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

Anonymous

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 device reliability. Given the rise of processor (un)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 relied on the availability of hardware recovery mechanisms. Although common in aggressive out-of-order machines, hardware support for speculative rollback and recovery is less common in lower-end commodity and embedded 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. New compiler analyses and algorithms are developed that enable Encore to provide this fault recovery at very low costs. By exploiting fine-grained idempotence analysis and cost-sensitive heuristics that only target statistically relevant code regions, Encore can achieve high recoverability coverage without the accompanying costs associated with traditional software-based checkpointing solutions. Experimental results show that Encore can recover from up to 95% of detected faults for certain applications and on average only imposes 6% of runtime performance overhead.

1. Introduction

Given the news coverage of the high-profile Toyota recalls and later articles chronicling Apple's antenna woes on their newly released iPhone, the reliability, or perhaps more appropriately the *unreliability*, of computer systems has taken center stage. Granted these headlining stories were mostly the result of faulty software and costly system engineering miscalculations. However, the public response to these events has highlighted the frustration that can arise when computers, and the systems they are associated with, do not function as advertised.

Although it is impossible to build a completely reliable system, hardware vendors target failure rates that are imperceptibly small. With the course of aggressive technology scaling that has been followed by the industry, many sources of unreliability are emerging in commercial microprocessors. One prominent source of unreliability, and the focus of this paper, is soft errors. Also known as transient faults, they can be induced by external sources e.g., electrical noise or high-energy particle strikes that result from cosmic radiation and chip packaging impurities. Additionally, in newly proposed architectures that embrace the principles of stochastic [24] and near threshold computing [5], they can also be the result of extreme timing speculation and/or frequency and voltage scaling. Whether they are the result of unexpected and uncontrollable forces or simply the cost of doing business for a cutting-edge architecture, transient faults have been and continue to be a source of unreliability for processor designers.

Unlike manufacturing or design defects, which persist and continually degrade system reliability, transient faults, as their name suggests, only intermittently cause errors in program execution. For the vast majority of time, a program executes undisturbed. Recog-

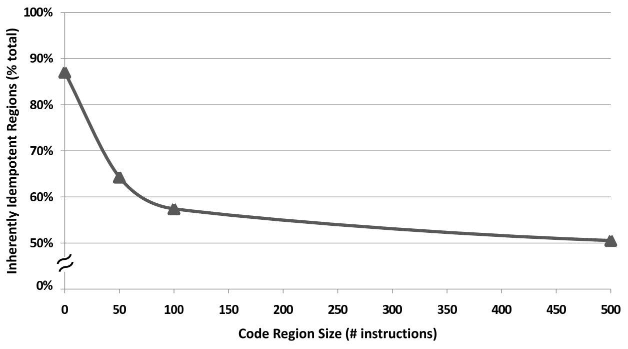


Figure 1: Inherent region idempotence as a function of region size. The data reported is the average across an assortment of SPEC2K and Mediabench workloads.

nizing this characteristic of transient faults, architects in the past have also designed systems that took periodic checkpoints (typically snapshots of processor and memory state) and could rollback to these checkpoints and resume execution in the event of a soft error. 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 an 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 [2, 12, 20, 22, 28]. However, the resultant overheads of these coupled detection and recovery scheme, a large component of which was the cost of creating checkpoints (requiring on the order of minutes to an hour), 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 typically 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 systems 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 estimates.

With that in mind we propose, Encore, a software-only fault recovery solution that seeks to provide high (but not necessarily guaranteed) coverage at the least possible cost. Furthermore, as an automated, compiler-driven technique it is able to utilize programmable heuristics that allow the end-user to dial in the degree of fault-tolerance that is desired 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 *idempotence* property of applications to reduce, and in certain situations nearly completely eliminate the overheads required to support re-execution based fault 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 based 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 generally means that there cannot exist any paths through the region that can read, modify, and then write to the same (or overlapping) memory location(s) [1].

To better understand the extent of idempotent code present in an application, Figure 1 shows the distribution of idempotent regions across a set of desktop and media benchmarks. Results are shown as a function of region size. The definition of regions and how this data was generated will be discussed later in the paper. For the time being, it is sufficient to treat regions simply as groups of connected basic blocks. The surprisingly large percentage of naturally occurring idempotent regions seen in Figure 1 is what initially encouraged the development of Encore. To the first order, the regions 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 these regions impractical.

This is not entirely unexpected since with more instructions comes the greater chance that there exists some sequence of instructions that violate the read-modify-write constraints required to maintain idempotence. This intuition is reinforced by the distribution shown in the figure which exhibits a sharp drop when moving from regions with just a handful of instructions to those with 50 or more. Lastly, it is also interesting to note that for the regions that do not exhibit full idempotence, many tend to be *nearly* idempotent, i.e., only a few offending instructions violate idempotence. Similarly, in others those that do break idempotence only occur along statistically unlikely paths.

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. Nevertheless, despite the advantages, unlike its mission-critical counterparts, Encore only provides probabilistic fault recovery (i.e., it is likely able to recover from most faults) and cannot provide provable guarantees on recoverability. Yet, this design decision allows it to transparently provide fault recoverability on commodity systems at a price that they can afford. The contributions of this paper are as follows:

- We demonstrate how low-cost transient fault recovery can be achieved for commodity systems without hardware support for aggressive performance speculation.
- Develop new compiler algorithms and heuristics for
  - Automatically identifying candidate idempotent regions in generalized code regions with support for cycles.
  - Trading off recoverability with performance overheads by exploiting application profiling statistics.
- Propose an analytical model for computing recoverability coverage without the aid of statistical fault injection.

[1] Idempotence also requires that no live-in registers are overwritten within the region as well, but this issue will be addressed separately.

Table 1: Comparison with conventional checkpointing schemes.

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

2. Recovering from Transient Faults

Generally speaking, transient fault tolerance consists of two components: 1) fault detection and 2) fault recovery. There is no shortage of examples in the literature that address each of these parts (see Section 6). Encore falls squarely on the side of fault recovery. That is to say, this paper is primarily concerned with being able to recover from a soft error event once the fault has already been detected. It assumes that fault detection itself is performed by some other low-cost means, some of which will be discussed in Section 6.

As previously mentioned, traditional high-reliability systems have chiefly relied upon heavy-weight, 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 (on the order of 100's to 1000's of instructions) without any specialized hardware support, and all at a cost that is significantly lower in terms of performance and storage space.

