Computer Sciences Department

Compiler Construction of Idempotent Regions

Marc de Kruijf
Karthikeyan Sankaralingam
Somesh Jha

Technical Report #1700

November 2011

Compiler Construction of Idempotent Regions

Marc de Kruijf, Karthikeyan Sankaralingam, Somesh Jha
University of Wisconsin – Madison
{dekruijf, jha, karu}@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 conservatively save more state than necessary.

This paper develops a compiler technique to recover program state without the overheads of explicit checkpoints. Our 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 in a compiler, and demonstrate low overheads and large region sizes using an LLVM-based implementation. Across a set of diverse benchmark suites, we construct idempotent regions almost as large as those that could be obtained with perfect runtime information. Although the resulting code runs slower, typical execution time overheads are in the range of just 2-12%.

1 Introduction

Recovery capability is a fundamental component of modern computer systems. Compiler recovery using idempotent regions is a recently developed software recovery technique [10] that enables recovery with low overhead by leveraging the inherent idempotence property in applications. Specifically, it harnesses the insight that applications naturally decompose into a continuous series of idempotent regions, i.e. application execution can be broken down into a set of regions, where each region is idempotent (re-execution has no side-effects). The property of idempotence allows these regions to be recovered by simple re-execution.

Although in principle a simple idea, recovery using idempotent regions presents two key challenges. First is the problem of ensuring that recovery by re-execution is possible even in the presence of execution failures. While regions may be statically idempotent at the time of construction, execution failures could potentially violate the idempotence assumption at execution time, and some scheme is needed to account for this possibility. Second is the problem of actually constructing program binaries into a series of idempotent regions. In particular, while a program naturally decomposes into a series of idempotent regions and these regions often have the potential to be very large, due to compilation artifacts, the sizes of idempotent regions in conventional binaries are often small. Unfortunately, small regions limit how far execution can proceed before an execution failure is detected and recovery is initiated. Large regions, in contrast, allow execution to proceed mostly in parallel with detection, even when detection latencies are relatively high. Hence, for recovery purposes, large idempotent regions are desirable. To be useful, idempotence-based recovery therefore requires that the inherent idempotence in applications is preserved as much as possible.

To meet these challenges, this paper develops a set of static analysis and compiler techniques that (1) enable recovery using idempotence even in the face of arbitrary execution failures and (2) also eliminate the compilation artifacts responsible for the small idempotent region sizes occurring under a conventional compiler. Our analysis identifies the regions in a function that are *semantically* idempotent. These regions are then compiled so that no

artifacts are introduced and their idempotence is preserved throughout code generation and program execution. To do this, our 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 technique partitions a function into idempotent regions close in size to the largest regions that would be constructed given 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 close to ideal.

The remainder of this paper is organized as follows. Section 2 presents an 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. First, we review the concept idempotence as it applies over groups of instructions. Second, we describe how statically-identified idempotence can be used to recover from a range of dynamic execution failures. Third, we present an example that illustrates how data dependences inhibit idempotence and how this information can be used to identify idempotent regions. Finally, we show how 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.

2.1 Idempotence at the Instruction Level

A region of code (a precise definition will follow) 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, which changes its behavior. A region's inputs are the variables that are defined before entry to the region and are correspondingly used at some point after entry to the region.

2.2 Using Idempotence for Recovery

Idempotence as a static program property can be determined through program analysis. Section 2.3 describes this analysis. First, however, 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 soft error occurrence in hardware, 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).

Fortunately, the problem is not as severe as it may initially seem. Let us first consider the case of branch misprediction. In this case, initial execution of an incorrect region (i.e. an incorrect instruction sequence) may overwrite an input of the correct region or its succeeding regions. If this happens, then the idempotence property is lost.

