Idempotent Code Generation: Implementation, Analysis, and Evaluation

Marc de Kruijf *
Google, Inc.
masher@google.com

Karthikeyan Sankaralingam
University of Wisconsin-Madison
karu@cs.wisc.edu

Abstract

Leveraging idempotence for efficient recovery is of emerging interest in compiler design. In particular, identifying semantically idempotent code and then compiling such code to preserve the semantic idempotence property enables recovery with substantially lower overheads than competing software techniques. However, the efficacy of this technique depends on application-, architecture-, and compiler-specific factors that are not well understood.

In this paper, we develop algorithms for the code generation of idempotent code regions and evaluate these algorithms considering how they are impacted by these factors. Without optimizing for these factors, we find that typical performance overheads fall in the range of roughly 10-15%. However, manipulating application idempotent region size typically improves the run-time performance of compiled code by 2-10%, differences in the architecture instruction set affect performance by up to 15%, and knowing in the compiler whether control flow side-effects can or cannot occur can impact performance by up to 10%.

Overall, we find that, with small idempotent region and careful architecture- and application-specific tuning, it is possible to bring compiler performance overheads consistently down into the single-digit percentage range. The absolute best performance occurs when constructing the largest possible idempotent regions; to this end, however, better compiler support is needed. In the interest of spurring development in this area, we open-source our LLVM compiler implementation and make it available as a research tool.

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

General Terms Algorithms, Design, Performance

* Marc de Kruijf was a research assistant at the University of Wisconsin-Madison while he was doing the research presented in this paper.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Introduction

Idempotence—the property that re-execution has no side-effects—has been recently proposed for compiler-based recovery in a variety of domains, including exception recovery [7, 10, 14], hardware fault recovery [8, 9], speculation recovery [11, 18], and concurrency bug recovery [19]. In these proposals, a compiler identifies *semantically* idempotent regions of code using static analysis techniques, and then generates code for these regions in such a way that idempotence is preserved by ensuring that certain live register or stack memory locations are not overwritten. The latter is accomplished either by integrating the concept of idempotence into the register allocation flow [7, 8, 14] or by checkpointing state that is live-in to an idempotent region [9–11, 18].

Among these two approaches, the former approach of preserving idempotence during register allocation is of particular interest because it enables a form of intelligent checkpointing that utilizes stack storage as a checkpointing resource only "as needed". For instance, some live-in variables may never be overwritten inside an idempotent region. These variables can be allocated to a register throughout a region and need not be checkpointed. Incorporating idempotence information into the register allocation algorithm allows the compiler to determine whether or not doing so is profitable leveraging existing register allocation heuristics. If it is not profitable, the compiler can choose to "checkpoint" the live-in variable to the stack by spilling it before region entry and then re-loading it as needed, as in the latter approach.

Despite the claimed benefits, however, no algorithms have been previously proposed for incorporating idempotence information into the register allocation flow. Additionally, the benefits of this approach compared to simple live-in register checkpointing has not been previously measured and its sensitivities with respect to architecture and application characteristics has not been evaluated. This paper addresses this gap in our understanding to contribute the following:

- It presents algorithms for the integration of idempotence information during register allocation. These algorithms allow the compiler to choose whether to (a) checkpoint a register to the stack or (b) preserve the register by preventing re-use.

- It evaluates the performance of the resulting code and explores sensitivities to application and architecture characteristics. Our main quantitative findings are that:
  1. with large idempotent regions, performance overheads quickly approach 0% as regions grow beyond roughly 50 instructions;
  2. small idempotent regions of 10-25 instructions typically entail performance overheads of roughly 10%;
  3. for small regions ISA effects typically influence performance by 2-3% (up to 15%); and
  4. a code generation algorithm that allows possible control flow side-effects performs only slightly worse—less than 1% worse on average—than a more specific code generation algorithm that allows them. However, in specific cases the impact can be as severe as 10%.
- To foster continued exploration of idempotence in compiler design and research into incorporating heuristics and transformations such as those explored in this paper, we publicly release the source code and documentation of our LLVM-based compiler implementation for use by the wider research community [2].

2. Background and Related Work

In the recent literature, researchers have proposed executing programs (either entirely or over some interval) as sequences of idempotent code regions [7–11, 14, 18, 19]. A compiler or programer demarcates these regions either in the source code or the program binary, and the underlying hardware utilizes the fact that regions are idempotent to recover from faulty events (such as hardware exceptions, faults, and misspeculations) by simple re-execution. Figure 1 illustrates the overall approach. It operates similarly to recovery using checkpoints, except that *there is no explicit checkpoint*—the regions themselves provide an *implicit* checkpoint by leveraging the property of idempotence.

To name specific research efforts, De Kruijf and Sankaralingam use this technique to simplify the microarchitecture of an in-order processor [7], Menon *et al.* use it to enable efficient exception recovery for GPUs [14], Hampton uses it to achieve virtual memory support on vector processors [10], Feng *et al.* apply it to enable low-cost fault recovery on commodity processors [9], and Zhang *et al.* use it for light-weight concurrency bug recovery support [19]. De Kruijf also creates a taxonomy that identifies constraints that must be met across a diverse range of applications [6]. In a similar fashion, this paper develops a tunable compiler algorithm that can be applied across a range of applications, and subsequently analyzes the role of applications, architectures, and compilers in idempotent region construction.

The remainder of this section presents background on the analysis and compiler construction of idempotent regions. We focus on the algorithms developed in our previous work [8] because these algorithms enable programs to be partitioned *entirely* into idempotent regions and hence they support the greatest diversity of uses; other algorithms, in contrast, allow only a selective partitioning [9, 10, 19].

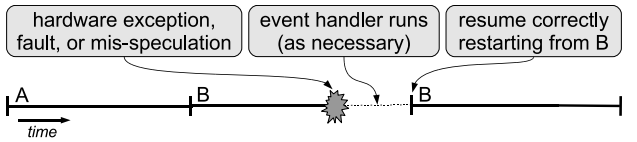


Figure 1. Recovery using idempotence.

2.1 Terms and Definitions

In our discussion, we assume the following terms and definitions. First, the terms *flow dependence* and *antidependence* refer to a read-after-write (RAW) data dependence and write-after-read (WAR) data dependence, respectively.

The term *region* refers to a subset of a function's control flow graph with a single entry point and multiple possible exit points. A region is uniquely defined by its entry point, and it contains all instructions reachable from its entry point to its exit points.

