Static Analysis and Compiler Design for Idempotent Processing

Marc de Kruijf Karthikeyan Sankaralingam Somesh Jha
University of Wisconsin – Madison
{dekruijf,karu,jha}@cs.wisc.edu

Abstract

Recovery functionality has many applications in computing systems, from speculation recovery in modern microprocessors to fault recovery in high-reliability systems. Modern systems commonly recover using checkpoints. However, checkpoints introduce overheads, add complexity, and often save more state than necessary.

This paper develops a novel compiler technique to recover program state without the overheads of explicit checkpoints. The technique breaks programs into *idempotent regions*—regions that can be freely re-executed—which allows recovery without checkpointed state. Leveraging the property of idempotence, recovery can be obtained by simple re-execution. We develop static analysis techniques to construct these regions and demonstrate low overheads and large region sizes for an LLVM-based implementation. Across a set of diverse benchmark suites, we construct idempotent regions close in size to those that could be obtained with perfect runtime information. Although the resulting code runs more slowly, typical performance overheads are in the range of just 2-12%.

The paradigm of executing entire programs as a series of idempotent regions we call *idempotent processing*, and it has many applications in computer systems. As a concrete example, we demonstrate it applied to the problem of compiler-automated hardware fault recovery. In comparison to two other state-of-the-art techniques, redundant execution and checkpoint-logging, our idempotent processing technique outperforms both by over 15%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers

General Terms Algorithms, Design, Performance, Reliability

Keywords Idempotent processing, idempotent regions

1. Introduction

Recovery capability is a fundamental component of modern computer systems. It is used to recover from branch misprediction and out-of-order execution [33, 38], hardware faults [32, 34], speculative memory-reordering in VLIW machines [13, 18], optimistic dynamic binary translation and code optimization [11, 14], and transactional memory [17, 28]. In each of these cases, recovery is used to repair the state of the program in the rare event that an execution failure (i.e. a fault or mis-speculation) occurs.

Checkpoints provide a conceptually simple solution, and have strong commercial precedent [11, 32, 37]. However, checkpoints are problematic for several reasons. First, software checkpoints often have high performance overhead and hence, to maintain reasonable performance, hardware support is often necessary. This hardware support, however, forces interdependencies between processor structures, occupies space on the chip, and entails recurring energy expenditure regardless of failure occurrence. Particularly for emerging massively parallel and mobile processor designs, the per-core hardware support comes at a premium, while the recovery support may be desirable only under specific or rare circumstances. Hardware checkpointing resources are also rarely exposed to software, and are even less often configurable in terms of their checkpointing granularity, limiting their wider applicability. Finally, checkpoints have limited application visibility and are often overly aggressive in saving more state than is required by the application [9, 27].

To combat these difficulties, *idempotence*—the property that re-execution is free of side-effects—has been previously proposed as an alternative to checkpoints. In contrast to explicit checkpoints, idempotence allows the architecture state at the beginning of a code region to be used as an implicit checkpoint that is never explicitly saved or restored. In the event of an execution failure, idempotence is used to correct the state of the system by simple re-execution.

Over the years, idempotence has been both explicitly and implicitly employed as an alternative to checkpoints. Table 1 classifies prior work in terms of its application domain and the level at which idempotence is used and identified [2, 9, 10, 12, 16, 19, 21, 23, 25, 31, 36]. One of the earliest uses is by Mahlke *et al.* in using restartable instruction sequences for exception recovery in speculative processors [23]. More recently, Hampton and Asanović apply idempotence to support virtual memory on vector machines [16], Tseng and Tullsen apply idempotence to support data-triggered parallel thread execution [36], and Feng *et al.* leverage idempotence for low-cost hardware fault recovery [12]. As the table shows, idempotence has historically been applied only under specific domains such as exception recovery and multithreading, often only under restricted program scope, and often using only limited or no static analysis.

In this paper, we develop an analysis framework to enable each of the above uses (and others yet to be invented), irrespective of their application domain and their underlying purpose, and across entire programs. In particular, we develop static analysis techniques and a compilation strategy to statically partition programs into large idempotent regions. We develop a provably correct region partitioning algorithm, demonstrate a working compiler implementation, and demonstrate application to at least one specific problem domain. We bring together the somewhat disparate uses listed in

PLDI '12, June 11–16, 2012, Beijing, China.

Technique name	Application domain	Program scope	Idempotence discovery technique
Sentinel scheduling [23]	Speculative memory re-ordering	Speculative code regions	Register-level compiler analysis
Fast mutual exclusion [2]	Uniprocessor mutual exclusion	Atomicity primitives	Programmer inspection
Multi-instruction retry [21]	Branch mispredictions and faults	Whole program	Compiler antidependence analysis
Atomic heap transactions [31]	Memory allocation	Garbage collector	Programmer inspection
Reference idempotency [19]	Reducing speculative storage	Non-parallelizable code	Memory-level compiler analysis
Restart markers [16]	Virtual memory in vector machines	Loops and vector code	Compiler loop analysis
Relax [9]	Hardware fault recovery	Selected code regions	Programmer inspection
Data-triggered threads [36]	Data-triggered multi-threading	Selected code regions	Programmer inspection
Idempotent processors [10]	Processor simplification	Whole program	Unspecified compiler analysis
Encore [12]	Hardware fault recovery	Selected code regions	Compiler interval analysis
iGPU [25]	GPU exceptions and speculation	Whole GPU program	PTX-specific compiler analysis

Table 1. Uses of fine-grained idempotence in hardware and software design (in chronological order).

Table 1 and unify them under a single paradigm we call *idempotent processing*, which allows the synthesis of multiple such uses in a single system or implementation artifact.

In brief, our approach operates as follows. First, we note that using a conventional compiler, idempotent regions are typically small. Our static analysis eliminates the compilation artifacts responsible for these small idempotent region sizes by identifying regions in a function that are *semantically* idempotent. These regions are then compiled in such a way that no artifacts are introduced and the idempotence property is preserved throughout code generation. To do this, the compiler limits register and stack memory re-use, which reduces locality and thereby introduces runtime overhead. However, typical overheads are in the range of just 2-12%.

In exchange for these overheads, our analysis partitions a function into regions that are close in size to the largest regions that would be constructed given near-perfect runtime information. We show that the problem of finding very large idempotent regions can be cast as a vertex multicut problem, a problem known to be NP-complete in the general case. We apply an approximation algorithm and a heuristic that incorporates loop information to optimize for dynamic behavior and find that overall region sizes are in most cases close to ideal. Overall, we make the following contributions:

- We perform a detailed analysis of idempotent regions in conventional programs and quantitatively demonstrate that conventional compilers artificially inhibit the sizes of the idempotent regions in programs, severely limiting their usefulness.
- We develop static analysis and compiler techniques to preserve the inherent idempotence in applications and construct large idempotent regions. We formulate the problem of finding idempotent regions as a graph cutting (vertex multicut) problem and optimize for runtime behavior using a heuristic that incorporates loop information.
- We present a detailed characterization of our idempotent regions, which can be applied in the context of the various uses previously proposed in the literature.
- We demonstrate our idempotent processing solution applied to the problem of recovery from transient faults in microprocessors. Our idempotence-based recovery implementation performs over 15% better than two competing state-of-the-art compiler-automated recovery techniques.

The remainder of this paper is organized as follows. Section 2 gives a complete overview of this paper. Section 3 presents a quantitative study of idempotent regions as they exist inherently in application programs. Section 4 presents our idempotent region construction algorithm. Section 5 gives details of our compiler implementation. Section 6 presents our quantitative evaluation. Section 7 presents related work. Finally, Section 8 concludes.

2. Overview

This section provides a complete overview of this paper. We define idempotence in terms of data dependences and present a motivating example that illustrates how data dependences can inhibit idempotence. We show how these data dependences can be manipulated to grow the sizes of idempotent regions and give an overview of our partitioning algorithm that attempts to maximize the sizes of these regions. Finally, we describe how statically-identified idempotence can be used to recover from a range of dynamic execution failures.

