PRESAGE: Protecting Structured Address Generation against Soft Errors

Vishal Chandra Sharma, Ganesh Gopalakrishnan
School of Computing, University of Utah
Email: {vcsharma,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 (**PR**ot**E**cting **S**tructured **A**ddress **GE**neration). 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.

```
L0:  cmp      0x2,%esi
L1:  jl       L12
L2:  xor      %eax,%eax
L3:  mov      0x1,%ecx
L4:  xorps    %xmm0,%xmm0
L5:  cvtsi2sd %ecx,%xmm0
L6:  cltq
L7:  movsd    %xmm0,(%rdi,%rax,8)
L8:  add      0x2,%eax
L9:  inc      %ecx
L10: cmp      %ecx,%esi
L11: jne      L4
L12: retq
```

Fig. 1: x86 representation of the foo() function

```
L0: void foo1(double* a, unsigned n){
L1:   double* addr=a;
L2:   for(int i=1;i<n;i++){
L3:     int id=2*i-2;
L4:     addr=&a[id];
L5:     *addr=i;
      }
    }
```

Fig. 2: foo1() function

```
L0: void foo2(double *a, int n){
L1:   double* addr=a;
L2:   int pid=0;
L3:   for(int i=1;i<n;i++){
L4:     int id=2*i-2;
L5:     int rid=id-pid;
L6:     addr[rid]=i;
L7:     pid=id;
L8:     addr=&addr[rid];
      }
    }
```

Fig. 3: foo2() function

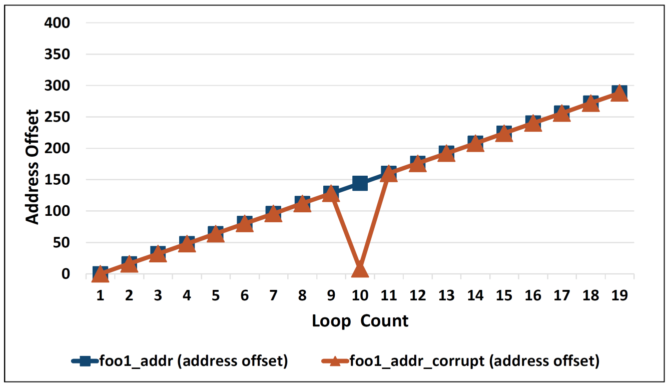


Fig. 4: Function `foo1` with no dependency-chains

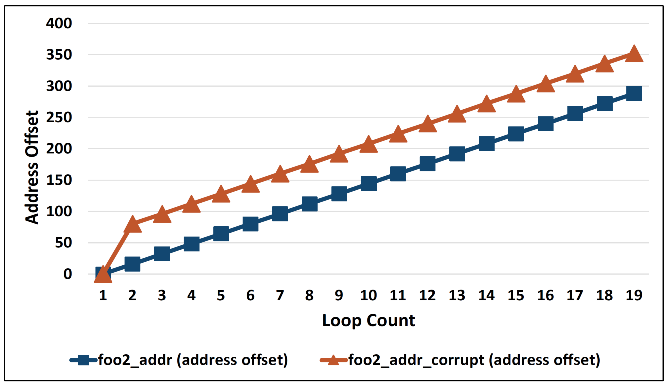


Fig. 5: Function `foo2` with a dependency-chain introduced

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.

Fault Site	Description
`i`	Loop iterator variable.
`id`	Absolute index variable.
`addr`	A variable containing an address of a location in the array `a[]`.
`rid`	Relative index variable (only present in `foo2`).

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

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

Term	Description
$\mathcal{F}$	A target function on which PRESAGE transformations are applied.
$\mathbf{b}$	A base address with at least one user in the target function.
$\mathcal{B}$	A basic block in the target function.
$E(\mathcal{B}_1, \mathcal{B}_2)$	A boolean function which returns *true* only if an edge exists from $\mathcal{B}_1$ to $\mathcal{B}_2$.
$\mathcal{L}_{\mathcal{B}_p}(\mathcal{B})$	A set of all immediate predecessor basic blocks of $\mathcal{B}$.
$\mathcal{L}_{\mathcal{B}_s}(\mathcal{B})$	A set of all immediate successor basic blocks of $\mathcal{B}$.
$\mathcal{L}_{\mathcal{B}_e}(\mathcal{F})$	A set of all exit basic blocks in the target function $\mathcal{F}$.
$\mathcal{L}_{\mathbf{b}}(\mathcal{F})$	A set of all immutable base addresses in $\mathcal{F}$.
$\mathcal{L}_{\mathcal{G}}(\mathcal{B}, \mathbf{b})$	A set of all `GEP` instructions in $\mathcal{B}$ which use the base address $\mathbf{b}$.
$\mathcal{M}_{\phi}$	A two-level nested hashmap with first key a basic block, second key a base address mapped to a `phi` node.
$\mathcal{M}_{\mathcal{G}}$	A two-level nested hashmap with first key a basic block, second key a base address mapped to a `GEP` instruction.