Finally, we distinguish between two different types of storage elements. The term *pseudoregister* refers to a function-local storage element such as a register or stack memory location and the term *non-local memory* refers to to all other types of storage elements—namely heap, global, and non-local stack memory.

2.2 Idempotence

Idempotence can be identified through the absence of *clobber antidependences*, which are antidependences with no prior flow dependence on the same variable [8]. That is, an idempotent region can contain an antidependence as long as the antidependent variable is defined before it is used. For example, a {*read*, *write*} sequence over a variable is not idempotent. However, a {*write*, *read*, *write*} sequence *is* idempotent due to the initial "protecting" write.

Some clobber antidependences are *semantic* while others are *artificial*. Artificial clobber antidependences act on pseudoregister locations. These resources are compiler controlled and can be allocated such that clobber antidependences do not actually emerge in compiled code, possibly at the expense of some additional register pressure. Semantic clobber dependences, in contrast, act on non-local memory. This memory is not under the control of the compiler; its location is fixed by the semantics of the program.

2.3 Idempotent Region Construction Algorithm

Our idempotent region construction algorithm attempts to partition entire functions into maximimally-sized idempotent regions [8]. The primary purpose in maximizing region size is that it is generally straight-forward to take large idempotent regions and partition them into smaller regions as desired, whereas the reverse problem is much harder.

The algorithm operates in three steps. In the first step, it transforms the function to remove artificial clobber antidependences and non-clobber antidependences, allowing

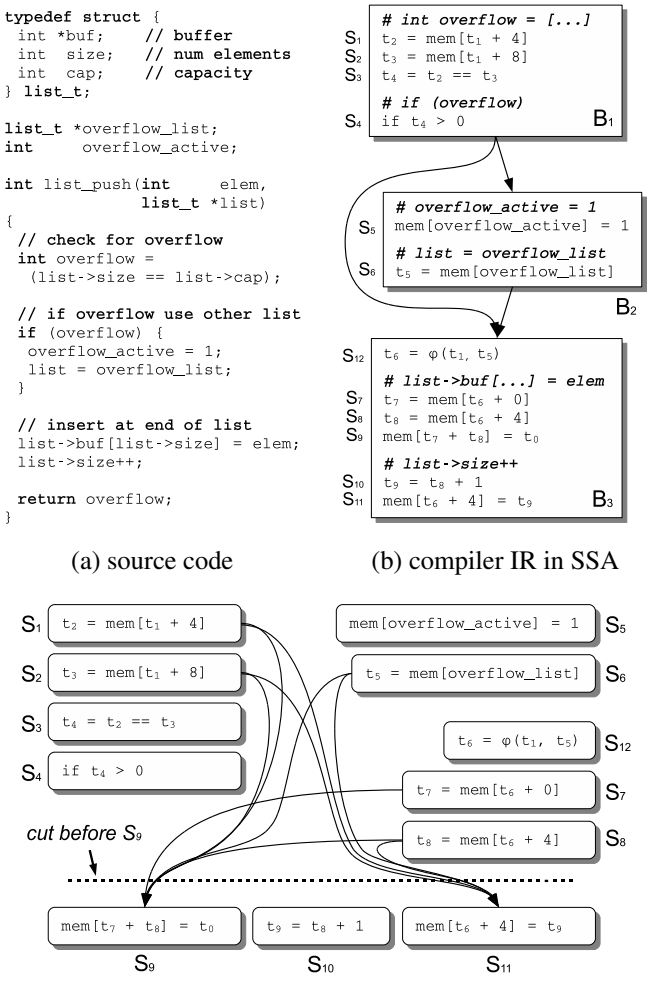


Figure 2. Idempotent region construction example.

the region construction problem to be formulated as a simple graph cutting problem. In the second step, the algorithm forms regions by "cutting" antidependences and placing region boundaries at the site of the cuts. Finally, in the third step, heuristics are introduced to maximize the sizes of regions as they occur dynamically at runtime.

Example. Figure 2 shows the algorithm applied to an example C function. Figure 2(a) shows the function's source code and Figure 2(b) shows its control flow graph with basic blocks B_1, B_2, and B_3. Inside each block are shown the low-level statements, S_i, and the use of pseudoregisters, t_i, to hold operands and read and write values to and from memory. The code in Figure 2(b) shows the pseudoregister assignments in SSA form; conversion to SSA is part of the initial transformation step of the construction algorithm. Figure 2(c) shows a dependence graph with semantic may-alias clobber antidependences indicated using solid black lines. These are the semantic clobber antidependences that remain after pseudoregister assignments are converted to SSA. A total of nine may-alias clobber antipendences exist in the function: $S_1 \rightarrow S_9$, $S_1 \rightarrow S_{11}$, $S_2 \rightarrow S_9$, $S_2 \rightarrow S_{11}$, $S_6 \rightarrow S_9$, $S_6 \rightarrow S_{11}$, $S_7 \rightarrow S_9$, $S_8 \rightarrow S_9$, and $S_8 \rightarrow S_{11}$ (the final one is must-alias)[1].

All semantic clobber antidependences involve a write to either memory location $\texttt{mem}[t_7 \texttt{ + } t_8]$ or memory location $\texttt{mem}[t_6 \texttt{ + 4}]$. For this simple example, it is possible to place a single cut that cuts all antidependences exactly between the statements S_8 and S_9 as shown in Figure 2(c). With this cut in place, the function is ultimately divided into two idempotent regions in total: one region with entry point at the beginning of the function and the other beginning at the site of the cut.

2.4 Recovery Using Idempotence

While intuitive that idempotence can be used to recover from failures with no visible side-effects such as traditional hardware exceptions, it is less obvious how it can be used to recover from failures that introduce incorrect control flow effects (e.g. branch mispredictions) or have other side-effects (e.g. transient hardware errors).

The approach to handling such side-effects varies. At one extreme, Hampton's work is limited to recovery from only virtual memory exceptions [10], which have no user-visible side-effects. Feng *et al.*, in contrast, propose recovery from arbitrary hardware errors by checkpointing live-in pseudoregister state and assuming memory stores due to erroneous control flow or erroneous address computations are contained [9]. For them, the possibility of incorrect control flow is irrelevant for pseudoregister state since all such state is checkpointed. Our previous work assumes that erroneous updates to non-local memory are contained, similarly to Feng *et al.*, and that the compiler protects against incorrect control flow effects with respect to pseudoregisters [8].