2.1 Identifying Idempotent Regions

A region of code (assume a linear sequence of instructions for now) is idempotent if the effect of executing the region multiple times is identical to executing it only a single time. Intuitively, this behavior is achieved if the region does not overwrite its inputs. With the same inputs, the region will produce the same outputs. If a region overwrites its inputs, it reads the overwritten values when it re-executes, changing its behavior.

A variable is an input to a region if it is *live-in* to the region. Such a variable has a definition that reaches the region's entry point and has a corresponding use of that definition after the region entry point. Below, we use this observation to derive a precise definition of idempotence in terms of data dependences. We use the term *flow dependence* to refer to a read-after-write (RAW) dependence and the term *antidependence* to refer to a write-after-read (WAR) dependence.

By definition, a live-in variable has a flow dependence that spans the region's entry point. Because the variable's definition must come before the entry to the region, the definition is not inside the region, and hence there is no definition that precedes the first use of that variable inside the region. Hence, *a live-in has no flow dependence before the first use of that variable inside the region*. Since a live-in has no flow dependence, overwriting a live-in must occur after the point of the use. Thus, *an overwritten live-in has an antidependence after the absence of a flow dependence*. It follows that a region of code is idempotent *if it contains no antidependences not preceded by a flow dependence*.

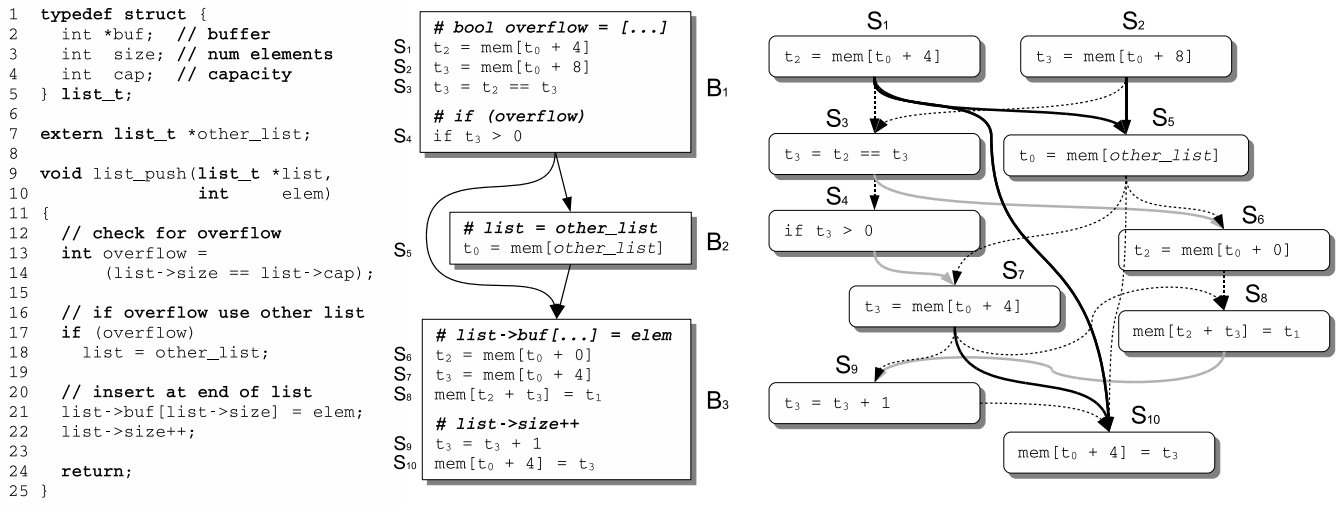
The table below shows three statement sequences involving a variable `x` and uses the above definition to identify whether the sequences are idempotent or not:

	RAW	RAW→WAR	WAR

Sequence	`x = 5`
	`y = x`
		`x = 5`
		`y = x`
		`x = 8`
			`y = x`
			`x = 8`

Idempotent?	Yes	Yes	*No*

The antidependence after no flow dependence chain that breaks the idempotence property as shown on the right we call a *clobber*



(a) A non-idempotent C function

(b) The control flow graph and compiler intermediate code

(c) A subset of the data dependence graph, showing flow dependences (dashed), clobber antidependences (solid dark), and non-clobber antidependences (solid light)

Figure 1. An example illustrating how clobber antidependences inhibit idempotence.

antidependence. Some clobber antidependences are strictly necessary according to program semantics. These clobber antidependences we label *semantic* clobber antidependences. The other clobber antidependences we label *artificial* clobber antidependences. The following example demonstrates semantic and artificial clobber antidependences.

A motivating example. For the remainder of this section, we use the C function shown in Figure 1(a) as a running example. The function, `list_push`, checks a list for overflow and then pushes an integer element onto the end of the list. The semantics of the function clearly preclude idempotence: even if there is no overflow, re-executing the function will put the element onto the end of the already-modified list, after the copy that was pushed during the original execution. As we will show, the source of the non-idempotence is the increment of the input variable `list->size` on line 22: without this increment, re-execution would simply cause the value that was written during the initial execution to be safely overwritten with the same value.

Figure 1(b) shows the function compiled to a load-store intermediate representation. The figure shows the control flow graph of the function, which contains three basic blocks B_1, B_2, and B_3. Inside each block are shown the basic operations, S_i, and the use of pseudoregisters, t_i, to hold operands and read and write values to and from memory. Figure 1(c) shows the program dependence graph of the function focusing only on flow dependences (dashed) and antidependences (solid). The antidependences are further distinguished as clobber antidependences (dark) and not clobber antidependences (light)[1]. The figure shows four clobber antidependences: $S_1 \rightarrow S_5$, $S_2 \rightarrow S_5$, $S_1 \rightarrow S_{10}$, and $S_7 \rightarrow S_{10}$. The first two clobber antidependences depend on S_5, which overwrites the pseudoregister t_0, and the second two depend on S_{10}, which overwrites the memory location at $t_0 + 4$.

The two clobber antidependences that depend on S_5 are unnecessary: they are *artificial* clobber antidependences. We can eliminate these clobber antidependences simply by writing to a different pseudoregister. Figure 2 shows the effect of replacing t_0 in S_5

[1] For simplicity, we assume the pointer argument `list`, the global `other_list`, and their respective buffer arrays are known not to alias.

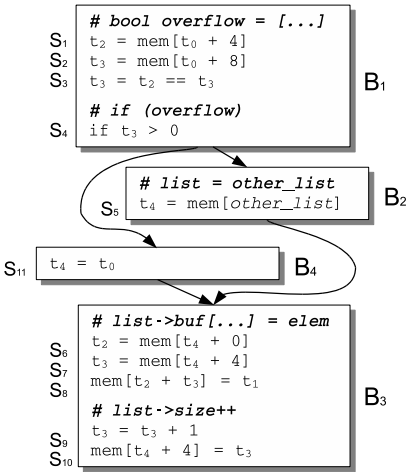


Figure 2. Renaming t_0 in S_5 to t_4.

with a new pseudoregister t_4. All uses of t_0 subsequent to S_5 are renamed to use t_4 as well, and a new statement S_{11} is inserted that moves t_0 into t_4 along the path where S_5 is not executed. S_{11} is placed inside a new basic block B_4 which bridges B_1 and B_3. A compiler can then permanently eliminate these artificial clobber antidependences by ensuring that t_4 and t_0 are not assigned to the same physical register or the stack slot during register allocation. If they are, then t_4 will overwrite t_0 and the two clobber antidependences will simply re-emerge. To do this, the register allocator can be constrained such that all pseudoregisters that are live-in to the region are also marked as live-out to the region. This enables idempotence in exchange for some additional register pressure.

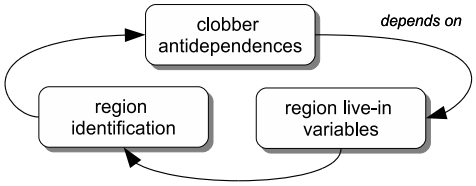
The final two clobber antidependences write to the memory location $t_0 + 4$ in S_{10}, which corresponds with the store of the `list->size` increment on line 22 of Figure 1(a). Unfortunately, the destination of this store is fixed by the semantics of the program: if we didn't increment this variable, then the function would not increase the size of the list, which would violate the semantics

of the function. Because we cannot legally rename the store destination, we label these clobber antidependences *semantic* clobber antidependences.

