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.**

***Keywords*—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.

```
1 x = m;  %live-in variable m = 8.
2 y = x + 1;
3 z = y;  %expected output x = 8, y = z = 9.
```

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 (b) 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.

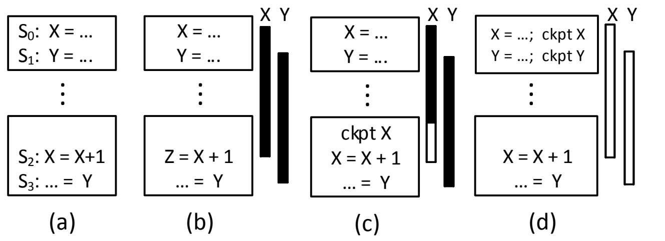


Fig. 2. Idempotent processing and vulnerability window. (a) original code (partitioned into regions in code boxes), (b) Idem[17], (c) Encore[18], (d) Bolt

C. Challenges

However, prior idempotence-based recovery schemes neither guarantee recovery without expensive hardware support. nor achieve insignificant performance overhead [17], [21], [18], [26].

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 S1, 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 VII-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 provides guaranteed recovery 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 for the subsequent regions, 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

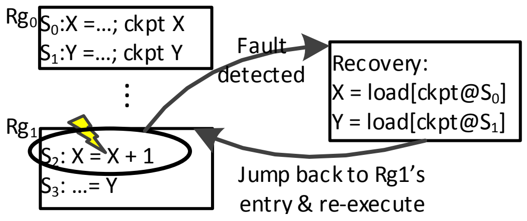


Fig. 3. Bolt's recovery model.

