PRESAGE: Protecting Structured Address Generation against Soft Errors

Vishal Chandra Sharma, Ganesh Gopalakrishnan
School of Computing, University of Utah
Email: {vchandra,ganesh}@cs.utah.edu

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
Email: sriram@pnnl.gov

Abstract—Modern computer scaling trends in pursuit of larger component counts and power efficiency have, unfortunately, lead to less reliable hardware and consequently soft errors escaping into application data (“silent data corruptions”). Techniques to enhance system resilience hinge on the availability of efficient error detectors that have high detection rates, low false positive rates, and lower computational overhead. Unfortunately, efficient detectors to detect faults during address generation have not been widely researched (especially in the context of indexing large arrays). We present a novel lightweight compiler-driven technique called PRESAGE for detecting bit-flips affecting structured address computations. A key insight underlying PRESAGE is that any address computation scheme that propagates an already incurred error is better than a scheme that corrupts one particular array access but otherwise (falsely) appears to compute perfectly. Ensuring the propagation of errors allows one to place detectors at loop exit points and helps turn silent corruptions into easily detectable error situations. Our experiments using the PolyBench benchmark suite indicate that PRESAGE-based error detectors have a high error-detection rate while incurring low overheads.

I. INTRODUCTION

High performance computing (HPC) applications will soon be running at very high scales on systems with large component counts. The shrinking dimensions and reducing power requirements of the transistors used in the memory elements of these massively parallel systems make them increasingly vulnerable to temporary bit-flips induced by system noise or high-energy particle strikes. These temporary bit-flips occurring in memory elements are often referred as soft errors. Previous studies project an upward trend in soft error induced vulnerabilities in HPC systems thereby pushing down their mean-time-to-failure (MTTF) [1], [2].

These trends drastically increase the likelihood of a bit-flip occurring in long-lived computations. Specifically, a bit-flip affecting computational states of a program under execution such as ALU operations or live register values, may lead to silent data corruption (SDC) in the final program output. Making matters worse, such erroneous values may propagate to multiple compute nodes in massively parallel HPC systems [3].

The key focus of this paper is to detect bit-flips affecting address computation of array elements. For example, to load a value stored in an array \( A \) at an index \( i \), a compiler must first compute the address of the location referred by the index \( i \). A compiler performs this operation under-the-hood by using the base address of \( A \) and adding to it an offset value computed using the index \( i \). This style of address generation scheme, which uses a base address and an offset to generate the destination address, is often referred as the structured address generation. Accordingly, the computations done in the context of structured address generation are referred as structured address computations.

Often, computational kernels used in HPC applications involve array accesses inside loops thus requiring structured address computations. For these kernels, there is a real chance of one of their structured address computations getting affected by a bit-flip. A structured address computation pertaining to an array, when subjected to a bit-flip may produce an incorrect address that still refers to a valid address in the address space of the array. Using the value stored at this incorrect but a valid address may lead to SDC without causing a program crash or any other user-detectable-errors.

In this paper, we demonstrate that the bit-flips affecting structured address computations for aforementioned class of computational kernels lead to non-trivial SDC rates. Also, we present a novel technique for detecting bit-flips impacting structured address computations. Given that the structured address computations involve arithmetic operations that use a CPU’s computational resources, we consider an error model where bit-flips affect ALU operations and CPU register files. We assume DRAM and cache memory to be error-free, a reasonable assumption because they are often protected using ECC mechanisms [4]–[7]. We further limit the scope of our error model by considering only ALU operations and register values that correspond to structured address computations.

Specifically, we make following contributions in this paper:

1) A fault injection driven study done on 10 benchmarks drawn from the PolyBench/C benchmark suite [8] demonstrating that structured address computations in those benchmarks, when subjected to bit-flips, lead to non-trivial SDC rates.

2) A novel scheme that employs instruction-level rewriting of the address computation logic used in structured address computations. This rewrite preserves an error in a structured address computation by intentionally corrupting all structured address computations that follow it. This requires creation of a dependency-chain between all structured address computation pertaining to a given array. Enabling the flow of error helps in following ways:
• **Strategic Placement of Error Detectors**: Instead of checking each and every structured address computations for soft errors (which is prohibitively expensive), we strategically place our error detectors at the end of a dependency chain.

• **Promoting SDCs to Program Crashes**: By enabling the flow of error in address computation logic, we increase the chances of promoting an SDC to a program crash (that is more easily detected).

3) We present a methodology for implementing our proposed scheme as a compiler-level technique called PRESAGE (PRotEcting Structured Address GEneration). Specifically, we have implemented PRESAGE using LLVM compiler infrastructure [9], [10] as a transformation pass. LLVM preserves the pointer related information at LLVM intermediate representation (IR) level (as also highlighted in recent works [11], [12]) while providing access to a rich set of application programming interfaces (APIs) for seamlessly implementing PRESAGE transformations. This is the key reason behind choosing LLVM as the tool-of-choice.