To allow for this possibility, we first derive a precise definition of a region. An idempotent instruction sequence we label an *idempotent path*, and we define an *idempotent region* as a collection of idempotent paths that share the same

entry point. That is, an idempotent region is a subset of a program's control flow graph with a single unique entry point and multiple possible exit points, and an idempotent path is a trace of instructions from such a region's entry point to one of its exit points. It follows that any variable used along some idempotent path belongs to the set of inputs for its containing region. With this definition of an idempotent region and its inputs, the idempotence property of a region is preserved regardless of the control flow decisions made inside of it.

For registers and local stack memory, the relatively larger amount of input state for an idempotent region compared to an idempotent path does not severely impact our analysis and the compiler implementation we describe in this paper comfortably preserves this larger amount of state. 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. Thus, for these types of memory we assume instead that stores are buffered and not released to memory 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 assuming invariable control flow, i.e. in terms of idempotent paths.

While at first glance the situation with soft errors appears much more dire, in practice techniques like ECC can effectively protect 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, the only two additional requirements are that instructions must write to the correct register destinations, and that execution must follow the program's static control flow edges. As with branch misprediction, stores must also 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, 13], and a variety of compatible hardware and software techniques have been previously developed [5, 16, 19, 20].

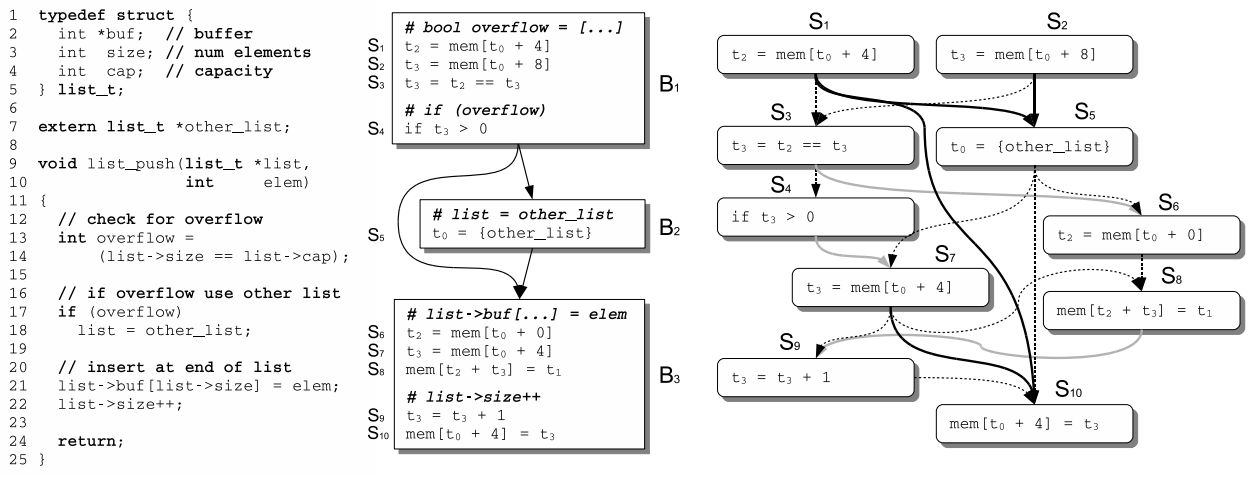
2.3 Identifying Idempotent Regions

In this section, we first derive a precise definition of idempotence in terms of data dependences. We then use an example to show how idempotence is inhibited by the occurrence of a specific chain of data dependences. By eliminating unnecessary occurrences of this chain, we show how to identify the inherently idempotent regions in a program.

Idempotence in terms of data dependences. In Section 2.1, we defined a region's input as a variable that has a definition (a write) that reaches the region's entry point and has a corresponding use (a read) of that definition after the region's entry point. This type of variable has a special name: it is called a *live-in* to the region (i.e. the variable is *live* at the entry point of the region). A region is idempotent if it does not overwrite its live-in variables.