In this paper, we assume, as in all prior work, that incorrect control flow effects with respect to non-local memory are contained using a store buffer or similar technique. For pseudoregister state, however, we develop code generation strategies for both the case where control flow is always the same and for where control flow can vary upon re-execution.

3. Overview

While integrating idempotence analysis into the register allocation flow naturally seems more efficient than simply checkpointing all live-in register state, the extent to which this is true depends on a variety of factors. This paper explores three such factors in detail, namely the impact of (a) the sizes of idempotent regions, (b) the instruction set architecture (ISA), and (c) the need to account for potentially variable control flow upon re-execution.

In this section, we elaborate on the significance of each of these factors. In Section 4, we then present our algorithms and data structures for integrating idempotence in-

[1] As an aside, the large number of may-alias dependences illustrates the difficulty in statically analyzing small functions with non-local side-effects such as the one shown.

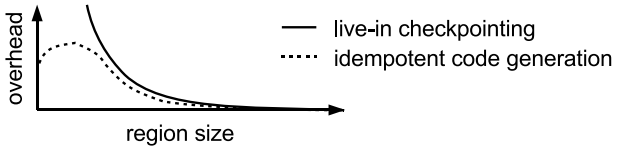


Figure 3. Run-time overhead in relation to region size.

formation with register allocation. Section 5 presents performance analysis and results. Finally, Section 6 concludes.

3.1 Idempotent Region Sizes

The type of failure idempotence is used to recover from affects the optimal idempotent region size in two ways: large idempotent regions are good when it is important to minimize the *detection stall latency* at region boundaries (execution may not proceed beyond the end of region until the its execution is verified correct); however, small region sizes are sometimes better because they minimize the *re-execution penalty* in the event of failure and recovery. Different types of failures have different characteristics with respect to these two features. However, this is fairly well understood.

What is less well understood is how region size affects the run-time overheads introduced by compiling to preserve the idempotence property itself, irrespective of the type of failure. In this paper, we explore in detail this relationship, which is quite different when using the approach of checkpointing all live-in state (*live-in checkpointing*) and integrating idempotence analysis into register allocation (*idempotent code generation*). The relationship for each of the two cases is as illustrated by Figure 3. The shape of each curve is as explained below.

Live-in checkpointing. When checkpointing, the amount of state that must be saved per idempotent region is simply the number of live-in registers at the entry point of the region. Let r represent the number of instructions to execute a region, and let the number of live registers at the entry point of a region be approximated by a constant, l. Assume moreover that the cost of checkpointing one live-in register corresponds with exactly one additional instruction. Then, the equation below computes the overhead under the live-in checkpoint approach o, as a fraction of r.

$$o = \frac{l}{r}$$

The fractional overhead o is inversely proportional to r, approaching zero as r approaches infinity, irrespective of l. This inverse relationship is what is shown for the corresponding curve of Figure 3.

Idempotent code generation. As with live-in checkpointing, idempotent code generation exerts some amount of additional register pressure on the function: at the entry point of a region each register that holds a value that could be used subsequent to that point must be preserved, even when such a register might be dead at some points inside the region and available for re-use. A key difference is that, in the case of live-in checkpointing, such a register is always spilled; with idempotent code generation, however, it may not need to be.

How the overhead of idempotent code generation correlates with region size is roughly as depicted by the corresponding curve of Figure 3. Note that the curve is neither monotonically increasing or decreasing with region size. Below, we explain the shape of this curve. For ease of explanation, we make the simplifying assumption that there are enough live variables at any given point in the program that all of a processor's physical registers can be usefully employed at all times. We also assume a fixed number of live-in registers, as we did in the live-in checkpointing case, as well.

The intuition is as follows: Initially assume each instruction forms its own idempotent region. This can be supported quite simply by modifying instructions that overwrite a source register to write to a separate register (spilling another register if necessary). The occurrence of such instructions is in most cases rare[2], and hence this introduces only modest additional pressure and incurs little run-time overhead as illustrated at the far left point of the curve.

As region size grows beyond one instruction, however, quickly register pressure grows and registers must be spilled to preserve input registers. Initially, the rate of growth can be as high as one register spill for each additional register-writing instruction. Over time, however, register pressure starts to diminish as live registers are forcibly spilled, allowing those registers to be re-used. Eventually the worst case is reached, where all live registers are pushed to the stack before the region's start. This worst case entails a fixed maximum cost that is amortized over the region's execution, and hence as region size approaches infinity, the fractional overhead approaches zero.

In Section 5.2 we concretely evaluate the impact of region size on the overhead of idempotent code generation, fitting the corresponding curve of Figure 3 with actual data. Overall, we produce two main findings. First, we find that performance overheads quickly approach zero as regions grow beyond roughly 50 instructions, and we identify a variety of program transformations and analyses that expose the inherent idempotence in applications and allow this to happen. Second, we find that code generation of small (10-25 instruction) idempotent regions introduces performance overheads commonly in the range of 10-15%. Heuristics that attempt to minimize register pressure reduce this by 1-4%.

3.2 ISA Sensitivity

For idempotent code generation, three ways in which instruction sets differ are of general significance: *register-memory* addressing support, *three-address* instructions support, and the number of *available registers*.

Register-memory vs. register-register. Whether an instruction set is a RISC-like load/store instruction set or not

[2] It is rare generally speaking, although it is not true for instruction sets with predominantly two-address instructions (e.g. x86). See Section 3.2.

can affect the performance overhead in terms of dynamic instruction count. x86 is not a load/store instruction set, and hence it is often able to account for additional register pressure by spilling instructions "for free", folding a register spill or reload into an existing instruction. A load/store instruction set such as MIPS or ARM does not share this capability.

Two-address vs. three-address instructions. Certain instruction sets, such as x86, implement many operations as so-called "two-address" instructions. These instructions read two source operands and overwrite one of them with the result. Such instructions are self-antidependent and hence *not idempotent*; when contained inside idempotent regions they must be preceded by some instruction that first defines the overwritten source register before it is overwritten. When no such instruction exists, a (logically redundant) move or copy instruction must be inserted to satisfy this requirement. This can artificially boost performance overheads particularly when idempotent region sizes are small. Three-address instructions, supported by instruction sets such as MIPS or ARM, avoid this inconvenience and hence are preferred.

