Leveraging Code Optimizations for Overhead-free Resilience against Transient Faults

Chao Chen
Georgia Institute of Technology
Atlanta, USA
chao.chen@gatech.edu

Greg Eisenhauer
Georgia Institute of Technology
Atlanta, USA
eisen@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
Atlanta, USA
santosh.pande@cc.gatech.edu

***Abstract*—Due to the system scaling, transient faults caused by heat fluxes and particle strikes have become a growing concern for the current and upcoming extreme-scale high-performance-computing (HPC) systems. Applications running on these systems are expected to experience transient errors more frequently than ever before (as per some estimates once or twice a week on production runs of HPC systems), which will either lead them to generate incorrect outputs or cause them to crash. However, since such faults are still quite rare, desirable solutions call for low (no) overhead systems that do not compromise the performance under no-fault conditions and also allow very fast fault recovery to minimize downtime.**

In this paper, we present IterPro, a light-weight resilience technique against transient faults. IterPro exploits side effects introduced by code optimization techniques such as strength reduction and loop unrolling, which are widely adopted by modern compilers. While these code optimization techniques were mainly designed to improve the execution speed, they introduce equivalent computation patterns and values (semi-redundancies) into the code, providing opportunities which can be exploited for resilience purposes. IterPro leverages these introduced semi-redundancies to repair the crashes caused by transient faults on-the-fly with negligible (if not zero) runtime overheads. Specifically, the crashed process can continue the execution in a normal manner instead of being simply terminated and restarted. We evaluated IterPro with 2 proxy applications (miniApps) derived from production HPC applications codes and 8 benchmarks from the NPB Benchmark suite. Our results show that IterPro can recover from up to 97.60% (82.67% on average) crashes. As compared to the prior state-of-the-art methods, this technique. improves the crash recovery rate by up to $\sim 2.91\times$, and $1.60\times$ on average. Since the IterPro runtime system doesn't reside in the normal execution paths of applications, it introduces no overhead under fault free execution.

***Index Terms*—Resiliency, Transient Fault, Soft Error, Fault Tolerance, Exa-scale Computing, Failure**

I. Introduction

As new computing architectures continue to boost system performance and energy efficiency with higher circuit density, shrinking transistor size and near-threshold voltage (NTV) operations, concern is growing in the HPC community about undesirable side-effects of these manufacturing trends, specifically transient faults caused by external noises, such as heat fluxes and high energy particles. For instance, Oliveira et al. [1] project that a hypothetical exa-scale machine built with $190,000$ cutting-edge Xeon Phi processors would experience daily transient errors even though their memory areas are protected with ECC (Error-Correcting Code) techniques. Unfortunately, it is quite difficult to mask these non-memory faults at the hardware level in a cost-efficient manner [2, 3], therefore applications running on these systems are expected to experience transient errors more frequently than ever before.

Transient faults can introduce two types of problems to scientific applications, including 1) Silent Data Corruption (SDC) which means applications can finish their executions "successfully" but with incorrect outputs, and 2) application crashes, called soft failures, due to the transient faults. Based on some observations, it is conjectured that SDCs might be occuring at the rate of 1 or 2 faults during heavy production runs at scale. Li et al. [4, 5] quantitatively classified these impacts on scientific applications by leveraging empirical fault injection experiments with a focus on transient faults manifested from the CPU logic assuming that memory is protected with ECC. They found that more than 30% of transient faults could manifest as soft failures, with an almost equivalent number of them leading to SDCs. However, while there has been significant amount of prior work on detecting and correcting SDCs [6–8], less research effort has been spent on handling soft failures, perhaps because the community takes it for granted that the standard Checkpoint/Restart (C/R) methods can provide adequate recovery. Unfortunately, while the C/R technique is quite effective for recovery from soft failures, it is also very costly in terms of lost opportunities (batch job slots), lost computation (everything since the last checkpoint) and I/O overheads (repeatedly writing checkpoint files). These costs are particularly significant for massively parallel jobs [9, 10] and also suffer from very poor scaling effects. While checkpointing has broader uses, as far as the recovery from specific faults is concerned, other alternatives exist. The aim of this work is to avoid or reduce the recovery costs by repairing a crashing application from its remaining uncorrupted state on-the-fly so that it can continue fault-free execution rather than being terminated.

A. Background

Complete protection from transient faults is prohibitively expensive, whether done in hardware or software. This has moved consideration towards "online recovery" which adds little overhead while still allowing recovery under certain circumstances. Specifically, these online recovery methods aim

to enable applications "catch" transient failures, and in some way patch their state in order to continue their execution instead of being simply terminated when facing soft failures. LetGo [11] and CARE [12] are two most related works in this direction. Although these techniques are effective and efficient for many scientific applications, they come with several constraints and limitations.

First, LetGo does not attempt to completely restore the crashing application, but instead relies on a set of heuristics to patch the corrupted state so that the computation can continue despite operating on a possibly-invalid patch. For example, if the crash was induced by a memory read at an invalid low address (potentially because of an address operand corrupted by a transient fault), LetGo substitutes a '0' as the result of that memory read and lets the application continue. While effective at getting the application to proceed to a "normal" termination, programs recovered in this manner are likely to generate incorrect results, essentially converting a soft failure into a SDC. In other words, LetGo's techniques are unsound and in general do not lead to real recovery but lead to a termination which is of little value.

