Encore: Low-Cost, Fine-Grained Transient Fault Recovery

Shuguang Feng†, Shantanu Gupta†, Amin Ansari†, Scott A. Mahlke†, and David I. August‡

†Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI
{shoe, shangupt, ansary, mahlke}@umich.edu

‡ Department of Computer Science
Princeton University
Princeton, NJ
august@cs.princeton.edu

ABSTRACT

To meet an insatiable consumer demand for greater performance at less power, silicon technology has scaled to unprecedented dimensions. However, the pursuit of faster processors and longer battery life has come at the cost of reliability. Given the rise of processor reliability as a first-order design constraint, there has been a growing interest in low-cost, non-intrusive techniques for transient fault detection. Many of these recent proposals have counted on the availability of hardware recovery mechanisms. Although common in aggressive out-of-order cores, hardware support for speculative rollback and recovery is less common in lower-end commodity processors. This paper presents Encore, a software-based fault recovery mechanism tailored for these lower-cost systems that lack native hardware support for speculative rollback recovery. Encore combines program analysis, profile data, and simple code transformations to create statistically idempotent code regions that can recover from faults at very little cost. Using this software-only, compiler-based approach, Encore provides the ability to recover from transient faults without specialized hardware or the costs of traditional, full-system checkpointing solutions. Experimental results show that Encore, with just 14% of runtime overhead, can safely recover, on average from 97% of transient faults when coupled with existing detection schemes.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault Tolerance; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Design, Reliability

1. INTRODUCTION

Although it is impossible to build a completely reliable system, hardware vendors attempt to target failure rates that are imperceptibly small. With the course of aggressive technology scaling that has been followed by industry, many sources of unreliability are emerging in commercial processors. One prominent source, and the focus of this paper, is soft errors. Also known as transient faults, they can be induced by a variety of phenomena like electrical noise and high-energy particle strikes that result from cosmic radiation and chip packaging impurities. Additionally, in newly proposed architectures that embrace the principles of stochastic [23] and near threshold computing [5], they can also be the result of extreme timing speculation and/or frequency and voltage scaling.

Traditionally, architects have designed systems that would take periodic checkpoints of processor and memory state. In the event of a soft error the system could rollback to an existing, fault-free snapshot and continue execution (rollback recovery). These highly robust fault recovery solutions have historically also relied on some form of modular redundancy to provide the necessary detection capabilities. Available in spatial and temporal variants, modular redundancy generally involved redundant execution (either on separate hardware or in separate software contexts) followed by detailed comparisons that would identify the presence of a fault [1, 27, 21, 19]. However, the resultant overheads of these coupled detection and recovery schemes, a large component of which was the cost of creating checkpoints, usually relegated their use to to high-end, enterprise systems [4].

These simple yet elegant techniques, having served those in the mission-critical server arena for decades, are not practical outside this niche domain. Although reliability cannot be completely ignored in lower-end systems, they are not usually designed to provide the "five-nines" of fault tolerance capable of sending someone safely to the moon. That said, the overheads associated with these conventional solutions are prohibitively expensive for budget-conscious designers with less demanding reliability requirements. In fact, this is the same argument made by [8], and to a similar extent [3], which argues that most commodity systems do not require reliability guarantees but will settle for probabilistic, best effort, fault tolerance.

This insight has sparked a recent interest in transient fault detection techniques that are able to maintain low runtime overheads by sacrificing a small degree of reliability, focusing primarily on addressing the bulk of faults that are relatively inexpensive to detect [29, 10, 8]. However, these techniques [29, 8] assume hardware provides rollback recovery, arguing that such hardware would already exist to support performance speculation. Although this argument may hold for aggressive out-of-order processors, such hardware support is not present in the majority of low-end commodity systems.

With that in mind, we propose, *Encore*, a software-only solution that seeks to provide probabilistic (best effort) rollback recovery capabilities at minimal costs. Encore was developed to complement emerging probabilistic detection techniques, enabling them

MICRO'11, December 3–7, 2011, Porto Alegre, Brazil.

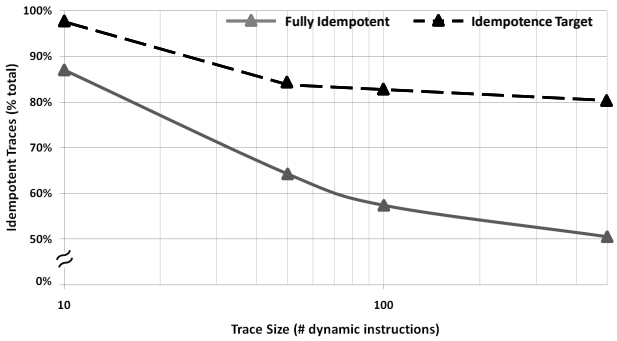


Figure 1: Percentage of dynamic instruction traces that are inherently idempotent as a function of size. The execution traces were extracted from an assortment of SPEC2K and Mediabench workloads. The *Idempotence Target* curve illustrates Encore's goal of exposing, and exploiting, greater degrees of idempotence through intelligent compiler analysis and transformations.

to be deployed in commodity systems without native hardware support for rollback recovery. As an automated, compiler-driven technique, Encore is able to utilize programmable heuristics that allow the end-user to dial in the desired degree of fault-tolerance and therefore only incur as much runtime overhead as they are able to budget. Encore can achieve this behavior by mimicking the same checkpoint, rollback, and re-execute model used by earlier enterprise systems. However, rather than performing, full-system, heavyweight checkpoints, Encore is able to exploit the *statistically idempotent* property of applications to reduce, and in certain situations completely eliminate the overheads required to supply rollback recovery.

At a high-level, an idempotent region of code is simply one that can be re-executed multiple times and still produces the same, correct result. In the context of rollback recovery, this means that at least to the first order, a fault occurring within an idempotent piece of code can be recovered from without any overhead for checkpointing state. This typically means that there cannot exist any paths through the region that can read, modify, and then write to the same (or overlapping) location(s), i.e. no write after read (WAR) dependencies.

To better understand the extent of idempotent code present in an application, Figure 1 shows the distribution of idempotent execution traces across a set of desktop and media benchmarks. Results are shown as a function of the trace size (dynamic instruction length). The surprisingly large percentage of naturally occurring idempotent regions of execution seen in Figure 1 is what initially encouraged the development of Encore. To the first order, the code regions corresponding to the traces that were identified as idempotent could be easily instrumented for rollback recovery with almost no impact on runtime performance. It is important to point out however, that although there is plenty of opportunity present, only a few of these regions actually span an entire function. Most are spread throughout the application, making manual inspection to identify them impractical.

This is not entirely unexpected. With more instructions comes the greater chance that there exists some sequence of instructions that violate the WAR constraints required to maintain idempotence. This intuition is reinforced by the data, which exhibits a sharp drop in the likelihood of being idempotent when moving from traces with just a handful of instructions to those with 50 or more. Lastly, it is also interesting to note that for the traces that do not exhibit full idempotence, many tend to be *nearly* idempotent, i.e., containing only a few offending instructions. Furthermore, these instructions often only occur along statistically unlikely paths. Encore seeks to expose even greater amounts of statistical idempotence by recognizing these properties of application behavior (*Idempotence Target* in Figure 1).

To make exploiting program idempotence feasible, this paper proposes techniques to automate the analysis and instrumentation within compiler optimization passes. We present the algorithms and heuristics developed that enable Encore to carefully partition application code into fine-grained regions with favorable idempotence behavior, and then to instrument them for rollback-recovery. By recognizing the statistically idempotent structure naturally present in many applications, Encore can transparently provide rollback recovery on commodity systems at prices that they can afford. The contributions of this paper are as follows:

- A low-cost transient fault recovery technique for commodity systems without hardware support for aggressive performance speculation.
- New compiler algorithms and heuristics for
  - Automatically identifying candidate idempotent regions in generalized code regions with support for cycles.
  - Selectively trading off recoverability with cost by performing code transformations that leverage application profiling statistics.
- A performance evaluation and analysis of Encore's ability to recover from transient faults with full-system simulations across a diverse set of representative workloads.

2. RECOVERING FROM TRANSIENT FAULTS

Transient fault tolerance requires the ability to *detect* and subsequently *recover* from soft error events. There is no shortage of examples in the literature that address each of these tasks (see Section 6). However, as we have already indicated, recent progress in achieving low-cost probabilistic transient fault detection has not been accompanied by similar advances in fault recovery. Encore, and the remainder of this paper, is concerned with being able to rollback and recover from a transient fault once it has already been detected by a low-cost, low-latency solution like ReStore [29] or Shoestring [8].