In summary, our error-detection approach is based on the following principle:

*The larger the fraction of system state an error corrupts, the easier it is to detect.*

The rest of the paper is organized as follows. Sec. §II provides a literary review of the closely related work done in this area. Sec. §III explains the key idea through a set of small examples. Sec. §IV formally introduces the key concepts and the methodology used to implement PRESAGE. In Sec. §V, we provide a detailed analysis of the experiments carried out to measure the efficacy of PRESAGE. Finally, Sec. §VI summarizes the key takeaways and future directions for this work.

## II. Background & Related Work

A previous work by Casas-Guix et al. [13] shows that an Algebraic Multigrid (AMG) solver is relatively immune to faults and can, often, recover to an acceptable final answer even after encountering a momentary bit-flip in the data state. However, they realize that any fault in the space of pointers often wreaks havoc, since the corrupted pointers tend to write data values into unintended memory spaces. As a solution, they propose the use of pointer triplication, which not only helps detect errors in the value of a pointer variable but also correct the same. Unfortunately, pointer triplication comes with a high overhead of runtime checks. Also, they do not focus on the scenarios where corruptions in structured address computations lead to SDC, which is the key focus of our work.

Another work by Wei et al. [11] highlights the difference between the results of the fault injection experiments done using a higher-level fault injector LLFI targeting instructions at LLVM IR level, and a lower-level, PIN based, fault injector performing fault injections at x86 level. This work highlights that LLVM offers a separate instruction called `getElementPtr` for carrying out structured address computations whereas at x86-level same instruction can be used for computing address as well as performing non-address arithmetic computations. Another recent work by Nagarakatte et al. [12] shows how, by associating meta-data and by using Intel’s recently introduced MPX instructions, one can guard C/C++ programs against pointer-related memory attacks. The key portion of this work is also implemented using LLVM infrastructure. The above two works, in a way, influenced our decision to choose LLVM for implementing PRESAGE.

Researchers have also explored the development of application-level error detectors for detecting soft errors affecting a program’s control states [14]–[16]. Another key area in application-level resilience is algorithm-based fault tolerance (ABFT), which exploits algorithmic properties of well-known applications to derive efficient error detectors [17], [18]. Researchers have also focused in the past to optimize the placement of application level error detectors at strategic program points. The information about these strategic location are usually derived through well established static and dynamic program analysis techniques [19]–[22]. To the best of our knowledge, none of the previous works have focused on protecting structured address generation leading to SDC, the focus of our work.

## III. Motivating Example

Fig. 2 presents a simple C function `foo1` performing store operations to even-indexed memory locations of an array `a[]` of size `2n` inside a for loop. It also stores the last accessed array address into a variable `addr` at the end of every loop-iteration. Fig. 1 represents the corresponding x86 code emitted for the `foo1` function when compiled using clang compiler with O1 optimization level. Registers `%esi` and `%ecx` represent the variable `n` and the loop iterator `i` of the function `foo1`, whereas registers `%rdi` and `%rax` correspond to the array’s base address and index respectively. In every loop iteration, a destination array address is computed by the expression `((%rdi,%rax,0x8)`) which evaluates to `(0x8+%rax+%rdi)`, the value in register `%rax` is incremented by 2, and the base address stored in `%rdi` remains fixed. It is worth noting that the final address computation denoted by the expression `( %rdi,%rax,0x8)` is not user-visible and is something compiler does under-the-hood.

In contrast to the fixed base address (FBA) scheme used in function `foo1`, function `foo2` (shown in Fig. 3), a semantically equivalent version of `foo1`, introduces a novel relative base address (RBA) scheme. Specifically, `foo2` uses an array address computed in a loop iteration (`addr`) as the new base address for the next loop iteration along with a relative index (`rid`) as shown in Fig. 2.

This simple but powerful scheme creates a dependency chain in the address computation logic as the computation of any new address would depend on the last computed address. Therefore, our RBA scheme guarantees that if an address computation of an array element gets corrupted then all subsequent address computations would also become erroneous.
This, in turn, enables us to strategically place error detectors at a handful of places in a program (preferably at all program exit points) thereby making the whole error detection process lightweight.

For example, in functions foo1 and foo2, the address of a new array element, computed during every loop iteration, is stored in the variable addr. The value stored in the variable addr may get corrupted in following scenarios:

**Error Scenario I**: A bit-flip occurs in the value stored in the loop-iterator variable i in functions foo1 and foo2.

**Error Scenario II**: A bit-flip affecting the value stored in the absolute index variable id in functions foo1 and foo2.

