as captured by fault filtration that occurs across the hardware/software stack [1]:

1) The fault falls within the micro-architectural don't-care set, thus effectively getting filtered.
2) The fault reflects as a visible micro-architectural state effect, but is filtered by the instruction set architecture (ISA), for example by being over-written by a good value at the beginning of the next micro-architectural epoch.
3) The fault is reflected into a programmer visible register, but falls within the don't-care set of the application logic, say by affecting a variable that does not form the "answer" returned by a function call.
4) The fault causes the machine to hang, results in a segmentation fault, or is otherwise clearly observed (say, by tripping a built-in hardware-level error detector).
5) The fault silently corrupts the output of a computation without tripping any observer or without being filtered.

We will use the term *Benign Fault* for categories 1–3 and *Silent Data Corruption (SDC)* for category 5. We will assume that *Segmentation Faults* are the only observed category 4 of faults.

Algorithm level fault detection can be studied by focusing either on memory faults or computational faults. Sorting algorithms and data structure resilience have recently been studied focusing on a DRAM fault model, where faults are assumed to occur only on the algorithm inputs [10], [11]. In this work, various algorithms are compared based on the k-unsortedness metric. More specifically, the lower the number of misplaced data items, the more resilient the algorithm is deemed to be. In contrast, in our case study we assume a more fine grained fault model that accommodates more fault categories, specifically, at the register and control flow level. We also compare algorithms based on the number of silent data corruptions.

One of our main contributions is the development of a fault injector called KULFI, based on the LLVM compilation infrastructure [12], [13]. KULFI can inject transient faults into a chosen data register of a randomly chosen program instruction at run time. Several previous studies have exploited fault injectors similar to KULFI [14], [15]. There are also efforts that directly inject faults into the hardware [16]. Hardware-based fault injection is less flexible [17] and not as programmable. Fault injectors can also be built by exploiting OS-level facilities [14], [15]. Other software-level fault injectors include those based on PIN [18], [19]. Specifically, the PDSFIS fault injector [18] uses Intel's PIN framework.

In contrast to the above works, KULFI uses the open-source LLVM compiler infrastructure, similarly to other recently reported fault injectors [19], [20]. The fault injector LLFI [20] is primarily geared towards injecting errors in soft-computing applications. LLFI and KULFI were developed concurrently, and currently they share many similar features. However, when we started working on this project, KULFI was the only tool available to us that had all of the required features. For example, KULFI provides fine-grained error injection control (briefly discussed in the next section), which suits well our requirements for performing the empirical evaluations described in this paper. The fault injector used in the Relax framework [19] also uses the LLVM compiler infrastructure. However, this fault injector is not publicly available. Furthermore, a recent informal study by a Relax user suggests that KULFI is easier to control and fine-tune, while also providing interesting command-line options not found in Relax [21].

There has been a significant amount of research on optimizing the placement of the fault detectors [22]–[25]. Research on using compiler-based techniques to detect hardware faults has also been reported [26], [27]. In recent work [28], the focus is primarily on fault recovery (not on fault detection) and custom annotations in the source language to convey the degree of resilience desired. Casas et al. [29] make a large-scale numerical application resilient by employing redundancy methods to guard pointers, and showing that some algorithms are able to recover from faults. Sahoo et al. [3] introduce an approach that employs likely program invariants for detecting hardware faults. SymPLFIED [30] is a formal framework that uses symbolic values to represent hardware faults, and performs symbolic execution to simulate the propagation of such faults. It analyzes fault propagation patterns to optimize the placement of fault detectors.

Several techniques have been proposed to detect transient faults that cause control flow variations. The work by Oh et al. [31] is based on assigning a unique signature to a basic block and tracking the signatures at runtime. Subsequently, Venkatasubramanian et al. [32] proposed a more refined and optimized solution. In contrast to these approaches, ours is based on the use of predicate abstraction [6].