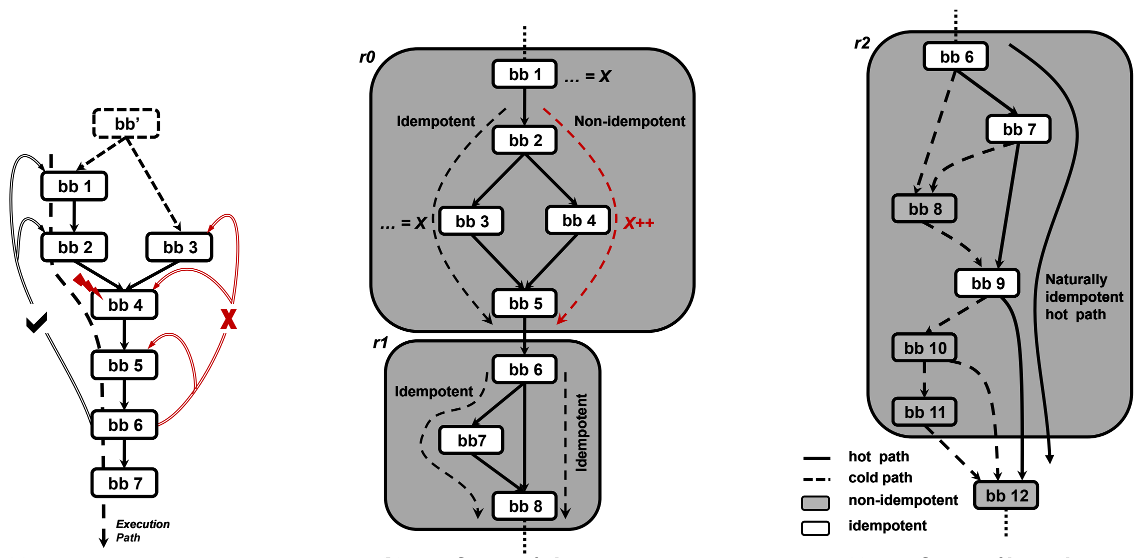
Although this initial comparison may suggest that Encore is strictly superior to these prior schemes, this is certainly not the case. In order to achieve the reductions in run time overhead without incurring hardware costs Encore has to sacrifice, albeit to a limited degree, reliability. Relative to enterprise solutions [4, 6, 14, 28], and to a lesser extent recently proposed architectural solutions [17, 27], which can provide guarantees on recoverability coverage, Encore can only provide probabilistic estimates. This small concession, consistent with the needs of systems along the lower spectrum of the commodity space [3, 8], enables numerous optimizations that allow it to maintain dramatically lower costs.

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 tolerated, and in a sense completely eliminated. This assumes that during the initial execution no read, modify, write dependencies overwrote state that could lead to erroneous behavior upon re-execution.

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 be able to rollback to 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 unecessarily re-executed. However, this would necessitate knowledge of the dynamic execution path of the program. Without hardware support, this would require some form of software-based dynamic control flow signature generation [32], an expensive proposition.

Instead, Encore only operates on single-entry, multiple-exit (SEME) regions. This frees it from having to account for which



(a) Re-execution options. **(b)** Significance of idempotence **(c)** Significance of hot paths

Figure 2: Fine-grained transient fault recovery with re-execution. (a), (b), and (c) illustrate some of the challenges and opportunities that exist when leveraging fine-grained re-execution to achieve fault recovery. (a) enumerates potential rollback destinations that execution can be redirected 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 in the region. (b) highlights how idempotence might constrain which code regions can actually be efficiently recovered. (c) depicts how otherwise non-idempotent regions can still frequently exhibit idempotent behavior (hot path). The region shown in (c) is taken from the CFG corresponding to the dominant hot function in *175.vpr*.

path of execution actually lead to the instruction corrupted by the fault. It can safely redirect execution to the top of the SEME region, the header. Details regarding the algorithm and heuristics that Encore uses to partition an application into these SEME regions are described in Section 3.

Figure 2a is a simple illustration of how this type of fault recovery is realized in the Encore system for a small subgraph $bb_1 - bb_7$. In this example a transient fault corrupted an instruction inside basic block bb_4. However, the fault is not detected until bb_6. Although expensive hardware detection schemes tend to have much shorter detection latencies (on the order of a few cycles), the lower-cost options that Encore is more likely to be paired with will typically have much greater latencies (on the order of dozens of instructions). Furthermore, these low-cost detectors are also less likely to be able to diagnose *exactly* the location of originating fault, e.g., bb_4. In this environment a potential recovery mechanism is left with the difficult task of deciding where control should resume.

When a transient fault is detected, Encore would like to to redirect control back to "closest" basic block along the path of execution that that preceded the fault site, namely bb_2. However, without additional information about where the fault occurred **and** what dynamic execution path originally led to the fault site, Encore is left with choosing between five potential candidates, $bb_1 - bb_5$. Even after eliminating bb_7 as a potential candidate, since there does not exist any path from bb_7 to bb_6, the choice of where to redirect control after a fault is detected is undecidable at compile-time. Fortunately, if Figure 2a were part of a SEME region, the decision could easily be made to conservatively rollback execution to the region header, bb'. Although some additional code would be re-executed uneccessarily, namely bb_1, this is still more agreeable than the alternative of dynamic-flow tracking.

2.2 The Role of Idempotence

Figure 2b illustrates how recognizing idempotent regions can greatly reduce the effort required to provide fault recoverability. The key principle behind using re-execution as a means of recovering from transient faults is the expectation that the portion of code being re-executed will produce the same results, the second time around, as it did during the initial execution, precisely the definition of idempotence. As long as no read, modify, write sequences to the same state throughout a region, the region is idempotent and it automatically becomes an attractive candidate for Encore's re-execution based fault recovery. In the example, since all paths through region r_1 are idempotent, it is more desirable than r_0, for which execution can be non-idempotent. Relying on conventional, heavy-weight, full-system checkpointing schemes (or even their more recent lighter-weight incarnations) 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 this property then selectively checkpointing X 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 high recoverability coverage.

Lastly, Figure 2c shows an actual subgraph taken from *175.vpr*, a benchmark from the SPEC2K integer 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}, bb_{11}, and bb_{12} 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, bb_9, and bb_{12} dominates the execution time. This suggests that although region r_2 is not strictly speaking idempotent, it does exhibit idempotent *behavior* for the vast majority of the time. This *probabilistic* idempotence is yet another property of applications that Encore exploits 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. Doing so is a promising yet challenging problem. A recent proposal by Kruijf et al. [3] overcame some of these challenges by resorting to manually inspecting and modifying of source code to take advantage of the function-wide idempotence and fault tolerant properties of multimedia and data-mining applications. Although the work is in the same spirit as Encore, by resorting to 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 idempotence to enable low-cost fault recovery was the purpose of developing Encore. The algorithms and heuristics needed to achieve this efficiently in the compiler is the main contribution of this paper. The remaining sections of this paper describe the analysis and instrumentation passes utilized by Encore (Section 3), the methodology to evaluate the recoverability coverage that can be achieved (Section 4), analysis of the resulting fault tolerance and performance overheads of Encore (Section 5), and concludes with a brief discussion of the the implications this has on low-cost fault-tolerant system designs (Section 7).