**Error Scenario III**: A bit-flip occurs in the value stored in the relative index variable rid, which is only present in the functions foo2.

**Error Scenario IV**: A bit-flip affecting the value stored in the variable addr in functions foo1 and foo2.

The above program-level sites are listed in Table I for easy reference. With respect to the error scenario IV, it is evident that only in the case of foo2, when the result of final address computation stored in addr is corrupted during one of the loop iterations, all subsequent address computations in the remaining loop iterations would also get corrupted due to the dependency chain introduced in the address computation logic. We further demonstrate the behavior of these dependency chains, introduced by our RBA scheme, through a small set of fault injection driven experiments. Fig. 4 presents the result of two independent runs for function foo1. The X-axis shows the number of loop iteration whereas the Y-axis shows the value stored in the variable addr. The execution with label foo1_addr represents a fault-free execution of foo1. The execution with label foo1_addr_corrupt represents a faulty execution of foo1 where a single bit fault is introduced at bit position 6 of the value stored in addr during the tenth loop iteration. Similarly, Fig. 5 presents the result of two independent runs for function foo2 such that a single bit fault is introduced at bit position 6 of the value stored in addr during the first loop iteration in the faulty execution represented by the label foo2_addr_corrupt. We can clearly notice that only in the case of function foo2, once an address value stored in addr gets corrupted, all subsequent address values stored in addr are also corrupted.
IV. METHODOLOGY

Sec. §III demonstrates that a simple rewrite of the address computation logic introduces a dependency chain thereby enabling the flow of error. Given that the address computation is often done in a user-transparent manner by the compiler, we implement our technique at the compiler-level. Specifically, we choose the LLVM compiler infrastructure to implement our technique as a transformation pass (here on referred to as PRESAGE) which works on LLVM’s intermediate representation (IR). Our implementation eliminates the need for any manual effort from programmers thereby allowing our technique to scale to non-trivial programs. LLVM’s intermediate representation (IR) provides a special instruction called `getelementptr` (here on referred as GEP for brevity) for performing address computation of Aggregate types including `Array` type\(^1\). Therefore, all analyses implemented as part of PRESAGE are centered around the GEP instruction. A GEP instruction requires a base address, one or more index values, and the size of an element to compute an address, often referred as structured address. Given the key focus of our work is to protect these structured addresses, the definition of an array on which PRESAGE transformations are applied closely follows the LLVM’s Array type definition with some restrictions as explained below:

**Definition 1:** An array in this paper always refers to a contiguous arrangement of elements of the same type laid out linearly in the memory.

**Definition 2:** All structured address computations protected using PRESAGE must always use only one index for address computations. It is important to note that this is needed only to simplify the implementation and does not limit the scope of PRESAGE as multi-dimensional arrays can be easily represented using single-indexed scheme. For example, a two-dimensional array could be laid out linearly in memory by traversing it in row-major or column-major fashion.

**Definition 3:** The base addresses used in all structured address computations protected by PRESAGE must not be updated. For example, if a PRESAGE transformation is applied on a callee function to protect its structured address computations then the callee function must not mutate the base addresses referenced in structured address computations protected by PRESAGE.

\(^1\)LLVM’s type system is explained in its language reference manual located at: http://llvm.org/docs/LangRef.html

<table>
<thead>
<tr>
<th>Fault Site</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Loop iterator variable.</td>
</tr>
<tr>
<td>id</td>
<td>Absolute index variable.</td>
</tr>
<tr>
<td>addr</td>
<td>A variable containing an address of a location in the array a[].</td>
</tr>
<tr>
<td>rid</td>
<td>Relative index variable (only present in foo2).</td>
</tr>
</tbody>
</table>

**TABLE I:** List of fault sites in functions `foo1` and `foo2`

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathcal{F})</td>
<td>A target function on which PRESAGE transformations are applied.</td>
</tr>
<tr>
<td>(b)</td>
<td>A base address with at least one user in the target function.</td>
</tr>
<tr>
<td>(\mathcal{B})</td>
<td>A basic block in the target function.</td>
</tr>
<tr>
<td>(E(B_1,B_2))</td>
<td>A boolean function which returns \textit{true} only if an edge exists from (B_1) to (B_2).</td>
</tr>
<tr>
<td>(\mathcal{L}_{B_1}(B))</td>
<td>A set of all immediate predecessor basic blocks of (B).</td>
</tr>
<tr>
<td>(\mathcal{L}_{B_1}(B))</td>
<td>A set of all immediate successor basic blocks of (B).</td>
</tr>
<tr>
<td>(\mathcal{L}_{\mathcal{B}}(\mathcal{F}))</td>
<td>A set of all exit basic blocks in the target function (\mathcal{F}).</td>
</tr>
<tr>
<td>(\mathcal{L}_{\mathcal{F}}(\mathcal{B}))</td>
<td>A set of all immutable base addresses in (\mathcal{F}).</td>
</tr>
<tr>
<td>(\mathcal{L}_{\phi}(B,b))</td>
<td>A two-level nested hashmap with first key a basic block, second key a base address mapped to a phi node.</td>
</tr>
<tr>
<td>(\mathcal{M}_{\phi})</td>
<td>A two-level nested hashmap with first key a basic block, second key a base address mapped to a GEP instruction.</td>
</tr>
</tbody>
</table>