On the other hand, CARE addressed this issue by attempting to replay a carefully selected set of instructions picked from just before the crash point. This replay might or might not be successful in recovering the failing application, depending upon the exact location of the fault and the context of uncorrupted state from which to recover. However, CARE would never convert a soft failure into an SDC, but would always either completely recover the application or terminate the application with the original failure. CARE specially examined how transient faults manifested into soft failures, and found that majority of (up to 98.95%) soft failures were due to invalid memory address accesses (`SIGSEGV` faults). For the segfault to occur, some corruption must occur in an address computations, e.g., array subscript computation, which lead the process to access invalid memory regions. In order to support recovering from such a fault, CARE modified the compile phase to build a specialized recovery kernel for every memory access instruction, essentially by cloning the instructions involved in calculating the address for that memory access. The set of instructions cloned for the kernel are limited to those which only feed directly into the address calculation and which do not modify any state that lives beyond that memory access. Whether a given kernel can successfully recover from the transient fault depends upon whether or not the actual transient fault occurred within or outside the live range of values needed for the replay of this set of cloned instructions. If the fault occurs within the live range, the values are available for replaying these instructions and will result in a corrected access and the process can continue with the correct semantics. If the fault occurs at program points outside of the live ranges of input parameters to the recovery kernel, the replay to reconstitute the correct memory access can not be done and thus no recovery is possible. In order to make this point more clear, refer to the example presented in Section II-A . Therefore, the odds of CARE recovering from a random fault strongly depend upon the proportion of instructions "covered" by these recovery kernels and whether the fault occurs during the live range of input values to the recovery kernel, which varies with different applications and can also be adversely impacted by code optimization.

```
1 for (i = 0; i < N; i++) {
2     sum += *(A++); // from sum += A[i];
3 }
```

Fig. 1: A sample example.

Unfortunately, based on our study (see Section II for detail), fault occurring "outside the live range of recovery kernel" case appears to be more prevalent than expected in many scientific workloads which operate on huge arrays. Inside these codes, *for* loops are used for iterating over array elements and array subscript calculations based on the induction variables are essential components for retrieving the array elements. Once induction variables are corrupted, the array subscript calculations get corrupted and processes may crash due to accesses to nonexistent array elements outside their bounds. Perhaps surprisingly, a significant portion of computations in these scientific workloads is based on the induction variables (see Section II), thus they stand a good chance of being corrupted by transient faults in ways that CARE could not recover from.

B. Contribution

This paper targets the above problem of recovery from corruptions to the induction variables, such that soft failures due to faults in induction variables can be repaired and then applications can continue their execution instead of being terminated. It is a challenging problem because induction variables typically employ in-place updates. For every update, the old value of the location (e.g., the register) holding the induction variable is overwritten by a new value. Therefore, once corrupted, it is not obvious how to recover a correct value. An intuitive way to address this problem is leveraging checkpoints of the variable itself, e.g. storing a backup of old value before every update. However, this would involve adding instructions leading to higher register demands inside application's inner loop. This will add undesirable run-time overhead to the most critical portions of these applications.

Instead, we address this problem by exploiting opportunities introduced by code optimization techniques, in particular, strength reduction and loop unrolling, that are widely deployed in modern compilers. While these techniques are of course designed to improve the code performance, they also tend to introduce some semi-redundancies into original codes. Those semi-redundancies are in the form of sets of state elements which are updated synchronously across loop iterations, a situation which allows a corruption in one of those elements to be potentially repaired by inferring its proper value from the uncorrupted value of another in that set. **IterPro** was created to exploit this observation, repairing corruptions from transient faults in one piece of the code by referencing the

uncorrupted state in other piece of codes instead of using expensive checkpoints. Designed as a compiler pass, **IterPro** locates these synchronously updated state elements by identifying semi-redundancies in codes (after codes are optimized by strength reduction and loop unrolling techniques), and constructs recovery codes by pairing and cross-referencing these equivalent computations.

For an intuitive example about how **IterPro** works, consider the simple code of Fig. 1. In this code, `A++` shares a similar computation pattern to `i++`. Thus, if a transient fault occurs during the `A++` update (corrupting A), the correct value for `A` can be simply recovered as $A = A_0 + (i - i_0)$ referencing to `i` if initial values i_0 for `i` and A_0 for `A` as well as current value i for `i` are available. Similarly, if `i` is corrupted its value could be recovered from `A`. Such patterns are not unheard of in application code, but they are very commonly introduced by optimization of, for example, multiple array accesses in a loop. Even this code pattern is not explicitly present, it can be introduced in critical regions without side effects. To this end, **IterPro** introduces two extra code transformations to reshape such code without introducing significant performance penalties, such that the introduced semi-redundancies can be explicitly exposed to the underlying recovery method.

In summary, this paper makes the following contributions:

1) Describing how to exploit the properties of modern code optimization techniques, coupled with two extra code transformations, for building a lightweight (if not zero-overhead) resilience mechanisms. To the best of our knowledge, this is the first work to examine how code optimization techniques can contribute to lightweight resilience mechanisms.
2) Demonstrating **IterPro**, a prototype based on the LLVM framework that implements these techniques for "free" soft failure recovery. It is worth to note that **IterPro** doesn't introduce any runtime overheads to the normal runs of applications.
3) Evaluating **IterPro** with 2 scientific proxy applications and 8 benchmarks from NPB benchmark suite. The results will show that **IterPro** significantly improved recovery rate by up to 2.9× (average 1.6×), as compared to our prior CARE framework.

The rest of paper is organized as follows: Section II introduces the motivation for **IterPro** and explains why it is important for many scientific applications; Section III presents the design and prototype of **IterPro**. Next, evaluation results are presented in Section IV, and the related state-of-the-art studies are discussed in Section V. Finally, we present our conclusion in Section VI.