Traditional high-reliability systems have chiefly relied upon heavyweight, full-system checkpointing mechanisms to support rollback and recovery. Some high-level characteristics of these traditional techniques are highlighted in Table 1. Compared to these conventional methods, Encore provides recoverability at much finer-granularities without any specialized hardware support. Although it cannot provide guaranteed recovery, the probabilistic nature of Encore allows it to be applied to commodity hardware at dramatically lower costs (in terms of runtime performance and memory footprint).

2.1 Recovery with Fine-grained Re-execution

At the high-level, one of the simplest ways to recover from a transient fault is by re-executing the application from a location far enough back along the control flow graph (CFG) so as to correctly reproduce the data that was corrupted by the fault. With this seemingly straight-forward maneuver, the effects of all but the most insidious transient faults can be completely eliminated. This, of course, assumes that during the initial execution no WAR dependencies overwrote state that could lead to erroneous behavior upon re-execution.

Table 1: Comparison with conventional checkpointing schemes.

Attributes	**Enterprise Recovery** [4, 13, 6, 27]	**Architectural Recovery** [26, 16]	**Encore**
Interval Length	~hours	100-500K instructions	100-1000 instructions
Storage Space	0.5 - 1 GB	0.5 - 1 MB	~10 - 100 B
Checkpoint Time	~minutes	~ms	~ns
Scope	Full System	Processor	Processor
Guaranteed Recovery	Yes	Yes	No
Extra Hardware	Sometimes	Yes	No

Note, that employing this form of fault tolerance requires, in addition to a detection mechanism, the ability to identify the location from which to initiate re-execution, i.e., deciding where the code should rollback to in the event of a fault. Ideally the system would rollback to point just before the fault site and no further. This would ensure correct forward progress while also minimizing the amount of "wasted work," the amount of code that was uneccessarily re-executed. Correctly reasoning about this location requires **1) that you can precisely diagnose where the fault occurred and 2) that you can identify the original path of execution that lead to the fault site**. Although conceptually simple, merely ascertaining 2) without specialized hardware support would require costly, software-based dynamic control flow signature generation [30]. Yet, even having established the original path of execution, accurately locating the site of the initial fault still remains an expensive, if not altogether impractical task.

Figure 2a helps illustrate these two challenges in more detail. Basic blocks $bb_1 - bb_7$ form a small subgraph of code taken from a larger CFG. In this example, a transient fault corrupts an instruction inside basic block bb_4. When the fault is detected at bb_6, we are left with the difficult task of determining where to redirect control to safely rollback and recover. Basic blocks bb_1, bb_2, and bb_4 are all viable options that would lead to safe recovery. However, rolling back to bb_5 would not be far enough, while re-executing from bb_3 could actually lead to other undesirable behavior since it was not on the original execution path. Ultimately, identifying the optimal location to redirect control after a fault is detected requires dynamic information and is undecidable at compile-time.

Yet, if the subgraph in Figure 2a were part of a single-entry, multiple-exit (SEME) region, the decision could be made to conservatively rollback execution to the region entry block (the dominating header), in this case bb'. This would not only free Encore from having to account for which path of execution originally lead to bb_4, but in this example it would also ensure that execution was resumed far enough "back" to regenerate any data corrupted by the original fault. Despite uneccessarily re-executing some code, namely bb_1 and bb_2, this is still a more agreeable alternative to specialized hardware additions or expensive dynamic control flow tracking. Although this effectively resolves challenge 2), obviously in the general case, resuming execution at the top of SEME regions is only effective against faults that are detected within the same region that they occur. Fortunately, for large SEME regions the probability of a fault being detected after control has left the region is reduced. Details regarding how Encore attempts to form these large SEME regions are described in Section 3.3.

2.2 The Role of Idempotence

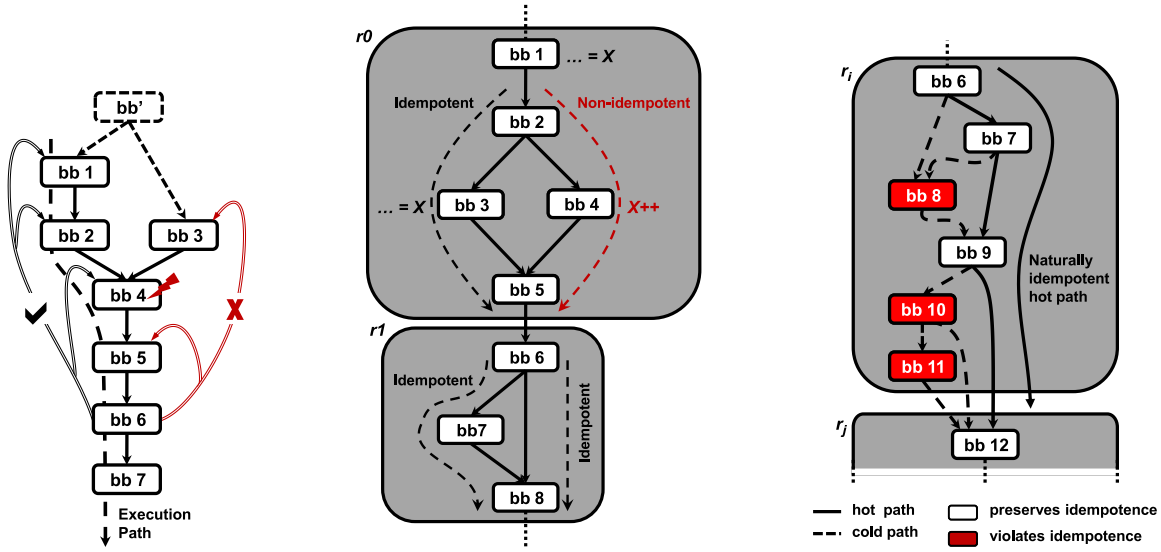
Figure 2b illustrates how recognizing idempotent regions can greatly reduce the overhead required to provide rollback recovery. Since idempotent regions by definition contain no WAR dependencies, they are attractive candidates for Encore's re-execution based fault recovery. In this example, since all paths through region r_1 are idempotent, it is more desirable than r_0, for which execution down the path containing bb_4 can be non-idempotent. Relying on conventional, full-system checkpointing schemes to ensure that a region like r_1 could be re-executed would be using a sledgehammer to crack the proverbial nut. Because the region is naturally idempotent, Encore can simply redirect all fault detection events initiated anywhere within the region to bb_6, the header of r_1. It is important to note here that although r_0 is not idempotent, if the increment of variable X in bb_4 were the only instruction violating idempotence then selectively checkpointing X prior to the increment would transform r_0 into a readily recoverable region. Small, cost-efficient transformations like these, described in greater detail in Section 3, are what enable Encore to achieve low-cost rollback recovery.

Lastly, Figure 2c shows an actual subgraph taken from *175.vpr*, a benchmark from the Spec2000 benchmark suite. It corresponds to a slice of the CFG from the function *try_swap*, which is the hottest function within the application (accounting for roughly half of its execution time). The details of the basic blocks and the surrounding CFG have been abstracted away for clarity. The shaded basic blocks, bb_8, bb_{10}, and bb_{11} are locations where the idempotence of the region can be violated. The code within these basic blocks is responsible for memory allocation to dynamic variables. Consequently, these are only executed the first time *try_swap* is called. For the remaining invocations of *try_swap*, the path through basic blocks bb_6, bb_7, and bb_9 dominates the execution time of r_i. This suggests that although region r_i is not strictly speaking idempotent, it does exhibit idempotent *behavior* the vast majority of the time. This *probabilistic* idempotence is yet another property of applications that Encore can exploit to reduce its overheads.

Admittedly the notion of idempotence is not new. For example, Kim et al. [9] leveraged idempotent properties of inner loops in Fortran applications to minimize the instances of storage overflows in a speculative execution system. However, relying on this property for low-cost, transient fault recovery in a systematic fashion has not yet been fully addressed. A recent proposal by Kruijf et al. [3] resorted to manually inspecting and modifying source code to take advantage of the function-wide idempotence and fault tolerant properties of multimedia and data-mining applications. Although their work is in the same spirit as Encore, and a great step in the right direction, utilizing domain/application-specific algorithmic knowledge to identify and condition candidate functions significantly limits the applicability of the approach. Establishing a generalized methodology for exploiting fine-grained, often statistical, idempotence to enable low-cost rollback recovery was the purpose of developing Encore.