**TABLE II:** Glossary of terms referred in this paper.

**A. Error Model**

We consider an error model where soft errors induce a single-bit fault affecting CPU register files and ALU operations. We assume that memory elements such as data cache and DRAM are error-free because they are usually protected using ECC mechanisms. We implement our error model by targeting runtime instances of LLVM IR level instructions of a target function for fault injection. For example, if there are \(N\) dynamic IR-level instructions observed corresponding to a target function, then we choose one out of \(N\) dynamic instructions with a uniform random probability of \(\frac{1}{N}\) and flip the value of a randomly chosen bit of the destination virtual register, i.e., the left-hand side of the randomly chosen dynamic instruction. Similar error models have been proposed in the past for various resilience studies and it provides a reasonable estimate of application-level resiliency of an application [16], [23]. Given that our focus is to study soft errors affecting structured address computation, we consider all fault sites which when subjected to a random single-bit bit-flip may affect the output of one or more GEP instructions of a target program. Specifically, we propose two following error models which mainly differ in the dynamic fault site selection strategy.

1) **Error Model I:** As described in Sec. §III, error scenario affecting structured addressed computations are broadly categorized into soft errors affecting index values and the final output of GEP instructions. Error model I considers the scenario where index values are corruption-free but the final output of one of the GEP instruction has a random single-bit corruption. This is done by randomly choosing from dynamic
instances of all GEP instructions of a target function and injecting a bit-flip in the final address computed the GEP instruction.

2) Error Model II: Error model II considers the case where the index value of one of the dynamic instances of GEP instructions are corrupted including the dynamic fault sites corresponding to the set of def-use leading to the index-value.

The above two error models are implemented using an open-source and publicly available fault injector tool VULFI [24], [25]. Also, note that in our error models, we do not target base addresses as these are small in numbers (one per array) and can be easily protected through replication without incurring severe performance or space overhead.

B. PRESAGE Transformations

We refer to two or more GEP instructions as same-class GEPs if they use the same base address. PRESAGE creates a dependency chain between same-class GEPs in a two-stage process.

1) Inter-Block Dependency Chains: The first stage involves enabling dependency chains between same-class GEPs in different basic blocks. Intuitively, it would require first GEP, for a given base address, appearing in all basic blocks be transformed in a manner such that it uses the address computed by the last same-class GEP in its predecessor basic block as the relative base. However, we need a bit more careful analysis as a basic block may have more than one predecessor basic blocks. Moreover, it might be possible that not all predecessor blocks have a same-class GEP or a predecessor block might be a back edge (i.e., there is a loop enclosing the basic block and its predecessor basic block). Therefore, we propose a three-step process for linking same-class GEPs in different basic blocks as explained by Figs. 6 and 7.

1: procedure UPDATEINTERBLKDEPC(N(E,F,M,G,P))
2: for all B in BFS(F) do
3: for all B in L_b(F) do
4: s ← ¬HasGEP(B_p,b,M_G)
5: s ← ¬HasPHI(B_p,b,M_G)
6: s ← s ∧ ¬IsBackEdge(B_p,B)
7: s ← s ∧ ¬HasPHI(B_p,b,M_G)
8: s ← s ∧ ¬IsBackEdge(B_p,B)
9: if s ∧ ¬IsBackEdge(B_p,B) then
10: f ← GetPHINode(B_p,b,M_G)
11: f_p ← GetPHINode(B_p,b,M_G)
12: SetIncomingEdge(B_p,B,f,f_p)
13: end if
14: end for
15: end for
16: end procedure

Fig. 6: Creating Inter-Block Dependency Chains

As a first step, as shown in Fig. 6, we iterate over all basic blocks of a target function F in a breadth-first order. In a given basic block B with an incoming edge count e, we insert a phi node for each unique base address appearing in L_b(F) for selecting a value from same-class incoming GEP values (each belonging to a unique predecessor basic block). For a given base address b, the respective phi node entry is used as the relative base by the first GEP (with base b) in the current basic block B. In case, B does not have a valid GEP entry for b, then we call B a pass-through basic block with respect to b. In this case, we simply pass the phi node value to the successor basic blocks.