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 CREATEINTERBLKDEPCHAIN($\mathcal{F}$,$\mathcal{M}_G$,$\mathcal{M}_\phi$)
2:     for all $\mathcal{B}$ in BFS($\mathcal{F}$) do
3:         $e \leftarrow$ GetIncomingEdgeCount($\mathcal{B}$)
4:         for all $\mathbf{b}$ in $\mathcal{L}_\mathbf{b}(\mathcal{F})$ do
5:             $\phi \leftarrow$ CreateEmptyPHINode($\mathbf{b}$,$e$)
6:             InsertPHINodeEntry($\mathcal{B}$,$\mathbf{b}$,$\phi$,$\mathcal{M}_\phi$)
7:             for all $\mathcal{B}_p$ in $\mathcal{L}_{\mathcal{B}_p}(\mathcal{B})$ do
8:                 if HasGEP($\mathcal{B}_p$,$\mathbf{b}$,$\mathcal{M}_G$) then
9:                     $\mathcal{G} \leftarrow$ GetGEP($\mathcal{B}_p$,$\mathbf{b}$,$\mathcal{M}_G$)
10:                    SetIncomingEdge($\mathcal{B}_p$,$\mathcal{B}$,$\phi$,$\mathcal{G}$)
11:                end if
12:            end for
13:        end for
14:    end for
15: 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 $\mathcal{F}$ in a breadth-first order. In a given basic block $\mathcal{B}$ with an incoming edge count e, we insert a `phi` node for each unique base address appearing in $\mathcal{L}_\mathbf{b}(\mathcal{F})$ for selecting a value from *same-class* incoming `GEP` values (each belonging to a unique predecessor basic block). For a given base address $\mathbf{b}$, the respective `phi` node entry is used as the *relative base* by the first `GEP` (with base $\mathbf{b}$) in the current basic block $\mathcal{B}$. In case, $\mathcal{B}$ does not have a valid `GEP` entry for $\mathbf{b}$, then we call $\mathcal{B}$ a *pass-through* basic block with respect to $\mathbf{b}$. 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 $\mathcal{B}$, 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 UPDATEINTERBLKDEPCHAIN($\mathcal{F}$,$\mathcal{M}_\phi$,$\mathcal{M}_\mathcal{G}$,$P$)
2:     for all $\mathcal{B}$ in BFS($\mathcal{F}$) do
3:         for all $\mathcal{B}_p$ in $\mathcal{L}_{\mathcal{B}_p}(\mathcal{B})$ do
4:             for all $\mathbf{b}$ in $\mathcal{L}_\mathbf{b}(\mathcal{F})$ do
5:                 $s \leftarrow \neg$HasGEP($\mathcal{B}_p$,$\mathbf{b}$,$\mathcal{M}_G$)
6:                 $s \leftarrow s \wedge$ HasPHI($\mathcal{B}_p$,$\mathbf{b}$,$\mathcal{M}_\phi$)
7:                 $s_1 \leftarrow s \wedge \neg$IsBackEdge($\mathcal{B}_p$,$\mathcal{B}$)
8:                 $s_1 \leftarrow s_1 \wedge (P = Pass1)$
9:                 $s_2 \leftarrow s \wedge$ IsBackEdge($\mathcal{B}_p$,$\mathcal{B}$)
10:                $s_2 \leftarrow s_2 \wedge (P = Pass2)$
11:                if $s_1 \vee s_2$ then
12:                    $\phi \leftarrow$ GetPHINode($\mathcal{B}$,$\mathbf{b}$,$\mathcal{M}_\phi$)
13:                    $\phi_p \leftarrow$ GetPHINode($\mathcal{B}_p$,$\mathbf{b}$,$\mathcal{M}_\phi$)
14:                    SetIncomingEdge($\mathcal{B}_p$,$\mathcal{B}$,$\phi$,$\phi_p$)
15:                end if
16:            end for
17:        end for
18:    end for
19: 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 $\mathcal{B}$ with each of its *pass-through* predecessor basic block with respect to a base address $\mathbf{b}$, 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 $\mathcal{B}_p$ to $\mathcal{B}$ (i.e., there is exists a loop enclosing $\mathcal{B}$ and $\mathcal{B}_p$) then $\mathcal{B}$ may receive invalid data from $\mathcal{B}_p$ as $\mathcal{B}_p$ is also successor basic block of $\mathcal{B}$. 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 $\mathcal{B}$ which do not have back edges to $\mathcal{B}$, are assigned to the respective `phi` node entries in $\mathcal{B}$. 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 $\mathcal{B}$ which do have back edges to $\mathcal{B}$.

```
procedure CreateIntraBlkDepChain($\mathcal{F}$,$\mathcal{M}_G$,$\mathcal{M}_\phi$)
    for all $\mathcal{B}$ in BFS($\mathcal{F}$) do
        for all b in $\mathcal{L}_\mathbf{b}(\mathcal{F})$ do
            for all $\mathcal{G}$ in $\mathcal{L}_\mathcal{G}(\mathcal{B},\mathbf{b})$ do
                if IsFirstGEP($\mathcal{G}$) then
                    $\phi \leftarrow$ GetPHINode($\mathcal{B}$,$\mathbf{b}$,$\mathcal{M}_\phi$)
                    $\mathbf{b}_r \leftarrow$ GetRelativeBase($\phi$,$\mathbf{b}$)
                    $pid \leftarrow$ GetPrevIdx($\phi$)
                end if
                $id \leftarrow$ GetCurrentIdx($\mathcal{G}$)
                $rid \leftarrow$ GetRelativeIdx($id$,$pid$)
                $\mathcal{G}_n \leftarrow$ CreateNewGEP($\gamma$,$rid$)
                $pid \leftarrow id$
                InsertGEP($\mathcal{G}_n$,$\mathcal{G}$)
                ReplaceAllUses($\mathcal{G}$,$\mathcal{G}_n$)
                DeleteGEP($\mathcal{G}$)
            end for
        end for
    end for
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 $\mathcal{B}$ of a target function $\mathcal{F}$ and for each unique base address $\mathbf{b} \in \mathcal{L}_\mathbf{b}(\mathcal{B})$, if there exist one or more *same-class* GEP instructions which use $\mathbf{b}$ 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 $\mathcal{B}$ with base $\mathbf{b}$, we extract the *relative base* information using the *phi* node entry ϕ created in the previous stage. At runtime, the phi node ϕ will receive the last address computed using the base address $\mathbf{b}$ from one of the predecessor basic blocks of $\mathcal{B}$. In summary, for each GEP instruction $\mathcal{G}$, an equivalent version $\mathcal{G}_n$ is created using the *relative base* and the *relative index* values. All uses of $\mathcal{G}$ are then replaced by $\mathcal{G}_n$ and $\mathcal{G}$ 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 $\mathcal{B}_e$, for each unique base address $\mathbf{b}$, PRESAGE makes available the value computed of the last run GEP instruction with base $\mathbf{b}$ and the *relative index* value used. Additionally, PRESAGE also makes available the *absolute index* value which along with the base address $\mathbf{b}$ can also be used to reproduce the output of the last run GEP instruction with base $\mathbf{b}$. The error detectors then simply check if the output $\mathcal{G}$ produced by the last run GEP instruction matches the recomputed value $\mathcal{G}_d$ using the base address $\mathbf{b}$ 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*.