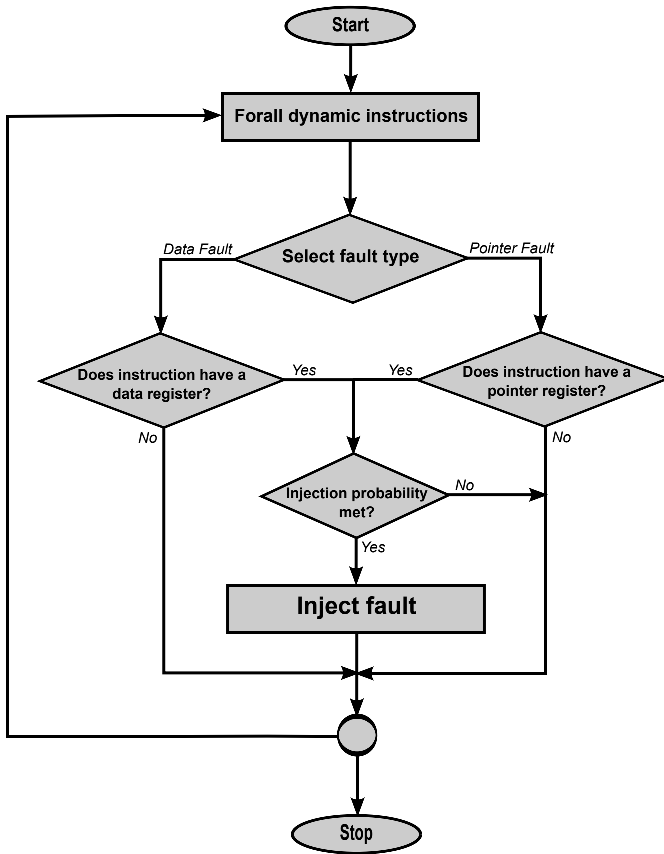

Fig. 1. Flowchart of Dynamic Fault Injection in KULFI

III. KULFI: A Fault Injector

We have developed an open-source instruction-level fault injector named Kontrollable Utah LLVM Fault Injector (or KULFI)[2] on top of the LLVM compiler infrastructure [12], [13]. KULFI is capable of injecting static and dynamic faults into programs written in C. Static faults model permanent faults and are injected to a fault site selected during compile time. Dynamic faults emulate transient faults and are injected to a fault site selected during program execution. KULFI can inject faults into both data and address registers, and currently it models only single-bit faults. It provides fine-grained control over the fault injection process by allowing a user to specify fault injection probability, injected byte location, fault site type (data, address, or both), limit on the number of injected faults, target functions to inject faults into, etc.

Figure 1 shows the flowchart of the dynamic fault injection done by KULFI. At a high level, the fault injection loop goes through all dynamic instructions. For each dynamic instruction, a type of fault to be injected is selected as either a data or pointer fault type. Subsequently, KULFI checks whether it is feasible to inject a fault with the chosen fault type into the selected instruction. If this check passes and the provided fault injection probability is met, then a fault is injected into the instruction. These steps are repeated for all dynamic instructions. Once the loop is finished going through all the dynamic instructions, the execution of KULFI terminates.

[2]Available from http://github.com/soar-lab/KULFI/.

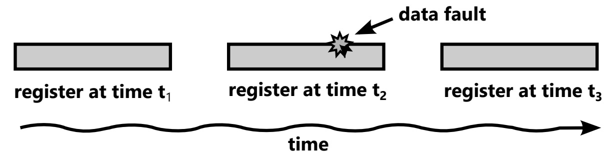

Fig. 2. Transient Fault Occurring in a Register

Given that transient faults are the main focus of this paper, we describe in more detail how KULFI models such faults. Figure 2 illustrates a transient fault occurring at register level. The shown register does not contain a fault at time t_1. At time t_2 a fault occurs, and then it disappears at time t_3. Dynamic fault injection capability of KULFI models such transient fault behavior. KULFI operates on the LLVM intermediate representation (IR) level (i.e., LLVM bitcode level) in the static single assignment (SSA) form. SSA ensures that every IR variable (i.e., logical register) is assigned only once, which is an advantage as opposed to operating at the source code level when modeling transient faults. More specifically, injecting a fault into an SSA logical register referenced by an instruction is a one-time occurrence affecting only the instructions that use that logical register. SSA naturally prevents references to the same source code variable in the later instructions from observing the injected fault. Note that the duration for which a transient fault persists in an actual hardware register varies. Therefore, it is possible that more than one instruction could get affected from a single transient fault. Currently we do not capture such timing-related behaviors of transient faults in the software emulation of faults done by KULFI. However, KULFI still provides a reasonable model of transient fault behaviors, and similar models were adopted by other error injectors [19], [20].

IV. An Empirical Case Study

We have performed an empirical case study that assesses the resilience of several popular sorting algorithms. Previous work suggests that algorithms and data structures that solve a particular problem not only vary in time and space complexity, but also in how resilient they are to faults [11]. In that line of work, a memory (i.e., DRAM) fault model is assumed to study the resilience of sorting algorithms [10]. However, the memory fault model is often too restrictive since it fails to cover classes of faults not directly tied to memory, such as register corruptions, control flow corruptions, and incorrect computation, which are prevalent in real-world systems. In our empirical study, we choose to use a more expressive fault model supported in KULFI. The chosen fault model considers all instructions of a program as candidate fault occurrence locations, including memory reads and writes, register operations, and control flow instructions.

In our case study, we consider implementations of five well-known sorting algorithms: BubbleSort (with preemptive

TABLE I
STATISTICS OF SORTING ALGORITHMS

Algorithm	LOC	SIC	MinDIC	MaxDIC	AvgDIC
BubbleSort	56	13	68k	61442k	14818k
RadixSort	61	39	30k	2040k	565k
QuickSort	65	25	34k	1110k	303k
MergeSort	70	38	79k	1269k	364k
HeapSort	77	28	15k	1519k	500k

termination criterion), RadixSort, QuickSort, MergeSort, and HeapSort.[3] All implementations take as input an array of integers to be sorted, and they output the sorted array. Since this is a preliminary study, we do not bias on the size and input data, i.e., the arrays are of random size (between 2000 and 10000) and contain random integer data. (As part of future work, we plan to experiment with various fixed data sizes and algorithm-specific inputs).

We perform a *fault injection campaign* for each sorting algorithm implementation using KULFI. Each fault injection campaign consists of 200 *fault injection experiments*. A single fault injection experiment comprises of 100 executions of an algorithm. Therefore, each algorithm is executed a total of 20000 times, which we split into 200 fault injection experiments so that we can later compute the statistical significance of our results. In each execution, the algorithm operates on a different randomly generated input array, while a single random bit-flip error is injected at runtime using KULFI. We describe the details of our fault injection strategy next.