Few registers vs. many registers. For load/store instruction sets, more registers allow the instruction set to accommodate additional register pressure more easily by simply allocating more available registers rather than spilling to the stack. However, as is the case without idempotence, more registers are only beneficial as long as they can be useful. Hence, the same trade-off in choosing the number of architectural registers exists with or without idempotence; idempotence only affects this trade-off to the extent that it exerts additional register pressure.

In Section 5.3 we evaluate the impact of these different ISA factors on the overhead of idempotent code generation. We find that, when region sizes are small, (1) memory-register support in x86 typically improves performance by 1-2% compared to ARM when register pressure is high, (2) three-address instruction support in ARM typically improves performance by 2-3% compared to x86 when register pressure is low, and (3) when register pressure is high and region sizes are small, achieving the same performance with idempotent code generation as without typically requires increasing the number of registers by as much as 60%.

3.3 Non-Deterministic Control Flow Effects

The region construction algorithm of Section 2.3 forms regions by ensuring that no region contains a clobber antidependence. Implicit in this formulation is the assumption that control flow does not vary upon re-execution. During recovery from mis-speculations or hardware faults, however, control flow may vary upon re-execution. In this case, the absence of clobber antidependences alone is not sufficient to guarantee idempotence, and thus a more general formulation is needed. To this end, below we introduce the more general concept of a *clobber dependence* (of which a clobber antidependence is a special type).

Clobber dependences. In the presence of potentially variable control flow, any variable whose value at the entry point of the region *might be read* after entry to the region must not be overwritten to preserve the idempotence property. This is in contrast to the clobber antidependence relationship, which only preserves variables that *must have been read* after entry to the region (with respect to the point where they are overwritten). The distinction is subtle, but important.

To support the non-determinstic control flow effects resulting from potentially erroneous control flow, we introduce the term *clobber dependence* to denote a write occurring inside a region where the write is to a variable that is *statically live-in* to the region. A clobber antidependence, described in similar terms, is a write to a variable that is *dynamically live-in* to a region *with respect to the point of the write*. In this light, it is easy to see that a clobber antidependence is simply a special type of clobber dependence.

In Section 5.4, we evaluate the differences in reasoning about idempotence in terms of clobber *dependences* compared to clobber *antidependences* during code generation. We find that using the more general concept of a clobber dependence, in addition to yielding a simpler code generation algorithm (see Section 4.3), increases the performance overhead of idempotent code generation by less than 1% on average, although in two specific cases the overhead increases by roughly 10%.

4. Idempotent Code Generation

This section presents our algorithms for integrating idempotence information during register allocation. Section 4.1 presents the new data structures we use and Sections 4.2 and 4.3 present our algorithms for deterministic and non-deterministic control flow, respectively.

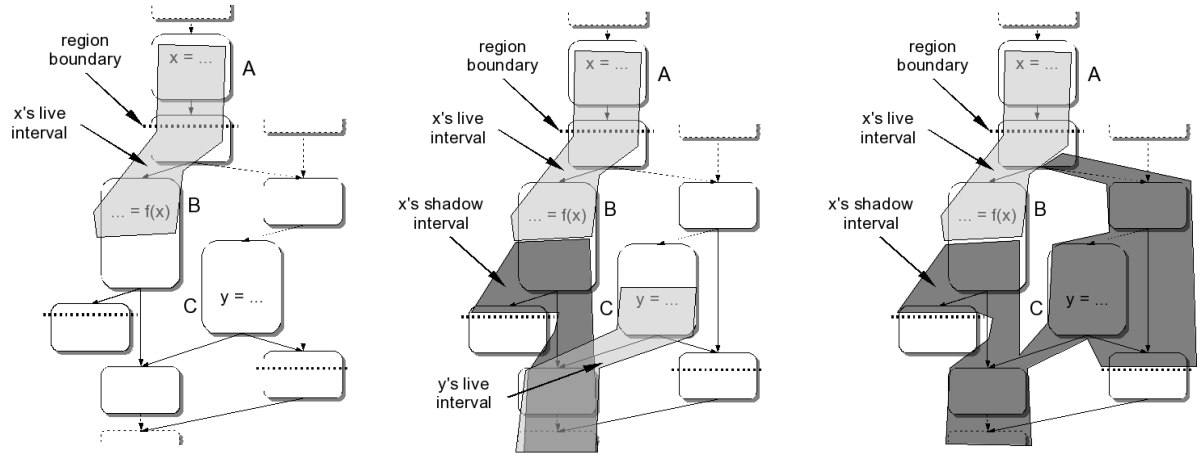
4.1 Region Intervals and Shadow Intervals

On the data structure side, we introduce the concepts of a *region interval* and a *shadow interval*, which are similar in spirit to the existing concept of a *live interval*—a compiler concept commonly used to track the liveness of variables.

A **live interval** spans the definition point(s) of a variable to all uses of that variable, and is easily computed using well-known data-flow analysis techniques [3]. Figure 4(a) shows the assignment to a variable x in basic block A and the sole use of the variable x in basic block B. The live interval of variable x shown using a transparent light gray overlay.

A **region interval** spans all basic block ranges contained inside a region, and a **shadow interval**, associated with each variable, spans the ranges where a variable must not be overwritten specifically to preserve idempotence. For simplicity, we define the shadow interval as disjoint from the live interval, since the live interval already prevents overwriting where the two interval types might otherwise overlap,

The shadow interval is computed in terms of live intervals and region intervals using an algorithm that varies based on assumptions about control flow. Below, the algorithm



(a) example CFG and live interval (b) "deterministic" shadow interval (c) "non-deterministic" shadow interval

Figure 4. An example control flow graph (CFG) subset annotated with the live interval and shadow interval of a variable x.

that assumes invariable (deterministic) control flow is presented in Section 4.2. The assumption of potentially variable (non-deterministic) control flow simplifies the algorithm, but makes it more conservative. This simplified algorithm is presented in Section 4.3.

4.2 Algorithm 1: Deterministic Control Flow

Under deterministic control flow assumptions, a variable write may be co-allocated with a variable that is statically live-in to the region as long as the live-in variable is dynamically dead with respect to the point of the write. This "dynamically dead" property guarantees that the variable will never be read, even upon re-execution, and even though it may be statically live at the point of re-execution. A live-in variable is known to be dynamically dead beyond some point if (a) no read of that variable's value may have occurred along any path leading up to that point and (b) no read of that variable's value can occur anywhere after that point. Hence, a variable *may not* be overwritten only in the inverse case, i.e. (a) a read of that variable's value may have already occurred or (b) a read of that variable's value can occur in the future. The live interval concept already prevents overwriting if (b) is true; hence, the shadow interval concept need only additionally prevent it for (a). The shadow interval of a live-in variable thus consists of the points inside a region interval that are reachable from any read inside the region (with all live interval ranges removed).