3. Encore

Achieving the goals laid out in the title of this work, low-cost transient fault recovery, involves identifying naturally occurring regions of code that are amenable to re-execution, and judiciously sacrificing fault tolerance to maintain low overheads. This section will elaborate on the means by which this is done in a methodical, automated fashion at compile-time.

Section 3.1 begins by describing the analysis that is used to determine the idempotence of candidate code regions, Section 3.2 discusses how idempotence is enforced cost-effectively in otherwise non-idempotent regions, Section 3.3 addresses how these code regions are actually formed, and lastly Section 3.4 elaborates on the heuristics developed to maximize Encore's recoverability coverage while maintaining minimal overheads.

3.1 Identifying Inherent Idempotence

Before the discussion proceeds any further, a few terms that will be used throughout this paper first need to be defined.

Region: in its unqualified form *region* will refer to a subgraph of basic blocks connected in the program CFG.

Reachable Store: at a given location p (e.g., instruction or basic block) in the CFG, a *reachable store*, relative to p, is a store instruction that may potentially execute after p.

Dominating Store: with respect to a given location p, a *dominating store* is one that is guaranteed to execute prior to reaching p.

Exposed Load: with respect to a given location p, an *exposed load* is one that is guaranteed to execute prior to reaching p and is unguarded. A load, l, has a corresponding *dominating store* (or stores) that writes to the same address along **all** possible paths to l is considered guarded. All others are not.

Inherent Idempotence: a property of a region that indicates that the region has no read-modify-write sequences to the same address that would prevent it from being safely re-executed without altering the live-outs of the region (i.e., is side-effect free).

Although these definitions (and text throughout this paper) only reference store and load 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*, *dominating store*, and *exposed-load* 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, $\mathbb{RS}_{bb_i}$. Note that the set notations used in subsequent equations ($\cup$ and $\cap$) are actually comparing the addresses being referenced and/or modified, not actually whether the instructions within the respective sets are actually identical [2].

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

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 $\mathbb{AS}_{bb_j}$ is the set of all stores within bb_j.

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 (all basic blocks with outgoing edges to blocks outside the region). As each basic block, bb_i, is encountered during these "reverse" post-order traversals, Equations 2 and 3 are used to update the corresponding dominating store and exposed load sets. Note that the dominating store set,$\mathbb{DS}_{bb_i}$, must be updated before the exposed load set, $\mathbb{EL}_{bb_i}$.

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

$$\mathbb{EL}_{bb_i} = \bigcup_{\forall bb_j \in \mathbb{C}_{bb_i}} \left(\mathbb{EL}_{bb_j} \cup f_{EL}(\mathbb{EL}^{local}_{bb_i}, \mathbb{DS}_{bb_j}) \right) \tag{3}$$

where,

$\mathbb{DS}_{bb_i}$:	is the set of dominating stores at bb_i.
$\mathbb{EL}_{bb_i}$:	is the set of exposed loads at bb_i.
$f_{EL}(\mathbb{X}, \mathbb{Y})$:	operates on sets of loads, $\mathbb{X}$, and stores, $\mathbb{Y}$, returning the set of all loads $x \in \mathbb{X}$ that are not aliased by any stores $y \in \mathbb{Y}$.
$\mathbb{EL}^{local}_{bb_i}$:	is the set of all loads in bb_i that are not preceded by a store, also within bb_i, that writes to the same address, effectively the set of *local* exposed loads for bb_i.

Once all the basic blocks within the region have been processed, and the associated $\mathbb{RS}$, $\mathbb{DS}$, and $\mathbb{EL}$ sets have been updated, a determination can be made as to whether r is idempotent according to Equations 4 and 5. Equation 5 essentially determines if idempo-

[2] This was done using standard, static memory alias analysis techniques which are necessarily conservative.

$$\text{Region } r \text{ is idempotent } \textbf{iff } I(bb_i) = \text{true}, \forall bb_i \in r \quad (4)$$

where,

$$I(bb_i) \begin{cases} \text{true}, & \textbf{iff } \mathbb{EL}_{bb_i} \cap \mathbb{RS}_{bb_i} = \emptyset \\ \text{false}, & \text{otherwise} \end{cases} \quad (5)$$

tence can be violated by executing basic block bb_i along any possible path through region r. Admittedly, identifying idempotence in this manner leads to conservative answers. By leveraging this analysis, Encore is essentially operating under worst-case assumptions. In fact, Equations 4 and 5 do not account for any correlations among branches between basic blocks and consequently may categorize regions as non-idempotent because of paths that can never be realized given the structure of the application. However, compared with path-sensitive alternatives, which are effectively intractable, the algorithm proposed here is efficient and reasonably accurate [7].

3.1.2 Incorporating Cycles

Up to this point the analysis has focused on acyclic regions. Introducing cycles can complicate matters dramatically. 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 boundaries of loops are encountered, header blocks during the forward post-order traversals and exit 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 load/store operations 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. The details of this process is described below.

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 dominating store and exposed load sets for each basic block within the loop are generated according to Section 3.1.1. The set of reachable stores for each basic block, however, are identified in a slightly different manner. Equation 6 describes how the set of reachable stores is generated for each basic block within the loop. Defining $RS^l_{bb_i}$ in this fashion ensures

$$\mathbb{RS}^l_{bb_i} = f_R(\mathbb{AS}^l, \mathbb{DS}^l_{bb_i}) \quad (6)$$

where,

$\mathbb{RS}^l bb_i$:	the set of reachable stores for basic block bb_i with respect to loop l.
$\mathbb{AS}^l$:	the set of all store instructions within loop l.
$f_R(\mathbb{X}, \mathbb{Y})$:	operates on sets of stores, $\mathbb{X}$ and $\mathbb{Y}$, returning the set of all stores $x \in \mathbb{X}$ with a destination not guaranteed to be overwritten by a store(s) $y \in \mathbb{Y}$.

that all cross-iteration, write-after-read dependencies are accounted for in the analysis. Once the loop-wide reachable store, dominating store, and exposed-load sets have been generated for all basic block within the loop, the loop itself can be treated as any other region and idempotence can be determined using Equation 4. Once loop idempotence has been determined the next step is to generate the meta-information associated with the loop.

[3] Not all cycles within a CFG can be converted into canonical form. In these extremely rare cases, only one was encountered during all of our analyses in Section 5, Encore chooses to ignore the enclosing region and leaves it unprotected (i.e., does not instrument it for recovery).

The goal of loop-wide meta-information is to capture and expose loop-wide side-effects to simplify subsequent region analyses. It is a summary of all the load/store operations within the loop so that the entire loop itself can 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. By virtue of being a cycle, this set effectively contains all stores within the loop.

$$\mathbb{RS}_{l_i} = \mathbb{RS}^{l_i}_{header} = \mathbb{AS}^{l_i} \quad (7)$$

  where $\mathbb{RS}_{header}$ is the set of reachable stores for the header block of l_i.

