Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery

Qingrui Liu†, Changhee Jung*, Dongyoon Lee* and Devesh Tiwari‡
†Virginia Tech, Blacksburg, Virginia, USA
‡Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
Email: lqingrui@vt.edu, chjung@cs.vt.edu, dongyoon@vt.edu, tiwari@ornl.gov

Abstract—This paper presents Bolt, a compiler-directed soft error recovery scheme, that provides fine-grained and guaranteed recovery without excessive performance and hardware overhead. To get rid of expensive hardware support, the compiler protects the architectural inputs during their entire liveness period by safely checkpointing the last updated value in idempotent regions. To minimize the performance overhead, Bolt leverages a novel compiler analysis that eliminates those checkpoints whose value can be reconstructed by other checkpointed values without compromising the recovery guarantee. As a result, Bolt incurs only 4.7% performance overhead on average which is 57% reduction compared to the state-of-the-art scheme that requires expensive hardware support for the same recovery guarantee as Bolt.

Key words—Reliability, Checkpointing, Compiler

I. INTRODUCTION

Due to various factors including technology scaling and near-threshold operation [1], [2], [3], [4], soft error resilience has become as important as power and performance in high-performance computing (HPC) systems. Soft errors (also known as transient faults) may lead to application crashes or silent data corruption (SDC) that could result in incorrect program outputs. Thus, effective techniques for soft error resilience are indispensable for HPC systems, and in fact it is one of the key Exascale research challenges [5], [6], [7], [8], [9], [10], [11].

The general idea behind soft error recovery is that when a fault is detected, the processor takes the recovery procedure to rollback to a fault-free state and continues execution. For example, traditional periodic checkpointing, an industrial-strength recovery paradigm, periodically checkpoints processor states [12], [13], [14], [15]. Upon an error, the system triggers a rollback to a fault-free snapshot and continues execution. However, periodic checkpointing is notoriously expensive due to its coarse-grained checkpoint-interval which is the period between two neighbouring checkpoints. First, coarse-grained checkpoint-interval means that a large number of states need to be checkpointed, incurring substantial performance/area overhead. Further, the longer the checkpoint-interval is, the more executed instructions are wasted upon recovery, imposing significant recovery overhead [16].

Instead, emerging idempotence-based recovery schemes become promising alternatives due to their fine-granularity (<100 instructions) and simple recovery mechanism [17], [18], [19], [20], [21], [22], [23], [24]. The compiler partitions and transforms the entire program into idempotent regions [17], [18]. At a high-level, a region of code is idempotent if it can be re-executed multiple times and still preserves the same, correct result [17], [18]. Therefore, the program can be recovered from a soft error by simply re-executing the idempotent region where the error occurred. Figure 1 shows an idempotent region where the expected outputs are $x = 8$, $y = z = 9$. Assume an error occurs in line 2, then $y$ and $z$ can be some random numbers. However, if the error is detected within the region, the program can simply jump back to the beginning of the region and re-execute from it to recover the expected output again.

Fig. 1. Idempotent region example

However, the existing idempotence-based recovery includes the following limitations. First, prior idempotence-based recovery schemes cannot provide guaranteed recovery without expensive hardware support (Section II), which greatly undermine the benefits brought by the fine-grained recovery [17], [18], [19], [20], [21]. Second, even with the expensive hardware support, prior schemes introduce a prohibitive performance overhead due to their instrumentation/Transformation.

In light of these challenges, this paper presents Bolt, a practical compiler-directed soft error recovery scheme that provides 1) guaranteed recovery without expensive hardware support, 2) negligible performance overhead for fault-free execution, and 3) fast and fine-grained error recovery on a fault. Bolt leverages the following two key insights: First, it is still possible to achieve the guaranteed recovery by checkpointing only the necessary architectural states for idempotent region boundaries without expensive hardware support. To this end, this paper proposes eager checkpointing to preserve the value of the registers that are live-in to the regions as soon as those registers are defined, obviating expensive hardware support.

Second, there are correlations among the checkpoints created by the eager checkpointing. That is, some checkpointed value can be reconstructed by other checkpointed values, thereby being removable without compromising the recovery guarantee. This insight enables Bolt’s checkpoint pruning.
a compiler technique to achieve negligible performance overhead. Bolt explores the program dependence graph (PDG) [25] among these checkpoints and identifies the subset of the checkpoints, which is essential for soft error recovery, to minimize the performance overhead.

Following are the major contributions of this paper:

- To the best of our knowledge, Bolt is the first fine-grained soft error recovery scheme without expensive hardware support and significant performance overhead.
- Bolt can correct soft errors even if the internal structures (e.g., register file, instruction queue) are corrupted by incorrect destination write events or multi-bit flips. Note that unlike previous schemes Bolt requires no hardware protection such as ECC for these internal structures.
- Bolt incurs only 4.7% runtime overhead across a large set of applications which benefits from Bolt’s novel compiler analysis to eliminate unnecessary checkpoints.
- To better understand the performance of Bolt, we also implemented two state-of-the-art fine-grained recovery schemes that require expensive hardware support for recovery guarantee. Bolt outperforms these schemes achieving 57% and 49% runtime overhead reduction on average.

II. BACKGROUND AND CHALLENGES

A. Terminologies

This paper refers the term inputs to the variables that are live-in to a region. Such a variable has a definition that reaches the region entry and thus has the corresponding use of that definition after the region entry. For instance, the variable \( m \) is an input to the region in Figure 1. This paper also refers the term anti-dependence to a write-after-read (WAR) dependence where a variable is used and subsequently overwritten.