3. ENCORE

Achieving low-cost transient fault recovery involves identifying naturally occurring regions of code that are amenable to re-execution, and judiciously sacrificing reliability to maintain low



(a) Rollback options **(b)** Significance of idempotence **(c)** Significance of hot paths

Figure 2: Challenges and opportunities for fine-grained transient fault recovery via rollback and re-execution. (a) enumerates potential rollback destinations that execution can be redirected to once a fault, striking at bb_4 is detected, at bb_6. Ideally bb_1 and bb_3 would share a common predecessor bb' that could serve as the rollback destination for all faults that are detected within the region. (b) highlights how idempotence violating instructions might constrain which code regions can actually be efficiently recovered. (c) depicts how otherwise non-idempotent regions can still frequently exhibit idempotent behavior along their hot paths. The region shown in (c) is taken from the CFG corresponding to the dominant hot function in *175.vpr*.

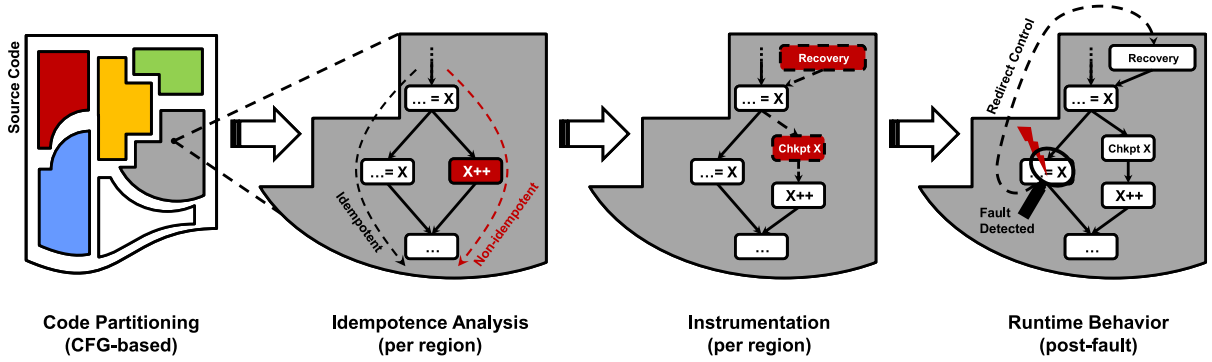


Figure 3: High-level Encore vision. At compile-time, application code is partitioned into SEME regions that are subsequently analyzed and instrumented to enable low-cost rollback recovery from transient faults. Flexible heuristics enable Encore to refine the partitioning and instrumentation passes, customizing their behavior to achieve the desired tradeoff between reliability and performance overheads.

overheads. Figure 3 illustrates the different high-level components of the Encore vision. Encore is designed as a series of compiler passes that analyzes, refines, and ultimately instruments the code with rollback recovery "hooks" that are coupled with a detection mechanism at runtime.

To start with, the application source code (specifically the CFG) is partitioned into SEME regions. These regions are then analyzed to determine their idempotence properties (Section 3.1). The results of this analysis are then used to instrument the regions for rollback recovery (Section 3.2) as well as refine the initial region partitioning (Section 3.3) using heuristics developed to maximize rollback coverage while maintaining acceptable overheads (Section 3.4). Lastly, when faults are detected at runtime, execution is redirected to a recovery block that restores any non-idempotent state from lightweight checkpoints before releasing control to the header block of the corresponding region.

3.1 Identifying Inherent Idempotence

Before the discussion proceeds any further, to help avoid any ambiguity that may arise, a few terms that will be used throughout this paper are explicitly defined below.

Region: in its unqualified form this will refer to SEME regions, a subgraph of basic blocks connected in the program CFG that contains a single (entry) block that dominates all other blocks and zero or more exiting blocks (basic blocks with branches to outside the region).

Reachable Store[1]**:** at a given point p (i.e., basic block) in the CFG, a *reachable store*, relative to p, is a store instruction that could potentially execute after control has passed through p.

Guarded Address: with respect to a given point p, a *guarded address* is one that is *guaranteed* to be overwritten by a store instruction prior to reaching p, along all possible paths to p.

[1]This should not to be confused with the concept of reaching definitions commonly used in dataflow analysis.

Exposed Address: with respect to a given point p, an *exposed address* is an address that *may* be referenced by an unguarded load prior to reaching p. A load l is guarded if, and only if, the address referenced by l is already a guarded address with respect to the location of l.

Inherent Idempotence: a property of a region indicating that the region contains no WAR sequences to the same address that could prevent it from being safely re-executed during rollback recovery without undesired side-effects.

These definitions, and the text throughout this paper, only make reference to load and store instructions. This is done simply in an effort to improve readability. In reality, all instructions that can potentially reference and/or modify memory are considered during the idempotence analysis. Additionally, register state is also initially ignored in the analysis and will be treated separately in Section 3.2.

3.1.1 Path Insensitive Analysis

Determining the idempotence of a region, r, begins by generating the region-wide *reachable store* ($\mathbb{RS}$), *guarded address* ($\mathbb{GA}$) , and *exposed address* ($\mathbb{EA}$) sets for all basic blocks $bb_i \in r$. This is done by performing multiple post-order traversals of the region's CFG. For the time being, the details surrounding these regions will be ignored with the exception of saying that they are limited to SEME subgraphs of basic blocks. Initially, the discussion will also be limited to acyclic regions. Cycles (i.e., loops) will be incorporated in Section 3.1.2 once the initial acyclic algorithm has been described.

The initial post-order traversal begins from the entry block to the region. As each basic block (bb_i) is encountered, Equation 1 is used to update the corresponding reachable store set, $\mathbb{RS}_{bb_i}$. Next, all edges in r are reversed and multiple post-order traversals are performed on this new subgraph starting from each of the region's exiting blocks. As each basic block, bb_i, is encountered during these "reverse" post-order traversals, Equations 2 and 3 are used to update the corresponding guarded address and exposed address sets. Note that the guarded address set, $\mathbb{GA}_{bb_i}$, must be updated before the exposed address set, $\mathbb{EA}_{bb_i}$. Furthermore, the set subtraction operation, "$-$", used in these and subsequent equations is supplied with standard, conservative, static memory alias analysis techniques [2].

$$\mathbb{RS}_{bb_i} = \bigcup_{\forall bb_j \in \mathbb{C}_{bb_i}} \left(\mathbb{RS}_{bb_j} \cup \mathbb{AS}_{bb_j} \right) \bigcup \mathbb{AS}_{bb_i} \tag{1}$$

where $\mathbb{RS}_{bb_i}$ is the set of reachable stores at bb_i; $\mathbb{C}_{bb_i}$ is the set of bb_i's children; and $\mathbb{AS}_{bb_i}$ is the set of all stores within bb_i itself.

$$\mathbb{GA}_{bb_i} = \bigcap_{\forall bb_j \in \mathbb{C}_{bb_i}} \left(\mathbb{GA}_{bb_j} \cup \mathbb{AS}_{bb_j} \right) \tag{2}$$

$$\mathbb{EA}_{bb_i} = \left(\bigcup_{\forall bb_j \in \mathbb{C}_{bb_i}} \mathbb{EA}_{bb_j} \right) \bigcup \left(\mathbb{EA}^{local}_{bb_i} - \mathbb{GA}_{bb_i} \right) \tag{3}$$

where $\mathbb{GA}_{bb_i}$ is the set of guarded addresses at bb_i; $\mathbb{EA}_{bb_i}$ is the set of exposed addresses at bb_i; $\mathbb{EA}^{local}_{bb_i}$ is the set of all addresses referenced by loads in bb_i that are not preceded by a store, also within bb_i, to the same address, effectively the set of *local* exposed addresses for bb_i.

Once all the basic blocks within the region have been processed, and the associated $\mathbb{RS}$, $\mathbb{GA}$, and $\mathbb{EA}$ sets have been generated, Equation 4 can be used to determine whether r is idempotent. It essentially checks if idempotence can be violated by executing a basic block bb_i along *any* possible path through r.