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

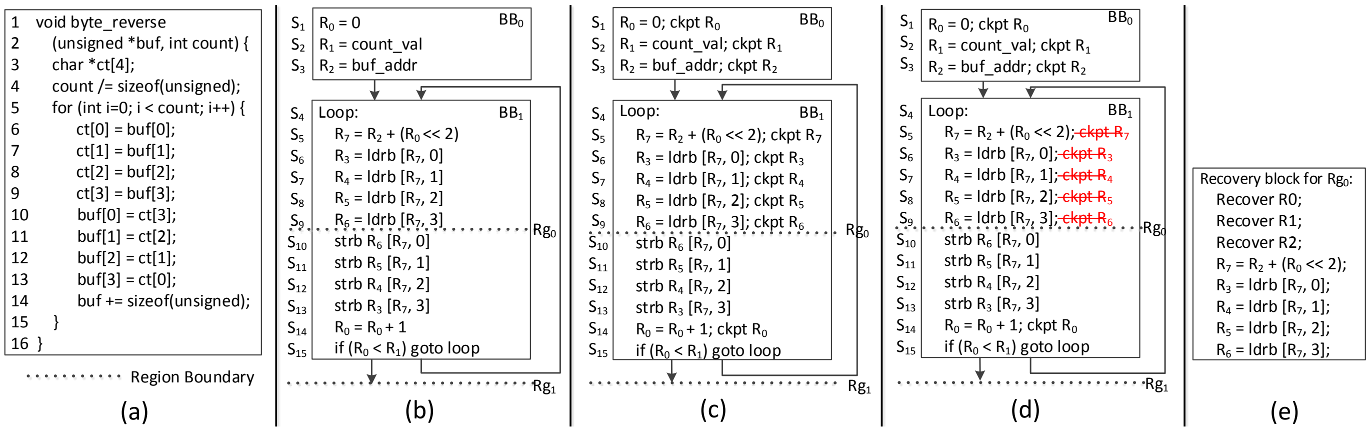
The overhead of idempotent processing is proportional to the number of checkpoints executed at runtime. With that in mind, we propose a novel compiler analysis that can identify unnecessary checkpoints in those eager checkpoints based on the following insight. *For a value corrupted due to a soft error, 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 (Rg_0,Rg_1) 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 Rg_0 are R_0~R_7. (d) minimizes the performance overhead with



```
void byte_reverse
    (unsigned *buf, int count) {
    char *ct[4];
    count /= sizeof(unsigned);
    for (int i=0; i < count; i++) {
        ct[0] = buf[0];
        ct[1] = buf[1];
        ct[2] = buf[2];
        ct[3] = buf[3];
        buf[0] = ct[3];
        buf[1] = ct[2];
        buf[2] = ct[1];
        buf[3] = ct[0];
        buf += sizeof(unsigned);
    }
}
```

```
S1   R0 = 0                                BB0
S2   R1 = count_val
S3   R2 = buf_addr
S4   Loop:                                 BB1
S5     R7 = R2 + (R0 << 2)
S6     R3 = ldrb [R7, 0]
S7     R4 = ldrb [R7, 1]
S8     R5 = ldrb [R7, 2]
S9     R6 = ldrb [R7, 3]
..................................... Rg0
S10    strb R6 [R7, 0]
S11    strb R5 [R7, 1]
S12    strb R4 [R7, 2]
S13    strb R3 [R7, 3]
S14    R0 = R0 + 1
S15    if (R0 < R1) goto loop
..................................... Rg1
```

```
S1   R0 = 0; ckpt R0                       BB0
S2   R1 = count_val; ckpt R1
S3   R2 = buf_addr; ckpt R2
S4   Loop:                                 BB1
S5     R7 = R2 + (R0 << 2); ckpt R7
S6     R3 = ldrb [R7, 0]; ckpt R3
S7     R4 = ldrb [R7, 1]; ckpt R4
S8     R5 = ldrb [R7, 2]; ckpt R5
S9     R6 = ldrb [R7, 3]; ckpt R6
..................................... Rg0
S10    strb R6 [R7, 0]
S11    strb R5 [R7, 1]
S12    strb R4 [R7, 2]
S13    strb R3 [R7, 3]
S14    R0 = R0 + 1; ckpt R0
S15    if (R0 < R1) goto loop
..................................... Rg1
```

```
S1   R0 = 0; ckpt R0                       BB0
S2   R1 = count_val; ckpt R1
S3   R2 = buf_addr; ckpt R2
S4   Loop:                                 BB1
S5     R7 = R2 + (R0 << 2); ~~ckpt R7~~
S6     R3 = ldrb [R7, 0]; ~~ckpt R3~~
S7     R4 = ldrb [R7, 1]; ~~ckpt R4~~
S8     R5 = ldrb [R7, 2]; ~~ckpt R5~~
S9     R6 = ldrb [R7, 3]; ~~ckpt R6~~
..................................... Rg0
S10    strb R6 [R7, 0]
S11    strb R5 [R7, 1]
S12    strb R4 [R7, 2]
S13    strb R3 [R7, 3]
S14    R0 = R0 + 1; ckpt R0
S15    if (R0 < R1) goto loop
..................................... Rg1
```

```
Recovery block for Rg0:
    Recover R0;
    Recover R1;
    Recover R2;
    R7 = R2 + (R0 << 2);
    R3 = ldrb [R7, 0];
    R4 = ldrb [R7, 1];
    R5 = ldrb [R7, 2];
    R6 = ldrb [R7, 3];
```

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

checkpoint pruning where (e) is the resultant recovery block for Rg_0. 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 $\mathbb{MIN\_CKPT}$
1: $\mathbb{REGION}$ ← region_formation(CFG)
2: $\mathbb{BASE\_CKPT}$ ← eager_checkpoint($\mathbb{REGION}$, CFG)
3: $\mathbb{MIN\_CKPT}$ ← checkpoint_pruning($\mathbb{BASE\_CKPT}$, 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[1].

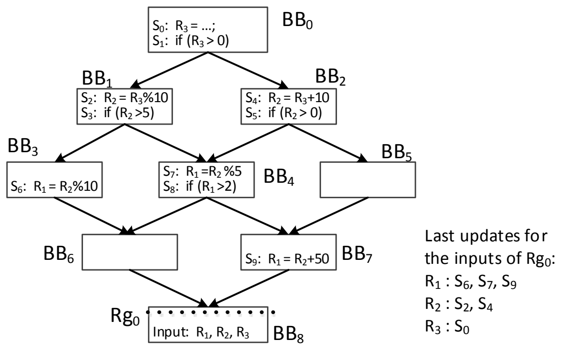


Fig. 5. An eager checkpointing example

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 $\mathbb{REGION}$, for each register input r of a region $Rg \in \mathbb{REGION}$, Bolt reverses

[1]Bolt's technique to avoid the overflow is implemented on top of Idem's region formation and used for other schemes in experiments for comparison.

the edges of CFG and traverses it starting from the entry of Rg 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 Rg_0 where the inputs are R_1, R_2, R_3.