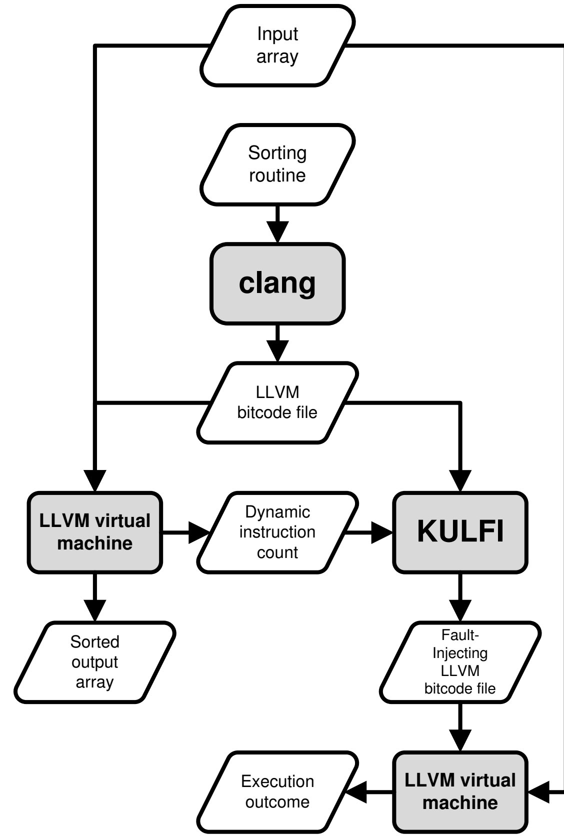

Fig. 3. Fault Injection Strategy

A. Fault Injection Strategy

Even with a fault injector such as KULFI available, selecting a realistic fault injection probability requires careful planning; we now present our approach in this regard. As noted in §II, fault filtration naturally occurs across the hardware/software stack where many faults fall into the "don't-care" sets of the higher layers. Specifically, Sanda et al. [1] report how an IBM POWER6 processor was actually bombarded with protons and alpha particles within an elaborate experimental setup. The authors estimated that the percentage of faults that actually reached the application logic was 0.2% of the overall number of latch-level faults. While this approach to fault simulation is quite realistic, such "bottom-up" fault injection approach (and its infrastructural overheads) are clearly out of reach to most researchers. On the other hand, there are a number of recent approaches targeting software-level resilience enhancing mechanisms (see §II). Therefore, we decided to focus our empirical study only on the effects of faults that *do reach* the application logic since those are of a particular interest to the software-level resilience community. We still had to devise a reasonable and fair fault injection probability.

Given the above discussion, we now define additional notions that help us elaborate our studies. By the term *dynamic instruction* we refer to a runtime instance of a static LLVM program instruction. We define the *dynamic instruction count* as the actual number of dynamic LLVM instructions executed corresponding to a specific program execution. For example, for a simple program consisting of five static instructions in a loop that iterates 1000 times, the static instruction count is five, while the dynamic instruction count is 5000. For our sorting algorithms, the dynamic instruction count varies depending on the algorithm considered and the input array, which we have to take into account to ensure that all dynamic instructions are considered for fault injection with equal probability. Table I gives various statistics for our sorting algorithms:

- LOC is the number of lines of code,
- SIC the number of static fault site instructions,
- MinDIC the minimum dynamic instruction count,
- MaxDIC the maximum dynamic instruction count, and
- AvgDIC the average dynamic instruction count.

We initially perform a faultless run of an algorithm on an input to compute the dynamic instruction count N for a particular execution. We then define the probability of fault injection for each dynamic instruction to be $1/N$. This ensures that all dynamic instructions are equiprobably considered for fault injection in subsequent runs of the program on the same input.

Figure 3 illustrates our fault injection strategy for this case study performed using KULFI. First, a sorting routine is compiled using LLVM's C/C++ front-end Clang into an LLVM bitcode file, which contains LLVM's intermediate representation. Then, we execute the generated bitcode file

[3]Source code of the examples and scripts for performing the experiments are also available from the KULFI website.

using the LLVM virtual machine (i.e., *lli*) and as input we provide a randomly generated input array. We record the sorted output array for later comparison. In the process, we also measure the dynamic instruction count N for this particular faultless execution. Using the dynamic instruction count we compute the probability of fault injection for each dynamic instruction as $1/N$. The original LLVM bitcode file and the computed fault injection probability are given as inputs to KULFI. The tool generates a fault-injecting LLVM bitcode file, i.e., an instrumented version of the original bitcode file in which a transient fault might be injected during execution into a dynamic instruction with the computed probability. The fault-injecting LLVM bitcode file is then executed on the same input array. We observe the number of injected faults and log only the executions during which exactly one fault is injected; we call such executions 1-fault executions. Executions where the number of injected faults is not equal to one are discarded. We record the outcome of every 1-fault execution to later analyze the effect of fault injection.

B. Experimental Results

We identify three possible outcomes of an execution of a sorting algorithm with a fault injected at runtime.

Benign Fault. In general, a transient fault is benign when the program state at the end of a faulty execution is the same as the program state obtained after a faultless execution. In the context of sorting algorithms, the output array obtained as the result of a faulty execution has to exactly match the sorted output array of the faultless execution.

Segmentation Fault. We classify a transient fault that causes a program to crash due to performing an invalid memory access as a segmentation fault.

Silent Data Corruption (SDC). A fault is classified as an SDC when the ordering, frequency, or the set of array elements at the output of the faulty execution is different from that of the faultless execution.

After each fault injection experiment (i.e., 100 1-fault executions), we log the number (i.e., fraction) of executions falling into each category. For example, here is how one such log entry might look like:

```
Benign: 41, Segmentation: 29, SDC: 30
```

In the end of every sorting algorithm fault injection campaign, we are left with 200 such log entries, one for every fault injection experiment. We perform statistical analysis of these logs and present our empirical results next.