We use a phi node because all PRESAGE transformations are applied at LLVM IR and LLVM uses the single static assignment (SSA) form thus requiring a phi node to select a value from one or more incoming values. For each phi node entry created in B, if valid incoming GEP values are available from one or more predecessor basic blocks, the phi node is updated with those values by calling the SetIncomingEdge routine.

1: procedure UPDATEINTERBLKDEPC(N(E,F,M,G,P))
2: for all B in BFS(F) do
3: for all B in L_b(F) do
4: s ← ¬HasGEP(B_p,b,M_G)
5: s ← ¬HasPHI(B_p,b,M_G)
6: s ← s ∧ ¬IsBackEdge(B_p,B)
7: s ← s ∧ ¬HasPHI(B_p,b,M_G)
8: s ← s ∧ ¬IsBackEdge(B_p,B)
9: if s ∧ ¬IsBackEdge(B_p,B) then
10: f ← GetPHINode(B_p,b,M_G)
11: f_p ← GetPHINode(B_p,b,M_G)
12: SetIncomingEdge(B_p,B,f,f_p)
13: end if
14: end for
15: end for
16: end procedure

Fig. 7: Updating Inter-Block Dependency Chains

At this point, we already have created phi node entries in each basic block (including all pass-through basic blocks), and have populated these phi nodes with incoming GEP values wherever applicable. As the next step, as shown in Fig. 7, for a basic block B with each of its pass-through predecessor basic block with respect to a base address b, the respective phi node Φ is updated with the predecessor's phi node entry Φ_p by calling SetIncomingEdge routine. If a back-edge exists from a pass-through predecessor basic block B_p to B (i.e., there is exists a loop enclosing B and B_p) then B may receive invalid data from B_p, as B_p is also successor basic block of B. Therefore, we invoke the procedure UpdateInterBlkDepChain in Fig. 7 twice. In the first pass, the phi node entries of all pass-through predecessor basic blocks of B which do not have back edges to B, are assigned to the respective phi node entries in B. In the second pass, we repeat the steps of the first pass with the exception that this time we select the phi node entries of all pass-through predecessor basic blocks of B which do have back edges to B.
1: procedure CreateIntraBlkDepChain(\( F, M_G, M_\phi \))
2: for all \( B \) in \( \text{BFS}(F) \) do
3: for all \( b \) in \( L_b(F) \) do
4: for all \( G \) in \( L_G(B, b) \) do
5: if IsFirstGEP(\( G \)) then
6: \( \phi \) ← GetPHINode(\( B,b,M_\phi \))
7: \( b_e\) ← GetRelativeBase(\( \phi, b \))
8: \( pid\) ← GetPrevIdx(\( \phi \))
9: end if
10: \( id\) ← GetCurrentIdx(\( G \))
11: \( rid\) ← GetRelativeIdx(\( id,pid \))
12: \( G_n\) ← CreateNewGEP(\( \gamma, rid \))
13: \( pid\) ← \( id\)
14: InsertGEP(\( G_n,G \))
15: ReplaceAllUses(\( G_{n} \), \( G \))
16: DeleteGEP(\( G \))
17: end for
18: end for
19: end for
20: end procedure

Fig. 8: Creating Intra-Block Dependency Chains

2) Intra-Block Dependency Chains: The second stage involves creating intra-block dependency chains. As shown in Fig. 8, for each basic block \( B \) of a target function \( F \) and for each unique base address \( b \in L_b(B) \), if there exist one or more same-class GEP instructions which use \( b \) as the base, we need to transform these GEPs to create a dependency chain. In other words, each GEP uses the value computed by the previous GEP as the relative base using our RBA scheme. For the first occurrence of GEP instruction in \( B \) with base \( b \), we extract the relative base information using the phi node entry \( \phi \) created in the previous stage. At runtime, the phi node \( \phi \) will receive the last address computed using the base address \( b \) from one of the predecessor basic blocks of \( B \). In summary, for each GEP instruction \( G \), an equivalent version \( G_n \) is created using the relative base and the relative index values. All uses of \( G \) are then replaced by \( G_n \) and \( G \) is then finally deleted.

C. Detector Design

The error detectors are designed to protect against single-bit faults injected using error model I. As shown in Fig. 9, in each exit basic block \( B_e \), for each unique base address \( b \), PRESAGE makes available the value computed of the last run GEP instruction with base \( b \) and the relative index value used. Additionally, PRESAGE also makes available the absolute index value which along with the base address \( b \) can also be used to reproduce the output of the last run GEP instruction with base \( b \). The error detectors then simply check if the output \( G \) produced by the last run GEP instruction matches the recomputed value \( G_d \) using the base address \( b \) and the absolute index value. Given that in error model I, we consider the base address and index value to be corruption-free, the error detectors are precise with respect to error model I as they do not report any false positives.