All those identified last update instructions form the baseline checkpoint set $\mathbb{BASE\_CKPT}$ where some checkpoints in the set might be eliminated by Bolt's checkpoint pruning techniques. Therefore, in the worst case, Bolt can just instruments right after these last update instructions in $\mathbb{BASE\_CKPT}$ 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 $\mathbb{BASE\_CKPT}$ as *checkpoints* in ease of illustration.

C. Checkpoint Pruning

To reduce the runtime overhead, Bolt prunes the checkpoints in $\mathbb{BASE\_CKPT}$ without compromising the recovery capability. The problem of checkpoint pruning is to find an minimal subset $\mathbb{MIN\_CKPT}$ out of $\mathbb{BASE\_CKPT}$ 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 Rg, if r's value can be **safely** reconstructed, the checkpoints for r's last updates are unnecessary for Rg.

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 Rg_1 at S_3 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_5. 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 Rg_1 at S_5 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 backtracking* 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.

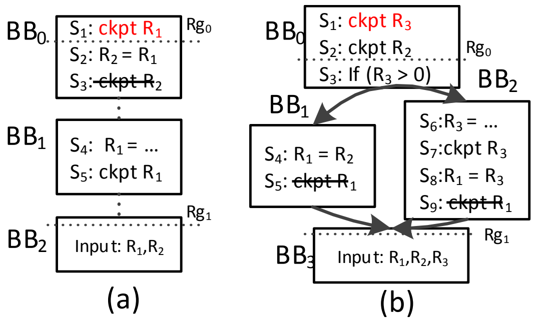


Fig. 6. Examples of unsafe checkpoint pruning.

*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 $v \xrightarrow[\delta^d]{r} v'$ 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 Rg, Bolt backtracks along a sequence of vertices ($RgE \xrightarrow[\delta^d]{r_1} v_1 \xrightarrow[\delta^d]{r_2} \dots \xrightarrow[\delta^d]{r_n} v_n$), where RgE represents the entry of region Rg 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 $\mathbb{MIN\_CKPT}$ 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 $r = const$, 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 in $\mathbb{MIN\_CKPT}$ 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 $v \xrightarrow[\Delta]{r} v'$, 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, $v_n \xrightarrow[\Delta]{r_n} v_{n-1} \xrightarrow[\Delta]{r_{n-1}} \dots \xrightarrow[\Delta]{r_1} RgE$, 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 $\mathbb{BASE\_CKPT}$ set and validate the vertex until Bolt find a safe checkpoint and place the checkpoint to $\mathbb{MIN\_CKPT}$. Then, Bolt terminates backtracking along this path. Note, whenever

Bolt puts a checkpoint into $\mathbb{MIN\_CKPT}$, 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.

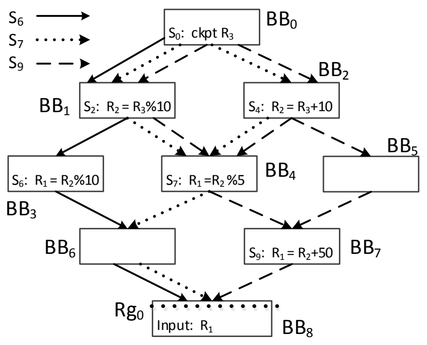


Fig. 7. A data dependence backtracking example

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 Rg_0. Assuming the checkpoint at S_0 is already in $\mathbb{MIN\_CKPT}$, 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_9) 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., $\forall v_i' \in \mathbb{V}$, $v_i \underset{\delta^d}{\xrightarrow{r}} v_i'$. 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 $Pred_{ncd}$ 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 $Pred_{ncd}$ 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 $v \underset{\delta^c}{\longrightarrow} v'$ to represent v is control dependent on v'. For each vertex v_i' in $\mathbb{V}$, Bolt inspects the sequence of vertices is $v_i' \underset{\delta^c}{\longrightarrow} v_{c0} \underset{\delta^c}{\longrightarrow} \ldots \underset{\delta^c}{\longrightarrow} v_{cn}$ where $v_{c0} \ldots v_{cn}$ are the control dependence predicates, and v_{cn} is the $Pred_{ncd}$ vertex. Then, Bolt validates the control dependence predicates, e.g., $v_{c0} \ldots v_{cn}$ by exploring the recovery slice of each predicate $\{v_{ci} | i \in \{0 \ldots n\}\}$. 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 RgE. In other words, every vertex in the recovery slice of v_{ci} should also be validated along the paths: $v_{ci} \underset{\Delta}{\longrightarrow} v_i \underset{\Delta}{\xrightarrow{r_i}} \ldots RgE$. Specifically, $v_{ci} \underset{\Delta}{\longrightarrow} v_i$ represents all the control flow paths that can reach v_i after v_{ci}.