Summary. Table 2 summarizes the differences between semantic and artificial clobber antidependences. Semantic clobber antidependences act on heap, global, and non-local stack memory, which we hereafter often refer to as just "memory". These memory locations are not under the control of the compiler; they are specified in the program itself and cannot be re-assigned. In contrast, artificial clobber antidependences act on "pseudoregister" locations: registers and local stack memory. These resources are compiler controlled, and assuming effectively infinite stack memory, can be arbitrarily re-assigned. While in practice stack memory is limited, our compiler does not grow the size of the stack significantly and we have no size-related difficulties compiling any benchmarks.

2.2 Constructing Idempotent Regions

Assuming we can eliminate all artificial clobber antidependences, we show in Section 3 that the idempotent regions that exist in application programs are potentially very large. However, the problem of statically constructing large idempotent regions remains surprisingly non-trivial. In principle, the problem should be as simple as merely identifying and constructing regions that contain no semantic clobber antidependences. However, this solution is circularly dependent on itself: identifying semantic clobber antidependences requires identification of region live-in variables, which in turn requires identification of the regions. This circular dependence is illustrated below:



Our solution to this problem is to transform the function so that, with the exception of self-dependent pseudoregister antidependences[2], *all antidependences are necessarily semantic clobber antidependences*. We then construct idempotent regions by identifying regions that contain no antidependences. Antidependence information does not depend on region live-in information, and hence the circular dependence chain is broken. During this process, self-dependent pseudoregister antidependences are optimistically assumed not to emerge as clobber antidependences; those that would emerge as clobber antidependences after the region construction are patched in a subsequent refinement step.

Considering only antidependence information, we show that the problem of partitioning the program into idempotent regions is equivalent to the problem of "cutting" the antidependences, such that a cut before statement S starts a new region at S. In this manner, no single region contains both ends of an antidependence and hence the regions are idempotent. To maximize the region sizes, we cast the problem in terms of the NP-complete vertex multicut problem and use an approximation algorithm to find the minimum set of cuts, which finds the minimum set of regions. This maximizes the average static region size, and we then employ heuristics to maximize the sizes of regions as they occur dynamically at runtime. We refer to the overall algorithm as our *idempotent region construction algorithm*. In Section 4 we describe the algorithm in detail and in Section 5 we discuss the specifics of our implementation.

[2] These are antidependences that occur across loop iterations and have assignments of the form $t_i = f(t_i)$.

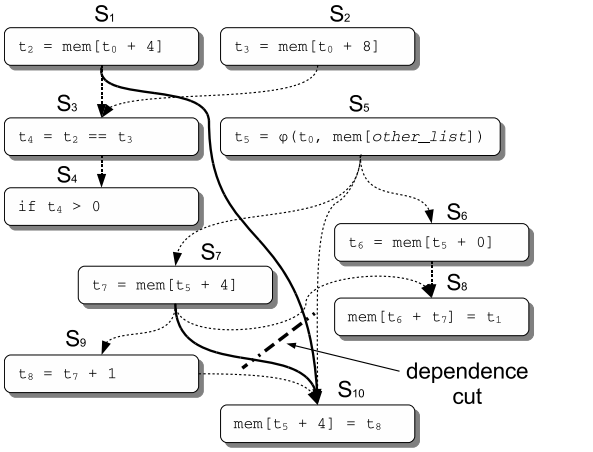


Figure 3. The data dependence graph in SSA form.

Figure 3 shows our algorithm applied to our running example. The initial step in the process is the conversion of all pseudoregister assignments to static single assignment (SSA) form [8]. The figure shows the dependence graph of Figure 1(c) simplified by the SSA transformation. The code structure is the same as in Figure 1(b) except that an SSA ϕ-node is placed at the head of B_3. The ϕ-node we label S_5 and we fold the original S_5 into it. Under SSA, the artificial clobber antidependences disappear and the semantic ones remain. Both semantic clobber antidependences write to memory location `mem[`t_5 `+ 4]` in statement S_{10}, one with a may-alias read in statement S_1 and the other with a must-alias read in statement S_7. In general, the problem of finding the best places to cut the antidependences is NP-complete. However, for this simple example the solution is straightforward: it is possible to place a single cut that cuts both antidependences. The cut can be placed before S_8, S_9, or S_{10}. Regardless of where the cut is placed, the function is ultimately divided into three idempotent regions in total: under our initial definition of a region as a linear instruction sequence, depending on the outcome of the control decision inside the function, two linear sequences exist up to the point of the cut, and after the cut there is one additional sequence.

2.3 Using Idempotence for Recovery

In this section, we address how statically-identified idempotence can be used to recover from dynamic execution failures. This is not obvious because an execution failure may have side-effects that can corrupt arbitrary state. For instance, as a result of microprocessor branch misprediction or due to a hardware soft error, a region's inputs may be accidentally modified, or state that is outside the scope of the current region may be modified (resulting in problems for other regions).

Tolerating control flow errors. We first consider the possibility of incorrect control flow, which can arise due to e.g. a branch misprediction and can cause an incorrectly executed region to overwrite an input of the correct region or its succeeding regions.

Fortunately, tolerating control flow errors is relatively straightforward with our method of placing region boundaries using cuts. First, we observe that each control flow path from a given cut C to any subsequent cut constitutes an idempotent region, and hence any incorrect execution *belongs to an idempotent region with the same entry point at C as the region of the correct execution*. Hence, by compiling in such a way that the inputs of *all* regions starting at C are preserved between C and all subsequent cut points, execution can be made idempotent regardless of control flow.

Type of Clobber Antidependence	Storage Resources
Semantic clobber antidependence	Heap, global, non-local stack memory ("memory")
Artifial clobber antidependence	Registers and local stack memory ("pseudoregisters")

Table 2. Semantic and artificial clobber antidependences and the resources on which they operate.

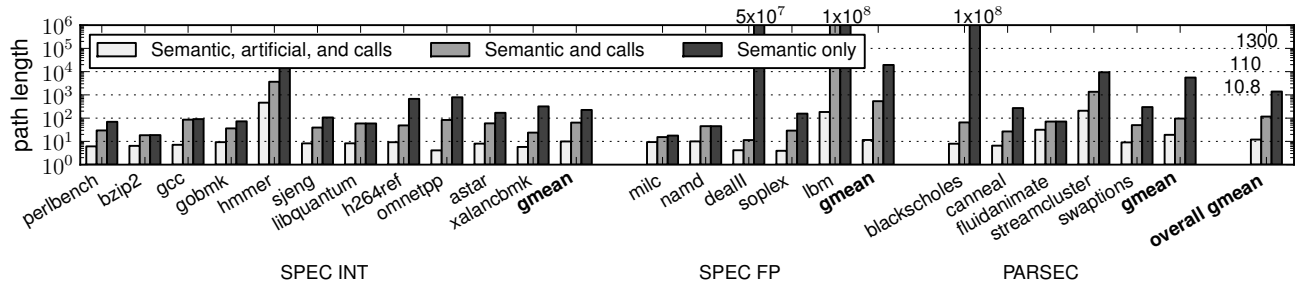


Figure 4. Average dynamic idempotent region sizes in the limit (y-axis is log-scale).

At this point, we rework our definition of idempotent region to achieve this property. Our previous definition of a region as an instruction sequence we now label more precisely as an *idempotent path*, and we redefine an idempotent region as a collection of idempotent paths that share the same entry point. (Thus, a *region* is a subset of a program's control flow graph with a single unique entry point and multiple possible exit points, and a *path* is a trace of instructions from such a region's entry point to one of its exit points.) An idempotent region preserves idempotence regardless of control flow by considering as inputs the collective inputs of all its containing paths. Unfortunately, however, it is now no longer possible to say that an input is necessarily overwritten only after the point of a read, as in the definition of idempotence in terms of a clobber antidependences; with this definition of region, a live-in may be written before it is read along an incorrect control flow path. This forces a more conservative compiler analysis.