1: procedure InsertDetectors(\( F, M_G, M_\phi \))
2: for all \( b_e \in L_{B_e}(F) \) do
3: for all \( b \) in \( L_b \) do
4: \( \phi \) ← GetPHINode(\( B,b,M_\phi \))
5: \( b_e\) ← GetRelativeBase(\( \phi, b \))
6: \( rid\) ← GetRelativeIdx(\( \phi \))
7: \( pid\) ← GetPrevIdx(\( \phi \))
8: \( G\) ← CreateNewGEP(\( b_e, rid \))
9: \( G_d\) ← CreateNewGEP(\( b, pid\))
10: InsertEqvCheck(\( G,G_d \))
11: end for
12: end for
13: end procedure

Fig. 9: Algorithm for Error Detectors

Fig. 10 shows the LLVM-level control-flow graph (CFG) of the function \( \text{foo1} \) presented in Sec. §III. Similarly, Fig. 11 shows the LLVM-level CFG of the PRESAGE transformed version of the function \( \text{foo1} \). The GEP instruction in function \( \text{foo1} \) (Fig. 10) which stores the computed address in register \%13 is replaced by a new GEP instruction (Fig. 11) in the PRESAGE transformed version of \( \text{foo1} \) which uses relative base and relative index value for address computation. The PRESAGE transformed version of \( \text{foo1} \) in Fig. 10 also has error detector code inserted in the exit basic block. Specifically, \%GEP_duplct represents the recomputed version of the address which is compared against the observed address value \%GEP_obsrvd. In case of a mismatch, the global variable @detectCounter is set to report error detection to the end user.

V. EXPERIMENTAL RESULTS

A. Evaluation Strategy

Our evaluation strategy involves measuring the effectiveness of the proposed error detectors in terms of SDC detection rate and performance overhead. In addition, we analyze the impact of PRESAGE transformations on an application’s resiliency using a fault injection driven study. We consider 10 benchmarks (listed in Table IV) drawn from the PolyBench/C benchmark suite [26]. These benchmarks represent a diverse set of applications from areas such as stencils, algebraic kernels, solvers, and BLAS routines. For each of these benchmarks, we perform four set of experiments, summarized in Table III. Each experiment set involves a fault injection campaign (FIC), consisting of 5000 independent experimental runs. In each experimental run, we carry out a fault-free and a faulty execution of a target benchmark using identical program input parameters and compare the outcome of the two executions. The program input parameters (such as array size used in the benchmark) are randomly chosen from a predefined range of values. During a fault-free execution, no faults are injected whereas during a faulty execution, a single-bit fault is injected in a dynamic LLVM IR instruction selected randomly using either error model I or error model II as explained in
Fig. 10: LLVM IR level CFG representation of the function foo

Experiment Set | Description
---|---
Native_FIC_EM-I | A fault-injection campaign (FIC) using error model I on the native version of a target benchmark.
Native_FIC_EM-II | Same as Native_FIC_EM-I except that error model II is used.
Presage_FIC_EM-I | A fault-injection campaign (FIC) using error model I on benchmarks transformed using PRESAGE.
Presage_FIC_EM-II | Same as Presage_FIC_EM-I except that error model II is used.

TABLE III: Summary of experiments

Sec. §IV. Note that we only target the key function(s) that implement the core logic of a benchmark for fault injection. For example, in the jacobi-2d benchmark, we only target the kernel_jacobi_2d which implements the core jacobi kernel and ignore the other auxiliary functions such as the function used for array initialization or the program’s main().

Given that the benchmarks chosen produce one or more result arrays as the final program output, we compare respective elements of the result arrays produced by the faulty and fault-free executions to categorize the outcome of the experimental run as:

SDC: The executions ran to completion, but the corresponding elements of the result arrays of the fault-free and faulty execution are not equivalent.

Benign: The corresponding elements of the result arrays of the fault-free and faulty execution are equivalent.

Program Crash: The program crashes or terminates prematurely without producing the final output.

We analyze the impact of PRESAGE transformations on an application’s resiliency by comparing the outcomes of the experiment sets Native_FIC_EM-I with Presage_FIC_EM-I, and Native_FIC_EM-II with Presage_FIC_EM-II.

B. Fault Injection Campaigns

Fig.12 shows the result of FIC done under each experiment set listed in Table III. Each column in the figure represents an FIC consisting of 5000 runs. Therefore, the total number of fault injections done across 10 benchmarks and 4 experiment sets stands at 0.2 million (4 experiment sets × 10 benchmarks × 5000 fault injections).