Region r is idempotent **iff**

$$I(bb_i) = \text{true}, \forall bb_i \in r \quad \text{where,}$$

$$I(bb_i) = \begin{cases} \text{true}, & \textbf{iff } \mathbb{EA}_{bb_i} \cap \mathbb{RS}_{bb_i} = \emptyset \\ \text{false}, & \text{otherwise} \end{cases} \tag{4}$$

Figure 4 illustrates how this path insensitive idempotence analysis is performed on a small example region. Figure 4 highlights the potential WAR dependencies (#,⋆,@,+) that exist between the relevant instructions 1-12. Figure 4b shows how the data structures in Equation 1 and Equations 2-3 are populated during the in-order traversal and reverse in-order traversal of the subgraph, respectively. Although there are four WAR dependencies that exist within this region, Encore is able to single-out the one dependence that can actually violate idempotence during runtime, the dependency between instructions 7 and 10 (⋆).

Admittedly, identifying idempotence in this manner leads to conservative answers. Equation 4 does not account for correlations among branches between basic blocks and consequently may categorize regions as non-idempotent because of paths that can never be realized given the design of the application. However, augmenting any compiler analysis with path sensitive information is generally considered an intractable problem [7]. Nevertheless, despite this limitation, the algorithm proposed here is efficient, scalable, and sufficiently accurate.

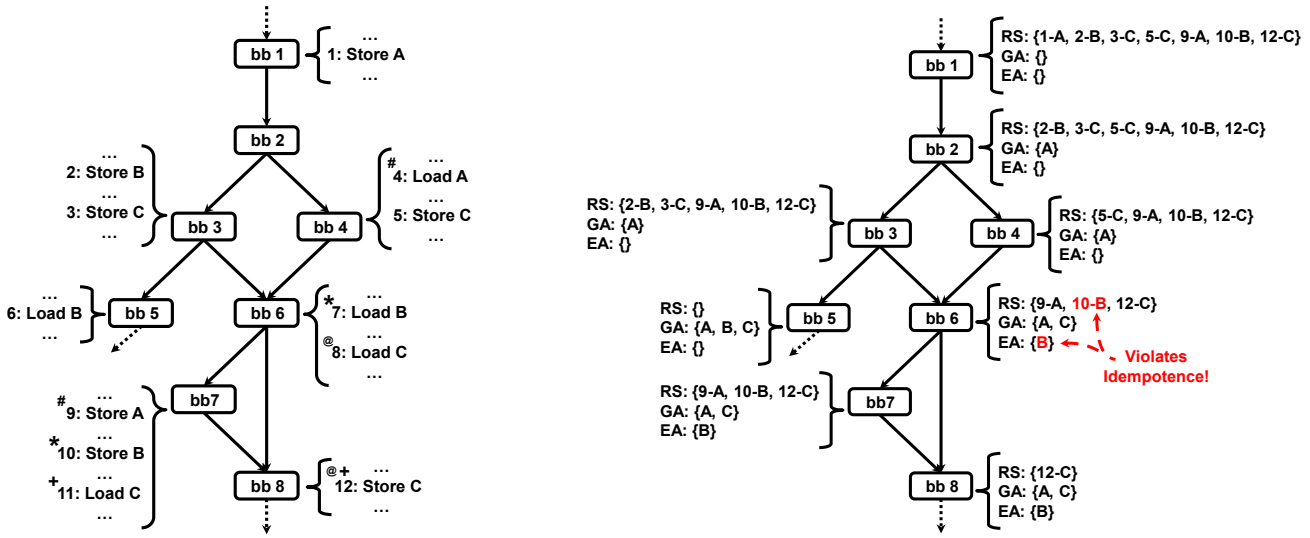
3.1.2 Incorporating Cycles

Up to this point, the analysis has focused on acyclic regions. Introducing cycles can dramatically complicate issues. To help maintain the scalability of the analysis, loops within a region are treated in a hierarchical manner. Initially, prior to idempotence analysis, a conventional compiler pass ensures that all loops are in a canonical form [3] (i.e., single header block and no side-entries). Next, whenever the boundaries of loops are encountered (header blocks during the forward post-order traversals and exiting blocks during the reverse post-order traversals) no attempt is made to enter the body of the loop. Instead, previously generated meta-information for each loop that summarizes the net impact of all the memory accesses within the loop is used to update idempotence data structures. This enables entire loops to be treated as if they were simply another basic block.

When analyzing regions containing cycles, all loops are processed first. If nested loops are present, they are analyzed from the inner-most loop outward. When processing an (inner-most) loop, the constituent basic blocks can initially be analyzed as if they were just a simple acyclic region. The guarded address and exposed address sets for each basic block within the loop are generated as described in Section 3.1.1. However, given the cyclic nature of loops, effectively all stores are potentially reachable from any point within (possibly across iterations). Therefore, the set of reachable stores for each basic block within a loop l, $\mathbb{RS}^l_{bb_i}$, is equivalent to the set of all stores within the loop, $\mathbb{AS}^l$. Defining $RS^l_{bb_i}$ in this fashion ensures that all cross-iteration WAR dependencies are accounted for. Once the loop-wide reachable store, guarded address, and exposed address sets have been generated for all basic block within the loop, the loop can be treated as any other region and

[2] Extending Encore to use more aggressive dynamic memory profiling is a promising area of future work.

[3] Not all cycles within a CFG can be converted into a canonical form. In these rare cases, Encore does not instrument the parent region for rollback recovery.



(a) A SEME region with four potential idempotence violating WAR dependencies: instructions 4 and 9 (#); 7 and 10 ($\star$); 8 and 12 (@); and 11 and 12 (+).

(b) Results of the Encore analysis that identifies the single dependency that actually requires checkpointing (instruction 10) to maintain idempotence. The number-letter pairs in the $\mathbb{RS}$ sets indicates the instruction number and destination address for the corresponding store.

Figure 4: Example illustrating Encore's idempotence analysis. Only the relevant instructions within each basic block are shown in (a). (b) shows how the data structures in Equation 1 and Equations 2-3 are populated during the in-order traversal and reverse in-order traversal of the subgraph, respectively.

idempotence can be assessed using Equation 4. Once loop idempotence has been determined, the next step is to generate the meta-information associated with the loop.

The goal of loop-wide meta-information is to capture and expose loop-wide memory side-effects to simplify subsequent region analyses. This allows the entire loop itself to be treated effectively as a simple basic block. The contents of this data structure are enumerated below and are used in an analogous fashion to their basic block counterparts.

Loop-wide reachable stores, $\mathbb{RS}_{l_i}$: the set of all stores that could potentially execute if control ever enters loop l_i. The cyclic nature of the loop ensures that $\mathbb{RS}_{l_i} = \mathbb{RS}^{l_i}_{header} = \mathbb{AS}^{l_i}$, where $\mathbb{RS}^{l_i}_{header}$ is the set of reachable stores for the loop header and $\mathbb{AS}^{l_i}$ is the set of all stores within the l_i.

Loop-wide guarded addresses, $\mathbb{GA}_{l_i}$: the set of all addresses that are that are guaranteed to be overwritten if and when loop l_i is executed. Since loops can have multiple exiting blocks this is effectively the intersection of all guarded addresses across all exiting blocks of l_i. In other words $\mathbb{GA}_{l_i} = \bigcap_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{GA}_{bb_i}$, where $\mathbb{X}_{l_i}$ is the set of exiting blocks for l_i.

Loop-wide exposed addresses, $\mathbb{EA}_{l_i}$: analogous to the definition of local exposed addresses for basic blocks, the exposed address set for l_i is the set of all addresses that may be referenced by an unguarded load along all possible paths through l_i. This is equivalent to the union of the $\mathbb{EA}$ sets across all the exit blocks, $\mathbb{EA}_{l_i} = \bigcup_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{EA}_{bb_i}$.

With the loop-wide meta-data populated for all loops within the CFG, analysis of any arbitrary region can proceed and treat loops hierarchically as just another basic block. The region traversals simply "step-over" loops whenever they are encountered and update idempotence data structures with the loop-wide meta-data.

3.2 Instrumentation

Once the idempotence of the various regions within an application has been determined, the next step is to identify whether inherently non-idempotent regions can be efficiently (with low runtime overheads) transformed into idempotent regions. For Encore, this transformation is achieved by instrumenting offending non-idempotent regions with instructions to checkpoint state that may otherwise be overwritten upon re-execution.