The algorithm for computing this shadow interval is shown in Algorithm 1. The algorithm takes as input a variable v and the set of region intervals R in v's function. It accumulates on the shadow interval s those ranges in a region r that are reachable from the "use" points of v (the points where v is read). The final step of the algorithm removes all ranges in v's live interval, l, from s.

The result of applying this algorithm to the example CFG of Figure 4(a) (again, with $v = \mathsf{x}$) is illustrated in Figure 4(b). The shadow interval of x is shown using a dark gray overlay. The live interval of the variable y is also shown in order to concretely motivate the the use of the shadow interval concept in contrast to simply extending the live intervals of idempotence-preserved variables so that live-in variables are also marked live-out, as previously suggested [8]. Doing so would not allow x and y to be co-allocated. With the shadow interval concept, however, a live interval may overlap a shadow interval as long as the definition point(s) of the live interval do not overlap the shadow interval.

Algorithm 1 COMPUTE-SHADOW-INTERVAL(v, R)

```
1: s ← ∅
2: l ← COMPUTE-LIVE-INTERVAL(v)
3: U ← GET-USE-POINTS(v)
4: for all r ∈ R such that (the entry point of r) ∈ l do
5:     for all u ∈ U such that u ∈ r do
6:         s ← s ∪ COMPUTE-REACHING-INTERVAL(u, r)
7:     end for
8: end for
9: return s \ l
```

Algorithmic Complexity. Assuming the union (logically "append") and difference (logically "remove") binary set operations used in Algorithm 1 have complexity $O(|z|)$, where z is the operand to the right, the worst case complexity of the loop is bounded by $|U| \cdot \sum_{r \in R} |r|$. The term $\sum_{r \in R} |r|$ is in turn bounded by $n \cdot s$, where n is the number of instructions in the function and s is the *sharing factor*—the maximum number of regions to which a single instruction may belong (our definition of region allows a single instruction to belong to multiple regions). While $|U|$ and s are both theoretically bounded by n—implying worst case complexity $O(n^3)$—in practice $|U|$ and s are both small with $|U| \ll n$ and $s \ll n$. Moreover, the mean $\overline{|U|}$ across all variable instances v in the function is a constant, while s can additionally be stati-

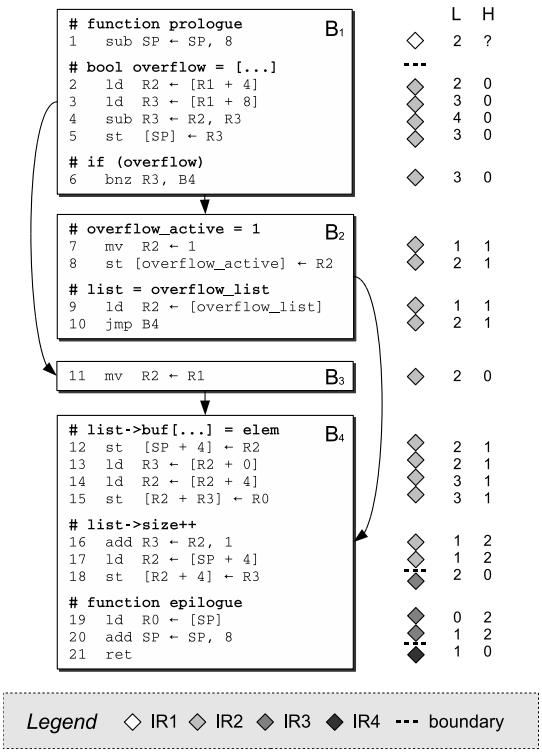


Figure 5. Code generation output with Figure 2(c) as input.

cally bounded if needed. Hence, in practice the *average* case complexity is only $O(n)$. To provide contrast, the average complexity of the live interval computation is also $O(n)$.

Code Generation Example. Figure 5 shows sample code generation output given as input the example function of Figure 2(a) partitioned as previously illustrated in Figure 2(c). The left side of the figure shows stylized assembly code with arrows indicating control flow edges. The diamonds on the right mark the region to which each instruction belongs, according to the legend shown at the bottom of the figure ("IR" stands for Idempotent Region). The compiler produces idempotence boundaries at either (1) register move and store instructions (a free "marker" bit is assumed for those instruction types), for which the boundary logically occurs *before* the instruction, or (2) at stack pointer register SP updates, for which the boundary logically occurs *after* the instruction (SP updates are a special kind of non-idempotent instruction that we assume can be supported as long as it terminates a region.) Finally, the far right shows two columns with the column headers L, for "live", and H, for "held". The live column marks the number of general-purpose registers (R0-R3) live immediately before each instruction. The held column marks the number of "held" registers immediately before each instruction. These registers are dead but not re-usable because their shadow interval extends out to this point.

In the code generation output, the cut placed before S_9 in Figure 2(c) translates to a cut before the store instruction on line 18. The compiler preserves the function argument

Algorithm 2 COMPUTE-VCF-SHADOW-INTERVAL(v, R)

1: $s \leftarrow \emptyset$
2: $l \leftarrow$ COMPUTE-LIVE-INTERVAL(v)
3: **for** $r \in R$ **such that** (the entry point of r) $\in l$ **do**
4: $\quad s \leftarrow s \cup r$
5: **end for**
6: **return** $s \setminus l$

`list` live at the entry of region IR2, initially in R1, by reassigning it to R2 in lines 9 and 11. To accomodate the increase in register pressure in block B_4 where R1 is held (dead but unavailable), R2 is spilled on line 12 and then reloaded on line 17. In total, 21 instructions are generated. For comparison, without idempotent code generation, only 17 instructions would be generated (see Figure 7).

4.3 Algorithm 2: Non-Deterministic Control Flow

In the presence of potentially variable control flow, *no* variable written in a region may be co-allocated with a variable that is statically live-in to a region. Otherwise, a clobber dependence might arise over the allocated resource. The shadow interval of a live-in variable is thus simply the region interval with all live interval ranges removed. The algorithm for computing the shadow interval is thus also very simple. Given the same inputs as in Algorithm 1, the algorithm is as presented in Algorithm 2.