```
procedure InsertDetectors($\mathcal{F}$,$\mathcal{M}_G$,$\mathcal{M}_\phi$)
    for all $\mathcal{B}_e$ in $\mathcal{L}_{\mathcal{B}_e}(\mathcal{F})$ do
        for all b in $\mathcal{L}_\mathbf{b}$ do
            $\phi \leftarrow$ GetPHINode($\mathcal{B}$,$\mathbf{b}$,$\mathcal{M}_\phi$)
            $\mathbf{b}_r \leftarrow$ GetRelativeBase($\phi$,$\mathbf{b}$)
            $rid \leftarrow$ GetRelativeIdx($\phi$)
            $pid \leftarrow$ GetPrevIdx($\phi$)
            $\mathcal{G} \leftarrow$ CreateNewGEP($\mathbf{b}_r$,$rid$)
            $\mathcal{G}_d \leftarrow$ CreateNewGEP($\mathbf{b}$,$pid$)
            InsertEqvCheck($\mathcal{G}$,$\mathcal{G}_d$)
        end for
    end for
end procedure
```

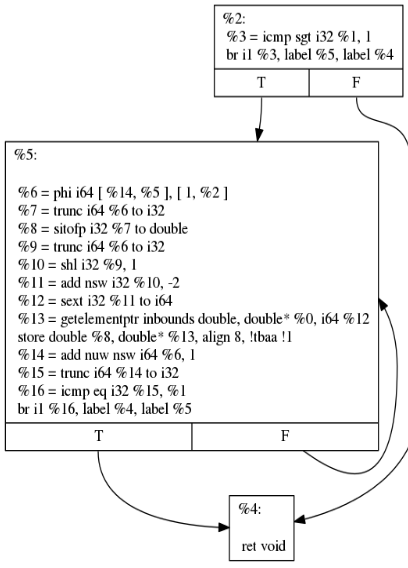
Fig. 9: Algorithm for Error Detectors

Fig. 10 shows the LLVM-level control-flow graph (CFG) of the function foo1 presented in Sec.§III. Similarly, Fig. 11 shows the LLVM-level CFG of the PRESAGE transformed version of the function foo1. The GEP instruction in function 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 foo1 which uses relative base and relative index value for address computation. The PRESAGE transformed version of 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 TableIII. 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