Non-trivial SDC Rates: The results for experiment sets Native_FIC_EM-I and Native_FIC_EM-II shown in Fig.12 demonstrate that non-trivial SDC rates are observed when structured address computations are subjected to bit flips. Specifically, for the experiment set Native_FIC_EM-I, we observe a maximum and a minimum SDC rates of 32.2% and 18.5%, for the benchmarks trmm and bicg, respectively. In case of Native_FIC_EM-II, we observe a greater contrast, with a maximum SDC rate of 43.6% and a minimum SDC rate of 2.3% for the benchmarks trmm and adi, respectively.

Promotion of SDCs to Program Crashes: When comparing the results of experiment sets Presage_FIC_EM-I and Presage_FIC_EM-II with that of Native_FIC_EM-I and Native_FIC_EM-II, we observe that PRESAGE transformations lead to a sizable fraction of SDCs getting promoted to program crashes. Specifically, Presage_FIC_EM-I reports an average increase of 12.5% (averaged across all 10 benchmarks) in the number of program crashes when compared to Native_FIC_EM-I, with a maximum increase of 19.3% reported
for the cholesky benchmark. Similarly, Presage_FIC_EM-II reports an average increase of 7.8% (averaged across all 10 benchmarks) in the number of program crashes when compared to Native_FIC_EM-II with a maximum increase of 16.8% reported for the jacobi-2d benchmark.

### C. Detection Rate & Performance Overhead

Fig. 13: SDC detection rate & performance overhead

Fig. 13 shows the percentage of SDCs reported in Fig. 12 under Presage_FIC_EM-I that are detected by the PRESAGE-inserted error detectors. Except for the benchmark fdt-d-2d, we are able to detect 100% of the SDCs caused by a random bit-flip injected using error model I. In case of fdt-d-2d, we are able to detect only 74% of the reported SDCs because a fraction of GEP instructions in fdt-d-2d have mutable base addresses. Recall that the PRESAGE transformations can only be applied to GEP instructions with immutable base addresses.

For the benchmarks adi, seidel-2d, gesummv, bicg, and cholesky, we notice that the error detectors incur almost negligible overheads ranging between 0.3% and 3.2%. Benchmarks lu and atax report overhead figures of less than 20% whereas the benchmarks jacobi-2d, fdt-d-2d, trmm, and atax report overhead figures of close to 40%.

### D. False Positives & False Negatives

We refer to the errors flagged during the execution of a PRESAGE-transformed program as a false positive when no faults are injected during the execution. Conversely, if there are no errors reported during the execution of a PRESAGE-transformed program while an error is actually injected during the execution, we regard it as a false negative. The basic philosophy of the PRESAGE detectors is to recompute the final address observed at the end point of a dependency chain and compare the recomputed address against the observed final address. Also, in error model I, the index and the base address of a GEP instruction are assumed to be corruption-free but the final address computed by it can be erroneous. Therefore, under error model I, whenever the recomputed address does not match the observed address, it attributes it to an actual bit-flip. In summary, the detectors never report false positives under error model I. Even in the case of error model II, where we subject the index value of a GEP instruction to a bit-flip, the value recomputed by the detectors would use the same corrupted index value to reproduce the same corrupted observed value. Thus even under error model II, the error detectors must not report false positive. However, it may report false negatives, including in cases where we inject bit flips into GEP instructions that have mutable base addresses, as in the case of fdt-d-2d benchmark.

### E. Coverage Analysis

Table IV provides an insight into the kind of coverage provided by the PRESAGE-based error detectors. Total SIC denotes the total static instruction count of the LLVM IR instructions corresponding to key function(s) of a benchmark that are targeted for fault injections. SIC-I and SIC-II represents the subset of instructions represented by SIC chosen using error model I and error model II respectively. Clearly, SIC-I and SIC-II represent the subset of instructions occupied by SIC using error model I and error model II respectively. The share of SIC-I ranging between 15.5% and 28.5% whereas that of SIC-II ranging between 63.3% and 21.7%. The ratio between SIC-I and SIC-II roughly varies from 1:3 (in case of seidel-2d) to 1:1 (in case of gesummv and bicg). Avg. DIC-I is a counterpart of SIC-I, representing the average
TABLE IV: Benchmark description