The result of applying this algorithm to the example CFG of Figure 4(a) (with v = x) is illustrated in Figure 4(c), with the shadow interval shown using a transparent dark gray overlay. It shows how the variable y cannot be co-allocated with variable x because its definition point falls in the shadow interval of x. The code generated using this algorithm happens to be the same as for Algorithm 1 (Figure 5).

5. Analysis and Compiler Evaluation

The region construction algorithm of Section 2.3 and the algorithms proposed in Section 4 were implemented with static verification using LLVM 3.0 [12]. In our evaluation, source code is compiled to LLVM IR using GCC with the LLVM DragonEgg plug-in [1]. The initial region construction is performed on the IR, and the code generator then takes the output of this construction and further refines and compiles the regions as the IR is gradually lowered to machine code. Our compiler targets both the x86 (x86-64) and ARM (ARMv7) instruction sets.

Benchmarks. In our evaluation, we consider three benchmark suites: SPEC 2006 [16], a suite targeted at conventional single-threaded workloads, PARSEC [4], a suite targeted at emerging multi-threaded workloads[3], and Parboil [17],

[3] Only five PARSEC benchmarks were chosen. These were the five benchmarks that (a) could be easily compiled for both x86-64 and ARMv7, (b) spend more than 90% of their execution time not in external library code, and (c) do not have an excessively long program setup phase.

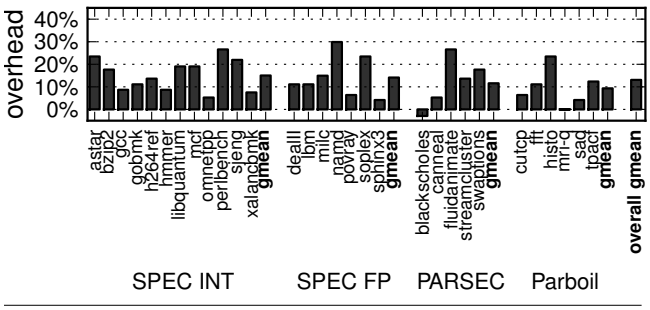


Figure 6. Baseline execution time overhead.

a suite targeted at massively parallel GPU-style workloads written for CPUs.

Measurements. We focus on two evaluation metrics: *execution time* and *region size*. Execution time is measured in terms of dynamic instruction count, which provides an architecture-neutral platform for compiler evaluation. For x86-64 we measure this using Pin [13], and for ARMv7 we use gem5 [5]. For region size, we specifically focus on the length of the instruction sequences dynamically executed though regions. We call this the *path length*, and measure it in terms of executed x86-64 instructions using Pin.

Execution. To account for the differences in instruction count between the compiled benchmark versions (i.e. with and without idempotent code generation), simulation time in Pin (x86) and gem5 (ARMv7) is measured in terms of the number of functions executed, which is constant between both versions. Initially, all benchmarks execute unmonitored for the number of function calls needed to execute at least 10 billion instructions for the benchmark version generated with the typical LLVM compiler flow. Execution is then monitored for the number of function calls needed to execute 10 billion additional instructions for this same version. For benchmarks with fewer than 20 billion instructions, the entire execution is monitored following the number of function calls needed to exit the setup phase of the benchmark.

5.1 Baseline Results

Figure 6 shows the baseline performance results that we compare against in the following sub-sections. The data was generated for the x86 instruction set using the region construction algorithm developed in our prior work [8] and using Algorithm 1. In Section 5.2 we vary how the regions are constructed, in Section 5.3 we vary instruction set features, and in Section 5.4 we vary the code generation algorithm.

A brief comparison with our prior work. Our prior work reports a roughly 8% geometric mean overhead while our baseline results show 12%. The main causes of difference are that: (1) our baseline does not include any loop optimizations, including loop unrolling; (2) our baseline ISA is x86, compared to ARM; (3) our compiler is built on top of LLVM 3.0, not LLVM 2.9, and with respect to register allocation there are very substantial differences between these two LLVM versions; (4) our compiler supports various architecture- and code-specific corner cases; and (5) we consider slightly different (overall, more) benchmarks.

5.2 Path Length Sensitivity

As explained in Section 3.1, it is often possible to reduce overheads by both decreasing and increasing idempotent path lengths. In this section, we consider both approaches separately in the two subsections below.

Increasing Path Length

In analyzing our applications, we observe that many benchmarks exhibit smaller path lengths than are actually achievable. *Limited aliasing information* in the compiler is in part responsible for short path lengths in specific benchmarks such as hmmer, lbm, and fft. Additionally, *overlooked loop optimizations*, *large storage arrays*, and *intra-procedural scope* provide three other reasons why path lengths are often unnecessarily small. In the discussion that follows, these four total problem sources are identified using the labels ALIASING, LOOP-OPT, ARRAYS, and SCOPE.

ALIASING. With limited aliasing information, a load/store pair may be believed to alias while in practice they could not, or would only alias under specific and rare circumstances. In some cases, the ambiguity is due to a lack of source-level annotations and/or inter-procedural scoping. However, in several cases the problem is simply that LLVM does not provide a flow-sensitive alias analysis. Flow-sensitivity helps particularly in the case of loops, for instance, where a load and store may only alias across iterations of some outer loop, or where a store and load may alias only when the store comes before the load inside the same iteration. Such loop-level aliasing information is a common feature of auto-parallelizing compilers [15], and although forthcoming in LLVM, it is not currently supported.

LOOP-OPT. Certain loop optimizations, such as loop fission/fusion, loop interchange, loop peeling/unrolling, and scalarization allow the construction of larger idempotent regions because they allow clobber antidependences to span longer distances and/or allow multiple of them to be cut simultaneously by a single boundary. Additionally, cuts inside loops require extra copies [8] and loop transformations can help with this as well.

ARRAYS. The region construction algorithm proposed in prior work assumes all *non*-clobber antidependences are eliminated by using local storage to hold the result of the "protecting" initial write and replacing the read to use the local storage [8]. This technique can be impractical, however, for a large or unknown number of initial writes (as in e.g. array initialization), since it effectively requires duplicating the non-local memory storage on the stack. For efficiency, our compiler does not duplicate this state on the stack, and hence it conservatively cuts the non-clobber antidependences that arise. For most benchmarks they are rare, with any additional cuts that they require masked by inter-procedural effects. However, certain applications exhibit

a specific pattern, the *initialization-accumulation* pattern, where these antidependences arise in abundance. In this pattern the initialization phase and accumulation phase together are idempotent but the accumulation phase alone is not.