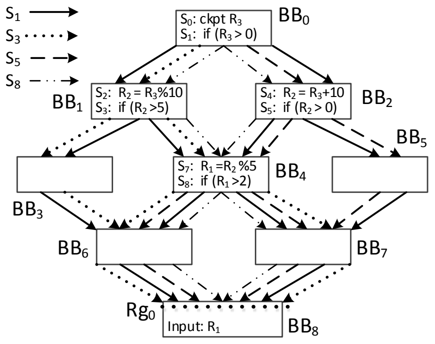


Fig. 8. A control dependence backtracking example

As shown in Figure 8, once all the paths via the data dependent vertices (e.g., S_6, S_7, S_9 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_8, S_9 as S_1. Then, from each data dependent vertex (i.e., S_6, S_7, S_9) to the nearest common dominating predicate, Bolt validates each control-dependent vertex, i.e., S_1, S_3, 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

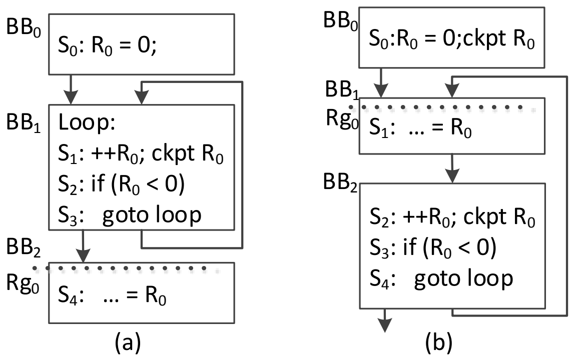


Fig. 9. An illustrating example of ineliminable checkpoints for loop

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, Rg_0 starts at BB_2 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, Rg_0 starts at BB_1, and R_0 is live-in to Rg_0. 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 $\mathbb{MIN\_CKPT}$.

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 $\mathbb{MIN\_CKPT}$ 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.

VI. Discussion

a) Error Detection: Detection schemes are orthogonal to our proposed recovery scheme. As with other region-based recovery schemes [17], [26], [18], soft errors must not escape 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 lengthen 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 register 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 *rdtscp* 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.

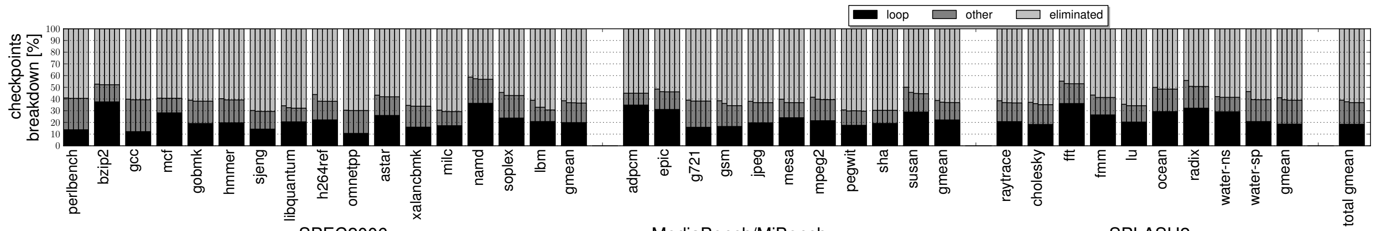
For the experiments, we use three sets of benchmarks: SPEC2006 [47] for general-purpose computation, MediaBench/MiBench [48], [37] for embedded systems, and SPLASH2 [49] for parallel systems. All the applications are compiled with a standard `-O3` optimization and fully simulated with appropriate inputs.

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., $\mathbb{BASE\_CKPT}$ (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 ($\mathbb{BASE\_CKPT}$).
- 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.



From left to right, the columns demonstrate the fraction of necessary (ineliminable and other) checkpoints after applying our checkpoint pruning optimization to prune the checkpoint candidates with different backtrack depths (5, 10, 20, 50, 100).