While performing the idempotence checks in Section 3.1, all offending stores that violate Equation 4 are recorded in a checkpoint set, $\mathbb{CP}$, associated with every region. If Encore decides to enable recovery (see Section 3.4), on a non-idempotent region, r_i, it will proceed to instrument each store, $s \in \mathbb{CP}$, with checkpointing instructions that checkpoints the data (and corresponding address) just prior to s. Additionally, in order to ensure that no WAR register dependencies violate idempotence, all live-in (with respect to r_i) registers that are overwritten within r_i are also checkpointed upon entering the region. The identification of register live-in values is a standard analysis in modern compilers and is omitted due to space constraints. Given the small amount (see Section 5) of storage required, the checkpointds are placed in a specially reserved region of the stack.

After instrumenting a region with the necessary checkpointing instructions, all that remains is to create a *recovery block*–the destination of all rollbacks, initiated if and when a fault is detected within the region. Encore instruments the header of each region with a simple store that updates a dedicated memory location with the address of the corresponding recovery block each time control enters the region. Existing low-cost, software-based detection schemes [29, 8] can be easily modified to redirect control to the address stored in this reserved memory location when a fault is detected.

Within the recovery block, all the previously checkpointed state (registers and memory) are restored before redirecting control back to the region header. Although this additional instrumentation also contributes to runtime overheads, it is only executed upon the detection of a transient fault. In fact, the conditional rollback to the

recovery block can also be amortized with the cost of the detection scheme. Further optimizations for reducing the overhead of this selective checkpointing are described in Section 3.4.

3.3 Region Formation

Having discussed how idempotence is analyzed (and enforced if necessary), we can now discuss how the CFG is initially partitioned, and subsequently refined, into these segments. Candidate region formation is done in Encore by building upon traditional interval analysis. In general, an interval is essentially a loop plus acyclic "tails" that dangle from the blocks within the loop. In practice, the initial loop at the "top" of the interval may not exist (i.e., an interval can simply be a small SEME subgraph that shares a single dominating header node).

Since it is a standard pass within most modern compilers, this section omits the details of how this *initial* partitioning is achieved and focuses on the subsequent refinement steps. However, there are two properties of this partitioning that are important to keep in mind: **1) All intervals are by definition SEME regions and 2) interval partitioning can be applied recursively**.

The first property of interval partitioning greatly simplifies the process of recovery. By ensuring that all regions are SEME, Encore can avoid the costly task of tracking dynamic execution paths (see Section 2.1). This property is what allows Encore to safely insert the recovery block described in Section 3.2 just before the region's header. Irrespective of which path lead to the actual fault site, redirecting control to this recovery block will ensure that it can be corrected.

The second property suggests that once a CFG is partitioned into intervals, the intervals themselves form an *interval graph* that can also be partitioned into intervals. Encore exploits the second property of interval partitioning to create candidate regions with varying sizes. By controlling the size of the regions, Encore is able to effectively manage the trade-off between fault tolerance and performance overhead. Generally speaking, the larger the region that Encore attempts to recover from, the greater the likelihood that the region is not inherently idempotent. Recall that non-idempotent regions require instrumentation to enable safe re-execution, which contributes to the overall runtime overhead. On the other hand, the larger the region, the more likely that a transient fault striking within the region will be detected before control exits the region and the fault is no longer recoverable. Section 3.4.2 will discuss how heuristics are used to identify the appropriate region size given a budget for acceptable performance overhead.

At this point, one might contend that merging two regions, r_i and r_j to form a larger region r' (with r_i preceding r_j) may not necessarily incur additional costs to enforce idempotence within r'. In fact, if r_j were non-idempotent, the fact that it is preceded by r_i could actually reduce the amount of checkpointing required in r_j if the idempotence violating instructions within r_j only referenced locations within the guarded address set for r_i. Although in principle this is correct, in practice these scenarios are rare and for the majority of cases, fusing regions together was not an effective means of *reducing* performance overheads.

3.4 Encore Heuristics

This section focuses on the heuristics used to glean the best reliability versus performance trade-offs from Encore. First, we discuss how profiling information can be used to statistically prune basic blocks from the idempotence analysis followed by the heuristic to identify which regions are chosen as candidates for rollback recovery.

3.4.1 Relaxing Idempotence Criteria

Unlike techniques targeting mission-critical systems that must provide guarantees on recoverability, Encore is free of such constraints and is at liberty to utilize profile-based, not necessarily provable, analyses.

Presented with this flexibility the algorithm described in Section 3.1 can selectively ignore any basic blocks that do not meet a certain "liveness" criteria. As previously mentioned, the idempotence determination made by Equation 4 is necessarily conservative since it accounts for all paths through the region. By exploiting profiling information, Encore can now exclude basic blocks that are along paths that have low probabilities of being traversed when updating $\mathbb{RS}$, $\mathbb{GA}$, and $\mathbb{EA}$ sets for each basic block. More formally, this means that Equations 1, 2, and 3 can be re-formulated limiting the union and intersection operations, which originally operated over all the children of a basic block, $\mathbb{C}_{bb_i}$, to a subset set of children $\mathbb{C}'_{bb_i}$ where the *dynamically-dead* children have been pruned away. The degree to which Encore filters these rarely executed basic blocks from its idempotence analysis is controlled by a heuristic parameter P_{min}. Any basic block with an execution probability less than P_{min} is selectively ignored.

3.4.2 Region Selection

Another opportunity for trading off reliability for performance is in the area of region selection. Encore can selectively decide **1) which regions should be instrumented for recovery (Section 3.2) as well as 2) when to terminate the process of merging existing intervals to form larger regions (Section 3.3)**. Exposing the heuristic parameters that control these decisions allows Encore to be customized by system designers.

Determining whether protecting a region is actually a profitable endeavor, is fairly straightforward. For inherently idempotent regions, the answer is almost always yes. The cost of updating the address for the current recovery block is negligible for all but the smallest possible regions. However, for small, non-idempotent code portions, the overhead incurred to preserve idempotence can potentially make it more attractive to simply concede fault coverage for those regions. To account for this possibility, only regions that have reasonable cost-to-coverage ratios are instrumented for selective checkpointing and rollback recovery. In other words $Coverage(r_i)/Cost(r_i) > \gamma$ must be satisfied for every candidate region, where γ is a heuristic threshold. The length of the hotpath through r_i is used as a compile-time surrogate for coverage, while the ratio of checkpointing instructions required to the number of total instructions within the hotpath is used as an estimate for cost.

Although some regions may initially be undesirable, Encore has the ability to merge adjacent regions to form larger, possibly more attractive candidates. Since merging regions has the potential to incur additional checkpointing instructions, it is only performed if the additional cost, $\Delta Cost$, is offset by the improvement in overall reliability, $\Delta Coverage$. Given two regions, r_i and r_j that are to be fused into r',

$$\Delta Coverage = \frac{Coverage(r')}{Max(Coverage(r_i), Coverage(r_j))} \quad (5)$$

At a given cost, this definition for $\Delta Coverage$ ensures that fusing two similarly sized regions, which earns more cost-effective returns in terms of improved reliability, is preferred over merging a large and a small region. Ultimately only when $\Delta Coverage/\Delta Cost > \eta$ does Encore consider merging existing regions. Small values of η predisposes the system to try and create the larger regions in

pursuit of greater reliability while larger η's shift the focus toward minimizing performance overheads.

4. EXPERIMENTAL METHODOLOGY

As with all reliability schemes dealing with transient faults, an ideal evaluation of Encore would involve electron beam experiments on real hardware running real-world applications. Yet, given limited resources an acceptable alternative has been statistical fault injection (SFI) on detailed system models (architectural, microarchitectural, RTL, etc.). Statistics related to fault masking, and to a lesser extent fault detection, can be highly dependent on the details of the underlying hardware. Consequently, for the full system results shown in Section 5.4, fault masking was determined by a series of Monte Carlo experiments that injected faults into a low-level Verilog model of an ARM926 embedded processor. Transient faults were injected into state elements and combinational logic and the overall average hardware masking rate was quantified.

However in contrast, the more important figure of merit for evaluating Encore, the amount of application code that can be cheaply re-executed, is far more sensitive to program structure and to some degree is (micro)architecturally neutral. The remaining details of the experimental methodology and the analytical model developed to evaluate Encore are described below.

4.1 Compilation Framework

The compiler analysis and instrumentation passes described in Section 3 were implemented in the LLVM compiler. An assortment of Spec2000 integer (SPEC2K-INT), Spec2000 floating point (SPEC2K-FP), and Mediabench applications serve as the representative workloads for our experiments. All applications were compiled with standard -O3 optimizations.