B. Idempotence-based Recovery

An idempotent region is a SEME (single-entry, multiple-exits) subgraph of the control flow graph (CFG) of the program. It can be freely re-executed without loss of correctness. More precisely, a region is idempotent if and only if it always generates the same output whenever the program jumps back to the region entry from any execution point within the region. To achieve this, the region inputs must not be overwritten, i.e., no anti-dependence on the inputs, during the execution of the region. Otherwise, re-executing the region can generate unexpected output since the inputs do not remain the same when the program jumps back to the beginning of the region.

In light of this, researchers propose different kinds of techniques to preserve the inputs. Any recovery schemes must preserve both the memory and register inputs with regard to the region boundary for correct recovery. Interestingly, previous techniques [17], [21] have developed simple algorithms to elegantly dismiss the overhead for preserving the memory inputs by partitioning the regions such that the memory inputs will never be overwritten in the regions (i.e., no anti-dependence to the memory input). Therefore, preserving the register inputs becomes the only source of cost.

De Kruijf et al. [17] (renamed as Idem hereafter) leverages register renaming to eliminate the anti-dependence on the register inputs, thus achieving idempotence at the expense of increasing the register pressure. Figure 2 (a) shows how Idem renames \( X \) to \( Z \) at \( S_2 \) to eliminate the anti-dependence on register \( X \) in the bottom region of the original code in Figure 2 (a). In contrast, Feng et al. [18] (renamed as Encore) preserve the register inputs by logging at the region entry only the register inputs that have anti-dependence. Figure 2 (c) shows how Encore preserves the register inputs by checkpointing only \( X \) at the region entry. Once a fault is detected during the execution of the region, Encore consults the checkpointed value to restore the inputs to the region for recovery.

C. Challenges

However, prior idempotence-based recovery schemes neither guarantee recovery without expensive hardware support, nor achieve insignificant performance overhead [17], [21], [18].

1) Lack of Recovery Guarantee: To provide guaranteed recovery, previous schemes must assume the following expensive hardware supports:

First, the contents in the internal structures (e.g., register file, instruction queue etc.) must remain intact. To illustrate, the vertical bars in Figure 2 (b) and (c) show the vulnerability windows of input \( X \) and \( Y \) for the bottom region; each time point of the window represents whether the input value in the register file (RF) is recoverable (white) or unrecoverable (black) from a fault at that time point. For example, if input \( Y \) is corrupted in the RF after defined at \( S_1 \), then both Idem and Encore fail to recover from the soft error because the re-execution starts from the corrupted states. Thus, previous schemes assume ECC protection to the RF and other internal structures which is excessively expensive in terms of power, delay, and area for low-cost systems. It is reported that ECC protection to RF can incur an order of magnitude larger power consumption [27], [28], up to 3X the delay of ALU operations [28] and 22% area overhead [29].

Second, all the writes events must write to the correct destination (e.g., in Figure 1 line 3, the value in \( y \) must be written to \( z \) instead of \( m \)). Thus, the read/write combinational logic to those internal structure must be hardened, which is exorbitant in commodity processors.
2) **Significant Performance Overhead:** Previous techniques incur significant performance overhead due to register renaming [17] or register logging [18]. We observe up to 40% performance overhead in our experiments. Taking into account that soft errors rarely occur (1/day in 16nm [30]), programmers are reluctant to use idempotence for such rare error correction at the cost of paying the high performance overhead all day.

### D. Fault Model

Except the aforementioned hardware support in the internal structures, Bolt shares the other assumptions in prior idempotent recovery schemes [17], [18], [26], [21], [19]:

1. Store queue, caches, and main memory are protected with ECC which has already existed in current commodity processors [31], [32].
2. As with branch misprediction, all the stores must be verified. They are buffered until the region reaches the end with no error detected. This is called store verification. For this purpose, gated store queue [33], [34], [35] is often used, and we evaluated its buffering overhead in our experiments (Section VI-C1).
3. PC and SP are protected as in prior schemes. However, we argue that only PC needs parity checking while in fact all other special registers can be handled by our scheme (see Section VI).
4. All the faults should be detected within the regions (see Section VI).  

### III. OVERVIEW OF BOLT

Bolt proposes two novel compiler techniques to address the above challenges and offers a practical fine-grained recovery scheme. Eager checkpointing guarantees recovered runtime without expensive hardware support. Checkpointing pruning minimizes the performance overhead by eliminating unnecessary checkpoints.

#### A. Eager Checkpointing: Guaranteed Recovery without Prohibitive Hardware Support

Bolt preserves the register inputs to the regions throughout their entire liveness period. To achieve this, Bolt eagerly checkpoints the value of register inputs to a region as soon as they are defined (Figure 2(d)). Such define-time checkpointing guarantees recovery of all the inputs to each idempotent code region. In particular, Bolt checkpoints once for each register input by tracking the last write. Even if an input is defined multiple times in one region, Bolt checkpoints only one time the value of the last definition.

**Artificial Define-Checkpoint Vulnerable Window:** One may be concerned about a vulnerable window where the register is defined and subsequently checkpointed. However, such vulnerable window is considered artificial because in eager checkpointing, the checkpoints in one region are *not* the current one. That is, even if a checkpoint is corrupted during the define-checkpoint window in a region \( r \), the checkpoint will not affect the recovery of the current region \( r \) and will be recreated (corrected) during the re-execution of \( r \) upon recovery. Thus, such a vulnerable window is implicitly eliminated in our eager checkpointing scheme. In case the checkpointing store may corrupt other memory locations, Bolt simply follows the aforementioned fault model and buffers those stores until they are verified as with branch misprediction.