- **Loop-wide dominating stores**, $\mathbb{DS}_{l_i}$: the set of all dominating stores within loop l_i, i.e., the stores that are guaranteed to have executed before control exits the loop. Since loops can have multiple exiting blocks (a basic block with an outgoing edge to another block outside l_i) this is effectively the intersection of all dominating store sets across all exiting blocks of l_i.

$$\mathbb{DS}_{l_i} = \bigcap_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{DS}_{bb_i} \quad (8)$$

  where $\mathbb{X}_{l_i}$ is the set of exiting blocks for l_i.

- **Loop-wide exposed loads**, $\mathbb{EL}_{l_i}$: analogous to the definition of local exposed loads for basic blocks, $\mathbb{EL}^{local}_{bb_i}$, the exposed load set for l_i is the set of all loads within l_i that are not dominated by a store(s) that writes to the same address along all possible paths.

$$\mathbb{EL}_{l_i} = \bigcup_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{EL}_{bb_i} \quad (9)$$

With the loop-wide meta-data generated for all loops within the CFG, analysis of any arbitrary region can proceed as if cycles did not exist. The region traversals simply "step-over" loops whenever they are encountered and update idempotence data structures with the loop-wide meta-data.

3.2 Preserving Idempotence

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 performance overhead) 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 the header block of r_i with checkpointing instructions that preserves all addresses that can be overwritten by any store in $\mathbb{CP}_{ri}$ with memory-to-memory moves (pairs of load and store instructions). Note that multiple offending store instructions at different locations within the region may only require a single checkpoint pair if they all alias the same memory location. Additional optimizations that help reduce the overhead of this selective checkpointing are described in Section 3.4.

Additionally, in order to ensure that no write-after-read register dependencies violate idempotence, all live-in (with respect to r_i) register values are also checkpoint ed with a register-to-memory move (a single store to memory). How register live-in values are identified for each region will not be discussed here since it is a standard analysis in modern compilers.

With the necessary checkpointing instructions identified and inserted into the region headers, all that remains is to create a *recovery block*–the destination of all rollbacks, initiated from exiting blocks, if and when a fault is detected. Within this 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. Furthermore, the conditional rollback to the recovery block can also be amortized with the cost of the detection scheme (assuming it is also software-based).

3.3 Region Formation

Having discussed how idempotence is analyzed, and enforced if necessary, for any arbitrary region it is time to tackle how the CFG is actually partitioned into these segments. Candidate region formation is done in Encore by building upon traditional interval analysis [1]. In general an interval, as defined by Aho et al., 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 partitioning typical application CFG's into disjoint intervals is a standard pass within most modern compilers, the details of how this initial partitioning is achieved are omitted. However the following two properties of this partitioning are important to keep in mind.

1. **All intervals are by definition SEME regions**: all blocks within a partition are dominated by a single the region header.
2. **Interval partitioning can be applied hierarchically**: once a CFG is partitioned into intervals, the intervals themselves form an *interval graph* that can also be partitioned into intervals. This repeated partitioning can be applied until the *limit flow graph* of the CFG is reached, which for most CFGs means that all the basic blocks are entirely contained within a single interval [4].

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 is corrected.

The second property of interval partitioning is what allows Encore 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. This instrumentation 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 (see Section 4.2.1). Section 3.4.2 will discuss how heuristics are used to identify the appropriate region size given a budget for acceptable performance overhead.

3.4 Encore Heuristics

Now that the basic procedures for forming regions and identifying/enforcing idempotence have been described, this section will focus on the heuristics used to glean the best reliability v. performance trade-offs from Encore. First we discuss how profiling information can be used to probabilistically prune basic blocks from the idempotence analysis followed by the heuristic used to identify which regions are chosen as candidates for rollback recovery.

3.4.1 Relaxing Idempotence

Since Encore is intended to supply probabilistic fault tolerance (specifically recoverability) for non-mission critical systems, one opportunity for optimization is to leverage application profiling data. Since conventional techniques targeting ultra-reliable systems must guarantee recoverability their mechanisms are limited to relying upon provable analyses. Encore, on the other hand, is not required to provide such guarantees and without such constraints is free 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 possible (and even impossible) 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 $\mathbb{RS}$, $\mathbb{DS}$, and $\mathbb{EL}$ 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 operating 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 Equation 10.

A basic block bb_j is considered dynamically-dead w.r.t. bb_i **iff**

$$W_{r_i}(bb_i, bb_j) \leq P_{min} \tag{10}$$

where,

$W_{r_i}(bb_i, bb_j)$: is the weight of the profiled edge from bb_i to bb_j, the probability that the transition from bb_i to bb_j occurs given that execution has entered region r_i.

P_{min}: is the heuristic threshold controlling the extent to which Encore prunes dynamically-dead code.

3.4.2 Region Selection

Another opportunity for trading off fault tolerance for performance is in the area of region selection. Since Encore has considerable control over the size of the regions that are created, Equation 11 describes heuristic that determines when it terminates the process of merging existing intervals to form larger regions.

$$\frac{C(I) - \sum_{i \in I} C(i)}{\sum_{i \in I} C(i)} \geq \beta \frac{O(I) - \sum_{i \in I} O(i)}{\sum_{i \in I} O(i)} \tag{11}$$

where,

I: is a larger interval that was formed by merging a set of smaller intervals $i \in I$., according to the algorithm from Section 3.3.

$C(I)$: is the coverage achievable by protecting interval I.

$O(I)$: is the performance overhead associated with protecting I, essentially the cost of instrumentation to preserve idempotence if I is otherwise inherently non-idempotent.

β: is a heuristic parameter, $[0, \infty)$, that configures Encore to target different reliability requirements.

[4] Technically, not all possible CFGs are reducible in this fashion, down to a single node. However, irreducible CFGs are rare and ignoring them did not measurably degrade Encore's coverage.

Only when Equation 11 is satisfied does Encore consider merging existing intervals to form a larger region. Small values of $\beta < 1.0$ predisposes the system to try and create the largest regions possible in pursuit of greater reliability. In contrast, larger value of $\beta > 1.0$ shift the focus toward minimizing performance overheads, preventing Encore from forming larger partitions unless they are also accompanied by significant improvements in coverage. The details of how coverage is evaluated in our experiments can be found in Section 4, however, during compilation Encore uses the distance of the hot path through a region as a compile-time surrogate for coverage, $C(I)$. Similarly, the ratio of checkpointing instructions inserted to total instructions along the hot path serves as a compile-time estimate of overhead costs, $O(I)$.

In addition to identifying the optimal region size, a decision must be made as to whether protecting a region is actually a profitable endeavor. For inherently idempotent regions the answer is almost always yes. The cost of instrumenting the exiting blocks of a region to rollback to the header when faults are detected 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 candidates for protection, and are subsequently instrumented with checkpointing and recovery instructions. In other words $O(I)/C(I) < \Phi$ must be satisfied for every region instrumented for recovery, where Φ is a heuristic threshold. It is important to mention that this constraint is imposed simply to keep performance overheads in check. Even in regions where the cost-to-coverage ratios are suboptimal the addition of checkpointing instructions will not negatively impact reliability (see Section 4.4).