II. Observations

IterPro is designed based on semi-redundancies introduced by code optimization techniques, including strength reduction and loop unrolling. These techniques are widely deployed in modern production compilers to improve the code execution efficiency. In this section, we first briefly introduce these techniques with a focus on how they produce semi-redundancies that can be exploited for resilience purposes with a simple example. This will be used to present the observations that motivated the design of **IterPro**.

```
1 c = 7;
2 for (i = 0; i < N; i++) {
3     y[i] = c * i;
4 }
```

(a) Original Example Code

```
1 c = 7, k = 0;
2 for (i = 0; i < N; i++) {
3     y[i] = k;
4     k = k + c;
5 }
```

(b) Transformed code using Strength Reduction

```
1 c = 7;
2 for (i = 0; i < N; i+=2) { // assume N%2 = 0
3     y[i] = c * i;
4     y[i+1] = c * (i + 1);
5 }
```

(c) Transformed code using Loop Unrolling.

Fig. 2: Semi-redundancy introduced by code optimizations

A. The Limitation of CARE

Consider the simple code segment in Figure 2a. Here, even in the lowest levels of optimization the induction variable `i` is likely stored only at a single location, and updated in place on every iteration. Suppose reference to `y[i]` causes a memory access violation which is caught by an online recovery tool like CARE. If the violation was the result of a transient fault in the computation of the actual fetch address (i.e. in the instructions involved in calculating `addr y + (i * sizeof(y[0]))`, then CARE is capable of recovering based on the uncorrupted values of `y` and `i`. But if instead the corruption occurred in `i++`, corrupting `i`, CARE would find this unrecoverable. This is because `i` is a state value that is likely updated in place in the register that holds it and there is no easy way to know (or in other words, the recovery kernel can not be replayed due to the updated value of `i` which is `i++`, the original value being lost. However, **IterPro** is capable of recovering from this fault, by extending CARE with more complex recovery routines but still maintaining CARE's near-zero overheads and ability to recover from simpler errors as well. The key is exploiting the side effects of code optimizations.

B. Strength Reduction

Strength reduction is a code transformation technique in modern compilers that replaces certain costly instructions with less expensive ones without changing programs' correctness. The classic example of strength reduction is to convert expensive multiplications into efficient additions. Although strength reduction is a global optimization, it is typically applied to computations in loops, since most of a program's execution time is typically spent in a small section of code which is

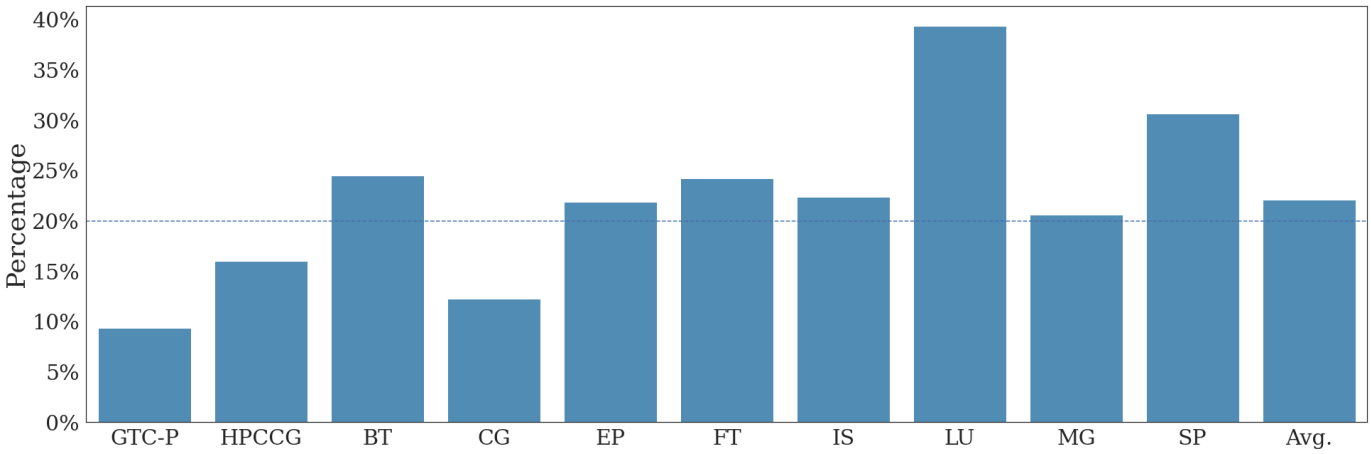

Fig. 3: The portion of computations (estimated) dedicated to updating induction variables

often inside loops that is executed over and over. (Similarly, this portion of code is also more highly likely to experience transient errors.) Strength reduction looks for expressions involving a loop invariant value (a value that doesn't change within the body of the loop) and an induction variable (a value which is changing by a known amount in each iteration of the loop). If applicable, strength reduction will transform these expressions into an equivalent but more efficient form. For illustration consider Fig. 2b which shows the transformed code after applying strength reduction on the code in Fig. 2a. As shown in the figure, the original multiplication operation `c * i` is replaced with (reduced to) a cheaper addition operation `k + c`, so the performance of the code is improved. However what's important for **IterPro** is that the introduced new expression `k + c` shares a similar computation pattern to `i++`. This provides an opportunity to recover the value of `i`, if it is corrupted, by referring to `k` as long as the initial and step values of these two variables and their updates are available. In particular, the correct value for `i` can be recomputed as `i = k / c` if `k` is in-tainted (The initial values for `i` and `k` are 0, and their step sizes are 1 and c respectively).

C. Loop Unrolling

In addition to strength reduction, loop unrolling is another compiler optimization technique that could introduce semi-redundancies to codes. The main goal of loop unrolling is to increase a program's speed by reducing (or eliminating) instructions that control the loop (such as end-of-loop tests on each iteration), reducing branch penalties, and hiding latency (e.g., the delay in reading data from memory). To eliminate these computational overheads, loop unrolling rewrites the loop as a repeated sequence of similar independent statements. Fig. 2c shows the transformed code after applying loop unrolling on the code in Fig 2a by unfolding the loop body twice. the transformation reduces the number of end-of-loop tests by almost half in the new code. Meanwhile, it also introduces two similar computing operations, including `i + 2` and `i + 1`. Although unlike the semi-redundancies introduced by the strength reduction in that both of these two instructions depend on variable `i`, they can also be exploited for resilience purposes through additional program analysis and code transformations. While the example code assumes $N\%2 = 0$, this technique can be implemented dynamically even if N is unknown at compile time.