In the derivation given in the following paragraph, we use the term region to refer specifically to a linear sequence of instructions (an idempotent *path*). This is because, as we will show, heap, global, and non-local stack memory interactions dictate the inherent idempotence of applications, and these memory interactions are analyzed in this linear context, as explained in the previous section. 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.

First, by definition, a live-in has a flow dependence that spans the entry point of the region. Second, a live-in that is overwritten is necessarily overwritten following the read of the flow dependence (otherwise it would not be live-in to the region's instruction sequence). That is, an antidependence follows the flow dependence that spans the entry point to the region. Finally, a flow dependence spanning a region's entry implies the absence of an initial flow dependence inside the region. This particular observation allows us to define a region's idempotence solely in terms of the region itself, and leads to our ultimate criterion for identifying idempotence: a sequence of instructions is idempotent if it



(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.

does not contain any antidependences that are not preceded by a flow dependence.

Semantic and artificial clobber antidependences. The antidependence after no flow dependence chain that breaks the idempotence property we call a *clobber antidependence*. Some clobber antidependences are strictly necessary according to program semantics. These clobber antidependences we label *semantic*. The other clobber antidependences we label *artificial*. The following example demonstrates semantic and artificial clobber antidependences.

An 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 compiler 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). It 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 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 first two clobber antidependences 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 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.

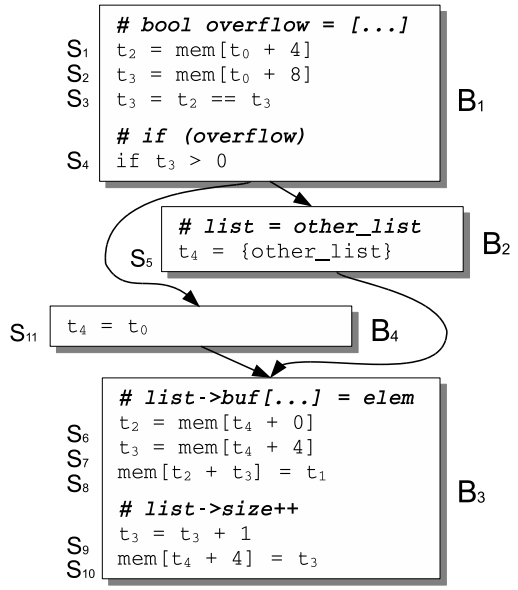


Figure 2: Renaming t_0 in S_5 to t_4.

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

Table 1: Semantic and artificial clobber antidependences and the resources on which they operate.

To permanently eliminate the artificial clobber antidependences, we must ensure that t_4 and t_0 are not allocated to the same physical register or the same stack slot. If they are, then t_4 will overwrite t_0 and the two clobber antidependences will simply re-emerge. To do this, we can place a constraint on the compiler's resource allocators to say that all pseudoregisters that are live-in to the region are also live-out to the region. This enables idempotence in exchange for additional register pressure in the compiler.

The remaining two clobber antidependences write to the memory location $t_0 + 4$ in S_{10}, which is the store corresponding with the increment of `list->size` 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 grow the size of the list, which would violate the function's semantics. Because we can't legally rename the store destination, we label these clobber antidependences *semantic* clobber antidependences.

Summary. Table 1 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.