4.2 Recoverability Coverage Model

As previously stated, Encore only targets the *recovery* aspect of processor reliability. Within this context, we define *recoverability coverage* as the percentage of application code that can be safely re-executed in the presence of a fault. In the case of Encore, this coverage is effectively equivalent to the percentage of execution time that is spent within dynamic code regions that are inherently idempotent or have been instrumented to preserve idempotence.

4.2.1 Impact of Detection Latency

Assume that the hot path through region r consists of instructions $i_0, i_1, ..., i_n$. If a fault corrupts the output of i_s (where $0 \leq s \leq n$) and the detection latency for the system is l instructions, Encore can recover from this fault if $s + l < n$. To account for the detection latency of the system we calculate a latency scaling factor α according to Equation 6.

$$\begin{aligned}\alpha_{r_i} =& Pr(s + l < n),\ \forall s \in [0, n], \forall l \in [0, D_{max}] \\ =& \int_0^n \int_0^s f(l)g(s)dlds \end{aligned} \tag{6}$$

where,

α_{r_i} : is the scaling factor associated with region r_i that accounts for detection latency.

n : is the number of (dynamic) instructions along the hot path through region r_i.

s : is a random variable, distributed over the interval $[0, n]$, representing the instruction (number) at which a transient fault occurs.

l : is a random variable, distributed over the interval $[0, D_{max}]$, representing the detection latency of a system with a maximum latency of D_{max}, measured in terms of instructions.

$Pr(s + l < n)$: the probability that a fault at instruction s is detected inside the boundary of region r_i.

$f(l)$: is the probability density function corresponding to the the detection latency of the system.

$g(s)$: is the probability density function corresponding to the fault sites within region r_i.

For the full-system fault coverage results presented in Section 5.4, we use an uniform distribution of fault sites, which assumes that every dynamic instruction over the course of an application's runtime has the same probability of being "struck" by a transient fault. This is consistent with the general body of reliability work that use uniform distributions to guide the selection of fault locations and times during SFI simulations. Similarly, the full-system results in Section 5.4 also assumes a uniform distribution of fault detection latencies. This is consistent with the detection latencies exhibited by techniques like Restore [29] and Shoestring [8], that exploit the anomalous software behavior that manifests in the wake of a soft error event. With these assumptions, Equation 6 can be re-written as Equation 7.

$$\begin{aligned}\alpha_{r_i} =& \int_0^n \int_0^{min(s, D_{max})} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dlds \\ =& \begin{cases} 1 - \frac{D_{max}}{2n}, & n \geq D_{max} \\ \frac{n}{2D_{max}}, & n < D_{max} \end{cases} \end{aligned} \tag{7}$$

4.3 Assumptions and Limitations

Performance modeling: The runtime performance overheads in Section 5.3 are presented in terms of dynamic instructions. The use of dynamic instructions may appear at first to be a less desirable alternative to running natively on a real machine and/or a microarchitectural simulator. However, this decision allows us to abstract away the details of the underlying hardware and present architecture-neutral results.

Faults corrupting control and/or address calculation: Both address faults that result in (over)writing data to erroneous locations, and faults that lead to deviations from the correct control path are not recovered by the current Encore system. Fortunately, these categories of faults also happen to be those that are most readily detected by low-cost detection mechanisms [29, 8]. These faults often result in highly visible symptoms, and are typically detected before they propagate to memory and/or divert control flow (i.e., before they become unrecoverable).

Multi-threaded Applications: We do not evaluate Encore in the context of multi-threaded workloads. However, the idempotence analysis described in Section 3 could be extended to handle multi-threaded applications. Encore's efficacy in these systems would be in large part dependent on the power of the memory analysis infrastructure. Steps would also need to be taken to ensure that rollback recovery instrumentation did not violate the semantics of synchronization events, either with manually annotated atomic regions or additional compiler analysis.

5. EVALUATION AND ANALYSIS

This section presents the quantitative evidence demonstrating Encore's ability to provide affordable rollback recovery. For the

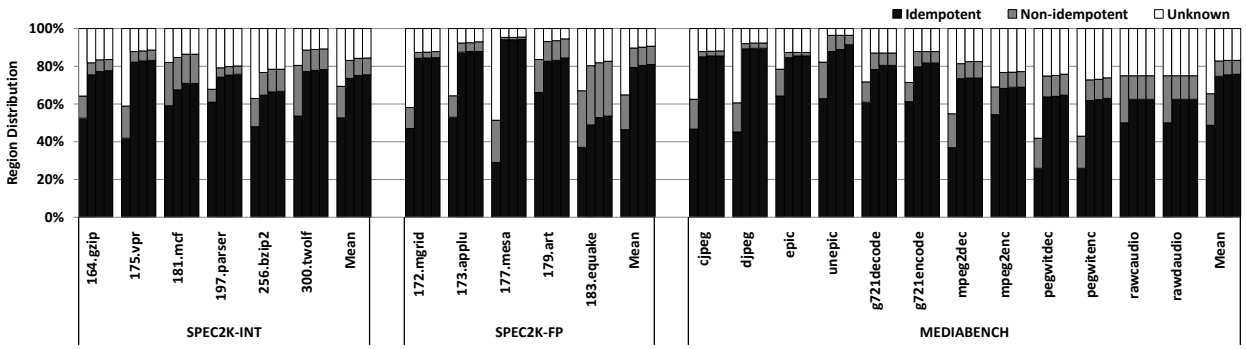


Figure 5: Inherent region idempotence as a function of P_{min}. From left to right, the columns illustrate the fraction of regions within each application that is inherently idempotent for different values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. With $P_{min} = \emptyset$, the left-most column for each application depicts the idempotence breakdown when no dynamically-dead code is pruned from the analysis. The *Unknown* segments correspond to portions of the application source code that could not be analyzed by Encore.

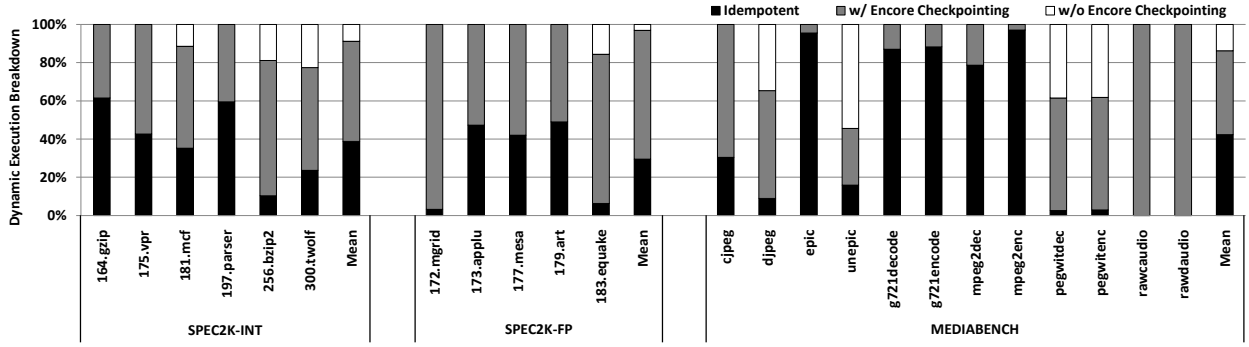


Figure 6: Breakdown of dynamic execution time. For each application the stacks represent the fraction of execution time spent within regions of the code that were inherently idempotent, non-idempotent but instrumented with selective checkpointing by Encore, and non-idempotent but too costly to checkpoint.

data presented in this section values for γ and η (Section 3.4.2) were empirically derived for each application to target an acceptable maximum runtime overhead of $\sim 20\%$.

5.1 Region Idempotence

Figure 5 examines the inherent idempotence of candidate recovery regions as a function of P_{min}. From left to right, the different columns for each application correspond to the idempotence for different values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. The different segments represent the fraction of regions that were identified to be inherently *idempotent*, *non-idempotent*, and *unknown*. Unknown regions contained code that Encore's compiler analysis was unable to process. This consisted of regions with calls to functions (mainly system and library function calls) for which relevant alias analysis information could not be easily obtained, preventing idempotence determinations.