D. Why is this important?

Modern scientific applications employ huge arrays to store computation states. The core computation of these applications is to update elements of these arrays which are typically expressed as loops. Updating loop inductions variable could contribute significant portion of computations in these applications. Fig. 3 roughly estimates the portion of induction-variable computations for several benchmarks (see Section IV for details). We statically counted the number of instructions that are involved in updating induction variables based on LLVM IR representations of applications. They were normalized to the total number of instructions inside loops of applications[1]. It surprisingly shows that, for these benchmarks, around $9\% \sim 40\%$ (average 22% in our applications) of computation is involved in induction variable updates. If this seems unintuitively large, (e.g. almost 40% of computation) recall that code optimization tends to convert regular patterns of array accesses into pointer-based induction variables and such accesses could dominate inner loops in these applications.

This result shows why these applications are particularly vulnerable to the sort of transient faults that **IterPro** is designed to recover from. Transient faults striking these instructions by their nature tend to corrupt address computations, lead the process to access incorrect array elements (leading to incorrect outputs) or invalid memory addresses (leading to process crashes). Sharma et al. [13] show that, based on single-bit-flip fault model, up to 80% of transient faults in loop induction variables would lead to process crashes. A multi-bit-flip event would be even more likely to result in a process crash. These observations motivated the design of **IterPro**.

[1]Only loops are considered is because they consume the majority of computation resources of an application

III. Design and Implementation of Dorothy

Given the observations above, **IterPro**'s focus is on recovery of induction variables. **IterPro** is designed as a compiler pass based on the LLVM framework. It works on LLVM IR, a light-weight low-level intermediate representation of programs. Based on the availability of LLVM front-ends, **IterPro** can support multiple languages, such as C/C++ (through Clang) and Fortran (through Flang or DragonEgg), in which majority of scientific applications are programmed. The design details will be presented in the rest of the section.

A. Design Principle and Challenges

The philosophy behind **IterPro** is pretty straightforward. For a given induction variable `i`, updated as `i = i + s`$_i$, **IterPro** will leverage scalar-evolution analysis to find another induction variable(s) `k` in the same code region, which is a loop, such that `k` is updated with a computation pattern (`k = k + s`$_k$) similar to `i` and `k` is not used with `i` at the same time to compute a memory address (e.g., `y[i+k]`). **IterPro** then pairs them together. And `k` is considered as a partner (or co-related induction variable) to `i`, such that if `i` is corrupted by a fault, **IterPro** will be able to recover it by referring to `k` (vice versa) based on the following equation:

$$i = \frac{k - k_0}{s_k} \times s_i + i_0 \tag{1}$$

where, i_0 and k_0 are initial values of i and k respectively.[2]

While it would very difficult to find such computation pairs in original source codes, the code optimization techniques deployed in modern compilers, such as strength reduction and loop unrolling, would introduce more opportunities in transformed codes (See section II), which are only accessible by compiler passes. To be able to successfully recover `i` when it is corrupted, **IterPro** must know or have accesses to initial values of `i` and `k` and their step sizes at runtime. In other words, when `i` is corrupted, **IterPro** should be able to: 1). find its partner `k`; 2) their initial values `i`$_0$ and `k`$_0$; 3) their step sizes `s`$_i$ and `s`$_k$; and 4) the current value of `k`. If these values are not constant numbers, **IterPro** has to make sure they are stored in somewhere (e.g., register or stack) during the code generation pass, such that they are available (the location storing them is not reused by others) regarding less when they are accessed during runtime.

Although semi-redundancies were introduced by the aforementioned code optimization techniques to the transformed codes, they might be not exploitable for resilience purposes due to following challenges:

1) No partner is exposed. In such case, even though these techniques introduced semi-redundancies, but they don't introduce new variables. An example is shown in Fig. 2c where `i + 1` shares a similar computation pattern to `i += 2`. However, it is useless to **IterPro** since they both depend on `i`. In particular, if accessing to `y[i]` failed because of a fault in `i`, there is no partner available for **IterPro** to recover it.
2) Sometimes initial values or step sizes are not available at runtime when a failure is detected as illustrated in Fig. 1. In this case, **IterPro** would be able to find the partner for `i`, which is `A`, but it may fail to find its initial value `A`$_0$. This is because the code generator typically maps `A` to a register, saying `%rax`, and updates it in-place simply with `add %rax, 8`. Therefore, the initial value for `A` is not preserved in applications' process address spaces.

[2]**IterPro** is based on CARE, and shares its approach of generating a specific recovery routine for every memory access, thus considering Figure 4a, the routine associated with the store at $y[i]$ recovers i from k, while the routine for $y[k]$ recovers k from i. These recovery routines are generated to external DLLs and loaded only on demand to avoid unnecessarily impacting application runtime.

```
1 c = 7;
2 for (i = 0, k=1; i < N; i+=2, k+=2) {
3     y[i] = c * i;
4     y[k] = c * k;
5 }
```

(a) Independent code promotion

```
1 S = A;
2 B = S;
3 for (i = 0; i < N; i++) {
4     sum += *(B++); // from sum += A[i];
5 }
```

(b) Micro-checkpoint

Fig. 4: Code Transformations in **IterPro**. C/C++ are used for illustration only. **IterPro** actually works on LLVM IR code.

In order to address these problems, **IterPro** introduces two additional code transformations, named independent compute promotion (ICP) and micro-checkpoint. For the first situation, **IterPro** leverages ICP to transform dependant computations into independent ones, if possible, by introducing new variables. And for "vanishing initial values" challenge, **IterPro** introduces code to store (checkpoint) related initial values in the stack, such that they are always available when they are needed for recovering corrupted induction variables. The following subsections present their details.