#### B. Checkpoint Pruning: Minimizing Performance Overhead

The overhead of idempotent processing is proportional to the number of checkpoints executed at runtime. With that in mind, we propose a novel compiler analysis that can identify unnecessary checkpoints in those eager checkpoints based on the following insight. *For a value corrupted due to a soft error, the original value can be restored without checkpointing it, as long as it can be recomputed by leveraging other checkpoints.* Without compromising the recovery capability, Bolt formulates the problem of checkpoint pruning as that of finding a *recovery slice*, which can recompute the value of the pruned checkpoints (Section IV-C). Such a slice is similar to traditional *backward slices* [36], however, with more constraints. If the recovery slice is successfully built, Bolt removes the corresponding checkpoint. On a fault, Bolt’s recovery runtime will simply execute the slice to reconstruct the original value.

The takeaway is that checkpoint pruning enables Bolt to effectively offload the runtime overhead of fault-free execution to the fault-recovery, which is indispensable taking into account the low soft error rate(1 error/day [30]).

#### C. Fault Recovery Model

Once a soft error is detected during the execution of the region, Bolt’s runtime system first discards the buffered stores in the faulty region. Then, it takes the control to execute a *recovery block* that restores all the inputs to the faulty region, i.e., the same live-in registers as they were at the beginning of the region before the fault occurred. Bolt can generate the recovery block either statically or dynamically (Section V-B).

Lastly, Bolt redirects the program control to the entry of the faulty region and re-starts from it. Figure 3 describes such a recovery model.

**Motivating Example:** Figure 4 shows how Bolt works as a whole for the `byte_reverse` function of sha in MiBench [37]. (a) shows the code snippet of the `byte_reverse` function. (b) illustrates the control flow graph divided into idempotent regions \( (R_{00}, R_{01}) \) by using an adapted region partitioning algorithm based on Idem [17] where dashed lines show the region boundaries. (c) shows Bolt’s eager checkpointing to provide guaranteed recovery where the register inputs to \( R_{00} \) are \( R_{00} \sim R_{07} \). (d) minimizes the performance overhead with
checkpoint pruning where (e) is the resultant recovery block for $R_{g0}$. For example, the value of $R_7$ can be reconstructed by executing the recovery block that consults the checkpointed values of $R_0$ and $R_2$. In this example, Bolt can achieve fine-grained guaranteed recovery without expensive hardware and over 80% performance improvement by pruning the checkpoints in the loop.

Algorithm 1 The High-level Bolt Algorithm

Inputs: CFG PDG

Outputs: Minimal Checkpoint Set MIN_CKPT

1. REGION ← regionformation(CFG)
2. BASE_CKPT ← eager_checkpoint(REGION, CFG)
3. MIN_CKPT ← checkpoint pruning(BASE_CKPT, CFG, PDG)

IV. BOLT COMPILER

Algorithm 1 shows a high-level Bolt algorithm which takes the control flow graph (CFG) and program dependence graph (PDG) as inputs. Bolt first partitions the entire CFG into different regions (Section IV-A). Then, it performs the eager checkpointing, that inserts checkpoints right after the last-updated registers in each region, to preserve the register inputs for guaranteed recovery without expensive hardware support (Section IV-B). Finally, Bolt prunes those checkpoints that can be reconstructed by other checkpointed value to minimize the performance overhead (Section IV-C, IV-D).

A. Region Formation

Bolt is versatile in that it is applicable to many region formation schemes [17], [18], [19], [20]. As discussed in Section II-B, previous idempotence-base recovery schemes [17], [21] have developed a simple region partition algorithm to guarantee no memory anti-dependence in the regions, making preserving register inputs the only cost.

For comparison of Bolt and other idempotence-based recovery schemes, this paper intentionally uses Idem’s region formation algorithm [17] to partition the entire program (CFG) into different idempotent regions. By doing so, we can fairly compare Bolt with the other schemes that leverage different methodologies for register input preservation.

Bolt also treats memory fences as region boundaries to obey underlying memory models and handles the I/O instructions as single instruction region as with [17], [18].

In particular, Bolt checks if the original idempotent regions overflow the store buffer, in which case such regions are split. This is particularly important for the store verification (See fault model in Section II-D). Note that prior schemes do not prevent the overflow, which is another reason why they cannot achieve the guaranteed error recovery.

B. Eager Checkpointing

To achieve guaranteed recovery in the absence of expensive hardware support, Bolt employs eager checkpointing that preserves register inputs to a region right after their definition. That is, Bolt is interested in the last update instructions that define the register inputs prior to the region entry. The identification of register inputs (live-ins) to a region is a standard analysis in modern compilers and is omitted due to space constraints. Given the partitioned region REGION, for each register input $r$ of a region $R_{g} \in$ REGION, Bolt reverses $^1$

\[ R_{g} = R_{0} + (R_{2} \times 2); \]

\[ R_{2} = ldrb \[R_{7}, 0\]; \]

\[ R_{3} = ldrb \[R_{7}, 1\]; \]

\[ R_{4} = ldrb \[R_{7}, 2\]; \]

\[ R_{5} = ldrb \[R_{7}, 3\]; \]

\[ R_{6} = ldrb \[R_{7}, 4\]; \]

\[ if (R_{0} < R_{6}) \text{goto loop} \]

Fig. 4. Motivating example with byte_reverse code of sha in MiBench: only interesting part is shown

Fig. 5. An eager checkpointing example
the edges of CFG and traverses it starting from the entry of $R_g$ in a depth first order, to search for the last update instructions of $r$. Figure 5 shows an example of the last updates to the inputs of region $R_g$ where the inputs are $R_1, R_2, R_3$.

All those identified last update instructions form the baseline checkpoint set $\text{BASE}_\text{CKPT}$ where some checkpoints in the set might be eliminated by Bolt’s checkpoint pruning techniques. Therefore, in the worst case, Bolt can just instructs right after these last update instructions in $\text{BASE}_\text{CKPT}$ with checkpointing stores to achieve the guaranteed recovery, but at a worse performance overhead. Upon recovery, Bolt’s runtime system simply recovers the checkpointed input as in Figure 3. Hereafter, we refer the last updates in $\text{BASE}_\text{CKPT}$ as checkpoints in ease of illustration.