For registers and stack memory, the impact of this conservative analysis is manageable and the static analysis and compiler implementation we describe in this paper preserves pseudoregister inputs regardless of control flow. However, for other types of storage—namely, heap, global, and non-local stack memory—we find the compiler's limited control over these resources too constricting. In particular, with the possibility for differing control flow on re-execution, an idempotent region would not be able to contain a store to a given address unless such a store occurred along all paths through the region. Thus, for memory we assume that stores are buffered and not released until control flow has been verified. We propose to re-use the store buffer that already exists in modern processors for this purpose. This allows us to reason about inputs from non-local memory more optimistically in terms of clobber antidependences, considering separately each path through an idempotent region. Considering our modified definition of a region, the first two idempotent paths (formerly, "regions") in the example of Figure 1 merge as one idempotent region.

Tolerating arbitrary errors. While the situation with hardware soft errors may appear much more dire, in practice techniques like error-correcting codes (ECC) can effectively protect existing memory and register state, leaving instruction execution itself as the only possible source of error. Since idempotence already precludes overwriting input state, regardless of whether the value written is corrupted or not, only three additional requirements are needed: (1) instructions must write to the correct register destinations, (2) execution must follow the program's static control flow edges, and (3) as with branch misprediction, stores must be buffered until they are verified. Altogether, these four total mechanisms—ECC, register destination verification, control flow verification, and store verification—are widely assumed in prior work on error recovery in software [6, 9, 21], and a variety of compatible hardware and software techniques have been previously developed [5, 24, 29, 30].

Recovering non-idempotent instructions. Finally, we note that some instructions such as certain memory-mapped I/O operations and some types of synchronization instructions are inherently non-idempotent. In this work, we consider such non-idempotent instructions as single-instruction idempotent regions that either terminate correctly or have no side-effects that make it unsafe for them to be re-executed for recovery.

3. Exploring the Potential: A Limit Study

To understand how much artificial clobber antidependences inhibit idempotent region sizes, we performed a limit study to ascertain the nature of the clobber dependences that emerge during program execution. Our goal is to understand the extent to which it would be possible to construct idempotent regions given perfect runtime information.

Methodology. We used the gem5 simulator [4] to measure the lengths of the idempotent paths that dynamically execute through a region across a range of benchmarks compiled for the ARMv7 instruction set. For each benchmark, we measured the distribution of path lengths occurring over a 100 million instruction period starting after the setup phase of the application. We evaluated two benchmark suites: SPEC 2006 [35], a suite targeted at conventional single-threaded workloads, and PARSEC [3], a suite targeted at emerging multi-threaded workloads.

We use a conventional optimizing compiler to generate program binaries, and measure idempotent path length optimistically as the number of instructions between dynamic occurrences of clobber antidependences. This optimistic (dynamic) measurement is used in the absence of explicit (static) region markings in these conventionally-generated binaries. We study idempotent regions divided by three different categories of clobber antidependences: (1) only semantic clobber antidependences, (2) only semantic clobber antidependences with regions split at function call boundaries, and (3) both semantic and artificial clobber antidependences with regions split at function call boundaries.

We consider as artificial clobber antidependences all clobber antidependences on registers and those with writes relative to the stack pointer, which are universally register spills for our compiler.

These are the clobber antidependences that can generally be eliminated by renaming pseudoregisters and careful register and stack slot allocation. We assume the remaining clobber antidependences are all semantic. We consider separately regions divided by semantic clobber antidependences that cross function call boundaries to understand the potential improvements of an inter-procedural compiler analysis over an intra-procedural one. To explore what is achievable in the inter-procedural case, we optimistically assume that call frames do not overwrite previous call frames. We also optimistically ignore antidependences that necessarily arise due to the calling convention (e.g. overwriting the stack pointer) and assume the calling convention can be redefined or very aggressive inlining can be performed such that this obstacle is weakened or removed.

Results and conclusions. Our experimental results, shown in Figure 4, identify three clear trends. First, we see that regions divided by both artificial and semantic clobber antidependences are much smaller than those divided by semantic clobber antidependences alone. The geometric mean path length considering both types is 10.8 instructions, while the length considering just semantic clobber antidependences is 110 instructions intra-procedurally (a 10x gain) and 1300 inter-procedurally (a 120x gain).

The second trend is a substantial gain (more than 10x) from allowing idempotent regions divided by semantic clobber antidependences to cross function boundaries. However, the gains are not reliably as large as the 10x gain achieved by removing the artificial clobber antidependences alone: the difference drops to only 4x when we drop the two outliers dealII and blackscholes.

The third and final trend is that path lengths tend to be larger for PARSEC and SPEC FP than for SPEC INT. PARSEC and SPEC FP benchmarks tend to overwrite their inputs relatively infrequently due to their memory streaming and compute-intensive nature.

Overall, we find that (1) there is a lot of opportunity to grow idempotent region sizes by eliminating artificial clobber antidependences, (2) an intra-procedural static analysis is a good starting point for constructing large idempotent regions, and (3) the greatest opportunity appears to lie with streaming and compute-intensive applications.

4. Region Construction Algorithm

In this section, we describe our idempotent region construction algorithm. The algorithm is an intra-procedural compiler algorithm that divides a function into idempotent regions. First, we describe the transformations that allow us to cast the problem of constructing idempotent regions in terms of cutting antidependences. Second, we describe the core static analysis technique for cutting antidependences, including optimizations for dynamic behavior. Finally, we describe our register and stack slot allocation to preserve the idempotence of our identified regions through code generation.

4.1 Program Transformation

Before we apply our static analysis, we first perform two code transformations to maximize the efficacy of the analysis. The two transformations are (1) the conversion of all pseudoregister assignments to static single assignment (SSA) form, and (2) the elimination of all memory antidependences that are not clobber antidependences. The details on why and how are given below.

The first transformation converts all pseudoregister assignments to SSA form. After this transformation, each pseudoregister is only assigned once and all artificial clobber antidependences are effectively eliminated (self-dependent artificial clobber antidependences, which manifest in SSA through ϕ-nodes at the head of loops, still remain, but it is safe to ignore them for now). The intent of this transformation is to expose primarily the semantic antidependences to the compiler. Unfortunately, among these antidependences we still do not know which are clobber antidependences and which are not, since, as explained in Section 2.2, this determination is circularly dependent on the region construction we are trying to achieve. Without knowing which antidependences will emerge as clobber antidependences, we do not know which antidependences must be cut to form the regions. Hence, we attempt to refine things further.

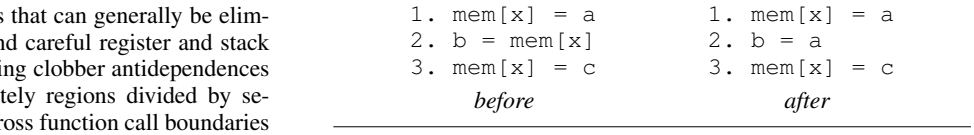


Figure 5. Eliminating non-clobber memory antidependences.

After the SSA transformation, it follows that the remaining antidependences are either self-dependent antidependences on pseudoregisters or antidependences on memory locations. For those on memory locations, we employ a transformation that resolves the aforementioned ambiguity regarding clobber antidependences. The transformation is a simple redundancy-elimination transformation illustrated by Figure 5. The sequence on the left has an antidependence on memory location x that is not a clobber antidependence because the antidependence is preceded by a flow dependence. Observe that in all such cases the antidependence is made redundant by the flow dependence: assuming both the initial store and the load of x "must alias" (if they only "may alias" we must conservatively assume a clobber antidependence) then there is no reason to reload the stored value since there is an existing pseudoregister that already holds the value. The redundant load is eliminated as shown on the right of the figure: the use of memory location x is replaced by the use of pseudoregister a and the antidependence disappears.