B. Independent Compute Promotion

Typically, semi-redundancies introduced by loop-unrolling exhibiting in the code in form of *derived induction values* (e.g., `i + 1` in Fig. 2c). Per discussion before, such semi-redundancies can't be directly exploited by **IterPro**, so we introduce compiler pass, ICP, which transforms these derived induction values into independent computations. It will create new induction variables along with their related update instructions to replace original derived induction values. For illustration, Fig. 4a shows the transformed code derived the code in Fig. 2c, in which a new variable `k` is created and original `i + 1` is replaced with `k` and `k + 2`. In particular, note that `k` is completely independent from `i`, therefore they can be inferred to recover each other if either one is corrupted.

Algorithm 1 The Pseudo Code for Independent Compute Promotion.

```
function doIndependencePromotion(loop)
  for every binary operator BO in loop do
    Expr ← getSCEVExpr(BO)
    isAddRec ← isSCEVAddRecExpr(Expr)
    isInAddr ← isUseInAddrCompute(BO)
    if isAddRec && isUsedInAddr then
      initVal ← getStartValue(Expr)
      stepVal ← getStepValue(Expr)
      IndPhi ← createPHINode(initVal)
      Inc ← createIncOp(IndPHI, stepVal)
      IndPhi → addIncomingValue(Inc)
      BO → replaceUsesWith(IndPhi)
    end if
  end for
end function
```

Algorithm 2 The pseudo code for micro-checkpoint

```
function doCheckpoints(loop)
  for every induction variable IV in loop do
    Latch ← getLoopLatch(loop)
    Init ← getStartValue(IV)
    Const ← isConstant(Init)
    Live ← isLiveAt(Init, Latch)
    if !Const && !Live then
      Var ← createLocalVariable
      createStore(Var, IV)
      Val ← createLoad(Var)
      IV → repalceAllUsesWith(Val)
    end if
  end for
end function
```

It is worthwhile to note that while ICP does demand an additional register, it doesn't introduce new computation. Such change is often hidden in superscalar processors. Hence, it has negligible penalties to applications' performance.

Algorithm 1 shows the core steps of independent compute promotion. For each loop in LLVM IR codes, ICP iterates over each binary operator in the loop. For those who are directly used (both directly and indirectly) in address computations, **IterPro** will create new induction variables to replace them, if they can be expressed in form of (`i = i + s`) based on scalar-evolution analysis, where `s` is a loop invariant value (it doesn't need to be a constant).

C. Micro-checkpoint

Micro-checkpoint is applied only to induction variables whose initial values are not live across the loop body. If a value is not live across the loop body, the location for holding this value could be reused by other variables at runtime, which means it could be not accessible by the recovery mechanism. For these induction variables, **IterPro** will checkpoint their initial values into the stack frame by creating new local variables and inserting a store instruction. The transformed code for the code in Fig. 1 is shown in Fig. 4b, in which a new local variable `S` is allocated to store the initial value (base address) of `A`. And a new variable `B` is introduced as an alias to `A` to iterate over elements in the array. And `B` will be identified as the partner to `i`. While `B = S` looks redundant, but it is not trivial. It provides **IterPro** heuristics about where to find initial values for `B`. Notably, the new code has substantially similar performance as the original code, since the instruction insertions are **outside** the loop body. The pseudo code for micro-checkpoint is shown in Algorithm 2. It iterates over each induction variable of a loop. If `init` is not a constant number, and it is not live (based on liveness analysis) at the end of corresponding loop, **IterPro** will then create a new local variable on the stack to store its initial value.

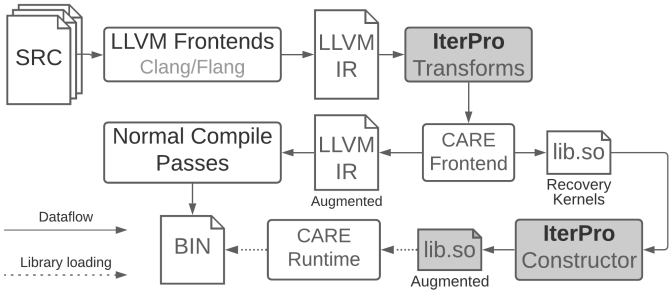

Fig. 5: **IterPro** Prototype Framework

Please note that while the C/C++ programming language were used for clarity in above examples, **IterPro** actually works on LLVM IR codes, which are an intermediate representations of applications.

D. Prototype

Using the techniques described above, we build a prototype implementation of **IterPro** on top of the CARE framework [12] as shown in Fig. 5. It reuses CARE's runtime system for performing actual recovery operations by interpreting the signal handler when a failure is detected by OS. **IterPro** augments CARE's compiler pass with the ability to construct recovery code for induction variables. For every induction variable involved in the recovery kernel for a memory access, **IterPro** will try to find its pair in the same loop, and add the necessary recovery code to the kernel.[3]

IV. Evaluation

This section presents evaluation results of **IterPro**. Generally, we are interested in the following questions: 1) What is **IterPro**'s performance in terms of failure recovery rates?; and 2) What is its runtime overheads to normal executions of applications? In the rest of the section, we will first introduce the evaluation methodology and the fault model, and then present evaluation results.

[3]In our current prototype, **IterPro** simply finds a partner for an induction variable guaranteeing that it can be repaired if it is corrupted. In future work, a multi-vote algorithm will be implemented to check whether it is actually corrupted before it is repaired to avoid potential case that the pair is corrupted

TABLE I: Scientific workloads from different scientific domains and implementing different algorithms

Workload	Language	Description
GTC-P	C	A 2D domain decomposition version of the GTC global gyrokinetic PIC code for studying micro-turbulent core transport. It solves the global, nonlinear gyrokinetic equation using the particle-in-cell method.
HPCCG	C++	A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors.
NPB	C	The NAS Parallel Benchmark (NPB) suite is a small set of programs derived from computational fluid dynamics (CFD) applications. It consists of 5 kernels and 3 pseudo-applications. In this work, NPB3.0-C version is used.