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. However, given limited resources an acceptable alternative has been statistical fault injection (SFI) on detailed system models (architecture, microarchitecture, RTL, models etc.). Nevertheless, we propose to evaluate Encore's capabilities using an analytical fault coverage model. Because recovery (through re-execution) is less sensitive to the details of the underlying hardware, an appropriate analytical model can provide an acceptable level of fidelity without necessitating time consuming simulations. The details of the experimental methodology and analytical model used for calculating coverage is described below.

4.1 Compilation Framework

The compiler analysis and instrumentation passes described in Section 3 were implemented in the LLVM compiler [10]. An assortment of SPEC2K integer (*164.gzip, 175.vpr, 181.mcf, 197.parser, 256.bzip2, 300.twolf*, floating point (*172.mgrid, 173.applu, 177.mesa, 179.art, 183.equake*), and Mediabench (*c/djpeg, un/epic, g721encode/decode, mpge2enc/dec, pegwitenc/dec, rawc/daudio*) applications serve as representative workloads for our experiments and are compiled with standard -O3 optimizations.

4.2 Recoverability Coverage Model

As previously stated, Encore is only targeting the *recovery* aspect of processor reliability. Within this context "coverage", will refer specifically to *recoverability coverage*–the ability of the system to recover from a transient fault once the fault has been detected. For software-only schemes like Encore that rely on rollback recovery, coverage is equivalent to the percentage of the application code that can safely be re-executed in the presence of a fault. Equation 12 describes the fraction of program execution that is potentially recoverable by Encore.

$$C(A) = \sum_{i=0}^{n} W^A(f_i) \sum_{j=0}^{m} W^{f_i}(r_j) \Big(W^{r_i}(p_{hot}) \cdot I(p_{hot}) \Big) \quad (12)$$

where,

$C(A)$: is the (recoverability) coverage for an application A.

$W^A(f_i)$: is the fraction of application A spent inside function f_i, the runtime weight of f_i relative to A.

$W^{f_i}(r_j)$: is the fraction of function f_i spent inside region r_j, the runtime weight of r_j relative to f_i.

$W^{r_i}(p_{hot})$: is the fraction region r_j spent along its hot path p_{hot}, the runtime weight of p_{hot} relative to r_j.

$$I(p_{hot}): \begin{cases} 1, & \text{if the hot path } p_{hot} \text{ is idempotent} \\ 0, & \text{otherwise.} \end{cases}$$

4.2.1 Impact of Detection Latency

Note that Equation 12 is incomplete since it assumes that any region that is idempotent, whether inherently or because it was instrumented, can be recovered. This neglects to account for the latency of the fault detection scheme. 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 only 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 13.

$$\begin{aligned} \alpha_{r_i} &= Pr(s + l < n), \ \forall s \in [0, n], \forall l \in [0, D_{max}] \\ &= \int_0^n \int_0^s f(l) g(s) dl ds \end{aligned} \quad (13)$$

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 $[0, D_{max}]$, which represents 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 before execution proceeds beyond the boundary of region r_i.

$f(l)$: is the probability density function for the detection latencies of the system.

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

In our evaluation of Encore in Section 5 we assume an uniform distribution for the fault distribution, insisting that every dynamic instruction over the course of an application's runtime has the same probability of being "struck" by a transient fault. In other words, every dynamic instruction has an equal chance of being the fault site. In reality, because of (micro)architectural masking not all instructions are necessarily equally vulnerable to a transient event. However, for this establishing the merits of Encore, it is an acceptable approximation.

Similarly, results in Section 5 also assume uniformly distributed fault detection latencies that are independent of fault location. Whereas detection latencies for hardware schemes are typically predictable, the low-cost mechanisms that complement Encore will likely rely more on software techniques, for which detection latencies are more erratic and dependent on complex interdependencies between instructions. A proper SFI framework could capture the effects of these behaviors. Unfortunately, the requisite computational resources necessary to generate statistically significant results can be prohibitive. Comparatively, our simplifying assumption provides an acceptable approximation, which can be easily updated with more accurate statistical models without needing to re-run massive Monte Carlo simulation sets.

Given independent, uniform distributions for fault distribution and detection latency, Equation 13 can be re-written as Equation 14. It is important to note that although experimental results were generated using uniform distributions for $f(l)$ and $g(s)$, Encore and the analytical model for coverage being proposed are not themselves limited to these distributions but are generally applicable.

$$\begin{aligned}\alpha_{r_i} &= \int_0^n \int_0^{min(y,D_{max})} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dxdy \\ &= \int_0^{D_{max}} \int_0^{min(y,D_{max})} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dxdy \\ &\quad + \int_{D_{max}}^n \int_0^{min(y,D_{max})} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dxdy \\ &= \int_0^{D_{max}} \int_0^{y} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dxdy \\ &\quad + \int_{D_{max}}^n \int_0^{D_{max}} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dxdy \\ &= \begin{cases} 1 - \frac{D_{max}}{2n}, & n \geq l \\ \frac{n}{2D_{max}}, & n < l \end{cases} \end{aligned} \tag{14}$$

Factoring in detection latency, $I(p_{hot})$ in Equation 12 can be re-written to account for α_l (Equation 15).

$$\begin{aligned} I'(p_{hot}) &= \alpha_{r_i} \cdot I(p_{hot}) \\ &= \begin{cases} 1 - \frac{D_{max}}{2n}, & L(p_{hot}) \geq D_{max} \\ \frac{n}{2D_{max}}, & L(p_{hot}) < D_{max} \end{cases} \end{aligned} \tag{15}$$

where $L(p_{hot})$ is the length of the hot path p_{hot}.

4.3 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, it allows us to abstract away the details of the underlying hardware and present, to some extent, architecture-neutral results. Since Encore only inserts a small fraction of additional instructions (that are actually executed at runtime) the instrumentation required for recovery should not significantly alter the cycles-per-instruction (CPI) an application can achieve on the hardware. This being the case, dynamic instruction counts can serve as a reasonably accurate performance metric.

4.4 Assumptions and Limitations

Below are some of the other assumptions made by our evaluation infrastructure.