From Figures 4–6, we observe that the values in log entries obtained after every fault injection campaign are strongly clustered, and there is a statistically significant distribution of the fractions for each outcome category. The larger shapes (*e.g.*, triangles) in the middle of clusters are indicative of the larger number of instances of faults that are closer to the middle. More specifically, the fractions of every category of outcomes across the 200 fault injection experiments follow the 68-95-99.7 (or three-sigma) rule of normal distribution. Therefore, using our empirical data we can draw statistically

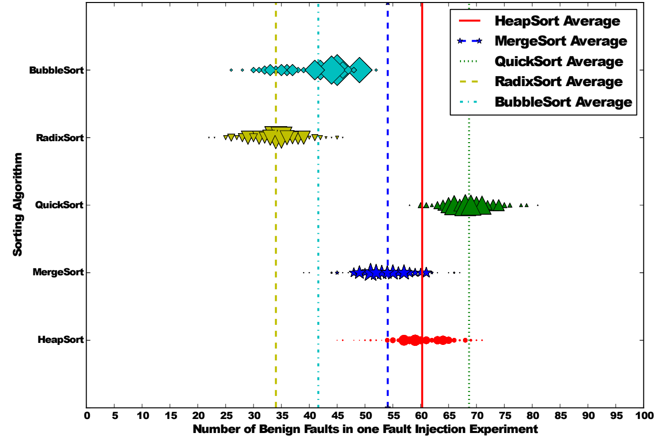

Fig. 4. Benign Faults

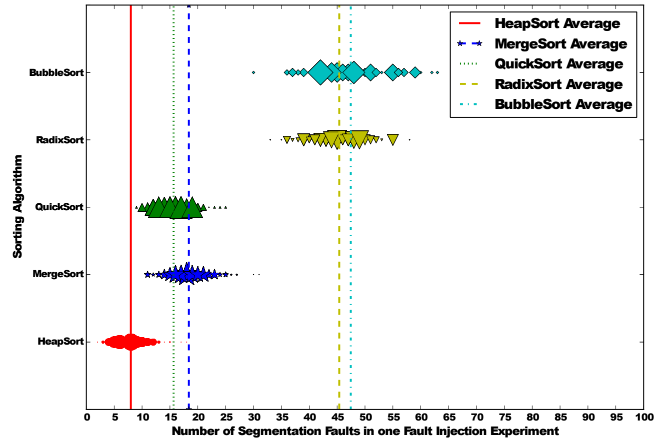

Fig. 5. Segmentation Faults

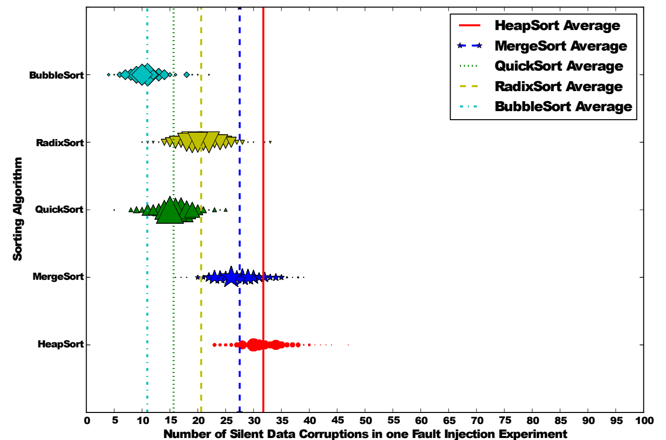

Fig. 6. Silent Data Corruptions

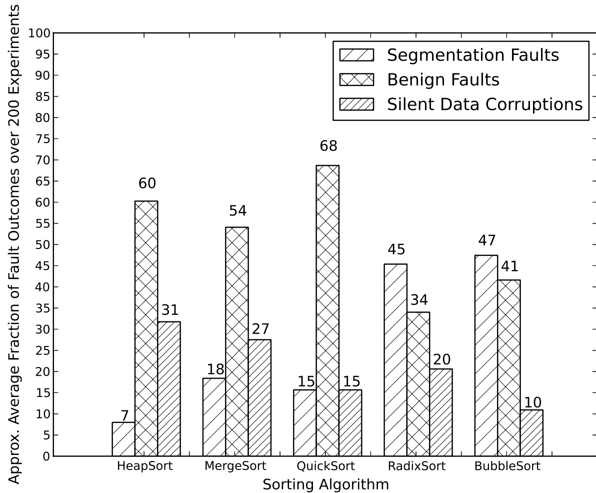

Fig. 7. Summary of Fault Injection Campaigns

significant conclusions about the behavior of the analyzed sorting routines in a faulty environment.

Figure 7 details the comparison of the sorting algorithms based on the average number of executions in each category of fault outcomes. For example, we observe that BubbleSort, though an algorithm with higher time complexity than HeapSort, leads to more detectable faults. In fact, HeapSort is the least resilient with respect to SDCs and results in either benign faults or SDCs in its fault injection campaign. QuickSort masks majority of injected faults and therefore its high number of benign faults. It is worthwhile to note that the three algorithms that have the least number of detectable faults (MergeSort, QuickSort, and HeapSort), follow a recursive divide-and-conquer algorithm design paradigm. On the other hand, BubbleSort leads to more segmentation faults.[4] We observe that approximately 85% of the executions of QuickSort and 90% of the executions of BubbleSort avoid SDCs. To sum up, QuickSort is the most resilient and available algorithm of the algorithms considered. The following summarizes the resilience-related observations, where the lower numbers are better (lower number of faults in those categories):

$$\text{SDCs} : \text{Bubble} < \text{Quick} < \text{Radix} < \text{Merge} < \text{Heap}$$
$$\text{Seg. Faults} : \text{Heap} < \text{Quick} < \text{Merge} < \text{Radix} < \text{Bubble}$$