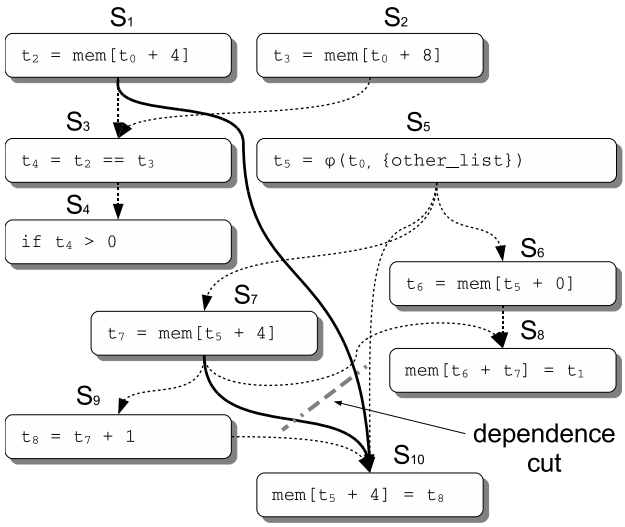
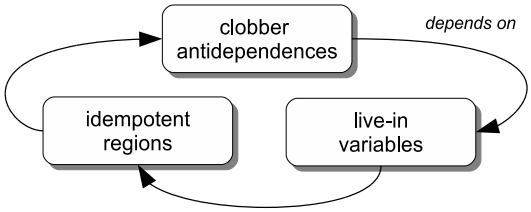


Figure 3: The data dependence graph in SSA form.

2.4 Constructing Idempotent Regions

Assuming we can eliminate all artificial clobber antidependences, we show in Section 3 that the dynamic 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 antidependence requires identification of a region's live-in variables, which in turn requires identification of the regions themselves. This circular dependence is illustrated below.



Our solution to this problem is to transform the function so that, with the exception of self-dependent pseudoregisters antidependences, which are handled later, all remaining antidependences are *necessarily* semantic clobber antidependences. Then, the problem of constructing idempotent regions can be phrased in terms of constructing regions that contain no antidependences. Antidependence information does not depend on live-in variable information, and hence the circular dependence chain is broken.

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. 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.

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 control flow is the same as in Figure 1(b) except that S_5 becomes a SSA ϕ-node and therefore moves to B_3. 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}. In all cases the result is the same: the function is divided into only two idempotent regions. The first region contains two possible execution paths through it (two dynamic idempotent regions) and the second has only one.

3 A Limit Study

To understand how much artificial clobber antidependences inhibit idempotent region sizes, we performed a limit study to understand 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 gem5 [4] to measure dynamic idempotent region sizes (the number of instructions dynamically occurring between region entry and exit) across a range of benchmarks compiled for the ARMv7 instruction set. For each benchmark, we measured the distribution of dynamic region sizes occurring over a 100 million instruction period starting after the setup phase of the application. We evaluated two benchmark suites: SPEC 2006 [22], 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 dynamic idempotent region size 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 (i.e. register spills). 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 constraint 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 for region sizes divided by both types of clobber antidependences is 10.8 instructions, while the sizes when we consider just semantic clobber antidependences are 110 intra-procedurally (a 10x gain) and 1300 inter-procedurally (a 120x gain).

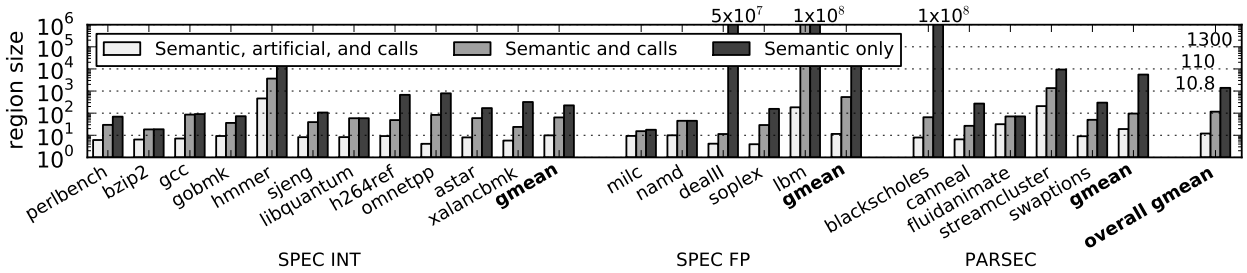


Figure 4: Average dynamic idempotent region sizes in the limit (y-axis is log-scale).