```
%2:
 %3 = icmp sgt i32 %1, 1
 br i1 %3, label %5, label %4
```

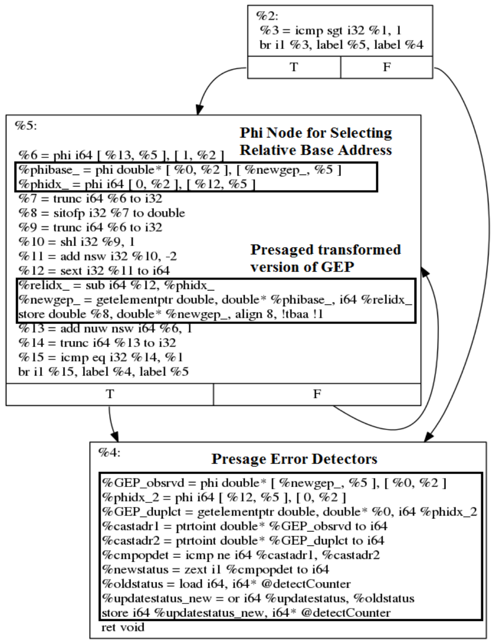
```
%5:

 %6 = phi i64 [ %14, %5 ], [ 1, %2 ]
 %7 = trunc i64 %6 to i32
 %8 = sitofp i32 %7 to double
 %9 = trunc i64 %6 to i32
 %10 = shl i32 %9, 1
 %11 = add nsw i32 %10, -2
 %12 = sext i32 %11 to i64
 %13 = getelementptr inbounds double, double* %0, i64 %12
 store double %8, double* %13, align 8, !tbaa !1
 %14 = add nuw nsw i64 %6, 1
 %15 = trunc i64 %14 to i32
 %16 = icmp eq i32 %15, %1
 br i1 %16, label %4, label %5
```

```
%4:

 ret void
```

Fig. 10: LLVM IR level CFG representation of the function `foo1`



```
%2:
 %3 = icmp sgt i32 %1, 1
 br i1 %3, label %5, label %4
```

T
F
Phi Node for Selecting Relative Base Address
Presaged transformed version of GEP

```
%5:

 %6 = phi i64 [ %13, %5 ], [ 1, %2 ]
%phibase_ = phi double* [ %0, %2 ], [ %newgep_, %5 ]
%phidx_ = phi i64 [ 0, %2 ], [ %12, %5 ]
 %7 = trunc i64 %6 to i32
 %8 = sitofp i32 %7 to double
 %9 = trunc i64 %6 to i32
 %10 = shl i32 %9, 1
 %11 = add nsw i32 %10, -2
 %12 = sext i32 %11 to i64
%relidx_ = sub i64 %12, %phidx_
%newgep_ = getelementptr double, double* %phibase_, i64 %relidx_
store double %8, double* %newgep_, align 8, !tbaa !1
 %13 = add nuw nsw i64 %6, 1
 %14 = trunc i64 %13 to i32
 %15 = icmp eq i32 %14, %1
 br i1 %15, label %4, label %5
```