Unfortunately, there is no program transformation that resolves the uncertainty for self-dependent pseudoregister antidependences. In the following section, we initially assume that these antidependences can be register allocated such that they do not become clobber antidependences (i.e. we can precede the antidependence with a flow dependence on its assigned physical register or stack slot). Hence, we construct regions around them, considering only the known, memory-level clobber antidependences. After the construction is complete, we check to see if our assumption holds. If not, we insert additional region cuts as necessary.

4.2 Static Analysis

After our program transformations, our static analysis constructs idempotent regions by "cutting" all potential clobber antidependences in a function. The analysis consists of two parts. First, we construct regions based on semantic antidependence information by cutting memory-level antidependences and placing region boundaries at the site of the cuts. Second, we further divide loop-level regions as needed to accommodate the remaining self-dependent pseudoregister clobber antidependences.

4.2.1 Cutting Memory-Level Antidependences

To ensure that a memory-level antidependence is not contained inside a region, it must be split across the boundaries between regions. Our algorithm finds the set of splits, or "cuts", that creates the smallest number of these regions. In this section, we derive our algorithm as follows:

1. We define our problem as a graph decomposition that must satisfy certain conditions.
2. We reduce the problem of finding an optimal graph decomposition to the minimum vertex multicut problem.

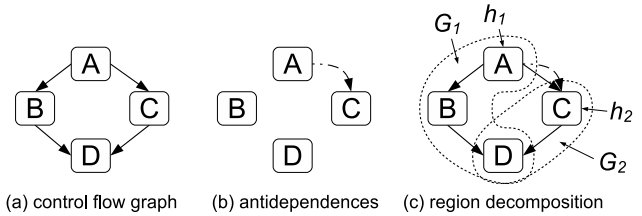


Figure 6. An example region decomposition.

3. To generate a solution, we formulate the problem in terms of the hitting set problem.
4. We observe that a near-optimal hitting set can be found efficiently using an approximation algorithm.

Problem definition. For a control flow graph $G = (V, E)$ we define a region as a sub-graph $G_i = (V_i, E_i, h_i)$ of G, where $h_i \in V_i$ and all nodes in V_i are reachable from h_i through edges in E_i. We call h_i the *header node*[3] of G_i. A *region decomposition* of the graph G is a set of sub-graphs $\{G_1, \cdots, G_k\}$ that satisfies the following conditions:

- each node $v \in V$ is in at least one sub-graph G_i,
- the header nodes for the sub-graph are distinct (for $i \neq j$, $h_i \neq h_j$), and
- no antidependence edge is contained in a sub-graph G_i for $1 \leq i \leq k$.[4]

Our problem is to decompose G into the smallest set of sub-graphs $\{G_1, \cdots, G_k\}$. Figure 6 gives an example. Figure 6(a) shows a control flow graph G and 6(b) shows the set of antidependence edges in G. Figure 6(c) shows a possible region decomposition for G. The shown region decomposition happens to be optimal; that is, it contains the fewest possible number of regions.

Reduction to vertex multicut. We now reduce the problem of finding an optimal region decomposition with the problem of finding a minimum vertex multicut.

Definition 1. (Vertex multicut) Let $G = (V, E)$ be a directed graph with set of vertices V and edges E. Assume that we are given pairs of vertices $A \subseteq V \times V$. A subset of vertices $H \subseteq V$ is called a *vertex multicut* for A if in the subgraph G' of G where the vertices from H are removed, for all ordered pairs $(a, b) \in A$ there does not exist a path from a to b in G'.

Let $G = (V, E)$ be our control flow graph, A the set of antidependence edge pairs in G, and H a vertex multicut for A. Each $h_i \in H$ implicitly corresponds to a region G_i as follows:

- The set of nodes V_i of G_i consists of all nodes $v \in V$ such that there exists a path from h_i to v that *does not* pass through a node in $H - \{h_i\}$.
- The set of edges E_i is $E \cap (V_i \times V_i)$.

It follows that a *minimum* vertex multicut $H = \{h_1, \cdots, h_k\}$ directly corresponds to an optimal region decomposition $\{G_1, \cdots, G_k\}$ of G over the set of antidependence edge pairs A in G.

[3] Note that, while we use the term *header node*, we do not require that a header node h_i dominates all nodes in V_i as defined in other contexts [1].

[4] This condition is stricter than necessary. In particular, an antidependence edge in G_i *with no path connecting the edge nodes*—implying that the the antidependence is formed over a loop revisiting G_i—is safely contained in G_i. However, determining the absence of such a path requires a path-sensitive analysis. We limit our solution space to path-insensitive analyses.

Solution using hitting set. The vertex multicut problem is NP-complete for general directed graphs [15]. To solve it, we reduce it to the hitting set problem, which is also NP-complete, but for which good approximation algorithms are known [7].

Definition 2. (Hitting set) Given a collection of sets $C = \{S_1, \cdots, S_m\}$, a *minimum hitting set* for C is the smallest set H such that, for all $S_i \in C$, $H \cap S_i \neq \emptyset$.

Note that we seek a set $H \subseteq V$ such that, for all $(a_i, b_i) \in A$, all paths π from a_i to b_i have a vertex in H (in other words, H is a "hitting set" of $\Pi = \cup_{(a_i,b_i)\in A}\pi_i$, where π_i is the set of paths from a_i to b_i). This formulation is not computationally tractable, however, as the number of paths between any pair (a_i, b_i) can be exponential in the size of the graph. Instead, for each $(a_i, b_i) \in A$, we associate a single set $S_i \subseteq V$ that consists of the set of nodes that dominate b_i but do not dominate a_i. We then compute a hitting set H over $C = \{S_i | S_i \text{ for } (a_i, b_i) \in A\}$. Using Lemma 1 it is easy to see that for all antidependence edges $(a_i, b_i) \in A$, every path from a_i to b_i passes through a vertex in H. Hence, H is both a hitting set for C and a vertex multicut for A.

We use a greedy approximation algorithm for the hitting set problem that runs in time $O(\sum_{S_i \in C} |S_i|)$. This algorithm chooses at each stage the vertex that intersects the most sets not already intersected. This simple greedy heuristic has a logarithmic approximation ratio [7] and is known to produce good quality results.

Lemma 1. Let $G = (V, E, s)$ be a directed graph with entry node $s \in V$ and (a, b) be a pair of vertices. If $x \in V$ dominates b but does not dominate a, then every path from a to b passes through x.

Proof: We assume that a pair of vertices (a, b) are both reachable from the entry node s. Let the following conditions be true.

- **Condition 1:** There exists a path from (a, b) that does not pass through the node x.
- **Condition 2:** There exists a path from s to a that does not pass through x.

If conditions 1 and 2 are true, then there exists a path from s to b that does not pass through x. This means x cannot dominate b. In other words, conditions 1 and 2 imply that x cannot dominate b.

Given that x dominates b, one of the conditions 1 and 2 must be false. If condition 1 is false, we are done. If condition 2 is false, then x dominates a, which leads to a contradiction. □

4.2.2 Cutting Self-Dependent Pseudoregister Antidependences

After memory antidependences have been cut, we have a preliminary region decomposition over the function. From here, we consider the remaining category of clobber antidependences—the self-dependent pseudoregister antidependences—and allocate them in such a way that they do not emerge as clobber antidependences.

In SSA form, a self-dependent pseudoregister antidependence manifests as a write occurring at the point of a ϕ-node assignment, with one of the ϕ-node's arguments data-dependent on the assigned pseudoregister itself. Due to SSA's dominance properties, such self-dependent pseudoregister assignments alway occur at the head of loops. Figure 7(a) provides a very simple example. Note that in the example the self-dependent "antidependence" is actually two antidependences, $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_1$. We refer to it as only a single antidependence for ease of explanation.