The second trend is that there exists 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 region sizes tend to be larger for PARSEC and SPEC FP than for SPEC INT. This is because the former benchmarks overwrite their inputs relatively infrequently due to their 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 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 how to perform register and stack slot allocation such that the idempotence property of identified regions is preserved through code generation.

4.1 Program Transformation

Before we apply our static analysis, we first perform two code transformations in order 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 is to convert 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, 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 the overview section, this determination is circularly dependent on the region construction that we are trying to achieve. Without knowing which antidependences will emerge as clobber antidependences, we do not know which antidependences need to be cut to form the regions. Hence, we attempt to

```
1.  mem[x] = a        1.  mem[x] = a
2.  b = mem[x]        2.  b = a
3.  mem[x] = c        3.  mem[x] = c
     before                after
```

Figure 5: Eliminating non-clobber memory antidependences.

refine things further.

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 are able to 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 re-load the stored value since there is an existing pseudoregister that already holds the value. The redundant load can be 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 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). 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 prove that an optimal graph decomposition is equivalent to a minimum vertex multicut.
3. We reduce the problem of finding a minimum vertex multicut to the hitting set problem.
4. We observe that a near-optimal hitting set can be found efficiently using an approximation algorithm.

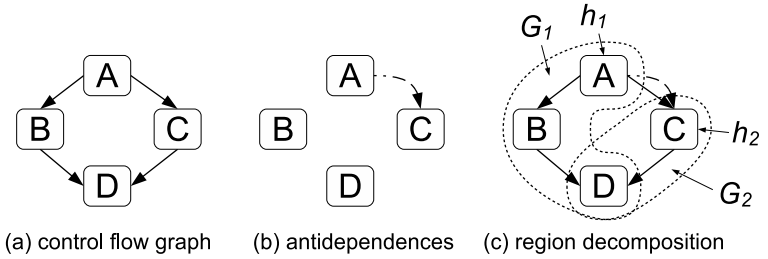


Figure 6: An example region decomposition.

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*[1] 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 $1 \leq i \leq k$.

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.

Equivalence to vertex multicut. We now equate the problem of determining an optimal region decomposition with the minimum vertex multicut problem.

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 vertex multicut $H = \{h_1, \cdots, h_k\}$ directly corresponds to a region decomposition $\{G_1, \cdots, G_k\}$ of G, and the problem of finding an optimal region decomposition is equivalent to finding a minimum vertex multicut, H, over the set of antidependence edge pairs A in G.

Equivalence to hitting set. The vertex multicut problem is NP-complete for general directed graphs [11]. We reduce the problem of finding a vertex multicut to the *hitting set problem*, which is also NP-complete, but for which good approximation algorithms are known [7]. Given a collection of sets $C = \{S_1, \cdots, S_m\}$, a minimum hitting set for

[1]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].

C is the smallest set H such that each S_i in C is intersected in at least one element by H; that is, for all $S_i \in C$, $H \cap S_i \neq \emptyset$.

For each antidependence edge $(a_i, b_i) \in A$ we associate a set $S_i \subseteq V$ which consists of a set of nodes that dominate b_i but do not dominate a_i. Let $C = \{S_1, \cdots, S_k\}$, and let H be a hitting set for C. Using Lemma 1 it is easy to see that for all antidependence edges $(a_i, b_i) \in A$ the following condition is true: 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 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 the 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 potential clobber antidependences—the self-dependent pseudoregister anti-dependences—and allocate them in such a way that they do not emerge as clobber antidependences.

In SSA form, a self-dependent pseudoregister antidependence has its 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 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 modification 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.