```
%4:
%GEP_obsrvd = phi double* [ %newgep_, %5 ], [ %0, %2 ]
%phidx_2 = phi i64 [ %12, %5 ], [ 0, %2 ]
%GEP_duplct = getelementptr double, double* %0, i64 %phidx_2
%castadr1 = ptrtoint double* %GEP_obsrvd to i64
%castadr2 = ptrtoint double* %GEP_duplct to i64
%cmpopdet = icmp ne i64 %castadr1, %castadr2
%newstatus = zext i1 %cmpopdet to i64
%oldstatus = load i64, i64* @detectCounter
%updatestatus_new = or i64 %updatestatus, %oldstatus
store i64 %updatestatus_new, i64* @detectCounter
ret void
```

Fig. 11: LLVM IR level CFG representation of PRESAGE transformed version of the function `foo1`

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 $\times$ 10 benchmarks $\times$ 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

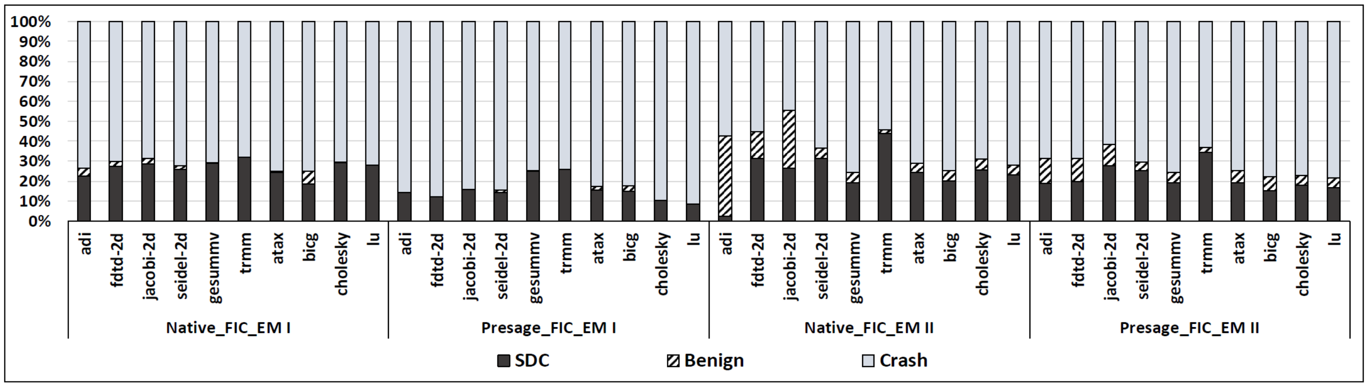


Fig. 12: Outcomes of the fault injection campaigns

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*

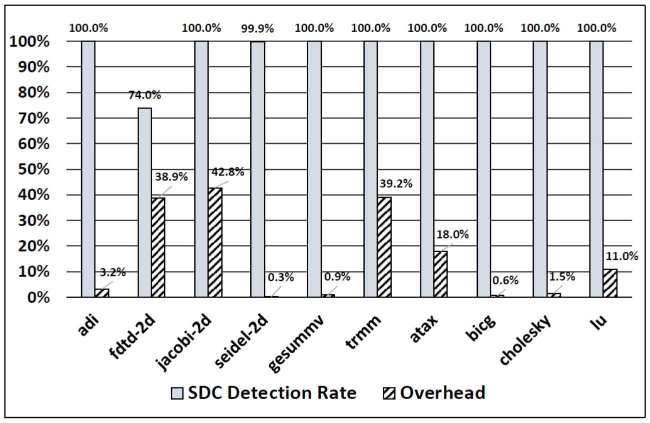


Fig. 13: SDC detection rate & performance overhead

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 `fdtd-2d`, we are able to detect 100% of the SDCs caused by a random bit-flip injected using error model I. In case of `fdtd-2d`, we are able to detect only 74% of the reported SDCs because a fraction of `GEP` instructions in `fdtd-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`, `fdtd-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 `fdtd-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 a significant portion of SIC with the share of SIC-I ranging between 15.5% and 28.5% where as 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

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

TABLE IV: Benchmark description

dynamic instruction count averaged over DIC observed during each experimental run of an FIC done under the experiment set Native_FIC_EM-I. Similarly, Avg. DIC-II denotes the average dynamic instruction count averaged over DIC observed during each experimental run of an FIC done under the experiment set Native_FIC_EM-II. Clearly, the fault sites considered under error model I and II constitute a significant part of the overall static instruction count of the benchmarks considered in our experiments.

F. Limitations of Our Approach

We have conducted preliminary investigations on the elevated overhead figures associated with some of our benchmarks. A significant portion of these overheads is attributable to the core PRESAGE transformations that introduce dependency chains. In general, such serial dependence chains can cause: (i) increased register pressure leading to register spills, and (ii) potential loss in optimization opportunities such as vectorization. Register pressure escalations can, in general, be expected due to *structured address computations* in a basic block requiring a previously computed address from one of its predecessor basic blocks. An added side effect of such dependency chains can be the elimination of vectorization opportunities.

We provide here a summary of our envisaged approaches to mitigate these limitations. One approach is to split dependency chains into shorter chains, striking a good balance between detection rates and overhead. Another approach is to create dependency chains across instruction accesses situated a certain stride apart; this has the potential to retain a sufficient amount of exposed instruction-level parallelism while also creating address calculation chains. Last but not least, it appears worthwhile to investigate how the advantages of vectorization and dependence chains can be obtained simultaneously. Our long-term research agenda is to continue to improve PRESAGE in such a way that we can realize its full potential in a predictable way on a broader array of examples than studied here.