In addition to logging execution outcomes, we also maintained a mapping of the dynamic instruction where the fault was injected to the outcome that was produced in that execution. We draw some interesting observations from this mapping. In BubbleSort, half of the faults injected into registers that are employed in computing the index of the array access in the expression `Array[i-1]` produce segmentation faults. In HeapSort, injecting faults into the instructions executed just before and just after the recursive calls causes a high percentage (close to 75%) of SDC faults. Such precise profiling of fault injection sites enables us to observe critical instructions, specific to a sorting algorithm, where error injection leads to SDCs. Note that, as an area of future work, one can potentially extend the notion of such critical regions of an algorithm to other algorithms that follow similar design patterns. Furthermore, targeted fault detection and recovery mechanisms can be employed around these critical regions to provide cheap and effective means for improving resilience of programs to transient faults.

[4]As to the reliability and repeatability of measuring segmentation faults, one has to choose virtually identical runtimes and memory layouts as well as mappings of user variables to memory locations. We will address these considerations in future work.

```
    bubbleSort(array[], size)
    {
PP0:  //state --> XXX
L0:   for (i = (size - 1); i > 0; i--)
      {
PP1:    //state --> TXX
L1:     for (j = 1; j <= i; j++)
        {
PP2:      //state --> TTX
L2:       if (array[j-1] > array[j])
          {
PP3:        //state --> TTT
L3:         swap(array[j-1],array[j])
PP4:        //state --> TTF
          }
          else
          {
PP5:        //state --> TTF
L4:         skip
          }
        }
PP6:    //state --> TFX
      }
    }
```

Fig. 8. BubbleSort with Encoded Predicate States

V. Towards a Formal Approach

In order to formally explain the behavior of programs under faults, we found it natural to adopt a predicate-abstraction-based approach. Our overall goal is to observe and explain faults in terms of their effect on abstract predicate state transitions. To the best of our knowledge, the use of predicate abstraction in resilience research is a novel research direction. In this work, we adopt the predicate-abstraction-based approach introduced by Ball [33]; while Ball used predicate abstraction to define a novel program coverage metric, we use it to study faulty behaviors in a more manageable abstract state space. In addition, in §V-D we briefly discuss our preliminary fault detector based on reachable abstract state space deviations, where our detector reports a fault when encountering an erroneous transition.

A. Predicate Transition Diagram for BubbleSort

We first present our approach that leverages predicate abstraction using BubbleSort as a running example (see Figures 8 and 9). Figure 8 gives the source code of BubbleSort for which we generate a predicate transition diagram. In order to as precisely as possible capture the effect of faults in BubbleSort, for our set of relevant predicates we choose the following: $i > 0$, $j \leq i$, and $array[j-1] > array[j]$. These predicates govern the major control-flow steps in the program, and therefore are important for understanding its behaviors. The chosen set of predicates defines our abstract states. We

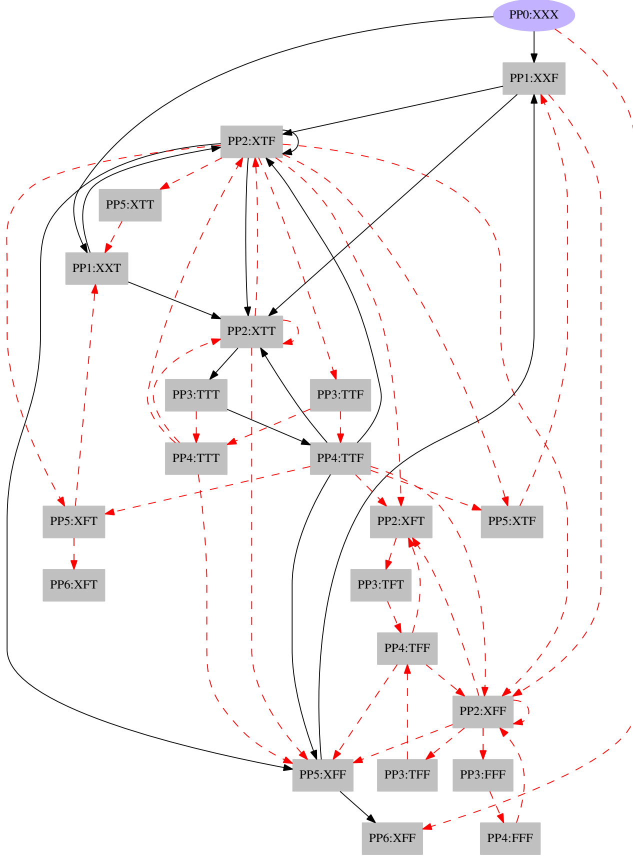

Fig. 9. Abstract Predicate Transition Diagram of BubbleSort

evaluate a vector of these predicates at chosen critical program locations, typically where one of the predicates governs control flow, thereby effectively computing reachable abstract states at those locations. In our example, we mark these program locations `PP0` through `PP6`. The evaluation of a predicate at a program location depends on whether the predicate is in scope (more specifically, all the variables in the predicate are in scope). For example, at `PP1` only the first predicate is in scope, and at `PP0` none of the predicates are in scope (i.e., none of the variables in any predicate are in scope). The truth values of predicates not in scope are recorded as `X`, which stands for "unknown."