To prevent self-dependent pseudoregister antidependences from emerging as clobber antidependences, the invariant we must enforce is that a loop containing such an antidependence either contains no cuts or contains at least two cuts along all paths through the loop body. If either of these conditions is already true, no modi-

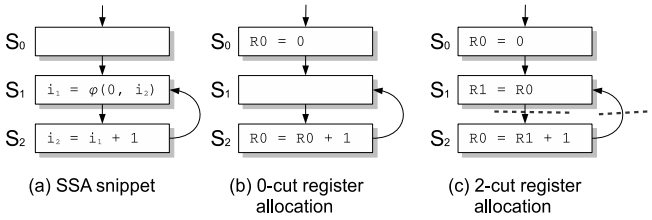


Figure 7. Clobber-free allocation of self-dependent pseudoregister antidependences.

fication to the preliminary region decomposition is necessary. Otherwise, we insert additional cuts such that the second condition becomes true. The details on why and how are provided below.

Case 1: A loop with no cuts. Consider the self-dependent antidependence shown in Figure 7(a). For a loop that contains no cuts, this antidependence can be trivially register allocated as shown in Figure 7(b). In the figure, we define the register (which could also be a stack slot if registers are scarce) of the antidependence outside the loop and hence across all loop iterations all instances of the antidependence are preceded by a flow dependence.

Case 2: A loop with at least 2 cuts. A self-dependent antidependence for a loop that contains at least two cuts can also be trivially-register allocated as shown in Figure 7(c). Here the antidependence is manipulated into two antidependences, one on R0 and one on R1, and the antidependences are placed so that they straddle region boundaries. Note that, for this to work, at least two cuts must exist along *all paths* through the loop body. This is obviously true in Figure 7(c) but in the general case it may not be.

Case 3: Neither case 1 or 2. In the remaining case, the self-dependent antidependence is in a loop that contains at least one cut but there exist one or more paths through the loop body that do not cross at least two cuts. In this case, we know of no way to register allocate the antidependence such that it does not emerge as a clobber antidependence. Hence, we resign ourselves to cutting the antidependence so that we have at least two cuts along all paths in the loop body, as in Case 2. This produces a final region decomposition resembling Figure 7(c).

4.3 Optimizing for Dynamic Behavior

Our static analysis algorithm optimizes for static region sizes. However, when considering loops, we know that loops tend to execute multiple times. We can harness this information to grow the sizes of the dynamic paths that execute through our regions at runtime.

We account for loop information by incorporating a simple heuristic into the hitting set algorithm from Section 4.2.1. In particular, we adjust the algorithm to greedily choose cuts at nodes from the outermost loop nesting depth first. We then break ties by choosing a node with the most sets not already intersected as normal. This improves the path lengths substantially in general, although there are cases where it reduces them. A better heuristic most likely weighs both loop nesting depth and intersecting set information more evenly, rather than unilaterally favoring one. Better heuristics are a topic for future work.

4.4 Code Generation

With the idempotent regions constructed, the final challenge is to code generate—specifically, register and stack allocate—the function so that artificial clobber antidependences are not re-introduced.

To do this, we constrain the register and stack memory allocators such that all pseudoregisters that are live-in to a region are also live-out to the region. This ensures that all registers and stack slots that contain input are not overwritten and hence no new clobber antidependences emerge.

5. Compiler Implementation

We implemented the region construction algorithm of Section 4 using LLVM [20]. Each phase of the algorithm is implemented as described below.

Code transformation. Of the two transformations described in Section 4.1, the SSA code transformation is automatic as the LLVM intermediate representation itself is in SSA form. We implement the other transformation, which removes all non-clobber memory antidependences, using an existing LLVM redundancy elimination transformation pass.

Cutting memory-level antidependences. We gather memory antidependence information using LLVM's "basic" alias analysis infrastructure. The antidependence cutting is implemented exactly as described in Section 4.2.1.

Cutting self-dependent pseudoregister antidependences. We handle self-dependent register antidependences as in Section 4.2.2 with one small enhancement: before inserting cuts, we attempt to unroll the containing loop once if possible. The reason is that inserting cuts increases the number of idempotent regions and thereby reduces the size of the loop regions. By unrolling the loop once, we can place the second necessary cut in the unrolled iteration. This effectively preserves region sizes on average. It also improves the performance of the register allocator by not requiring the insertion of extra copy operations between loop iterations (enabling a form of double buffering).

Optimizations for dynamic behavior. We optimize for dynamic behavior exactly as described in Section 4.3.

Code generation. We extend LLVM's register allocation passes to generate machine code as described in Section 4.4. To maintain the calling convention, functions that contain only a single region are split into two regions to allow parameter values to be overwritten by return values as necessary.

6. Evaluation

For evaluation, we first present the characteristics (region sizes and runtime overhead) produced by our idempotent region construction across a range of applications. We then present results evaluating our technique for recovery from transient hardware faults.

6.1 Methodology

We evaluate benchmarks from the SPEC 2006 [35] and PARSEC [3] benchmark suites. We compile each benchmark to two different binary versions: an *idempotent binary*, compiled using our idempotent region construction implemented in LLVM; and an *original binary*, generated using the regular optimized LLVM compiler flow.

Our performance results are obtained for the ARMv7 instruction set simulating a modern two-issue processor using the gem5 simulator [4]. To account for the differences in instruction count between the idempotent and original binary versions, simulation length is measured in terms of the number of functions executed, which is constant between the two versions. All benchmarks are fast-forwarded the number of function calls needed to execute at least 5 billion instructions on the original binary, and execution is then simulated for the number of function calls needed to execute 100 million additional instructions on the original binary.

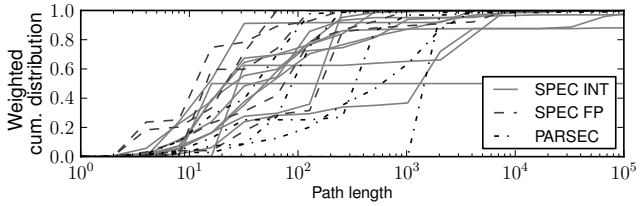


Figure 8. Distribution of idempotent path lengths.

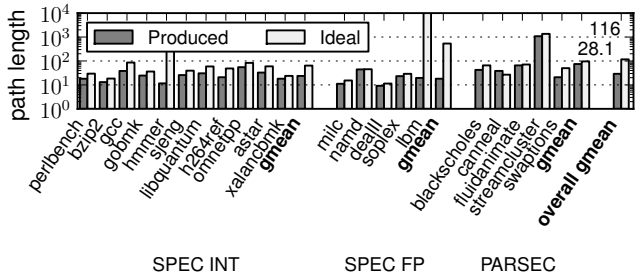


Figure 9. Average idempotent path lengths.

6.2 Idempotent Region Characteristics

Region sizes and path lengths. An important characteristic of our region construction is the size of the idempotent regions it statically produces. More important, however, is the length of the paths that dynamically execute through the idempotent regions at runtime. In general, longer path lengths are better for two reasons. The first has to do with runtime overhead: in the limit as path length approaches infinity, the relative cost to preserve a region's live-in state approaches zero. In the worst case, all live state must be pushed to the stack before the region's start, and this fixed maximum cost is amortized over the region's execution. The second reason has to do with detection latencies: longer path lengths allow execution to proceed speculatively for longer amounts of time while potential (but presumably unlikely) execution failures remain undetected.

In practice, however, optimal path length (and hence, region size) depends on a variety of factors. These factors include the effects of register pressure, aggravated by divergent control flow as regions grow beyond a small set of basic blocks (for reasons relating to potentially incorrect control flow as described in Section 2.3). Hence, larger regions are not always better. Additionally, while longer path lengths better tolerate long detection latencies, minimizing the recovery re-execution cost favors shorter path lengths. This is important particularly when failures are relatively frequent, as with e.g. branch prediction. In future work, we plan to explore this optimization space in detail. For this work, we aim to produce the longest possible paths, observing that path lengths are often easily reduced as needed to suit application demands.