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Avg. DIC-I (in millions)</th>
<th>Avg. DIC-II (in millions)</th>
<th>SIC-I</th>
<th>SIC-II</th>
<th>Total SIC</th>
<th>%SI-I</th>
<th>%SI-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>adi</td>
<td>59.2</td>
<td>157.5</td>
<td>30</td>
<td>69</td>
<td>161</td>
<td>18.6%</td>
<td>42.8%</td>
</tr>
<tr>
<td>fdtl-2d</td>
<td>63.7</td>
<td>24.8</td>
<td>68</td>
<td>98</td>
<td>249</td>
<td>27.3%</td>
<td>39.3%</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>74.8</td>
<td>36.8</td>
<td>42</td>
<td>114</td>
<td>180</td>
<td>23.3%</td>
<td>63.3%</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>64.2</td>
<td>97.1</td>
<td>56</td>
<td>112</td>
<td>196</td>
<td>28.5%</td>
<td>57.1%</td>
</tr>
<tr>
<td>gesumnv</td>
<td>0.4</td>
<td>0.7</td>
<td>5</td>
<td>5</td>
<td>22</td>
<td>22.7%</td>
<td>22.7%</td>
</tr>
<tr>
<td>trmm</td>
<td>39.1</td>
<td>107.1</td>
<td>14</td>
<td>39</td>
<td>90</td>
<td>15.5%</td>
<td>43.3%</td>
</tr>
<tr>
<td>atax</td>
<td>0.5</td>
<td>0.7</td>
<td>22</td>
<td>26</td>
<td>91</td>
<td>24.1%</td>
<td>28.5%</td>
</tr>
<tr>
<td>bicc</td>
<td>0.4</td>
<td>0.7</td>
<td>5</td>
<td>5</td>
<td>23</td>
<td>21.7%</td>
<td>21.7%</td>
</tr>
<tr>
<td>cholesky</td>
<td>0.3</td>
<td>0.8</td>
<td>16</td>
<td>39</td>
<td>89</td>
<td>17.9%</td>
<td>43.8%</td>
</tr>
<tr>
<td>lu</td>
<td>0.6</td>
<td>1.9</td>
<td>15</td>
<td>35</td>
<td>77</td>
<td>19.4%</td>
<td>45.4%</td>
</tr>
</tbody>
</table>

VI. CONCLUSIONS & FUTURE WORK

Researchers in the HPC community have highlighted the growing need for developing cross-layer resilience solutions with application-level techniques gaining a prominent place due to their inherent flexibility. Developing efficient lightweight error detectors has been a central theme of application-level resilience research dealing with silent data corruption. Through this work, we argue that, often, protecting *structured address computations* is important due to their vulnerability to bit flips, resulting in non-trivial SDC rates. We experimentally support this argument by carrying out fault injection driven experiments on 10 well-known benchmarks. We witness SDC rates ranging between 18.5% and 43.6% when instructions in these benchmarks pertaining to *structured address computations* are subjected to bit flips.

Next, guided by the principle that maximizing the propagation of errors would make them easier to detect, we introduce a novel approach for rewriting the address computation logic used in *structured address computations*. The rewriting scheme, dubbed the RBA scheme, introduces a dependency chain in the address computation logic, enabling sufficient propagation of any error and, thus, allowing efficient placement of error detectors. Another salient feature of this scheme is that it promotes a fraction of SDCs (user-invisible) to program crashes (user-visible). One can argue that promoting SDCs to program crashes may lead to a bad user experience. However, a program crash is far better than an SDC, whose insidious nature does not raise any user alarms while silently invalidating the program output.

We have implemented our scheme as a compiler-level technique called PRESAGE developed using the LLVM compiler infrastructure. In Sec.§IV, we formally presented the key steps involved in implementing the PRESAGE transformations which include creating inter-block and intra-block dependency chains, and a lightweight detector placed strategically at all exit points of a program. We reported high detection rates ranging between 74% and 100% with the performance overhead ranging between 0.3% and 42.8% across 10 benchmarks. When faults are injected using error model I, the PRESAGE-transformed benchmarks witness an average and a maximum increase of 12.5% and 19.3%, respectively, in program crashes as compared to their original versions. These figures stands at 7.8% and 16.8%, respectively, when error model II is used instead of error model I for fault injection.

Our current work identifies some challenges we plan to
address as part of the future work. Specifically, we observe relatively higher detection overheads for some of the benchmarks, potentially due to increased register pressure and/or loss in vectorization opportunities caused due to the introduction of dependency chains as explained earlier. In the future, we plan to explore efficient ways of mining GEP instructions in a program that are best suited for PRESAGE transformations and also vectorize the dependency chains (wherever possible) to minimize the performance impact. Although, the main focus of our work is to provide coverage explicitly for error model I, we also observe that PRESAGE provides partial coverage for error model II by promoting a fraction of SDCs to program crashes. As future work, we plan to explore techniques used in the context of verification and polyhedral transformations to develop comprehensive error detection mechanisms for error model II. Finally, through this work, we hope to bring to the resilience community’s notice the importance and the need for developing efficient error detectors for protecting structured address computations.

VII. ACKNOWLEDGEMENT

This work was supported in part by the U.S. Department of Energies (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award 66905. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830. The Utah authors were supported in part under the same DOE project with award number 55800790, NSF Award CCF 1255776, and SRC Tasks 2425.001, 2426.001. Also, thanks to Vinu Joseph, Arnab Das, Ian Briggs, and Hari Sundar for their feedback.

REFERENCES