C. Checkpoint Pruning

To reduce the runtime overhead, Bolt prunes the checkpoints in $\text{BASE}_\text{CKPT}$ without compromising the recovery capability. The problem of checkpoint pruning is to find an minimal subset $\text{MIN}_\text{CKPT}$ out of $\text{BASE}_\text{CKPT}$ that still allows Bolt’s recovery runtime to restore all the register inputs of a faulty region in the event of a soft error. To address this, Bolt leverages the following axiom:

Axiom 1: Given a register input $r$ of a region $R_g$, if $r$’s value can be safely reconstructed, the checkpoints for $r$’s last updates are unnecessary for $R_g$.

At the first glance, the checkpoint pruning problem simply seems like a program slicing problem [36], [38] by exploring the backward slice of the register inputs. However, traditional backward slicing cannot guarantee the value of the register inputs to be safely reconstructed, i.e. restoring the register inputs to their original value. Note that such a guarantee is required according to axiom 1. Therefore, eliminating checkpoints with the traditional backward slicing is unsafe. For instance, Figure 6 shows examples of unsafe checkpoint pruning with the traditional backward slicing. (a) tries to eliminate the checkpoint for input $R_2$ of region $R_g$ at $S_1$ by leveraging $S_1$ and $S_2$. However, the checkpoint for $R_1$ at $S_1$ is unsafe as it will be overwritten by the checkpoint at $S_2$. Note, each register has one checkpointing location in a specially reserved region of the stack frame. Thus, the value of $R_2$ cannot be recovered upon recovery. (b) attempts to eliminate the checkpoints for input $R_1$ of region $R_g$ at $S_2$ and $S_9$ leveraging backward slicing to recover input $R_1$. However, the checkpoint of $R_3$ at $S_1$ is unsafe as it might be overwritten by $S_7$, thus failing to recover $R_1$ due to the lack of control flow consideration.

Therefore, Bolt introduces recovery slice which guarantees to restore all the register inputs upon recovery. To determine whether it is safe to eliminate the checkpoints for a register input, we must guarantee that the integrity of both the data flow and control flow of the recovery slice such that the recovery slice can precisely reconstruct the value of register inputs without eagerly checkpointing them.

To construct a recovery slice from the program dependence graph (PDG), Bolt must depend on (1) Data dependence back-tracking to ensure that the resulting slice recomputes the value of register input from only safe checkpoints and statements; and (2) Control dependence backtracking to guarantee the right control flow in the recovery slice. Note, in a PDG, all the statements are represented as vertices and connected with edges annotating control/data dependence relationship.

1) Data Dependence Backtracking: To reconstruct the data flow of register inputs, Bolt traverses backwards through the vertices of the PDG via the data-dependent edges in a depth-first search manner. We use the notation $v \xrightarrow{\delta^D} v'$ to denote $v$ is data dependent on $v'$, i.e., $v'$ defines the register $r$ that can reach $v$, and $v$ uses $r$. Given a register input $r$ of a region $R_g$, Bolt backtracks along a sequence of vertices $(R_gE \xrightarrow{r_1} ... \xrightarrow{r_n} v_n)$, where $R_gE$ represents the entry of region $R_g$ which is data-dependent on $r_1$ and $v_n$ is the last node in a PDG path. In particular, the backtracking terminates along the path when one of the following is met:

- The vertex $v_n$ has no data-dependent edge;
- The vertex $v_n$ has already been in $\text{MIN}_\text{CKPT}$ set;
- The vertex $v_n$ is an unsafe statement.

First, if there is no vertex on which $v_n$ depends, it is in form of $r = \text{const}$. On a fault, the register can be re-assigned with the constant value, thus Bolt can recover $r$ without checkpointing it.

Second, if $v_n$ has already been in $\text{MIN}_\text{CKPT}$ set, it means $v_n$ fails in previous data/control-dependence backtracking. Therefore, Bolt terminates backtracking along the path and validates whether $v_n$ is a safe checkpoint to ensure data flow integrity. That is, the checkpoint must not be overwritten along all the reachable control flow paths (RCFP). We use the notation $v \xrightarrow{\Delta} v'$, where $v'$ defines $r$ used by $v'$, to denote the control flow paths on which $v$ can reach $v'$ without intervening definition of $r$ along the path. To validate a checkpoint $(v_n)$ along the path, $v_n \xrightarrow{\Delta} v_{n-1} \xrightarrow{\Delta} ... \xrightarrow{\Delta} R_gE$, Bolt simply traverses the path to ensure there is no other checkpoints for the same register $r_n$. If the validation succeeds, Bolt terminates backtracking along this path. Otherwise, Bolt return to the most recent vertex $(v_i)$ that are in the $\text{BASE}_\text{CKPT}$ set and validate the vertex until Bolt find a safe checkpoint and place the checkpoint to $\text{MIN}_\text{CKPT}$. Then, Bolt terminates backtracking along this path. Note, whenever
Bolt puts a checkpoint into `MIN_CKPT`, Bolt needs to verify whether the checkpoint breaks any existing recovery slices, i.e., the checkpoint overwrites the checkpoints depended by the existing recovery slices. In such cases, Bolt needs to invalidate those broken recovery slices and re-construct them.