Figure 8 plots the cumulative distribution (weighted by execution time) of the dynamic path lengths executed through our idempotent regions across the SPEC and PARSEC benchmark suites (in the interest of space, applications from the same suite are not individually labeled). The figure shows, for instance, that most applications spend less than 20% of their execution time executing paths of length 10 instructions or less. The figure also shows that path length distributions are highly application dependent. Generally, the PARSEC applications tend to have a wider, more heavy-tailed distribution, while SPEC FP applications have a narrower, more regular distribution. SPEC INT has applications in both categories.

Figure 9 shows the average length of our idempotent paths compared to those measured as ideal in the limit study from Section 3 (the ideal measurement is for "semantic and calls"—intra-procedural regions divided by dynamic semantic clobber antidependences). Our geometric mean path length across all benchmarks is roughly 4x less than the ideal (28.1 vs. 116). Two benchmarks, hmmer and lbm, have much longer path lengths in the ideal case. This is due to limited aliasing information in the region construction algorithm; with small modifications to the source code that improve aliasing knowledge, longer path lengths can be achieved. If we ignore these two outliers, the difference narrows to roughly 1.5x (30.2 vs. 44.9; not shown).

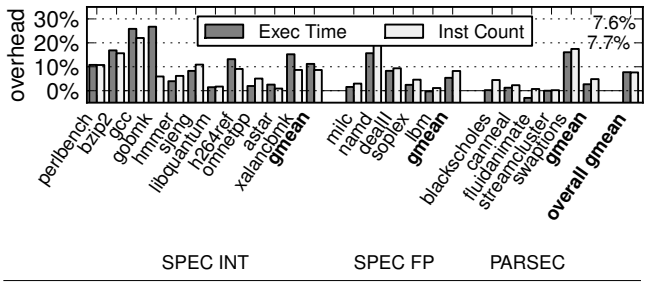


Figure 10. Execution time and instruction count overheads.

Runtime overheads. Forcing the register allocator to preserve input state across an idempotent region adds overhead because the allocator may not re-use live-in register or stack memory resources. Instead, it may need to allocate additional stack slots and spill more registers than might otherwise be necessary.

Figure 10 shows the percentage execution time and dynamic instruction count overheads. Across the SPEC INT, SPEC FP, and PARSEC benchmarks the geometric mean execution time overheads are 11.2%, 5.4%, and 2.7%, respectively (7.7% overall), These overheads are closely tracked by the increase in the dynamic instruction count: 8.7%, 8.2%, and 4.8% for SPEC INT, SPEC FP, and PARSEC, respectively (7.6% overall).

The one case where execution time overhead and instruction count overhead are substantially different is for gobmk, which has a 26.7% execution time overhead but only a 5.9% instruction count overhead. For this particular benchmark, some code sequences that were previously efficiently expressed using ARM predicated execution transform to regular control flow due to liveness constraints, resulting in more control flow sensitivity. Additionally, most of the added instructions are load and store (spill and refill) instructions, which have longer latency than regular move instructions.

For other applications, the differences mostly trend along the nature of the fundamental data type—floating point or integer. Integer applications such as those from SPEC INT tend to have higher execution time overheads because ARM has fewer general purpose registers than floating point registers (16 vs. 32). Hence, these applications are more reliant on register spills and refills to preserve liveness. In contrast, floating point benchmarks such as SPEC FP and PARSEC have many more available registers. Additionally, the comparatively long path lengths of PARSEC benchmarks tend to allow better register re-use since the cost of pushing live-ins to the stack is amortized over a longer period of time.

6.3 Idempotence-Based Recovery

Our compiler implementation and static analysis are general and span entire programs, and hence they can be used in the context of prior works using idempotence such as those presented in Table 1. As a concrete example, however, we consider the case for software-only recovery from transient hardware faults (soft errors). We evaluate against two other compiler-based recovery techniques, and for all techniques assume compiler-based error detection using

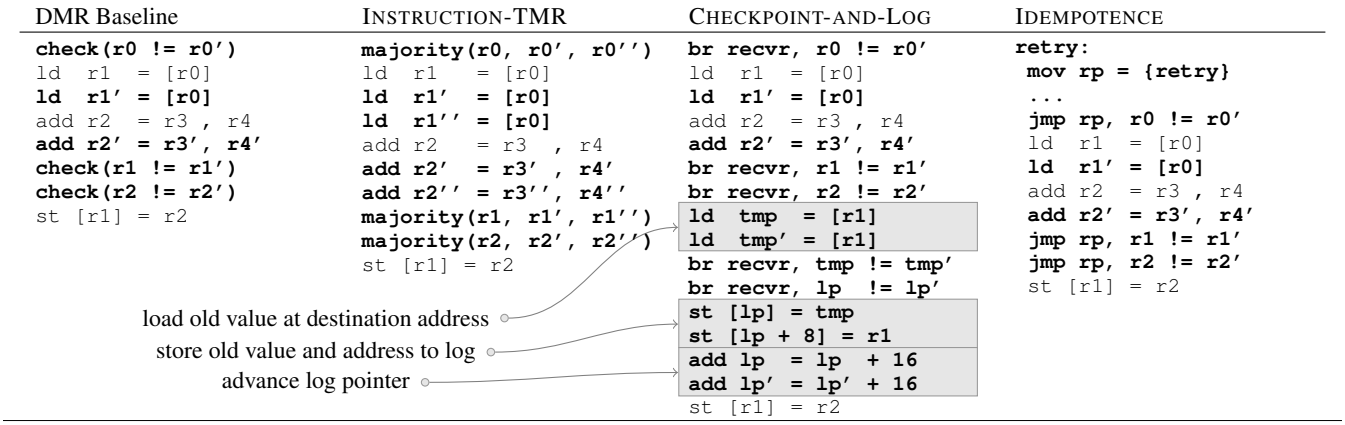


Figure 11. Three software recovery techniques on top of instruction-level DMR. Changes over original load-add-store sequence in bold.

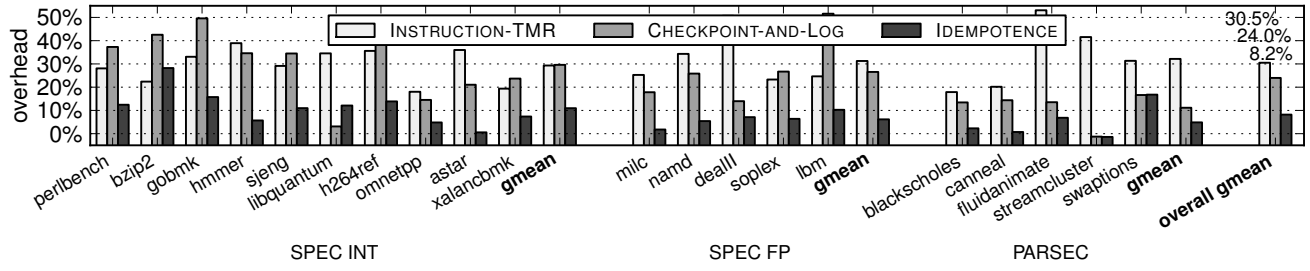


Figure 12. Overhead of three software recovery techniques relative to the DMR baseline.

instruction-level dual-modular redundancy (DMR). The DMR uses detection at load, store, and control flow boundaries, as previously proposed by Reis *et al.* [29] and Oh *et al.* [26]. We assume registers and memory are protected by ECC; thus, errors arise through instruction execution alone. Figure 11 illustrates the behavior of each recovery alternative.

The first recovery technique, INSTRUCTION-TMR, implements TMR at the instruction level. Our implementation attempts to replicate the work of Chang *et al.* [6], which adds a third copy of each non-memory instruction and use majority voting before load and store instructions to detect and correct failures. We support the majority voting as a single-cycle operation.

The second technique, CHECKPOINT-AND-LOG, is an implementation of software logging similar to logging in software transactional memory systems [17]. Before every store instruction, the value to be overwritten is loaded and written to a log along with the store address, and the pointer into the log (assigned a dedicated register, `lp`) is incremented. In our implementation, as the log fills, the log is reset and a register checkpoint is taken, which starts a new checkpointing interval. We assume a 16KB log size (1K stores per checkpoint interval) with intelligent periodic polling for log overflow using a technique similar to that proposed by Li and Fuchs [22]. In our simulations, all log traffic writes through the L1 cache, and we optimistically assume that both the register checkpointing and periodic polling contribute no runtime overhead.