Having instrumented the BubbleSort program with appropriate predicate evaluators, we run it with and without injected faults, observe the abstract state transitions made, and superimpose the transitions in a predicate transition diagram. Multiple such runs are performed, one for each permutation of a fixed-size input array, in order to accomplish a higher degree of coverage of all possible executions of BubbleSort. That empirically ensures that our predicate transition diagrams indeed represent over-approximations of the reachable state space even though they are generated dynamically. The outcome is the diagram of Figure 9. We use the following conventions in this figure:

- The solid (black) edges are *valid transitions* observed during both fault-free and faulty executions, under some input.
- The dotted (red) edges are *invalid transitions* that are never observed during fault-free runs; they are spurious state transitions caused by fault injection.

TABLE II
STATISTICS OF PREDICATE TRANSITION DIAGRAMS

Algorithm	Invalid Transitions	Valid Transitions	Total
BubbleSort	38 (71%)	16 (29%)	54 (100%)
RadixSort	64 (72%)	25 (28%)	89 (100%)
QuickSort	35 (53%)	32 (47%)	67 (100%)
MergeSort	67 (47%)	76 (53%)	143 (100%)
HeapSort	56 (66%)	29 (34%)	85 (100%)

Note that in our current prototype implementation, we carry out the described predicate evaluation concretely at runtime. In our future work, we plan to employ symbolic techniques (e.g., [7], [8], [33]) to obtain predicate transition diagrams automatically.

The benefit of obtaining predicate transition diagrams with and without fault injection is that it provides an instantaneous visualization of the effect of faults on the overall program execution. In our future work, we plan to construct these diagrams not just using control-flow predicates, but also interesting program invariants. Furthermore, §V-D provides a preliminary assessment of our fault detector synthesized using the predicates employed in predicate transition diagrams. One of the outstanding challenges in this line of research is to synthesize such fault detectors that impact the computation the least, and are effective in trapping the most number of faults.

B. Generating Predicate Transition Diagrams

The general approach to build predicate transition diagrams of the kind illustrated in Figure 9 is as follows:

- Given a collection of predicates and instrumentation locations in the program, we instrument each of these locations with a call to a predicate evaluation function. This function takes predicates as inputs, evaluates them at the given program location, and returns a three-valued vector containing values true (`T`), false (`F`), and unknown (`X`) for each predicate. An unknown value is generated for a predicate if there is a variable in the predicate that is undefined at the program point.
- For a given program, let δ be its fault-free predicate transition relation and δ_F its predicate transition relation under faults. These transition relations are defined for a general program input.
- To combine these transition relations into a meaningful and informative predicate transition diagram, we create a solid (black) edge for every transition in δ and a dotted (red) edge for every transition in $\delta_F \setminus \delta$.

C. Results

We apply the procedure described earlier on all our sorting algorithms to generate their predicate transition diagrams.