Lastly, Bolt also terminates backtracking if $v_n$ is an unsafe statement, e.g., `load`, `call`. Same with validating checkpoints, if the value in the memory location of the `load` is overwritten along RCFP, Bolt cannot rely on the value to build the recovery slice. Thus, Bolt must validate the `load` same as validating checkpoints to ensure the no stores overwrite the same memory location. As Bolt limits itself to intra-procedural in its current form, Bolt also stops backtracking from call instructions and applies the same procedure to call instructions as with dealing unsafe checkpoints.

\[ \begin{align*}
\text{Fig. 7. A data dependence backtracking example} \\
\end{align*} \]

Figure 7 shows a data dependence backtracking example following the example in Figure 5 for a live-in register $R_1$ with respect to region $R_{g0}$. Assuming the checkpoint at $S_0$ is already in `MIN_CKPT`, all data-dependent paths terminate at $S_0$ during the data dependence backtracking. The reachable control flow paths (RCFP) via $R_1$’s last update instructions (i.e., $S_6$, $S_7$, and $S_0$) are shown in with different type of line. The checkpoint for $R_3$ at $S_0$ is not overwritten along all the RCFP. Thus, Bolt ensures the integrity of the data flows for register input $R_1$.

2) Control Dependence Backtrack: Suppose a vertex $v_i$ has a set of data-dependent vertices ($\mathbb{V}$) for register $r$, i.e., $\forall v'_i \in \mathbb{V}, v_i \xrightarrow{\delta} v'_i$. Once all the data-dependent paths via $v_i$ successfully finish data dependent backtracking, Bolt should ensure the control flow integrity so that the recovery slice can produce the expected value of $r$ at $v_i$. We consider the following 3 cases according to the size of $\mathbb{V}$, i.e., the number of data-dependent vertices of $v_i$, and whether all these dependent vertices are checkpointed or not:

- If $|\mathbb{V}|$ is 1, the control always reaches $v_i$ after its singular data-dependent vertex.
- If $|\mathbb{V}|$ is greater than 1 and all the data-dependent vertices are checkpointed, there is no need to distinguish the checkpoints as they store the checkpointed value to the same location reserved in the stack frame. Thus, we can restore the value of $r$ from the location.
- If $|\mathbb{V}|$ is greater than 1 but not all of the data-dependent vertices are checkpointed, Bolt needs to distinguish them by tracking their control flow.

Only for the third case, Bolt backtracks control dependence to ensure that only one of the values produced by the data-dependent vertices in $\mathbb{V}$ can reach the vertex $v_i$. To achieve this, Bolt first computes the nearest common dominating predicate $P_{\text{Pred} \_n}$ of the data-dependent vertices based on a dominator tree [39]. Then, it traverses control-dependent edges backwards from each vertex in $\mathbb{V}$ up to $P_{\text{Pred} \_n}$ validating the visited control predicates.

To simplify presentation, our discussion below assumes that each basic block in CFG contains at most one predicate. We use the notation $v \xrightarrow{\delta} v'$ to represent $v$ is control dependent on $v'$. For each vertex $v_i$ in $\mathbb{V}$, Bolt inspects the sequence of vertices $v'_i \xrightarrow{\delta} v_{c0} \xrightarrow{\delta} \cdots \xrightarrow{\delta} v_{cn}$, where $v_{c0} \ldots v_{cn}$ are the control dependence predicates, and $v_{cn}$ is the $P_{\text{Pred} \_n}$ vertex. Then, Bolt validates the control dependence predicates, e.g., $v_{c0} \ldots v_{cn}$ by exploring the recovery slice of each predicate \{ $v_{ci} \mid i \in \{0 \ldots n\}$ \}. Note that every vertex in the recovery slice of $v_{ci}$ should be validated by traversing the reachable control flow paths not only to $v_{ci}$, but also to the region entry $R_{gE}$. In other words, every vertex in the recovery slice of $v_{ci}$ should also be validated along the paths: $v_{ci} \xrightarrow{\Delta} v_1 \xrightarrow{\Delta} \cdots \xrightarrow{\Delta} R_{gE}$. Specifically, $v_{ci} \xrightarrow{\Delta} v_1$ represents all the control flow paths that can reach $v_1$ after $v_{ci}$.

\[ \begin{align*}
\text{Fig. 8. A control dependence backtracking example} \\
\end{align*} \]

As shown in Figure 8, once all the paths via the data dependent vertices (e.g., $S_6$, $S_7$, $S_0$ in Figure 5) of register input $R_1$ are successfully backtracked, Bolt starts the control dependence backtracking. First, Bolt determines the nearest common dominating predicate of $S_6$, $S_7$, $S_0$ as $S_1$. Then, from each data dependent vertex (i.e., $S_6$, $S_7$, $S_0$) to the nearest common dominating predicate, Bolt validates each control-dependent vertex, i.e., $S_1$, $S_8$, $S_5$ and $S_8$. In this example, Bolt can validate the checkpoint $S_0$ in the predicate $S_3$’s recovery slice along the reachable control flow path (RCFP) to $S_3$ ($BB_0 \rightarrow BB_1$) as well as every RCFP from $S_3$ to the region entry. In Figure 8, as the checkpoint for $R_3$ is not overwritten along all the RCFP via each control dependence predicate. Therefore, Bolt consider the control flow of the recovery slice
of register input \( R_1 \) are well-formed and the checkpoints for the last updates of \( R_1 \) (\( S_6, S_7, S_9 \)) can be safely eliminated.

**D. Checkpoint Pruning for Loop**

![Fig. 9. An illustrating example of ineliminable checkpoints for loop](image)

Incorporating loops into the checkpoint pruning complicates the analysis, since they cause issues in the data/control dependence backtracking. Figure 9 (a) corresponds to the case where the data dependence backtracking fails in the loop while Figure 9 (b) to the case where the control dependence backtracking fails. Suppose that \( R_0 \) is live-in to a region, \( R_{B0} \) starts at \( BB_0 \) in Figure 9 (a). The last update of \( R_0 \) happens in the instruction \( S_1 \). As Bolt tries to eliminate the checkpoint for \( S_1 \), it runs into infinite backtracking as \( S_1 \) is data dependent on itself. To solve the problem, Bolt tracks the history of visited vertices and treats any of them as unsafe statements during the backtracking.