A. Setup

We evaluated **IterPro** on an X86_64 platform equipped with 48 cores and 128GB of memory. Two scientific proxy applications, GTC-P and HPCCG, as well as 8 benchmarks from the NPB benchmark suite were used in our experiments. Table I briefly presents their properties. These benchmarks are derived from production scientific applications for evaluating system performance. They contain compute-intensive kernels which typically dominate the execution of production scientific applications. Therefore, in production applications, these portions of codes are more likely to experience transient faults. All of these codes were compiled into LLVM IR codes with clang using the "-O1" flag. In addition, strength reduction and loop unrolling code optimization passes were also applied. **IterPro** works on these optimized LLVM IR codes (referred as "baseline" in the rest of the section), performing the required code transformations, building recovery kernels and generating the final binaries.

B. Fault Model and Methodology

We evaluated **IterPro** based on empirical fault injection experiments, which were widely used in prior studies [4, 5, 14, 15]. Similar to these studies, **IterPro** focuses on faults in the CPU logic, assuming memory regions are protected with other techniques, such as ECC. The tool introduced in [12] is used in this work. To emulate the the impact of transient faults from the CPU logic, it injects a fault to the "destination" operand of a randomly selected instruction right after the instruction is executed. Then the execution of the process is continued. A "destination" operand is one of architecture states, e.g. a register, or a memory cell, that is updated by the instruction. For instructions having implicit destination operand(s), such as X86 `idiv` $\%ecx$ which divides the value in $\%edx : \%ead$ by $\%ecx$ and store results in $\%eax$ and reminder in $\%edx$, one of the implied destinations, e,g, $\%eax$, is selected. For each run of an application, only one injection is performed. The single-bit-flip fault model, which is widely used in previous studies [11, 15], is used in this work, which means, for each injection, it randomly flips a bit in the destination operand.

In this work, we are particularly interested in faults that lead to process crashes and specifically how many of them can be successfully recovered by **IterPro**. We compared **IterPro** to our prior CARE framework. To ensure the fairness of comparison, the same set of faults were injected for both of these two frameworks. To achieve this, two rounds of empirical fault injection experiments were performed. In the first round, we first profiled the number of executions for each static instruction (from applications only) using the Intel Pin tool. Then we randomly select a static instruction for injection based on the numerical distribution of their executions, and also generate a random number based on the executions of the selected instruction to determine the program point at which the fault would be injected at runtime. In other words, a dynamic instruction is approximately represented by a pair (I, n), which means the fault will be injected to the instruction I after it is executed n times. For each run of an application, only one injection is performed, and for each workload, we performed 5000 injections. In all, we get $788 \sim 2791$ (depends on applications) injections that lead to process crashes. Then these injection were replayed in the second round of experiments which is performed to actually evaluate the performance of **IterPro** (as well as CARE).

TABLE II: Number of recoverable induction variables respectively in baseline and **IterPro** transformed codes.

Benchmark	# of Loops	Baseline	**IterPro**	Improvement
GTC-P	333	145	167	15.17%
HPCCG	30	38	43	13.16%
BT	177	253	277	9.49%
CG	38	8	40	500%
EP	12	0	4	BIG
FT	53	46	48	4.35%
IS	7	0	12	BIG
LU	189	340	370	8.82%
MG	81	32	64	200%
SP	316	364	474	30.22%

C. The efficiency of **IterPro** code transformations

In this subsection, we evaluate the utility of the introduced code transformations. These extensions to the normal LLVM code generation are key to the success of **IterPro** in that they significantly add to the set of faults from which **IterPro** can recover by introducing "partner" induction variables for some cases where none naturally exist, and storing away necessary initial values where they would not have been otherwise available. In other words, they introduce more "recoverable" introduction variables into codes increasing their resilience. Table II shows the impact of introduced code transformations by comparing the number of recoverable induction variables in baseline codes and in **IterPro** generated codes. As shown in the table, **IterPro**'s additional LLVM passes increased the number of recoverable induction variables by $4\% \sim 500\%$ (72.65% on average) for 8 out of 10 benchmarks. For two others (EP and IS from NPB), **IterPro**'s additional code transformations introduce a recovery opportunity for induction variables where none existed before (marked by BIG).