- **Faults corrupting control and/or address calculation:** Both address faults that result in writing or overwriting data to erroneous locations, and faults that lead to deviations from the correct control path cannot be 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 [8, 31]. Oftentimes, these faults can be detected before they propagate to memory and/or divert control flow (i.e., before they become unrecoverable).
- **Faults corrupting instrumentation code:** Although the instructions inserted by Encore can themselves be subject to transient faults, we assume that Encore, working in tandem with the detection mechanism, can choose to elide rollback recovery events initiated by faults detected in the instrumentation code itself. This does not appreciably impact recoverability coverage since the checkpointing code does not perform substantive computation (i.e., does not influence region live-outs). Although a fault in the instrumentation code would temporarily disable Encore's ability to rollback correctly, it would require a fault to corrupt a checkpoint instruction followed almost immediately by a subsequent fault elsewhere within the same region in order for this to impact coverage. Given a standard single event upset model this scenario is highly improbable.
- **Cold-path execution:** To reduce the dependence on path-sensitive analysis the coverage results reported in this paper are limited to recoverable hot paths. We conservatively assume that any execution time spent along "colder" paths are not recoverable. Obviously this under-estimates coverage, but was necessary to reign in the complexity of our evaluation framework.
- **Masking (software, micro/architectural):** We do not account for masking at the various levels of the stack. We assume all dynamic instructions can lead to equally deleterious results if subject to a transient fault. This necessarily results in conservative estimate of coverage. Analysis of characteristics like architectural, and similarly software, vulnerability factors [13] can be used to enhance improve results presented in Section 5 but were beyond the scope of this work.
- **Dynamically-linked library/system calls:** Consistent with similar reliability works in the literature we do not consider faults outside of code that is visible to the compiler. The common practice is to assume that these portions are protected by other means, although they can be addressed by Encore if their source is available.
- **Multi-threaded Applications:** Since multi-threaded programs are less common (albeit growing) in the low-cost, commodity domains being targeted we do not explicitly evaluate Encore in the context of these workloads. However, the idempotence analysis described in Section 3 could 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. Of course step would also need to be taken to ensure that rollback recovery instrumentation did not violate the semantics of synchronization events.

5. Evaluation and Analysis

This section demonstrates quantitative evidence of Encore's ability to provide recoverability coverage without incurring appreciable overheads. Section 5.1 begins by analyzing the idempotence of candidate recovery regions. This is followed by coverage results in Section 5.3. Lastly, Section 5.3 concludes with a discussion of performance overheads. All experimental results reported in this section were generated with $\beta = 0.25$, $\Phi = 0.1$, and values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. A value of $\emptyset$ for Φ means that no dynamically-dead blocks are pruned from the analysis and a value of 0.0 indicates only code that is *never* observed to execute during profiling is pruned.

5.1 Region Idempotence

Figure 3 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 the 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 *idempotent*, *non-idempotent*, and *unknown*. Unknown regions are those that Encore's compiler analysis was unable to process (e.g., mainly system and library function calls), 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. This suggests that a good portion of the instrumentation optimizations described in Section 3 can be achieved without incurring any measurable risk.

Not surprisingly the SPEC2K floating point and Mediabench applications exhibit slightly better idempotence behavior than the SPEC2K integer 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 the three benchmark suites are comparable (a few poor-performing applications in the floating point and Mediabench groups drag down their averages).

More importantly it is encouraging to observe that even in control-heavy SPEC2K integer applications there is still a considerable fraction of code that is *inherently* idempotent. On average, across all applications, 47% of regions are inherently idempotent without pruning and 76% are idempotent with $P_{min} = 0.0$. This suggests that little, if any instrumentation code will need to be inserted by Encore, across much of the application, to maintain idempotence for recovery rollback

5.2 Recoverability Coverage

Next we examine the recoverability coverage that can be achieved by Encore. Figure 4 presents coverage numbers for different values of D_{max}. Recall coverage here refers to the fraction of runtime execution that can be recovered once a fault has been detected. Due to space limitations results are only shown for $P_{min} = 0.0$, meaning that idempotence analysis only filters out completely dynamically-dead code (anything that *never* executed during profiling runs). This corresponds to the best coverage Encore can achieve, without running the risk of introducing faults (because idempotence was not maintained) during rollback recovery, by filtering out portions of code that have non-negligible probabilities of executing.

The segments labeled *Recoverable* are inherently idempotent and can be recovered effectively "for free," provided that the fault was detected in time. *Recoverable w/ Instrumentation* regions also contribute to the coverage Encore can achieve, but require instrumentation to ensure idempotence is preserved, i.e., to accommodate read-modify-write chains and register live-ins. The *Unrecoverable* segments refer to the portion of runtime execution that Encore fails to recover from because the fault was detected after execution had left the region containing the original fault site and therefore cannot be recovered. Lastly, the sections labeled *Unknown* correspond to time spent in regions of code that Encore could not analyze and are therefore conservatively considered unrecoverable.

Inspection of Figure 4 reveals the expected inverse relationship between coverage and expected detection latency, D_{max} (in instructions). Consistent with the idempotence results in Figure 3, coverage also exhibits better behavior for the more multimedia and embedded-centric applications. In fact, this application dependence is even more pronounced for coverage, supporting the claim that these benchmarks tend to spend the majority of their time inside tight, largely idempotent loops. For example with $D_{max} = 10$ (Figure 4d) the audio codec *g721encode* can recover from 85% of detected faults naturally, without *any* additional instrumentation to preserve idempotence. Similarly the floating point heavy *172.mgrid* is able to achieve a dramatic 95% coverage rate with the help of Encore's selective checkpointing. Those correspond to 6x and 20x improvements in the fault tolerance (in terms of recoverability coverage) of these two applications!

5.3 Performance Overheads

Lastly we investigate the performance overheads incurred by Encore. Figure 5a reports the runtime overheads for different values of P_{min}. At first glance the results may appear counter intuitive. One might expect that as the degree of dynamically-dead code filtering becomes more and more aggressive (i.e., increasing values for P_{min}) that there would be a corresponding decrease in runtime costs since more and more potentially idempotence-breaking instructions would be pruned and not require instrumentation. Many applications (e.g., *181.mcf*, *300.twolf*, and *183.equake*) support this intuition. However others (e.g., *164.gzip*, *175.vpr*, and *197.parser*) seems to contradict this idea, with performance actually degrading as more code is pruned.

Upon close inspection we discovered that in the latter set of benchmarks, the act of aggressively pruning dynamically dead code actually made certain regions (corresponding to the *Unknown* segments in earlier results) that previously could not be processed subject to idempotence analysis. As these formerly blacklisted regions were analyzed, some were identified as non-idempotent, which required Encore to insert code to preserve idempotence, leading to additional runtime overheads.

Yet, with few exceptions, most notably *181.mcf*, Encore imposes minimal performance overheads independent of the value of P_{min}. For $P_{min} = 0.0$, which corresponds to the coverage reported in Figure 4, on average across all applications only 6% of execution time can be attributed to Encore instrumentation. Upon further examination, we learned that the bulk of execution time for *181.mcf* was spent within a single function with a single hot region, a region which only had a roughly 20 instruction hot path. Unfortunately the region also contained 10 idempotence-violating stores and 14 register live-in values! Despite there being such a high cost for instrumenting this undesirable region, so much of the application's execution was spent within this portion of code that not to protect would have meant sacrificing dramatically on coverage. This was an example where even obeying the heuristic Φ, which is designed to prevent instrumenting poorly behaved code, could not avoid dramatic performance penalties. Although this was as close to a pathological case study as we saw in our experiments, a simple hard upper bound on performance penalty would prevent runaway overheads.