Figure 9 (b) demonstrates the infeasible case of the control dependence backtracking. Here, \( R_{B0} \) starts at \( BB_1 \), and \( R_0 \) is live-in to \( R_{B0} \). Even if the data dependence backtracking may be successful in this case (i.e., checkpoint at \( S_0 \) can be eliminated), Bolt cannot differentiate definitions of \( R_0 \) at \( S_0 \) and \( S_2 \) by the control dependence backtracking. Fortunately, these two cases only happen for basic induction variables and other non-linear induction variables [40]. To get around the problem, Bolt places the last updates of such variables into \texttt{MIN\_CKPT}.

**V. IMPLEMENTATION**

The Bolt compiler described in Section IV is a series of passes for the LLVM compiler framework [41]. After checkpoint pruning, Bolt instruments each last update instruction in \texttt{MIN\_CKPT} with corresponding checkpointing stores. In addition, Bolt also considers the following implementation details.

**A. Limiting the Backtrack Depth for Recovery Time**

In the limit, the exploration of recovery slice for a checkpoint explores all the way to the initial input of the program. That is, one can always recover by restarting the entire execution. Besides, this also influences scalability for compilation. Therefore, it is important for Bolt to set a reasonable backtrack depth which limits the depth of the dependence backtracking.

For example, a backtrack depth of 5 prevents Bolt from backtracking more than 5 data/control-dependent vertices in the program dependence graph (PDG). If the dependence backtracking does not terminate by itself within the depth, Bolt simply treats the last-visited vertex as if it is unsafe, applying the same procedure to deal with invalid statements. In particular, we discover that a small backtrack depth (10) is enough to achieve the significant reduction of checkpoint candidates compared with that of a high backtrack depth (100). The implication is two-fold: First, our compiler is scalable; Second, the fault-recovery execution time is reasonably small. Section VII evaluates different backtrack depths in more details.

**B. Just-in-time Recovery Slice Generation**

Before jumping back to the beginning of a faulty region where the error is detected, Bolt’s runtime system needs to executes a recovery slice to restore the inputs to the region (See Section III). For this purpose, Bolt can generate the recovery slice either statically or dynamically.

Bolt can statically generate the recovery slice during the exploration to prune the checkpoints. However, two problems limit a static approach: (1) It significantly increases the code size by generating the recovery block for each region, which is prohibited for low-end embedded systems. (2) More importantly, the static slice generation cannot exploit the opportunity to prune the checkpoints for the inputs to a function boundary which is also a region boundary. Recall that Bolt’s checkpoints store the register value to a reserved location in the stack frame of each function. Therefore, the checkpoints of the callee do not overwrite that of the caller, even if they save the same register for checkpointing. That is, we can leverage the checkpoints in the caller to reconstruct the checkpoints in the callee. However, it is very expensive to determine the calling contexts statically.

In contrast, a dynamic approach is much more preferable due to the lack of these problems. Code size will not increase, because the recovery slice is built dynamically. For the second problem of the static approach, Bolt’s runtime system first builds the program dependence graph by analyzing the binary. Then, it generates the recovery slice using the algorithm in Section IV-C. Note that since all required analyses are performed after the register allocation, there is no technical problem in achieving such just-in-time slice generation without relying on source code. By looking at the return address of the callee’s stack, Bolt can determine the caller function and continue the recovery slice exploration from the call site. In essence, Bolt can “inter-procedurally” generate the recovery slice. Section VII-D further investigates the overhead of Bolt’s just-in-time recovery slice generation.
the region where they occur to achieve full recoverability. Existing software [19], [20], [42], [30] and hardware [13], [43], [44] approaches can be employed for the previous work [17], [26], [18] to achieve full detection coverage within the region. For example, Idem [17] leverages dual-modular-redundancy (DMR) [42], [44] to detect the errors before leaving the faulty region. However, DMR-based detection schemes come with expensive performance/area overhead which might overwhelm the benefits brought by the fine-grained recovery schemes.

Fortunately, Clover [19], [20] proposes an efficient error detection scheme to contain the errors within the idempotent-region while incurring negligible area and moderate performance overheads. Clover detects all the soft errors before leaving the region where they occurred by using the acoustic sensors [13] and partial instruction duplication. In fact, Clover can further reduce their overhead: 1) The area overhead and detection latency of the sensors can be significantly reduced with careful placement of the sensors on top of the processor-die rather than naive mesh-like placement [45], [13]. 2) Extending the idempotent-region length for less instruction-duplication can dramatically reduce the performance overhead; previous work [21] shows that the region can be lengthened by orders of magnitudes with precise points-to analysis thus tolerating a much higher detection latency. Thus, we believe said scheme would serve well for Bolt.

b) Special Register Protection: There are various special registers in the register file such as stack pointer (SP), condition status register (e.g. EFLAGS) and program counter PC, etc. While all other special registers can be checkpointed in the stack frame as with the regular registers, Bolt checkpoints SP in a global array in case of multithreaded program. Thus, upon recovery, we first leverage instruction like rdtsrep to retrieve the thread ID. Then, Bolt safely reloads SP from the element in the global array with the thread ID. Last, using the recovered SP, Bolt can retrieve all other checkpointed register value safely from the stack frame.

c) Impact on HPC applications: As soft-errors are becoming more dominant in large scale HPC systems [11], it is critical to innovate new schemes that can reduce the overhead of checkpointing and soft-error recovery. Unfortunately, pure software-based schemes incur very high overhead and often have high barrier to entry for adoption since it may require changing the application source code. On the other hand, hardware-based schemes impose high chip area and performance overheads. Therefore, a compiler-based scheme, such as proposed in this study, is likely to positively impact scientific applications running on HPC systems, since it does not require expensive hardware support for register file protection and manual code changes. As shown in our evaluation section, Bolt reduces the soft-error recovery overhead by 95% for a number of pthread applications taken from SPLASH benchmark suites, and relieves chip designers from providing expensive ECC protection for register file and other internal structures such as instruction queue – potentially reducing the chip design and testing cost as well.