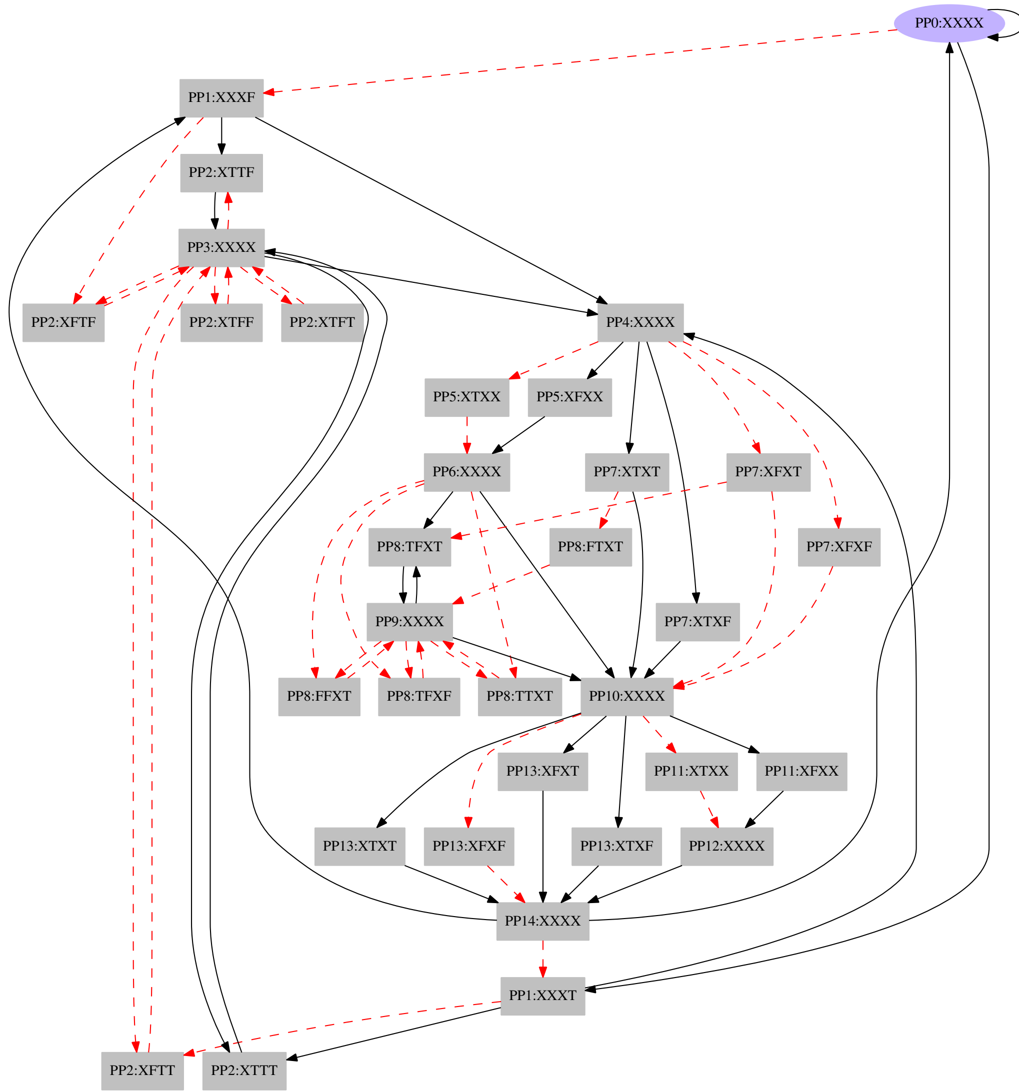

Fig. 10. Abstract Predicate Transition Diagram of QuickSort

Then, we perform a preliminary assessment of the usefulness of the generated diagrams for estimating resilience of our algorithms. Figure 10 shows the abstract predicate transition diagram of QuickSort.[5] If we do a visual comparison of Figures 9 and 10, we can notice a higher degree of invalid transitions (dotted edges) in the BubbleSort diagram. This corresponds to the fact that the number of benign faults in BubbleSort is much lower than in QuickSort (see Figure 7), which in turn lead us to compare the degree of valid transitions in our diagrams against the fraction of exhibited benign faults.

Table II gives statistics of the generated predicate transition diagrams with respect to the number of valid and invalid transitions. We compared the percentage of valid transitions against the percentage of benign faults, and as it turns out, there appears to be a rough correlation between these numbers: higher percentage of benign faults typically implies higher percentage of valid transitions. While this is a very preliminary, crude exploration that should be taken with a grain of salt, as an area of future work we are planning to explore this and similar connections further. For example, we could refine our diagrams to include probabilities of particular transitions being taken, which would enable us to reason more precisely about how often particular invalid transitions are actually taken.

[5] Note that detailed predicate transition diagrams for other sorting routines are provided at https://github.com/soar-lab/KULFI/wiki/Predicate-Transition-Diagrams-for-Various-Sorting-Algorithms.

D. From Predicate Transition Diagrams to a Fault Detector

In this section, we summarize our preliminary experiments that illustrate how a fault detector may be synthesized based on insights gained from predicate transition diagrams. In a predicate transition diagram, the presence of invalid transitions may be leveraged to detect occurrences of transient faults. Hence, we devise a simple approach that uses generated predicate transition diagrams to automatically detect faults occurring during execution of a sorting algorithm. First, we

TABLE III
FAULT DETECTION STATISTICS

Algorithm	% Faults Detected
BubbleSort	66.0%
RadixSort	88.4%
QuickSort	100.0%
MergeSort	100.0%
HeapSort	33.7%
Average	77.6%

compute a complete set of reachable abstract states for a fault-free sorting routine by running it on all possible inputs as described previously. Obviously, every subsequent run of the program under zero faults must visit only states within this complete set. If not, we can surmise that the program execution has experienced a transient fault. These transient faults may result in a segmentation fault, turn out to be benign, or worse still, result in an SDC.

In order to demonstrate the effectiveness of this approach, we used KULFI to empirically test it on our sorting routines. We run every sorting routine on an array of a given size where each element is initialized to a random value. During the execution, we inject exactly one transient fault using KULFI; in addition, we make sure that we only keep faults that cause SDCs and disregard others that we are not interested in. We also capture the set of reachable abstract states and check whether it is included in our initial complete set of abstract states. If not, we report a detected fault. We perform these experiments for all five sorting routines with input arrays of size 200; each routine is repeatedly executed until 1000 SDC-causing faults is reached. Table III shows the obtained fault detection statistics. Note that the fault detection statistics refer to the detection of only those faults which cause SDC in our experiments. The column "% Faults Detected" gives the percentage of faults (out of 1000 SDC-causing faults injected) that our approach successfully detected. The fault detection percentage varies from 33.7% all the way to 100%, with an average of 77.6%, which clearly shows the promise of the approach.

Clearly, sorting is an extreme example of a system where data (i.e., the items being sorted) directly affects control flow. In general, the degree to which data affects control will determine the success of error detection based purely on control-flow tracking. Another point worth noting is that for a small-scale study such as ours, to build a reachable predicate state space we execute a program on all possible inputs. In the future, and especially for larger experiments, we might use static analysis and/or sampling-based techniques for building the reachable state space. We will also incorporate lessons from previous work on control-flow tracking based error detection cited earlier. Our hope is to bring in the insight of predicate transition diagrams into this field.

VI. Conclusions and Future Work

In this paper, we presented a unique, thorough case study of resilience of several popular and widely used sorting algorithms to hardware faults. To be able to perform such an extensive resilience study, we first implemented a new, open source LLVM-level fault injector called KULFI. Faults injected by KULFI at the LLVM level provide a reasonable fault model for actual hardware faults. Using KULFI, we performed an extensive empirical study that observed behavior of sorting algorithms when faults are being injected. Based on the statistically significant results of this study, we drew informative conclusions about resilience of these algorithms. Our empirical results and conclusions aim to serve as a guidance for software developers that have to take resilience into account when choosing an appropriate sorting routine for their task.

Apart from the empirical case study, we also introduce our novel predicate-abstraction-based approach for analyzing resilience of programs. In the approach, we dynamically generate abstract predicate transition systems for fault-free and faulty executions. Then, we superimpose these systems in a meaningful way in order to generate abstract predicate transition diagrams that visualize fault propagation at the higher, abstract level. Our abstraction seems to often explain at the more manageable high-level the empirically measured resilience of algorithms. In the end, we leverage our predicate abstraction approach to build a simple fault detector, and we show its effectiveness on our set of benchmarks.