SCOPE. Several applications, particularly those written in an object-oriented style, tend to execute in sequences of relatively small functions. Naturally, small function bodies limit any intra-procedural alias analysis and our region construction additionally forces cuts at function boundaries, further constraining idempotent path lengths.

Assuming the capability to overcome the four limiting factors identified above, we wish to understand the resulting idempotent path lengths and associated overheads. We consider as a case study the PARSEC and Parboil benchmarks, which are relatively small and well-contained and thus suitable for manual inspection and modification. These benchmarks are limited by each of the four factors in the manner indicated by the second column of Table 1.

We overcome each of the limiting factors as follows (here, we primarily rely on manual annotations due to insufficiently advanced compiler support). For ALIASING, we assist the compiler in identifying no-alias memory relationships by manually providing source code annotations using the C/C++ *restrict* keyword and/or performing code refactoring. For LOOP-OPT, we manually modify the program source code to refactor loops and or scalarize non-local memory accesses. For ARRAYS, we modify the code generator to initially ignore potentially non-clobber antidependences in the antidependence cutting phase of the region construction. The non-clobber antidependences that then emerge are handled in a separate post-cutting phase "as-needed", in the same manner that loop pseudoregister antidependences are handled as a post-pass [8]. Finally, for SCOPE, we add annotations to force inlining.

Results. Columns 3-4 of Table 1 show that very large path length improvements can be achieved with these modifications compared to the baseline. In all cases, the geometric mean path length grows to at least one hundred instructions and in the vast majority of cases they grow to many thousands of instructions[4]. More importantly, however, we see examining columns 5-6 of the table that these transformations create a very sharp reduction in dynamic execution time overhead. In most cases, execution time overhead is effectively eliminated. The data thus strongly supports constructing larger idempotent regions to reduce the compiler-induced execution time overheads. The reason is intuitive: larger regions allow amortizing the cost of preserving region live-in state over longer distances.

Decreasing Path Length

Computationally-intensive programs, such as those from the Parboil and PARSEC suites, tend to have extractably large idempotent regions. However, programs written for general purpose processors—particularly those written in an object-oriented style with frequent updates to object member variables in heap memory—do not generally have such large regions. These programs often contain many ambiguous "may-alias" memory references of the same variety as those that inhibit automatic parallelization by compilers [15]. For these programs, the semantically idempotent regions are not generally large enough to usefully amortize the register pressure overheads of preserving their idempotence throughout the compilation process. These programs are sometimes better served by judiciously shortening region sizes to minimize overheads resulting from register pressure.

To explore the opportunity, we modified our compiler to take as argument an optional *pressure threshold*, which is a value from 1 to 10. When this value is set, the compiler examines the entry point of each basic block and computes the 10 "heaviest" (least likely to be spilled) pseudoregister values at those points using various weighting heuristics such as loop depth, etc. Among those 10 pseudoregisters it computes the number whose values would be preserved purely for idempotence purposes. If that number meets or exceeds the pressure threshold, then an idempotence boundary is placed at the entry point of the basic block.

The algorithm examines the entry point of basic blocks because that is where the number of held registers is typically highest due to registers becoming dynamically dead as a result of control divergence. This can be seen examining the compiler output previously shown in Figure 5. At the start of B_2 register R1 is a held register because it is dead

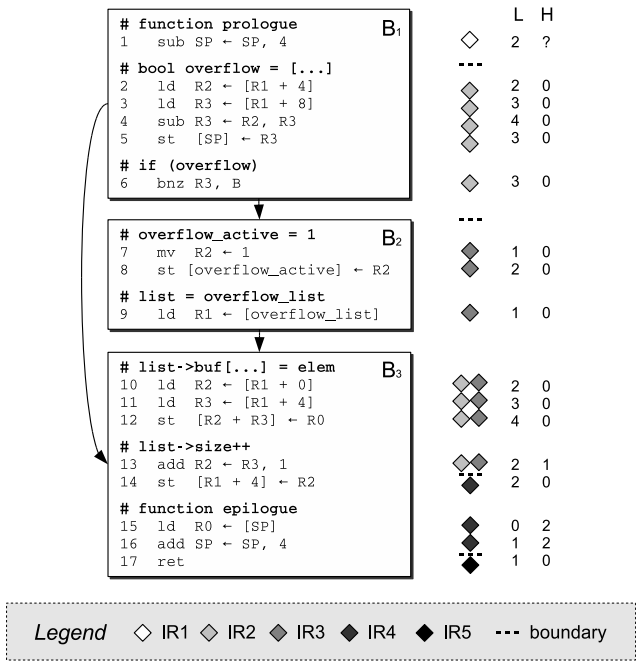


Figure 7. Code generation output given Figure 2(c) as input with a pressure threshould value greater than or equal to 1.

[4] As desired, path lengths can still be arbitrarily reduced by employing loop blocking and other similar techniques.

Benchmark	Limiting Factor(s)	Length Before	Length After	Overhead Before	Overhead After
blackscholes	ALIASING, SCOPE	78.9	>10,000,000	-2.93%	-0.05%
canneal	SCOPE	35.3	187.3	5.31%	1.33%
fluidanimate	ARRAYS, LOOP-OPT, SCOPE	9.4	>10,000,000	26.67%	-0.62%
streamcluster	ALIASING	120.7	4,928	13.62%	0.00%
swaptions	ALIASING, ARRAYS	10.8	210,674	17.67%	0.00%
cutcp	LOOP-OPT	21.9	612.4	6.44%	-0.01%
fft	ALIASING	24.7	2,450	11.12%	0.00%
histo	ARRAYS, SCOPE	4.4	4,640,000	23.53%	0.00%
mri-q	—	22,100	22,100	0.00%	0.00%
sad	ALIASING	51.3	90,000	4.17%	0.00%
tpacf	ARRAYS, SCOPE	30.2	107,000	12.36%	-0.02%

Table 1. PARSEC and Parboil benchmarks, limiting factors, and characteristics before and after addressing limiting factors.

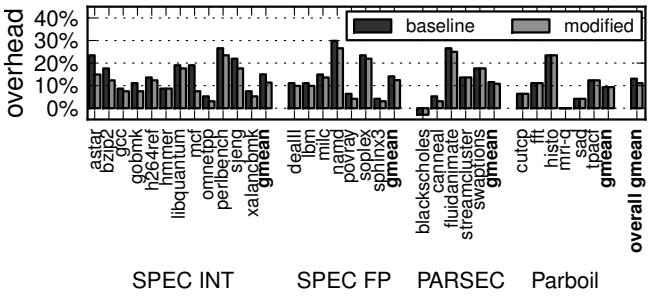


Figure 8. Overhead with a pressure threshold value of 5.