VII. Evaluation and Analysis

To evaluate Bolt, we first analyze the checkpoint pruning optimization and how different backtrack depths impact this optimization. In particular, how this affects the number of checkpoints to be removed. Then, we evaluate Bolt’s overhead and compare with the state-of-the-art idempotence-based soft error recovery schemes. Finally, we evaluate our fault-recovery execution overhead after fault occurrence.

A. Experimental Methodology

We conduct our simulations on top of the Gem5 simulator [46] with the ARMv7 ISA, modeling a modern two-issue out-of-order 2 GHz processor with L1-I/D (32KB, 2-way, 2-cycle latency, LRU), and L2 (2MB, 8-way, 20-cycle latency, LRU) caches. The pipeline width is two; and the ROB, physical integer register file, and load/store buffer have 192, 256, and 42 entries, respectively.

For the experiments, we use three sets of benchmarks: SPEC2006 [47] for general-purpose computation, Mediabench/MiBench [48], [37] for embedded systems, and SPLASH2 [49] for parallel systems. All the applications are compiled with a standard --O3 optimization and fully simulated with appropriate inputs.

B. The Breakdown of Checkpoint Candidates

Figure 10 shows how Bolt’s checkpoint pruning works across different backtrack depths with the breakdown of checkpoint candidates set, i.e., BASE_CKPT (See Section IV-B). In the breakdown of each bar, the top portion corresponds to the checkpoints that can be eliminated while the rest correspond to the necessary checkpoints that must be instrumented. We classify the necessary checkpoints as loop and other. The loop checkpoints are the ones Bolt cannot prune due to limitations for loop discussed in Section IV-D, thus they remain the same regardless of the backtrack depth. In contrast, other checkpoints are affected by different backtrack depths because they are the ones identified by the recovery slice exploration. For each application, we show such five breakdown bars corresponding to the backtrack depths of 5, 10, 20, 50, and 100, respectively (from left to right in Figure 10). We make the following observations:

- Bolt’s checkpoint pruning technique is effective at decreasing the number of checkpoints. It can eliminate on average more than 60% of the checkpoint candidates (BASE_CKPT).
- For most of the applications, the portion of the eliminated checkpoints start to saturate when the backtrack depth is greater than 10. Such small backtrack depth is beneficial in two-fold: (1) the compilation will be scalable as the number of backtrack is dramatically reduced, (2) The recovery time is reasonably small after fault occurrence because at most 10 instructions will be executed to recover one register input. With those in mind, we empirically determine the backtrack depth as 10 for the remaining experiments.
C. Overheads

We compare performance overhead with that of the state-of-the-art’s recovery techniques. For comparison, all the techniques employ the same region construction algorithm described in Idem [17]. We set the baseline to the original application binary without any recovery support. Before presenting more detailed discussion, we categorize and summarize each technique as follows:

- **Idem** [17] uses register renaming to preserve the live-in registers that have anti-dependence. Idem may increase the register pressure, thus degrading the performance due to the resulting register spillings and reloadings. It requires expensive RF protection to guarantee soft error recovery.
- **Encore** is our version of Encore [18]. It checkpoints the live-in register, that have anti-dependence, at the region entry. As with Idem, Encore also assumes RF protection.
- **Bolt-no-opt** checkpoints all the live-in registers as soon as they are defined (Section IV-B), offering guaranteed soft error recovery without RF protection.
- **Bolt** is equipped with the checkpoint pruning, that eliminates unnecessary checkpoints, offering guaranteed soft error recovery at low overhead without RF protection.

We present performance overhead in two forms. The first one reflects the architectural effect of store buffering which buffers the unverified stores until their region ends (See fault model in Section II). For the other form, we provide a architecture-neutral performance overhead in terms of total dynamic instruction count [17], [18], [21].

1) **Runtime Overhead with Architectural Effect**: Figure 11 shows the architectural performance overhead in terms of execution time (cycles) where the y-axis represents the overhead as percentage compared to that of the original program. We modified the simulator to model the effect of store buffering, i.e. the processor holds the stores in one region until the region ends. Then, those stores will be drained to the caches if there is available bandwidth between the store queue and caches. For those regions (<0.001%) that contain stores more than the size of store queue (42 entry in intel i7 haswell), we can place additional region boundaries to break those regions into smaller ones so that they won’t overflow the store queue.

Store buffering might adversely degrade the performance. If the stores in the previous region cannot be drained to cache due to the bandwidth congestion, those stores have to stay in the store queue. Thus, later stores in the current region cannot get executed if the store queue is full, causing pipeline stall.
However, we found out that such situation is rare and store buffering trivially affects the performance by < 1% (not shown in the figure).

Idem incurs more overhead in most of the applications compared to other schemes. Idem can introduce on average an 11% performance loss and up to a 30% performance loss. This is reasonable as Idem preserves the region inputs by register renaming which may ends up with more spillings and reloadings. Worse, reloading instructions (load) are on the critical path of processor which might greatly degrade the performance. In contrast, other schemes just need to pay the overhead of checkpointing stores which are off the critical path. Besides, the performance overhead of Idem will further degrade given the effect of ECC protection to the register file. It’s interesting to observe that Idem outperforms Encore and Bolt-no-opt in the SPLASH2 applications. This is because the regions in the applications of SPLASH2 generally have larger region sizes which hide Idem’s overhead as register pressure is higher in larger regions. This phenomenon is corroborated by the previous work [17], [21].