There are numerous avenues for future work we are planning to explore. While abstract predicate transition diagrams are a useful high-level visualization of the effects of faults, more research is needed in coming up with the right visual metaphors. Currently, our predicate abstraction prototype tool performs abstraction dynamically, at runtime. In the long run, we would like to employ well-known symbolic predicate abstraction techniques instead. That would enable us to also leverage automatically generated predicates of global system invariants, instead of just relying on predicates we syntactically observed in source code. Apart from improving our foundational techniques, we will also perform similar empirical resilience studies with other classes of algorithms. When we find good ways to estimate resilience through empirical studies and formal approaches, we will have the basis for selecting among a class of algorithms those that emerge to be more resilient. Finally, in addition to observing faults at a higher-level, formal approaches may also help us design better fault detectors, which is clearly an exciting direction of future research. All these ideas will be experimented on a much larger collection of benchmark examples drawn from scientific computing and other areas.

VII. Acknowledgement

We would like to thank Pedro Diniz, Prabhakar Kudva, Shuvendu Lahiri, and Karthik Pattabiraman for their insights and feedback on the early drafts of this paper. We are also grateful to Sui Chen for trying out early versions of KULFI and providing us with detailed feedback that helped us to improve it.

REFERENCES

[1] P. N. Sanda, J. W. Kellington, P. Kudva, R. N. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, “Soft-error resilience of the IBM POWER6 processor,” *IBM Journal of Research and Development*, pp. 275–284, 2008.

[2] J. A. Rivers, M. S. Gupta, J. Shin, P. N. Kudva, and P. Bose, “Error tolerance in server class processors,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 945–959, 2011.

[3] S. K. Sahoo, M. lap Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou, “Using likely program invariants to detect hardware errors,” in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 70–79.

[4] F. Perry and D. Walker, “Reasoning about control flow in the presence of transient faults,” in *Static Analysis Symposium (SAS)*, 2008, pp. 332–346.

[5] “KULFI: An instruction level fault injector,” http://github.com/soar-lab/KULFI/.

[6] S. Graf and H. Saïdi, “Construction of abstract state graphs with PVS,” in *International Conference on Computer Aided Verification (CAV)*, 1997, pp. 72–83.

[7] S. Das, D. Dill, and S. Park, “Experience with predicate abstraction,” in *International Conference on Computer Aided Verification (CAV)*, 1999, pp. 160–171.

[8] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of C programs,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 203–213.

[9] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[10] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, “The price of resiliency: a case study on sorting with memory faults,” *Algorithmica*, vol. 53, no. 4, pp. 597–620, 2009.

[11] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, “Experimental study of resilient algorithms and data structures,” in *International Conference on Experimental Algorithms*, 2010, pp. 1–12.

[12] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[13] “The LLVM compiler infrastructure,” http://llvm.org/.

[14] V. Sieh, “Fault-injector using UNIX ptrace interface,” in *Internal Report 11/93, IMMD3, Universität Erlangen Nürnberg*, 1993.

[15] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A flexible software-based fault and error injection system,” *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995.

[16] H. Madeira, M. Z. Rela, F. Moreira, and J. a. G. Silva, “RIFLE: A general purpose pin-level fault injector,” in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 1994, pp. 197–216.

[17] M.-C. Hsueh, T. Tsai, and R. Iyer, “Fault injection techniques and tools,” *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[18] A. Jin, J. Jiang, J. Hu, and J. Lou, “A PIN-based dynamic software fault injection system,” in *International Conference for Young Computer Scientists (ICYCS)*, 2008, pp. 2160–2167.

[19] M. de Kruijf, S. Nomura, and K. Sankaralingam, “Relax: An architectural framework for software recovery of hardware faults,” in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 497–508.

[20] A. Thomas and K. Pattabiraman, “LLFI: An intermediate code level fault injector for soft computing applications,” *Silicon Errors in Logic System Effects (SELSE)*, 2013, informal proceedings.

[21] S. Chen, personal communication, 2013.

[22] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, “Application-based metrics for strategic placement of detectors,” in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2005, pp. 75–82.

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

[24] J. Wei and K. Pattabiraman, “BLOCKWATCH: Leveraging similarity in parallel programs for error detection,” in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[25] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, “Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 123–134.

[26] J. Yu, M. Garzaran, and M. Snir, “Efficient software checking for fault tolerance,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008, pp. 1–5.

[27] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, “Automated derivation of application-aware error detectors using static analysis: The trusted Illiac approach,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 1, pp. 44–57, 2011.

[28] S. Hukerikar, P. C. Diniz, and R. F. Lucas, “Programming model extensions for resilience in extreme scale computing,” in *Euro-Par Parallel Processing Workshops*, 2012, pp. 496–498.

[29] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, “Fault resilience of the algebraic multi-grid solver,” in *International Conference on Supercomputing (ICS)*, 2012, pp. 91–100.

[30] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, “SymPLFIED: Symbolic program-level fault injection and error detection framework,” in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 472–481.

[31] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[32] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, “Low-cost on-line fault detection using control flow assertions,” in *IEEE On-Line Testing Symposium (IOLTS)*, 2003, pp. 137–143.

[33] T. Ball, “A theory of predicate-complete test coverage and generation,” in *International Conference on Formal Methods for Components and Objects (FMCO)*, 2005, pp. 1–22.