Fig. 10. Checkpoint Breakdown

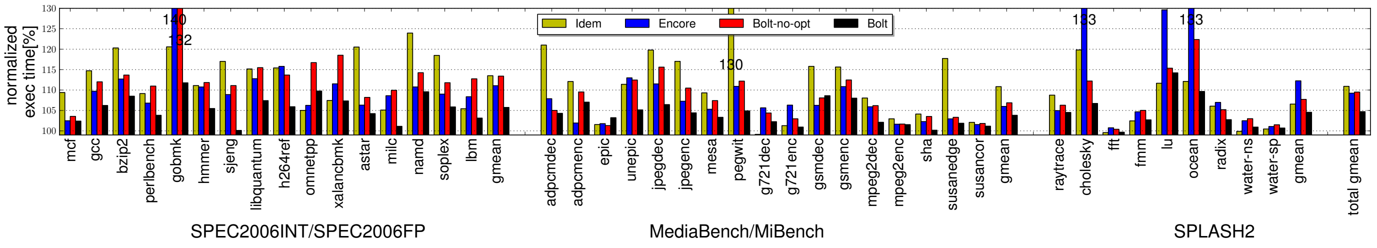


Fig. 11. Performance overhead in terms of execution time (cycles) considering the architectural effect of store buffering.

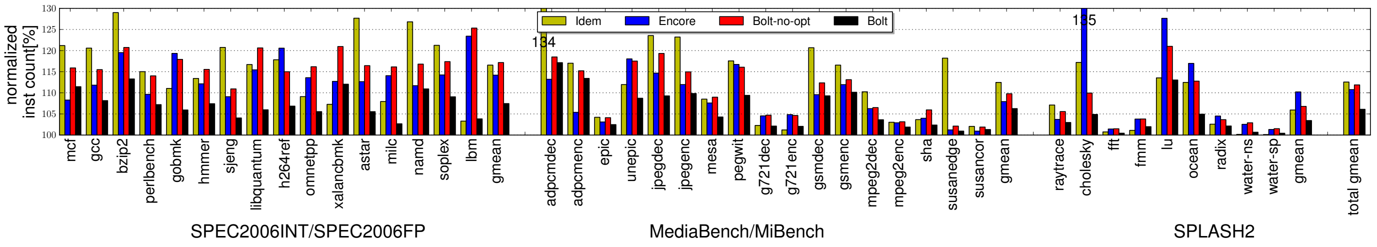


Fig. 12. Architectural-neutral performance overhead in terms of total dynamic instruction count.

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 under-estimated 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 approache 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.

Generation time (ms)	$\leq$5	$\leq$10	$\leq$20	$\leq$50
Ratio in all regions (%)	95.72	3.02	0.97	0.28

TABLE I
DISTRIBUTION OF THE TIME TO GENERATE RECOVERY SLICE.

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 backtrack depth to 10, the maximum number of instructions in the slice is less than 10 $\times$#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.

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 multi-threading (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 original execution to two more redundant executions and recovers the error by majority voting among those three versions. However, they also introduce significant performance overhead preventing their adoption in commodity systems.

In contrast, idempotent-based recovery is a promising recovery approach. Our proposed schemes eliminate the performance/hardware overhead problem with our novel compiler analysis making our idempotence-based recovery scheme realistic to be applied in low-cost commodity systems.

IX. Summary

This paper presents Bolt, a lightweight soft error recovery scheme. Bolt guarantees 100% recovery without expensive register file protection. It can recover from soft errors even in the case when the RF is corrupted. To the best of our knowledge, Bolt is the first compiler-directed recovery solution that does not require expensive RF protection mechanisms for idempotence.

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% performance overhead on average which is 57% and 49% reduction compared to two state-of-the-art schemes that require expensive 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

[1] L. Wang and K. Skadron, "Implications of the power wall: Dim cores and reconfigurable logic," *IEEE Micro*, pp. 40–48, 2013.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376, 2011.

[3] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[4] M. B. Taylor, "Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1131–1136, 2012.

[5] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, G. Grider, R. Haring, J. Hittinger, A. Hoisie, D. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, and R. Stevens, "Top ten exascale research challenges," tech. rep., U.S. Department of Energy ASCAC Subcommittee, Boston, MA, USA, Feburary 2014.