Encore in general performs better than Bolt-no-opt as it only checkpoints the live-in register with anti-dependence in the region resulting in 9.2% overhead on average. In a few applications (e.g. gobmk, cholesky, lu etc.), Encore incurs much more overhead (upto 40%) than Bolt-no-opt. This is because Bolt-no-opt employ eager checkpointing and thus avoid putting some checkpoints in the loops. As with Idem, Encore also have to pay the overhead of RF protection.

Without RF protection, Bolt-no-opt achieves a comparable performance overhead with Encore and Idem resulting in 9.5% on average. Since the effect of adding ECC to a RF is not considered in our experiments, the power efficiency and performance hits to Encore and Idem are not visible. But, with good reason Bolt-no-opt should dominate both Encore and Idem respectively.

To further reduce and shift the runtime overhead of fault-free execution to that fault-recovery execution, Bolt applies the checkpoint pruning optimization. This pruning greatly shrinks overall overhead leading to an average overhead of 4.7%, which is 57% and 49% reduction compared to Idem and Encore respectively. Again, the improvements are underestimated since our simulation does not reflect the ECC delay of the prior work. In summary, Bolt provides guaranteed soft error recovery at low overhead without expensive RF hardware protection schemes.

2) Architecture-Neutral Performance Overhead: Figure 12 shows architecture-neutral performance overhead across different schemes. A similar trend as in Figure 11 is observed in the dynamic instruction count overhead. As expected, Bolt executes much less instructions than Idem and Encore. Note that a checkpoint approach is in general preferable over a register renaming approach as stores are off the critical path in modern out-of-order processors. The downside of the checkpoint approach is that it needs to restore checkpointed data for recovery making it slightly slower. However, as soft errors happen once in a while, it is much more desirable to make the common fault-free case faster.

D. Fault-Recovery Overhead

After a fault occurrence, Bolt invokes the exception handler to generate the recovery slice and execute the recovery slice to recover the region input before releasing the control to the entry of faulty region. Thus, fault-recovery time in Bolt contains two parts: (1) time to generate recovery slice and; (2) time to execute the slice and the faulty region.

As we limit the backtrace depth to 10, the maximum number of instructions in the slice is less than 10 × #RF, where #RF is the size of register file. Since the region sizes are < 30 instructions on average, the time to execute the slice and faulty region are trivially small relative to the time to generate recovery slice.

We perform dynamic recovery slice generation to examine Bolt’s practicality. For each recovery slice, we first generate the program dependence graph (PDG) and dominator tree (DT) information. Then, we generate the recovery slice based on the PDG and DT information with our adapted algorithm. Then the total cycles are recorded. Table I shows the recovery time distribution for all the regions generated from all the applications reported. We use a 2GHz processor frequency to calculate the time in millisecond (ms) scale. As we can see, over 95.72% of regions can generate their recovery slice within 5 ms., and 99.99% of the region can generate their recovery slice within 50 ms, which is negligible for user as soft error happens rather infrequently.

<table>
<thead>
<tr>
<th>Generation time (ms)</th>
<th>≤5</th>
<th>≤10</th>
<th>≤20</th>
<th>≤50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio in all regions (%)</td>
<td>95.72</td>
<td>3.02</td>
<td>0.97</td>
<td>0.28</td>
</tr>
</tbody>
</table>

TABLE I: DISTRIBUTION OF THE TIME TO GENERATE RECOVERY SLICE.

VIII. OTHER RELATED WORK

This section describes the prior works related to soft error recovery. We also explain how our proposed scheme advances the state-of-art and differs from previous approaches in this domain.

There exists a large body of work on soft-error recovery with software/hardware approaches. For commercial systems, hardware/software recovery schemes [12], [13], [14], [15] involve taking a snapshot of the system status including register file and memory. To achieve that, they usually maintain multiple copies of register files and a memory log to checkpoint the whole system status. For example, Upasani et al. [13] requires two additional copies of the architectural state units (register files, RAT, etc.) with their ECC protection. Besides, they modify the cache structure and its coherency protocol for memory logging. Such recovery schemes usually introduce exorbitant performance/energy/area overhead making them only viable in high-end commercial server systems.

Flushing the pipeline to recover from a soft error [50], [51] is another alternative. However, such recovery schemes require the errors to be detected before the instruction is
committed in the pipeline implying a high-cost detection scheme. Other techniques also explore simultaneous multithreading (SMT) [44] to recover the leading thread from the trailing thread which occupy the computing resource leading to performance degradation. Chang et al. enable fault recovery at the granularity of a single instruction by incorporating triple-modular-redundancy (TMR) [52]. TMR essentially copies the performance/hardware overhead problem with our novel compiler recovery approach. Our proposed schemes eliminate the perf-
formance degradation. Chang [44] and reconfigurable logic,
We also demonstrate that Bolt can effectively shift the runtime overhead of fault-free execution to that of fault-
recovery execution for lightweight idempotent processing. The experiment results show that Bolt incurs only 4.7% perfor-
ance overhead on average which is 57% and 49% reduction compared to two state-of-the-art schemes that require expen-
sive hardware support for the same recovery guarantee as Bolt.

X. ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments. This work was in part supported by the National Science Foundation under the grant CCF-1527463 and Google Faculty Research Awards. This work was also supported by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

REFERENCES

[10] S. Arahage, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elno-
ing temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems,” in Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, pp. 25–36, June 2014.
[22] G. Gupta, S. Sridharan, and G. S. Sohi, “Globally precise-restartable ex-