It is also important to mention that these overheads do not account for the time required to recover from a fault once it is detected (i.e., restoring state if needed and re-executing). Although this additional overhead is dependent on the specific fault rate that is expected, for the usage scenarios described in Section 2 the fault rates are typically high but still orders of magnitude low enough that the fault-free execution case dominates the negligible rollback recovery overhead. This approximation holds for Encore-style recovery (very fine-grained with minimal, if any, state restoration) but breaks down if traditional system-wide checkpointing and re-execution of large portions of code (e.g., entire functions) is required.

Finally, although Section 3.4.2 presented a heuristic that could be used to temper runtime overhead by sacrificing on coverage, in practice not much appreciable coverage was obtained by increas-

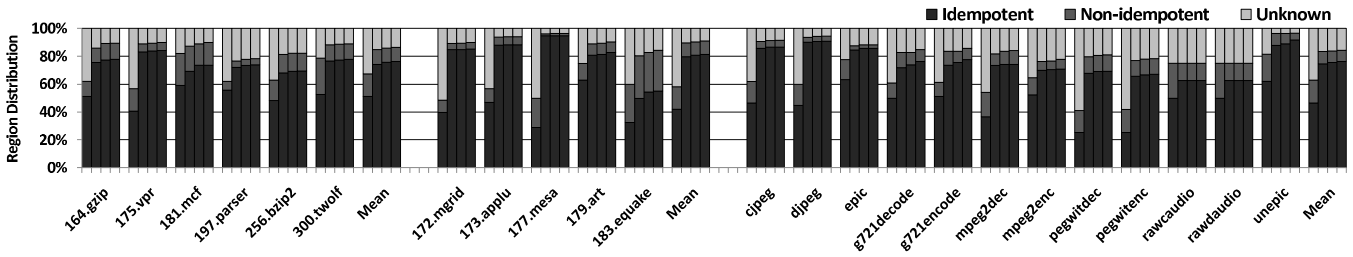


Figure 3: Inherent region idempotence as a function of P_{min}. From left to right, the columns illustrate the fraction of regions within each application that is inherently idempotent for different values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. With $P_{min} = \emptyset$, the left-most column 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.

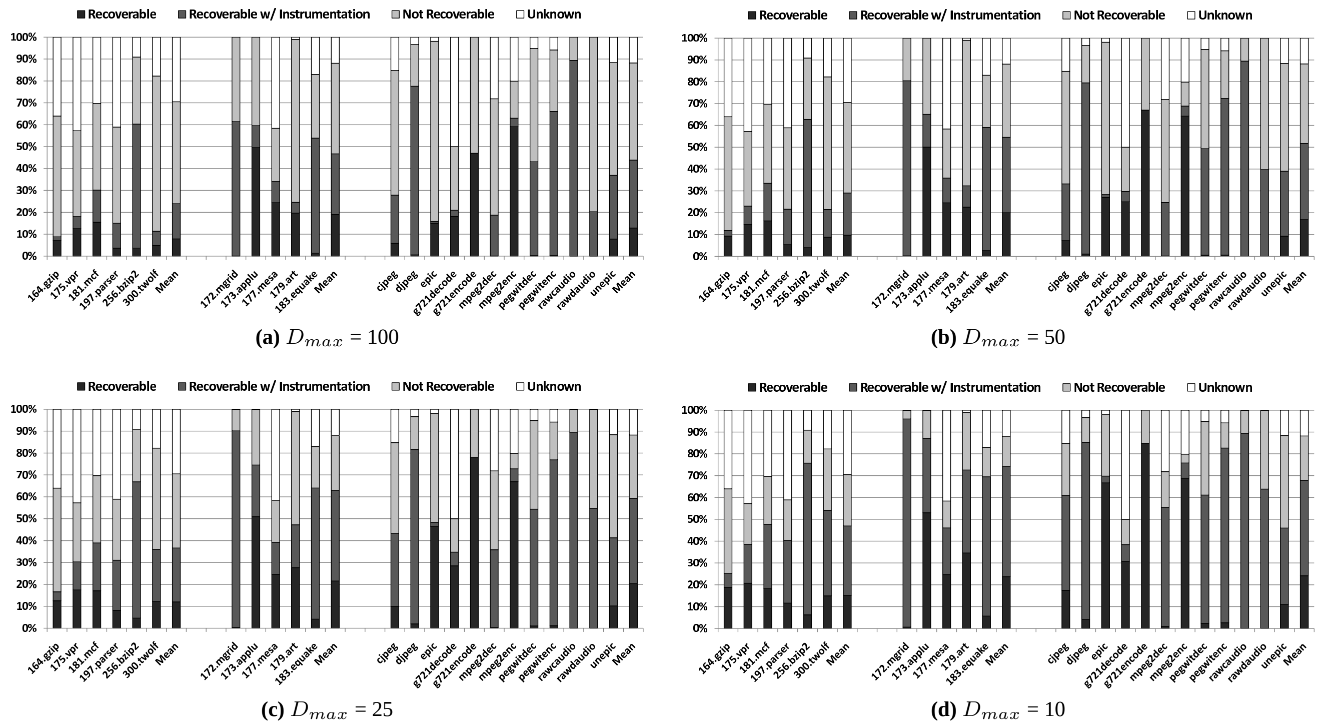


Figure 4: Recoverability coverage for different values of maximum fault detection latency, D_{max} (in instructions). Coverage is presented as the % of program execution *time* (excluding system/library calls) that can be protected by Encore. Results shown correspond to $P_{min} = 0.0$. *Unknown* segments correspond to execution time spent in portions of the application source code that could not be analyzed.

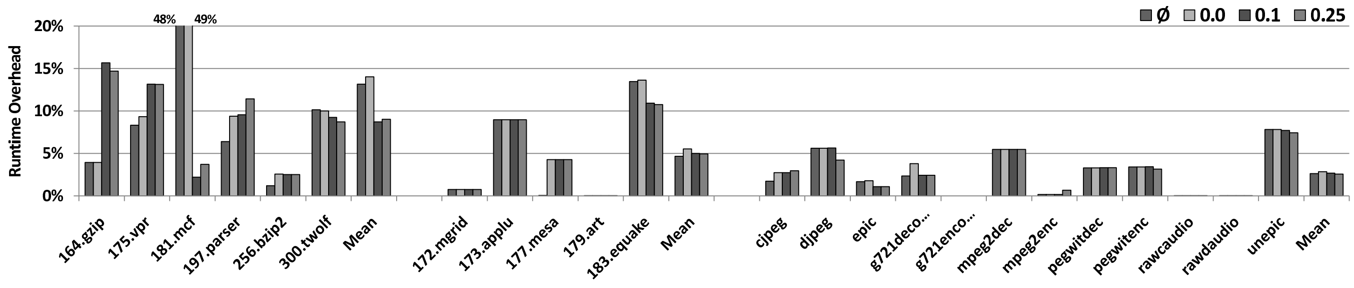
ing region sizes beyond an initial partitioning. Consequently, performance and coverage results in this section correspond to regions formed without subsequent applications of interval partitioning.