Note that, as expected, the fraction of regions that are deemed idempotent grows as more dynamically-dead code is pruned (increasing values of P_{min}). Furthermore, nearly all of the benefit can be garnered by simply pruning the code that was *never* executed during profiling runs ($P_{min} = 0.0$). This suggests that a good portion of the instrumentation optimizations described in Section 3 can be achieved without incurring any measurable risk. For the remainder of this section, all data presented is for $P_{min} = 0.0$.

Not surprisingly the SPEC2K-FP and Mediabench applications exhibit slightly better idempotence behavior than the SPEC2K-INT benchmarks. As suggested by Kruijf et al. [3], the multimedia and embedded-type codes typical of emerging applications tend to have fewer memory side-effects, great for idempotence. However, it is interesting to note that at least in terms of static code, on average, the extent of idempotence present across all three benchmark suites are comparable. It is encouraging to observe that even in control-heavy SPEC2K-INT applications, there is still a considerable fraction of code that is *inherently* idempotent. On average, across all applications, 49% of regions are inherently idempotent without pruning and 75% are idempotent with $P_{min} = 0.0$. This suggests that Encore would only have to insert minimal, if any, checkpointing instrumentation code for most applications to enforce idempotence.

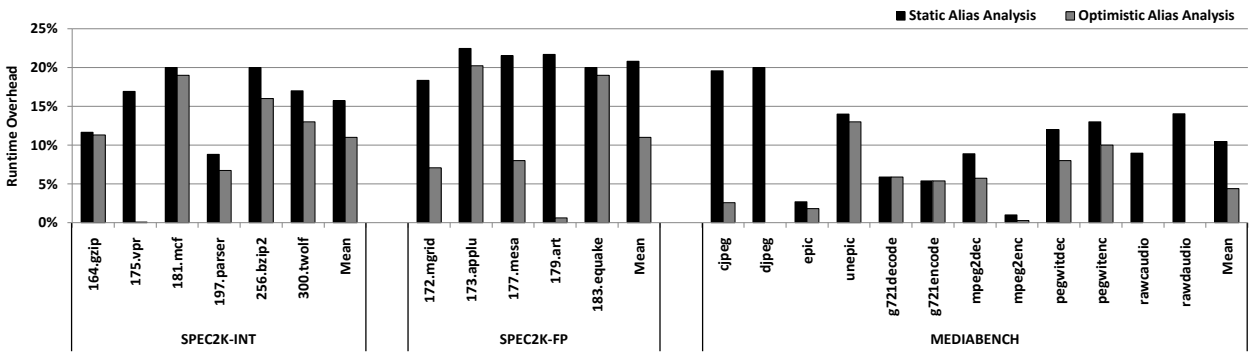
5.2 Dynamic Execution Breakdown

Figure 6 takes a closer look at the execution of these workloads and presents the breakdown of execution time (calculated in terms of the percentage of total dynamic instructions) spent in different regions of the code. The segments labeled *w/ Encore Checkpointing* correspond to execution within regions that were non-idempotent but were selectively instrumented to preserve idempotence, while *w/o Encore Checkpointing* represents execution time spent in regions that were inherently non-idempotent but were too expensive to checkpoint. Execution time represented by the *w/o Encore Checkpointing* segment corresponds to lost recoverability coverage.

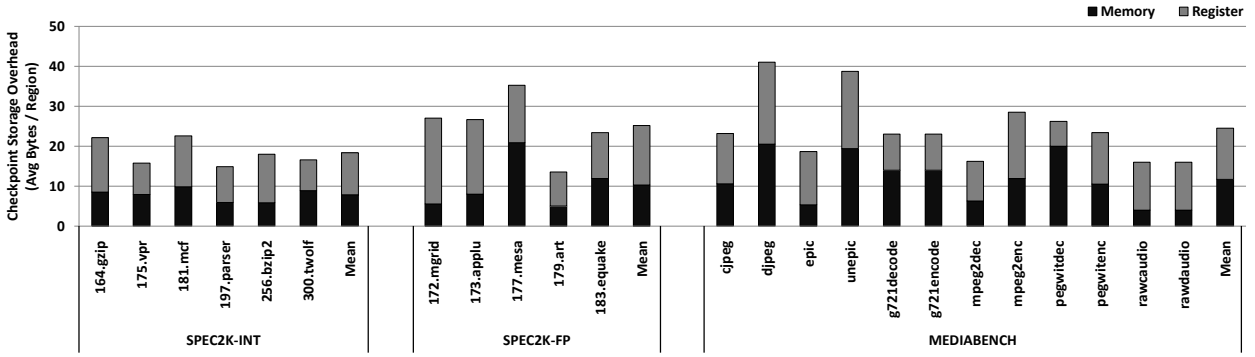
Despite having roughly the same amount of idempotent static code, the SPEC2K-FP and Mediabench workloads spent significantly more runtime within naturally idempotent and easily checkpointed code regions, i.e., Encore recoverable code.

5.3 Overheads

Figure 7a reports the runtime overheads corresponding to the recoverability coverage results reported in Figure 6. The *Static*



(a) Runtime performance overhead. The *Static Alias Analysis* bar reports the current runtime overheads for Encore while the *Optimistic Alias Analysis* bar provides an approximate lower-bound for future Encore designs that could utilize more robust (potentially dynamic) alias analysis frameworks.



(b) Overheads reported as the average number of bytes required per region to store checkpointing information. The stacked breakdowns highlight the contributions from memory and register checkpointing.

Figure 7: Encore runtime and storage overheads.

Alias Analysis bar shows the current runtime overheads for Encore while the *Optimistic Alias Analysis* bar provides an approximate lower-bound for future Encore designs that could utilize more robust alias analysis frameworks. Encore must currently checkpoint a significant number instructions because the limited alias analyses available to it cannot effectively disambiguate their addresses. Future, systems with more powerful, potentially dynamic, alias analyses could determine that a large fraction of these currently idempotence-violating instructions are in fact innocuous.

Nevertheless, even in its current form Encore only imposes a 14% runtime overhead, on average, across all benchmarks. Although Encore was given a 20% performance budget, obviously not all workloads incurred this overhead. Some, like *172.mgrid, epic,* and *mpeg2enc* were able to instrument all regions for recovery without requiring the full performance budget. Others, like *181.mcf* and *177.mesa* were not able to meet (approach) the 20% target without incurring significant reductions in recoverability coverage.

Similarly, Figure 7b reports the storage overheads required to hold the selective checkpointing information generated by Encore. Note that for register checkpoints, the checkpointing information only consists of the register data, whereas for memory checkpoints both data and address must be stored to enable proper recovery. On average, Encore must only store 24 bytes of information per region, orders of magnitude less than the memory footprint of conventional, full-system checkpointing techniques.

5.4 Full-system Reliability

Lastly, Figure 8 examines the full-system fault tolerance that can be achieved by a commodity system augmented with Encore for rollback recovery and a Shoestring-like mechanism for fault detection. The *Masked* segments represent the fraction of transient faults that are naturally masked by the underlying hardware and do not require any additional intervention. As mentioned in Section 4, this masking rate was identified with Monte Carlo-based SFI experiments on a Verilog model of an ARM926 commodity processor [2].

In addition to the "free" fault coverage due to hardware masking, the fraction of faults the system can also recover from with Encore-enabled rollback recovery is represented by the *Recoverable w/ Idempotence* and *Recoverable w/ Encore Checkpointing* segments. Portions of the bars labeled *Not Recoverable* correspond to faults that either occurred within regions of the application code that Encore chose not to protect, or were the result of faults that were not detected before execution had already left the region containing the original fault site.

The different columns in Figure 8 correspond to the use of detection schemes with different latencies. From left to right, the columns represent the coverage for systems with detection latencies (uniformly distributed) of 1000, 100, and 10 instructions. The middle column illustrates the coverage achievable for a system experiencing fault detection latencies consistent with existing techniques like Shoestring [8] and Restore [29]. The leftmost bar shows that Encore can even benefit systems with hardware speculation support. Since aggressive out-of-order machines typically only support rollback of 10-100 instructions, Encore could enhance the recoverability of these systems by supporting rollback even in cases where detection latencies reached 1000 instructions. Lastly, the rightmost bar presents the potential fault coverage that can be achieved in future systems with further constrained detection latencies.