[6] S. Borka, "The exascale challenge," in *International Symposium on VLSI Design Automation and Test*, 2010.

[7] J. Torrellas, D. Quinlan, A. Snavely, and W. Pinfold, "Thrifty: An exascale architecture for energy-proportional computing," 2013.

[8] J. Ang, B. Carnes, P. Chiang, D. Doerfler, S. Dosanjh, P. Fields, K. Koch, J. Laros, M. Leininger, J. Noe, T. Quinn, J. Torrellas, J. Vetter, C. Wampler, and A. White, "Exascale hardware architectures working group," tech. rep., Lawrence Livermore National Laboratory, 2011.

[9] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji, P. C. Diniz, A. Koniges, and M. Snir, "Exascale programming challenges," in *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*, U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Jul 2011.

[10] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, R. Harrison, W. Harrod, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling, "Exascale software study: Software challenges in extreme scale systems," 2009.

[11] M. Snir, R. W. Wisniewski, J. A. Abraham, V. Adve, S. Bagchi, P. Balaji, J. Belak, F. C. P. Bose, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, M. E. C. Engelmann, S. Fazzari, A. Geist, R. Gupta, F. Johnson, Krishnamoorthy, S. Leyffer, T. M. D. Liberty, Mitra, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *"International Journal of High Performance Computing Applications"*, vol. 28, no. 2, 2014.

[12] P. Ramachandran, S. K. S. Hari, M. Li, and S. V. Adve, "Hardware fault recovery for i/o intensive applications," *ACM Trans. Archit. Code Optim.*, vol. 11, pp. 33:1–33:25, Oct. 2014.

[13] G. Upasani, X. Vera, and A. Gonzalez, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery.," in *ISCA*, pp. 37–48, 2014.

[14] D. Sorin, M. Martin, M. Hill, and D. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 123–134, 2002.

[15] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "Ibm's s/390 g5 microprocessor design," *Micro, IEEE*, vol. 19, pp. 12–23, Mar 1999.

[16] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting 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.

[17] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 475–486, ACM, 2012.

[18] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 398–409, ACM, 2011.

[19] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, LCTES'15, (New York, NY, USA), pp. 2:1–2:10, ACM, 2015.

[20] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. XX, no. X, 2016.

[21] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2013.

[22] G. Gupta, S. Sridharan, and G. S. Sohi, "Globally precise-restartable execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 181–192, 2014.

[23] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, "Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution," in *Proceedings of the Eighteenth International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 113–126, 2013.

[24] D. A. Popescu, E.-D. Tirsa, M. I. Andreica, and V. Cristea, "An application-assisted checkpoint-restart mechanism for java applications.," in *International Symposium on Parallel and Distributed Computing (ISPDC)* (N. Tapus, D. Grigoras, R. Potolea, and F. Pop, eds.), pp. 190–197, IEEE, 2013.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[26] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 497–508, ACM, 2010.

[27] ARM, "Developer suite," 2003. Version 1.2.

[28] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors.," in *DATE*, pp. 586–591, 2005.

[29] D. H. Yoon and M. Erez, "Memory mapped ecc: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 116–127, 2009.

[30] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 385–396, ACM, 2010.

[31] ARM., "Cortex-a57 technique reference manual."

[32] Intel., "Xeon e7 processor - ras servers white paper."

[33] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing&trade; software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pp. 15–24, 2003.

[34] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016.

[35] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016.

[36] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.

[37] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.

[38] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[39] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[40] M. Wolfe, "Beyond induction variables," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, (New York, NY, USA), pp. 162–174, ACM, 1992.

[41] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[42] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[43] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 210–222, IEEE, 2007.

[44] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *International Symposium on Fault Tolerant Computing*, pp. 84–91, 1999.

[45] G. Upasani, X. Vera, and A. Gonzlez, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. 65, pp. 5–18, Jan 2016.

[46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011.

[47] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[48] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[49] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pp. 24–36, June 1995.

[50] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 169–180, IEEE, 2007.

[51] G. Upasani, X. Vera, and A. Gonzalez, "Framework for economical error recovery in embedded cores," in *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pp. 146–153, IEEE, 2014.

[52] G. Reis, J. Chang, and D. August, "Automatic instruction-level software-only recovery," *Micro, IEEE*, vol. 27, pp. 36–47, Jan 2007.