The final technique, IDEMPOTENCE, is our idempotence-based recovery technique. Here, as each idempotent region boundary is encountered, its address is written to the register `rp`. In the event that a fault is detected, execution jumps to the address contained in `rp` (the use of a register to hold the restart address is necessary to handle potentially overlapping control flow between regions).

Results. Figure 12 presents results comparing the overhead of the three techniques relative to performance of the underlying DMR. Across all benchmarks, INSTRUCTION-TMR performs worst with 30.5% geometric mean performance overhead, CHECKPOINT-AND-LOG has 24.0% overhead, and IDEMPOTENCE performs best with only 8.2% overhead.

Compared to INSTRUCTION-TMR, CHECKPOINT-AND-LOG performs worse for applications with frequent memory interactions, such as several SPEC INT applications, but better for all other applications where its per-instruction overheads are lower. Overall, IDEMPOTENCE outperforms both techniques by a significant margin. It avoids the redundant operations added by INSTRUCTION-TMR to correct values in-place, and avoids the overheads associated with unnecessary logging in CHECKPOINT-AND-LOG. In particular, when logging only the first memory value written to a particular memory location is required to be logged. However, under CHECKPOINT-AND-LOG the occurrence of the first write is not statically known and cannot be efficiently computed at runtime. IDEMPOTENCE also preserves local stack memory more efficiently with its fully-integrated compile-time approach.

7. Related Work

The application of idempotence in compiler-based recovery has been previously explored, primarily in the context of exceptions and also hardware fault recovery. For exception recovery, Hampton and Asanović explore the use of idempotent regions for exception recovery in vector processors [16], De Kruijf and Sankaralingam use them for exception recovery in general purpose processors [10], and Mahlke *et al.* propose restartable (idempotent) instruction sequences under sentinel scheduling for exception recovery in VLIW processors [23]. For fault recovery, Feng *et al.* and De Kruijf *et al.* both explore mechanisms to opportunistically

employ idempotence over code regions that together cover large parts—but not all parts—of a program. While De Kruijf *et al.* manually identify idempotent regions, Feng *et al.* identify them using a compiler interval analysis. We build upon this prior work and make several additional contributions: we develop a compiler analysis to uncover the minimal set of *semantically* idempotent regions across *entire* programs, we describe the algorithmic challenges in compiling for these regions, and we explore how they can be used to recover across a range of different types of execution failures.

In other related work, Shivers *et al.* [31] and Bershad [2] both explore using idempotence to achieve atomicity on uniprocessors. The work of Li *et al.* on compiler-based multiple instruction retry is also similar in that they breaks antidependences to create recoverable code regions [21]. However, they do so over a sliding window of the last N instructions rather than over static program regions. As such, they do not distinguish between clobber antidependences and other antidependences; all antidependences must be considered clobber antidependences over a sliding window since any flow dependence preceding an antidependence will eventually lie outside the window. Our use of static program regions allows for the construction of large recoverable regions with low overheads.

More general work on compiler-based recovery includes the work of Chang *et al.* on recovery of hardware transient faults at the granularity of single instructions using TMR on top of DMR [6]. Chang *et al.* also explore two partial recovery techniques in addition to TMR. However, for full recovery functionality, the overheads remain effectively the same as TMR, and our results show that idempotence-based recovery has potentially better performance than TMR. Finally, compilers have been proposed for coarse-grained, checkpoint-based recovery as well. Li and Fuchs study techniques for dynamic checkpoint insertion using a compiler [22]. To maintain the desired checkpoint interval, they periodically poll a clock to decide if a checkpoint should be taken.

8. Conclusion

The capability for fast and efficient recovery has applications in many domains, including microprocessor speculation, compiler speculation, and hardware reliability. Unfortunately, most prior software-based solutions typically have high performance overheads, particularly for fine-grained recovery, while hardware-based solutions involve substantial power and complexity overheads.

In this paper, we identified idempotence as a basic program property that can be used for general and efficient recovery in software. While it is intuitive that idempotence can be used to recover from execution failures with no visible side-effects (e.g. hardware exceptions), we showed how failures resulting in only incorrect control flow (e.g. branch misprediction) can also be supported in a straightforward manner. Additionally, we described how failures involving a wider range of side-effects (e.g. soft errors) can be recovered with a relatively small set of supporting mechanisms.

We demonstrated the potential for idempotence-based recovery by building a compiler that partitions programs into idempotent regions, enabling the paradigm of *idempotent processing*—execution in sequences of idempotent regions. We presented a static analysis that partitions applications into *semantically* idempotent regions and showed a compiler that generates code preserving the idempotence of these regions with low performance overhead (commonly less than 10%). As an example, we showed how our analysis and compiler can be used to recover from hardware transient faults efficiently, purely in software. However, our technique is general and applies to many other uses of idempotence as well.

While we demonstrated idempotence as a powerful primitive for program recovery, several questions remain for future research. First, while we showed that idempotent regions can be large, limited program knowledge sometimes inhibits region sizes unnecessarily. Better programmer aliasing information and/or the use of more declarative programming styles may allow the construction of much larger idempotent regions. Second, while we constructed these large regions in part as a first-order approximation towards minimizing performance overheads, in practice optimal region size depends on a variety of factors. An important topic of future work is to characterize the performance overheads based on such factors. Finally, future work exploring the applicability of idempotence with respect to specific failure scenarios will further help in understanding its full potential. Regardless of the outcomes to these questions, however, idempotent processing and the concept of semantic idempotence are likely to remain as valuable building blocks for use in future research on low-overhead software recovery solutions.

Acknowledgements

We thank the anonymous reviewers for comments and the Wisconsin Condor project and UW CSL for their assistance. Support for this research was provided by NSF under the following grant: CCF-0845751. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2007.

[2] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS '92*.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, Aug. 2011.

[5] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06*.

[6] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *DSN '06*.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. POPL '89.

[9] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA '10*, 2010.

[10] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *MICRO '11*.

[11] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03*.

[12] S. Feng, S. Gupta, A. Ansari, S. Mahlke, and D. August. Encore: Low-cost, fine-grained transient fault recovery. In *MICRO '11*.

[13] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *ASPLOS '94*.

[14] M. Gschwind and E. R. Altman. Precise exception semantics in dynamic compilation. In *CC '02*.

[15] J. Guo, F. HÃijffner, E. Kenar, R. Niedermeier, and J. Uhlmann. Complexity and exact algorithms for vertex multicut in interval and bounded treewidth graphs. *European Journal of Operational Research*, 186(2):542 – 553, 2008.

[16] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *ICS '06*.

[17] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[18] Intel. *Itanium Architecture Software Developer's Manual Rev. 2.3*. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm.

[19] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst.*, 28:942–965, September 2006.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*.

[21] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu. Compiler-based multiple instruction retry. *IEEE Transactions on Computers*, 44(1):35–46, 1995.

[22] C.-C. J. Li and W. K. Fuchs. CATCH – Compiler-assisted techniques for checkpointing. In *FTCS '90*.

[23] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *ASPLOS '92*.

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

[25] J. Menon, M. de Kruijf, and K. Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *ISCA '12*, 2012.

[26] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, March 2002.

[27] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software – Practice & Experience*, 29(2):125–142, 1999.

[28] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO '01*.

[29] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. Swift: software implemented fault tolerance. In *CGO '05*.

[30] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *ISCA '05*, pages 148–159.

[31] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP '99*.

[32] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[33] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[34] D. J. Sorin. *Fault Tolerant Computer Architecture*. Morgan & Claypool, 2009.

[35] Standard Performance Evaluation Corporation. *SPEC CPU2006*, 2006.

[36] H.-W. Tseng and D. Tullsen. Data-triggered threads: Eliminating redundant computation. In *HPCA '11*.

[37] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[38] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO '91*.