6. Related Work

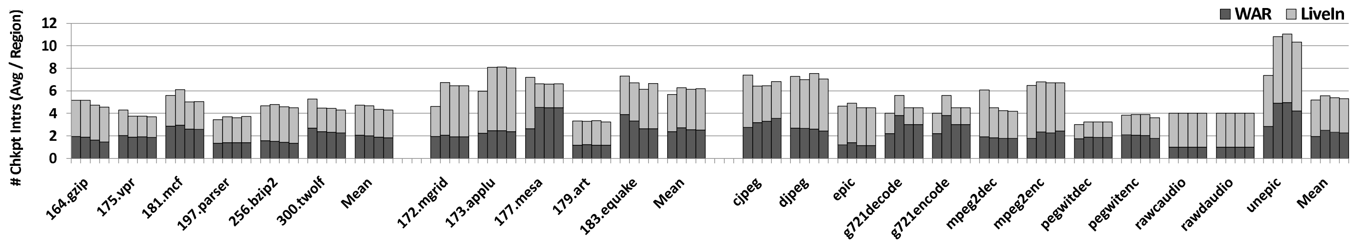
Fault tolerance in microprocessors can be broadly divided into three requisite steps: 1) detection/diagnosis of fault, 2) system repair/reconfiguration and 3) recovery to an error-free state. While dealing with transient faults, the second step becomes uneccessary since they do not cause any persistent damage. Thus, fault tolerance for transient faults boils down to simply *fault detection* and *system recovery*. As the focus of our paper is on system recovery, this section provides only a brief overview of detection solutions, while providing a more detailed discussion of previous efforts in system recovery.

6.1 Fault Detection

There exists a large body of research addressing the challenge of fault detection [8, 11, 12, 16, 19, 21–23, 30]. These efforts can be broadly divided into three 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 [22] and dual-core execution [26] are good examples of this. Second are solutions that exploit temporal redundancy, where the same work is re-executed on the same hardware resource. Instruction duplication [21] and selective instruction duplication [15] are well known techniques that fall into this category. Third category is of techniques that rely on high-level symptoms [18, 23, 30] or specialized detectors [11, 32] to catch a fault. Finally, there have also been recent proposals that formulate hybrid solutions [8, 19] that combine techniques from the above categories for a lower cost and higher coverage.



(a) Overall runtime performance overhead.



(b) The average number of checkpointing (static) instructions inserted per region. Enforcing idempotence in otherwise non-idempotent regions requires two types of checkpointing instructions: 1) *W*AR - needed for write-after-read memory dependences and 1) *L*iveIn - needed for register to memory checkpoint of live-in registers. *W*AR instructions are simply memory-to-memory moves (load + store) and *L*iveIn instructions are register-to-memory moves (store).

Figure 5: Encore overheads. In both (a) and (b) overheads, for each application, are shown for different values of P_{min}. From left to right the columns correspond to values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. For $P_{min} = \emptyset$ no dynamically-dead code is pruned from the idempotent analysis. Overheads are shown for fault-free executions. In (a), the negligible performance for recovery is omitted since fault events are still orders of magnitude "rarer" than the fault free cases. For (b), the instructions required for recovery (i.e., branch and state restoration) are not included because these can typically be amortized with the detection scheme.

6.2 System Recovery

Once a fault has been 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 (in hardware or software) to enable the rollback. The most popular category of recovery solutions are checkpoint based. In their simplest incarnation, checkpoint-recovery solutions periodically save off the entire system state, and revert to the most recent version in the event of a fault. In the remainder of this section, we examine few prominent examples of checkpoint-recovery schemes in greater detail.

Enterprise-level Recovery. Traditionally, checkpoint-recovery solutions have been used in large scale enterprise systems to provide the guarantee the often touted "5 nines" of reliability. These systems, with 100-1000s of nodes, take periodic snapshots of the entire memory system, usually storing it onto a globally accessible disk [4]. To maintain consistency, all the nodes in the system take their checkpoints at the same time, causing a surge in bandwidth requirements. In order to accelerate this process faster, the original application running on the system is stalled while the checkpoint is being created. Enterprise-level checkpoints are usually created using software libraries [14], some of which are even open source [6]. In additions to these library based solutions, some IBM mainframes also rely on small hardware modifications like register file checkpointing mechanisms and write-through caches to recover from processor and memory system errors [28]. In general these enterprise-level solutions are appropriate for their domain, but the cost of these creating these checkpoints (requiring on the order of minutes) are prohibitively high outside all but the most mission-critical systems.

Architectural Recovery. A cheaper alternative to taking a complete system snapshot is to record incremental changes to the system state in a log. In the event of a failure, these changes can be unrolled as needed. SafetyNet [27] and ReVive [17] are two examples of such log-based recovery solutions. SafetyNet maintains the log in local cache and memory buffers, storing the first overwrite to every memory location. By using a distributed mechanism, checkpoint creation and recovery times are very short. ReVive, on the other hand, does not modify the cache hierarchy, and stores the log in the main memory. This allows it to recover even if a node is permanently lost. However, this ability comes at the cost of extra network and memory traffic, resulting in a larger performance overhead. Although log-based recovery solutions are scalable to more frequent checkpoints, and smaller intervals, the overheads from the required hardware additions and the accompanying complexity that is introduced as a result are not insignificant. This makes them impractical for budget weary commodity systems.

Opportunistic Recovery. This last category of recovery solutions may not technically be recovery schemes in the conventional sense. Works like Relax [3] and others [24, 25, 29] have 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. Consequently, an attempt is not always made to correct the effects of a transient fault. Only when the fault is expected to significantly impact the "quality" of externally visible results is proper rollback recovery every initiated [3].

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 day computing. Yet despite this growing reliability concern, we would argue that instead of appropriating large transistor budgets (or processor cycles) to hedge against growing fault rates, system architects should embrace the high degree of fault tolerance that can be had simply by trading in provable guarantees for probabilistic estimates. Such tradeoffs may be the most attractive for low-end commodity and embedded markets, where systems often cannot afford to devote a substantial portion of their computing resources to anything other than actually performing computations. With recoverability coverage at 70% on average for floating point and embedded applications, and as high as 85% and 95% for individual programs, relying on Encore would be sufficient. Simply budgeting a 6% performance overhead could provide sufficient transient tolerance, freeing designers to focus their attention on other aspects of the system architecture.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] 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.

[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. 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] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

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

[12] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[13] S. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *International Symposium on Microarchitecture*, pages 29–42, Dec. 2003.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones. Stochastic computation. 2010.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, 2007.

[31] 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.

[32] 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.