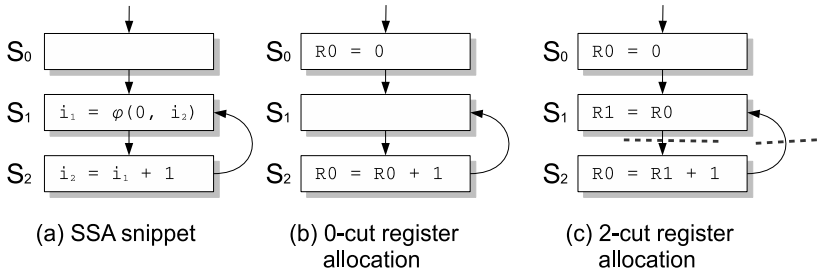


Figure 7: Clobber-free allocation of self-dependent pseudoregister antidependences.

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 antidependence 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 improve the sizes of our dynamic idempotent regions at execution time.

Our simple heuristic is to first select antidependence cuts in order of loop nesting depth, from outermost to innermost. We adjust the greedy hitting set algorithm used in Section 4.2.1 to greedily choose nodes with the lowest loop nesting depth first. We then break ties by choosing a node with the most sets not already intersected, as normal. This improves dynamic region sizes substantially in general, although there are cases where it reduces region sizes. A better heuristic most likely weighs both loop depth and intersecting path 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 [12]. 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 antide-pendences. 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: even though the number of static regions grows, the number of static instructions grows roughly proportionally as well. 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.

6 Evaluation

In this section, we present our evaluation methodology and results measuring the characteristics of the regions produced by our region construction algorithm.

6.1 Methodology

We evaluate benchmarks from the SPEC 2006 [22] and PARSEC [3] benchmark suites. Each benchmark we compile to two different binary versions: an *idempotent binary*, which is compiled using our idempotent region construction implemented in LLVM; and an *original binary*, which is generated using the regular LLVM compiler flow. Both versions are compiled with the maximum level of optimization.

Our performance results are obtained for the ARMv7 instruction set, simulating a two-issue out-of-order 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.

6.2 Results

Below, we present results for the region sizes and runtime overheads across our suite of benchmarks.

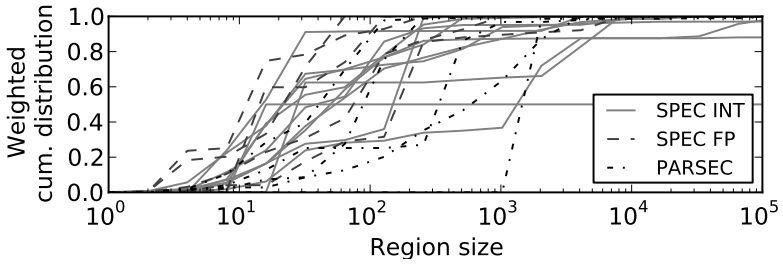


Figure 8: Distribution of region sizes.

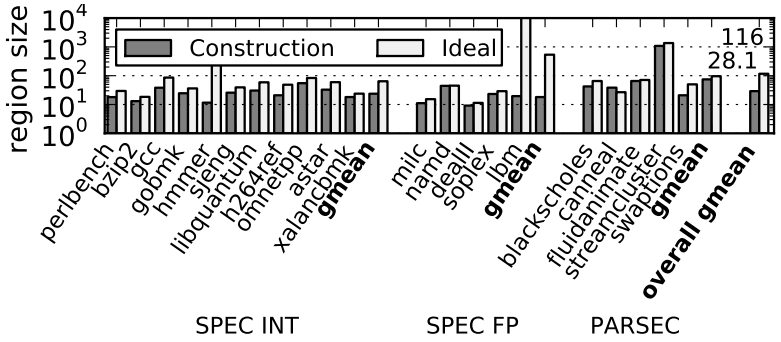


Figure 9: Average idempotent region sizes.

Region sizes. An important characteristic of our region construction is the sizes of the regions it produces. Larger region sizes are generally better because the size dictates how long execution can proceed speculatively while potential (but unlikely) execution failures remain undetected.