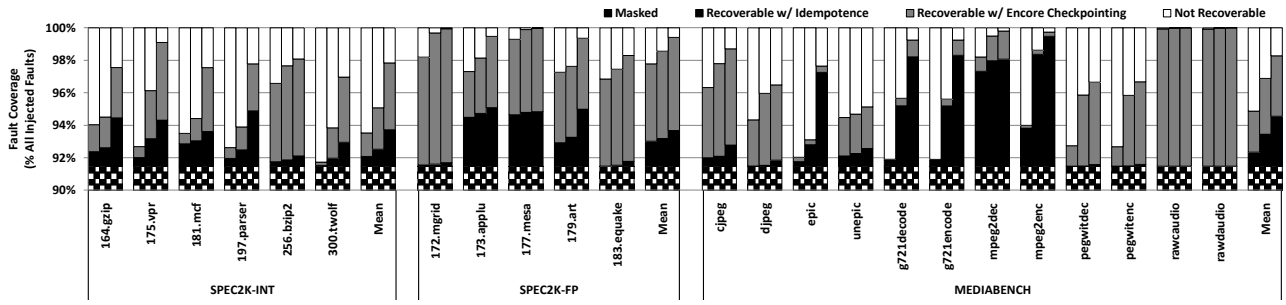


Figure 8: Full-system fault coverage for a low-cost commodity system using Encore for rollback recovery and fault detection schemes with different latencies. From left to right, the columns represent the % of all transient fault events that can be effectively tolerated given a system with detection latencies (uniformly distributed) of 1000, 100, and 10 instructions.

Nevertheless, even with present day latencies, Encore can safely recover from 97% of faults, on average, across all benchmarks and nearly all faults for certain workloads like *172.mgrid*, *177.mesa*, *mpeg2dec*, and *rawcaudio*. Although these coverage results may seem less impressive when compared with the base masking rate of 91%, one must view these gains in the proper context. By supplying this 66% reduction in the number of transient events that can cause system failures, Encore can enable low-end commodity systems to meet reliability targets that may otherwise be out of reach.

6. RELATED WORK

Transient fault tolerance requires two steps: 1) detecting the fault event and 2) recovering to an error-free state and resuming execution. Since Encore targets system recovery, this section only contains a brief overview of fault detection solutions, while providing a more detailed discussion of previous efforts in fault recovery.

6.1 Fault Detection

There exists a large body of research addressing the challenge of fault detection [12, 8, 15, 18, 29, 21, 20, 22]. These efforts can be broadly divided into four categories. First, there are solutions that utilize some form of spatial redundancy to execute multiple copies of an application simultaneously, periodically comparing results. Redundant multi-threading [21] and dual-core execution [25] are good examples from this class. The second category consists of solutions that exploit temporal redundancy, where the same work is re-executed on the same hardware resource. Compiler-based instruction duplication [20] and hardware-based selective replication [14] are well known techniques that fall into this group. Lately, a third class of techniques have emerged that rely on high-level software symptoms [29, 17, 22] to identify faults, sometimes with the help of specialized detectors [12, 30]. Finally, there have also been recent proposals that formulate hybrid solutions [8, 18] combining multiple techniques to drive costs even lower while maintaining high detection coverage.

6.2 System Recovery

Once a fault is detected, the system must rollback in order to continue execution from a previous clean state. Recovery solutions are tasked with maintaining this clean state, and providing an interface to enable the rollback. The most popular category of recovery solutions are checkpoint based. In their simplest form, checkpoint-recovery solutions periodically save off the entire system state, and revert to the most recent version in the event of a fault.

Enterprise-level Recovery: Traditionally, checkpoint-recovery solutions have been used in large-scale enterprise systems to guarantee the often touted "five-nines" of reliability. These systems, with 100-1000s of nodes, periodically suspend their program execution and take snapshots of the entire memory system, usually stored on globally accessible disks [4]. To maintain consistency, all the nodes in the system take their checkpoints simultaneously, often causing bottlenecks due to disk bandwidth limitations. In general these solutions are appropriate for their domain, but the cost of creating these checkpoints are prohibitively high for all but the most mission-critical systems.

Architectural Recovery: A cheaper alternative to taking a complete system snapshot is to log incremental changes to the system state. In the event of a failure, these changes can be unrolled as needed. SafetyNet [26] and ReVive [16] are two examples of such solutions. Although these log-based recovery solutions are scalable to more frequent checkpoints, and smaller intervals, the additional complexity and overheads introduced from potential hardware additions makes them less attractive for budget-wary commodity systems.

Opportunistic Recovery: This last category of recovery solutions may not technically be recovery schemes in the conventional sense. Work by Li and Yeung [11], subsequently reaffirmed by others [24, 23, 28, 3], recognized that not all applications, or even functions within an application, require the same degree of "correctness." Many, especially multimedia and embedded codes can naturally tolerate a non-trivial amount of errors. Li and Yeung exploit this notion of application-level correctness by manually inserting checkpointing code that only saves the program counter, architectural register file, and stack state at the top of outer loops. Relax [3] takes this principle even further. Functions are manually instrumented with recovery blocks that are allowed to select between re-executing code, returning default values, or simply ignoring the faults depending on how such actions are expected to impact the "quality" of externally visible results. Encore shares the same basic principles with these other lightweight recovery solutions. However, this work is the first to present an automated (compiler-based, without manual inspection), generalized (beyond inner/outer loops and functions) solution for achieving selective rollback recovery. Furthermore, the application-level correctness notions [28] that existing works benefit from are complementary to Encore and can be supplied to the compilation framework to further enhance reliability.

7. CONCLUSION

Whether due to environmental phenomena or ambitious designs pushing the envelop of low power architectures, transient faults are re-emerging as a prominent reliability issue in modern computing. Yet despite this growing reliability concern, we would argue that instead of appropriating large transistor budgets (or processor cy-

cles) to hedge against growing fault rates, system architects should embrace the high degree of fault tolerance that can be had simply by sacrificing provable guarantees. Such tradeoffs are the most attractive for low-end commodity and embedded markets, where systems often cannot afford to devote a substantial portion of their resources to anything other than actually performing useful computations. With the ability to recover from, on average, 97% of transient faults (when paired with existing detection mechanisms), Encore is poised as an attractive solution. Realizing this coverage at a modest 14% average performance overhead, it frees system designers to refocus their attention back onto other aspects of the system architecture.

8. ACKNOWLEDGEMENTS

We thank the anonymous referees for their valuable comments. We also thank David Meisner and Mojtaba Mehrara for their feedback during the development of this work. This research was supported by National Science Foundation grants CCF-0916689 and CNS-0964478 and the Toyota InfoTechnology Center.

9. REFERENCES

[1] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[2] J. A. Blome, S. Gupta, S. Feng, S. Mahlke, and D. Bradley. Cost-efficient soft error protection for embedded microprocessors. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 421–431, 2006.

[3] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, June 2010.

[4] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. A case study of incremental and background hybrid in-memory checkpointing. In *Proc. of the 2010 Exascale Evaluation and Research Techniques Workshop*, 2010.

[5] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb. 2010.

[6] J. Duell, P. Hargrove, and E. Roman. The design and implementaion of berkeley lab's linux checkpoint/restart, 2002.

[7] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of the 1997 International Conference on Computer Aided Design*, pages 598–604, 1997.

[8] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft-error reliability on the cheap. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[9] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–11, 2001.

[10] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2008.

[11] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 181–192, Feb. 2007.

[12] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[13] H. G. Naik, R. Gupta, and P. Beckman. Analyzing checkpointing trends for applications on the ibm blue gene/p system. *Proc. of the 2009 International Conference on Parallel ProcessingWorkshops*, pages 81–88, 2009.

[14] N. Nakka, K. Pattabiraman, and R. Iyer. Processor-level selective replication. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, 2007.

[15] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 99(PrePrints), 2009.

[16] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002.

[17] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 169–180, Feb. 2007.

[18] V. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–94, Oct. 2006.

[19] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simulataneous multithreading. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[20] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

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

[22] S. K. Sahoo, M.-L. Li, P. Ramchandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants for hardware reliability. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, 2008.

[23] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones. Stochastic computation. In *Proc. of the 47th Design Automation Conference*, 2010.

[24] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Proc. of the 2010 International Conference on Dependable Systems and Networks*, 2010.

[25] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 223–234, 2006.

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

[27] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999.

[28] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2009.

[29] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, June 2006.

[30] N. J. Warter and W.-M. W. Hwu. A software based approach to achieving optimal performance for signature control flow checking. In *Proc. of the 1998 International Symposium on Fault-Tolerant Computing*, pages 442–449, 1990.