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 Energys (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

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[2] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[3] R. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C. Cher, and P. Bose, "Understanding the propagation of transient errors in HPC applications," in *International Conference for High Performance Computing,Networking, Storage and Analysis (SC), Austin, TX, USA*, 2015.

[4] C. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[5] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, Sept 2005.

[6] J. Kim, M. Sullivan, S. Gong, and M. Erez, "Frugal ECC: efficient and versatile memory error protection through fine-grained compression," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA*, 2015, pp. 12:1–12:12.

[7] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: strong, safe, and flexible codes for reliable computer memory," in *International Symposium on High Performance Computer Architecture, HPCA, Burlingame, CA, USA*, 2015, pp. 101–112.

[8] "PolyBench/C: The polyhedral benchmark suite," http://web.cs.ucla.edu/~pouchet/software/polybench/.

[9] 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.

[10] "The LLVM Compiler Infrastructure," http://llvm.org/.

[11] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA*, 2014, pp. 375–382.

[12] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Everything you want to know about pointer-based checking," in *Summit on Advances in Programming Languages (SNAPL)*, 2015, pp. 190–208.

[13] M. Casas, B. Supinski, G. Bronevetsky, and M. Schulz, "Fault Resilience of the Algebraic Multi-Grid Solver," in *Proc. of the 2012 International Conference on Supercomputing (ICS)*. IEEE Computer Society Press, 2012, pp. 00–00.

[14] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, pp. 111–122, 2002.

[15] D. S. Khudia and S. A. Mahlke, "Low Cost Control Flow Protection Using Abstract Control Signatures," in *LCTES*, 2013, pp. 3–12.

[16] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards Formal Approaches to System Resilience," in *PRDC*, 2013.

[17] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *International Conference on Dependable Systems and Networks (DSN)*, 2013.

[18] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-sum: A novel online abft scheme for general iterative methods," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016, pp. 43–55.

[19] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-Based Metrics for Strategic Placement of Detectors," *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 75–82, 2005.

[20] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 385–396.

[21] M. a. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," *Programming Language Design and Implementation (PLDI)*, 2012.

[22] S. Sastry Hari, S. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application resiliency analyzer for transient faults," *in IEEE Micro*, vol. 33, no. 3, pp. 58–66, May 2013.

[23] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: an intermediate code-level fault injection tool for hardware faults," in *International Conference on Software Quality, Reliability and Security, QRS Vancouver, BC, Canada*, 2015, pp. 11–16.

[24] V. C. Sharma, G. Gopalakrishnan, and S. Krishnamoorthy, "Towards reseiliency evaluation of vector programs," in *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[25] "VULFI - an LLVM based fault injection framework," fv.cs.utah.edu/fmr/vulfi.

[26] "Polybench benchmark suite," https://sourceforge.net/projects/polybench.