Figure 8 shows a plot of the weighted cumulative distribution of dynamic idempotent region sizes across the SPEC and PARSEC benchmark suites (in the interest of space, applications from the same suite use the same annotation and are not individually labeled). The weighting is by execution time contribution; the figure shows, for instance, that most applications spend less than 20% of their execution time in regions of size 10 instructions or less. The figure also shows that region size 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 sizes of our dynamic idempotent regions 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 region size across all benchmarks is roughly 4x less than the ideal (28.1 vs. 116). Two benchmarks, hmmer and lbm, have much larger dynamic idempotent regions 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, larger region sizes can be achieved. If we ignore these two outliers, the difference narrows to roughly 1.5x (30.2 vs. 44.9; not shown).

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 of our region construction. 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

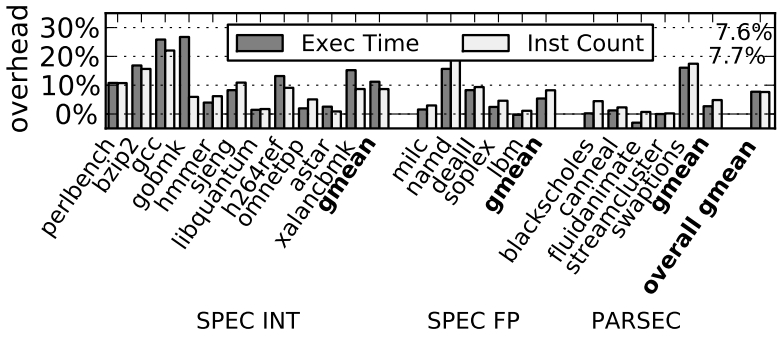


Figure 10: Percentage execution time and dynamic instruction count overheads.

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 transition to regular control flow due to liveness constraints, resulting in a growth in the branch misprediction rate and resulting squash cycles. Additionally, essentially all of the added instructions are load and store (spill and refill) instructions, resulting in increased load-store queue and cache contention, and also more memory ordering violations.

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 large region sizes of PARSEC tend to allow better register re-use since the cost of pushing live-ins to the stack when necessary is amortized over a longer time period of time. An interesting corollary is that improvements to idempotent region sizes generally have the benefit of also reducing execution time overheads.

7 Related Work

This section presents related work in compiler-based and hardware-based recovery.

Compiler-based recovery. Similar to ours is the work of Li *et al.* on compiler-based multiple instruction retry [13]: they also break antidependences to create recoverable regions. 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 lower performance overheads.

Also similar is the work of Mahlke *et al.* on exception recovery using restartable (idempotent) instruction sequences under sentinel scheduling [15]. However, their treatment is brief and they apply the idea only to recovery from speculative memory reordering over specific program regions.

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 [14]. To maintain the desired checkpoint interval, they periodically poll a clock to decide if a checkpoint should be taken.

Hardware-based recovery. For hardware faults, related work in hardware checkpoint recovery includes ReVive I/O [18] and SafetyNet [21]. ReVive I/O is particularly notable for its use of idempotence to recover certain I/O operations efficiently. While ReVive I/O exploits idempotence at the system level, our work exploits it at the instruction level. Checkpoint recovery has also been explored in detail for mis-speculation recovery [2, 17]. For both mis-speculation and fault recovery, there is opportunity to apply hardware recovery mechanisms in combination with our technique.

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 described a compiler algorithm for supporting recovery using idempotent regions. The technique has relatively low performance overhead (commonly less than 10%), and adds no hardware overhead. Our algoorithm identifies and constructs application regions that are *semantically idempotent*. These regions can be recovered simply by re-execution. Overall, we find that the analysis of idempotent regions provides a promising foundation for future research on low-overhead compiler-based recovery solutions.

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] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO '03*.

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

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[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. 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.

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

[13] 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.

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

[15] 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*.

[16] 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.

[17] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA '03*.

[18] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReVive I/O: Efficient handling of I/O in highly available rollback-recovery servers. In *HPCA '06*.

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

[20] 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.

[21] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02*.

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