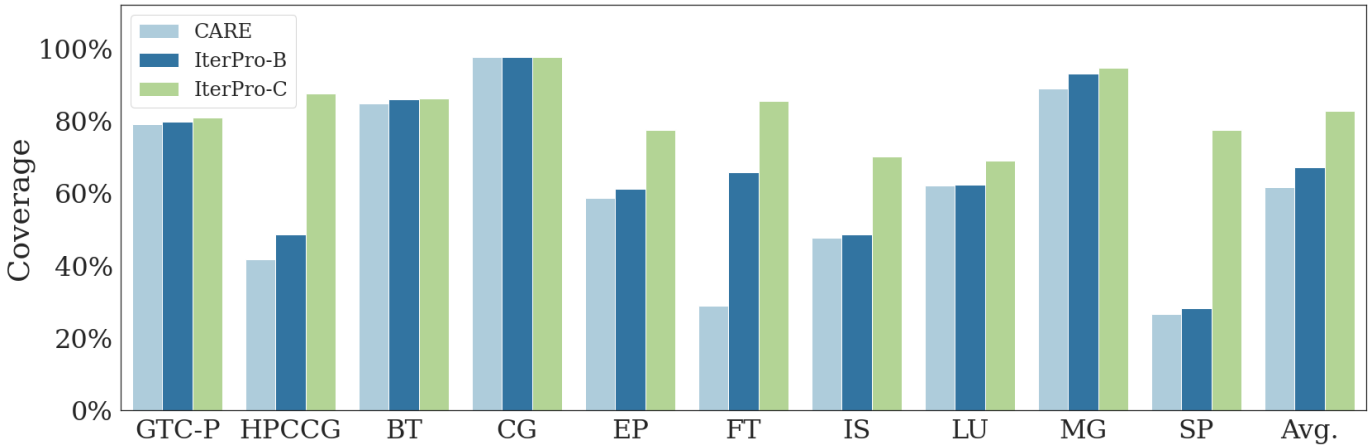

Fig. 6: Failure Recovery Rates of CARE and **IterPro**. It shows the advantage of exploiting side-effects of code optimizations and the efficiency of **IterPro** code transformations.

D. Recovery rate

In this subsection, we evaluate the performance of **IterPro** by comparing it to CARE framework. For **IterPro**, we consider two setups: 1) a fractional evaluation when **IterPro**'s enhanced code transformations are **not** applied (that is **IterPro**'s methods are applied to unaugmented LLVM-generated code), and 2) a comprehensive evaluation when **IterPro** code transformations are applied. They are respectively labeled as **IterPro**-B and **IterPro**-C. Both **IterPro**-C and CARE build recovery kernels based on the transformed codes, while **IterPro**-B works on the baseline codes. This allows us to both understand the relative contribution of **IterPro**'s induction variable recovery scheme over the simpler recovery approach of CARE, and to see the relative importance of the additional transformation passes in **IterPro** (and the attendant minor instruction additions and register reservations implied by those additional passes).

Fig. 6 presents the failure recovery rate for each considered scheme. As shown in the figure, **IterPro**-C improved recovery rate for 9 out of 10 evaluated benchmarks as compared to CARE. On average, **IterPro**-C can recover 82.67% of injected *SIGSEGV* faults, while CARE recovers 61.64% of these failures. For 3 of them, including FT, SP and HPCCG, it improved the recovery rate by more than $2\times$. On an average, it improved recovery rate by $1.6\times$ across all benchmarks. **IterPro**-C can achieve such significant improvements mainly because of its ability to recover from corruptions in induction variables, which is not available in CARE. The figure also shows the contribution of **IterPro**'s code augmentations for resilience by comparing the recovery rate of **IterPro**-B and **IterPro**-C. As shown in the figure, the average recovery rate for **IterPro**-B is 67.11%, which is about 6% higher than CARE, but around 15% lower than **IterPro**-C. As compared to **IterPro**-B, **IterPro**-C achieved significant improvements ($1.37\times$ on average) for many benchmarks. This is a significant improvement in recovery rates, but unlike **IterPro**-B, those passes do involve code generation changes, hence we must consider their impact on application performance. In the rest of the paper, **IterPro** is referred as to **IterPro**-C.

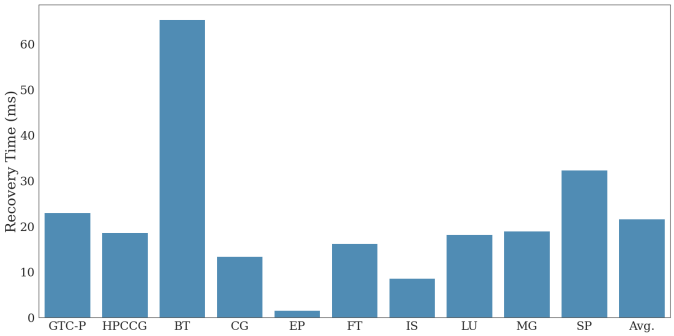

Fig. 7: Recovery time of **IterPro**

E. Recovery Time of **IterPro**

Recovery time measures the time required by **IterPro** to recover from a fault. Figure 7 shows that **IterPro** can recover a process from a *SIGSEGV* fault with only a few tens of milliseconds. As inherited from CARE, it only replays a few instructions related to address computations. It could add 4 more instructions, including `sub`, `div`, `mul` and `add`, to repair each involved induction variable used inside the kernel, but executing these instructions takes very few resources. Such a fast recovery mechanism could help to mask the impacts of soft failures on modern scientific applications.

F. Performance Penalties introduced by **IterPro**

As inherited from CARE, **IterPro** runtime system and recovery kernels don't reside in normal execution paths of the application and are actually loaded dynamically in the case of a fault. Therefore **IterPro**'s recovery mechanism has no performance interference on normal runs of applications. However, the novel LLVM passes that enhance recoverability through independent compute promotion and micro-checkpoint potentially have some minor impact. They could slightly increase register pressure and introduce more memory-to-register data

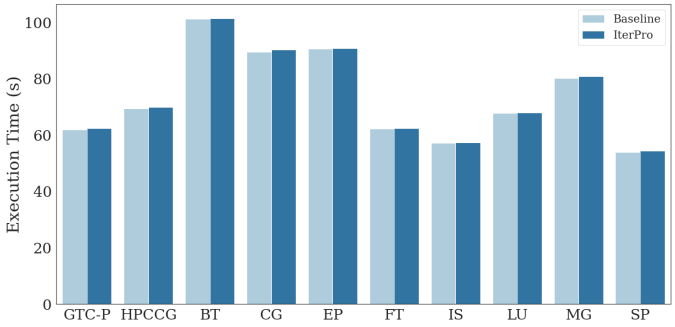

Fig. 8: Runtime overhead of **IterPro**

movements, therefore impact the binary code performance. However, these effects are likely to be negligible or non-existent depending upon exact details of code and architecture. Fig. 8 compares execution times for binaries compiled from baseline and **IterPro** transformed codes. It shows these two set of binaries almost have the same execution times (with around 0.51% differences), which implies that these effects are too small to be easily detectable on whole application runs with any of our example applications.

V. Related Work

Detection and recovery from failures are not new topics in HPC and other environments [6, 8, 11, 15, 16], In this section, we present a brief survey of prior work that is most related to **IterPro**.

Rx[17] continues the execution of applications by rolling them back to previous safe status and modifying their execution environments with minor changes. Rx is motivated by the observation that many program bugs are associated with the setup of process environments, so changing the environment setup could avoid the crashes. Its techniques could help handle transient faults by simply replaying the computation *without* changing the environment, however its basic operation requires at least partial application checkpoints which are likely to have significant costs. RCV [18] and LetGo [11] are two other online failure recovery techniques. They share a similar idea of exploring a set of heuristics for recovery. Upon a failure, they reset architecture states to a predefined value, and then continue the execution of the application. For example, they return zero as the default result of the divide for divide-by-zero errors (*SIGFPE*) and reads for invalid memory accesses (*SIGSEGV*. Writes to invalid memory address will be simply discarded. These techniques are computationally inexpensive and may succeed in getting the application to continue, Obviously such heuristic based method could lead to SDCs, which is another challenge problem in HPC community.

CARE [12] is the most related work to **IterPro**. It undertakes a proper recovery process with regards to the maligned address computation by recomputing it as per the program semantics and through the use of in-tainted values by synthesizing a very lightweight function. It develops careful correspondence mechanism to co-relate the recovery handlers to the fault causing instruction at runtime.

IterPro shares similar goal and design to the above mentioned works in that they all aim to help applications to survive failures by replacing the default signal handler with their own one to provide recovery services. **IterPro** is built on top of CARE, and extends it with the ability to recover from failures when initial values for address computation are corrupted. As inherited from CARE, **IterPro** is more accurate than other methodologies (not introduce SDCs), since it leverages untainted values to recompute the corrupted addresses. Meanwhile **IterPro** has a stronger recovery capability than CARE in that it can recover corruptions in induction variables by exploiting side effects in code optimization techniques.

VI. Conclusion and Future Work

Resilience is projected to be a critical challenge for HPC systems due to system scaling trends in higher circuit density, smaller transistor size and near-threshold voltage (NTV) operations. These technology trends would make the system more susceptible to transient faults caused by such things as high-energy particle strikes and heat flux. Transient faults could not only lead scientific applications to generate incorrect outputs, they could also crash the execution of an application, which requires the application to be restarted from a latest checkpoint, and to redo the lost computation.

In this paper, we present and evaluate **IterPro**, a lightweight and compiler-assisted soft failure recovery technique that allows processes to survive crashes caused by certain transient faults, such that the applications can continue their execution. **IterPro** exploits semi-redundancies introduced by modern compiler optimization techniques, including strength reduction and loop unrolling, for resilience purposes. Coupled with two new code transformations, **IterPro** leverages these semi-redundancies to repair soft failures caused by faults in induction variables, which has almost equivalent opportunity of experiencing transient faults as other core computations in many scientific workloads. We build a prototype of **IterPro** on top of the CARE, and evaluated it with 2 scientific proxy applications and 8 benchmarks from the NPB benchmark suites. As compared to CARE, it improved recovery rate by up to $2.9\times$ due to its capability of recovering corruptions in induction variables. In particular, **IterPro** introduces (almost) zero runtime overheads to normal runs of applications. To the best of our knowledge, **IterPro** is the first technique that explores side-effects of code optimization techniques for resilience purposes. In our future work, we plan to extend **IterPro** in the direction of generating resiliency codes against not only soft failures but also SDCs without introducing significant performance penalties.

References

[1] D. Oliveira, L. Pilla, N. DeBardeleben *et al.*, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New

York, NY, USA: ACM, 2017, pp. 28:1–28:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126960

[2] M. A. Heroux, "Toward resilient algorithms and applications," in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, ser. FTXS '13. New York, NY, USA: ACM, 2013, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/2465813.2465814

[3] V. Sridharan, N. DeBardeleben, S. Blanchard *et al.*, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 297–310. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694348

[4] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 57:1–57:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389074

[5] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of bluegene/q compute chip through hardware proton irradiation and software fault injection," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 587–596. [Online]. Available: https://doi.org/10.1109/SC.2014.53

[6] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996142

[7] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Chicago, IL, USA: IEEE, May 2016, pp. 730–739.

[8] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande, "Ladr: Low-cost application-level detector for reducing silent output corruptions," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208043

[9] J. Dongarra, P. Beckman, T. Moore *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[10] J. Elliott, K. Kharbas, D. Fiala *et al.*, "Combining partial redundancy and checkpointing for hpc," in *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ser. ICDCS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 615–626. [Online]. Available: https://doi.org/10.1109/ICDCS.2012.56

[11] B. Fang, Q. Guan, N. Debardeleben *et al.*, "Letgo: A lightweight continuous framework for hpc applications under failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: ACM, 2017, pp. 117–130. [Online]. Available: http://doi.acm.org/10.1145/3078597.3078609

[12] C. Chen, G. Eisenhauer, S. Pande, and Q. Guan, "Care: Compiler-assisted recovery from soft failures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356194

[13] V. C. Sharma, G. Gopalakrishnan, and S. Krishnamoorthy, "Presage: Protecting structured address generation against soft errors," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 252–261.

[14] R. A. Ashraf, R. Gioiosa, G. Kestor *et al.*, "Understanding the propagation of transient errors in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 72:1–72:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807670

[15] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of sdc propagation," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: ACM, 2017, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/3078597.3078617

[16] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442533

[17] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failures," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 235–248, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095809.1095833

[18] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," *SIGPLAN Not.*, vol. 49, no. 6, pp. 227–238, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594337