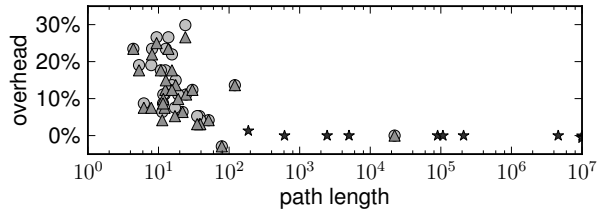


Figure 9. Path length versus execution time overhead.

along that control flow path. Figure 7 shows the output after setting a pressure threshold value of 1 or more. It causes the register pressure exerted by R1 along B_2's control flow path to be subsequently removed. As a result, the number of register spills is reduced by 1 and the code is more compact with a total of only 17 instructions (which happens to be ideal).

Results. Figure 8 shows that the modified compiler, using a pressure threshold value of 5, produces lower execution time overheads relative to the baseline. However, the differences are not large; while the baseline compiler yielded runtime overheads of roughly 9-16%, the modified compiler reduces these overheads to 9-13%—only a modest 3% reduction at the high end. Across a range of threshold values, the difference is similarly not large (not shown).

Summary and Implications

Figure 9 plots the correlation between geometric mean path length and run-time overhead for the baseline results (circles), after increasing path lengths (stars), and after decreasing path lengths (triangles).

With respect to large regions, it suggests that path lengths of 50 instructions and more are sufficient to yield negligible overhead overall; benchmarks omnetpp, blackscholes, canneal, sphinx3, and sad all have mean path lengths in the range of 30-80 instruction under the baseline, and in all cases the overhead is relatively low, between -2 and 5%. The main exception is streamcluster, which has path lengths of over 100 instructions but still has overheads in the 10% range for x86. The compiler compiles this benchmark's most critical loop differently when compiling for idempotence, inserting one extra instruction and growing the size of the loop from 7 instructions to 8 instructions. The extra instruction appears redundant, and indeed, comparing against the same code compiled for ARMv7 (see Figure 10), the loop is compiled without any extra instructions and the overall overhead becomes negligible. We thus conclude that this difference is purely due to noise that our extensions introduce into the compiler register allocator algorithm.

With respect to small regions, however, no clear rule-of-thumb emerges. Evidently, the overheads of compiling for small idempotent regions is heavily dependent on existing application register pressure and control flow behavior; while in some cases constructing smaller regions to reduce register pressure provides a substantial improvement in performance, in other cases it provides little or no improvement.

5.3 ISA Sensitivity

We now consider and evaluate the extent to which ISA register-memory addressing support, three-address instructions support, and the number of available registers affect the compiler-induced overheads of idempotent code generation.

With respect to register-memory addressing and three-address instruction support, a perfectly controlled experiment is challenging given that no mainstream register-to-register two-address instruction set (nor register-to-memory three-address instruction set) exists. As a compromise, we evaluate the impact of both features at the same time by comparing dynamic instruction count increase across the x86-64 and ARMv7 instruction sets; both instruction sets have the same number of general purpose integer registers (16) and we compile then with the same floating point support (x86-

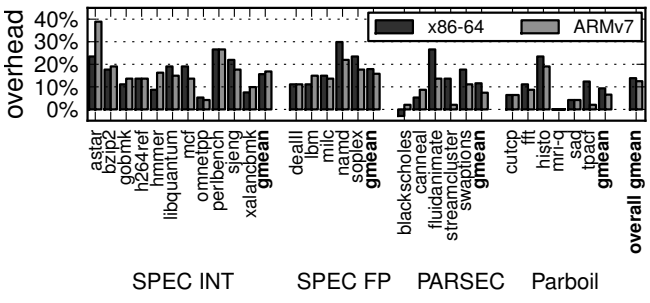


Figure 10. Execution time overhead x86-64 vs. ARMv7.

64 has 16 single-precision FP registers with SSE2 and we compile ARMv7 to similarly use only 16 single-precision FP registers, even though 32 are available with NEON).

To evaluate the impact of the number of available registers, we attempt to quantify the number of additional registers that would be needed in the ISA for idempotent code generation to achieve achieve comparable performance to normal code generation. For this piece, we focus specifically on benchmarks with small idempotent regions, since intuitively as regions grow larger the impact of the number of available registers approaches insignificance.

Results. Figure 10 compares the overheads in execution time for x86-64 and ARMv7 in terms of dynamic instruction count[5]. Overall, the figure indicates a lower geometric mean overhead for ARMv7, at 12.6%, than for x86-64, which has 13.9% overhead. Curiously the differences closely correlate with the type of benchmark—integer (SPEC INT) vs. floating point (SPEC FP, PARSEC, and Parboil). For floating point benchmarks, ARM has substantially lower overhead (typically by 2-4%) because register pressure is lower, which enhances the benefit of three-address instruction support in ARM relative to memory-register support in x86. With integer benchmarks, however, we see the opposite effect; x86 has lower overhead (typically by 1-3%) than ARM because register pressure is higher, which enhances the benefit of memory-register support in x86 relative to three-address instruction support in ARM. Overall, the benefits of register-memory addressing and three-address instructions can be quite substantial (up to 15%), as we see for `astar` and `fluidanimate`, respectively.

Finally, to understand the impact of the number of available registers, Figure 11 shows the effect on the execution time overhead for ARMv7 as we reduce the number of general purpose integer registers (GPRs) from 16 to 14, 12, and 10. We focus only the SPEC INT integer benchmarks, which are most affected, and see that performance of having only 10 (removing 6 out of 16) general purpose registers roughly corresponds with the performance of compiling with a pressure threshold value of 5, for which the data is also shown on the right using a hatched bar. The implication is that, assuming region sizes are constrained and register pressure is continually high (as is presumably the case with only 10 registers), compiling for idempotence with no performance loss generally requires a roughly $\frac{6}{10} = 60\%$ increase in the number of available registers. This effective 60% increase in register pressure seems high. However, it also seems intuitive that idempotent region sizes in the range of 5 to 15 instructions, which is typical for the SPEC INT benchmarks, would have a roughly proportional number of live-in registers that must be preserved for idempotence.

[5] Data for SPEC INT's `gcc` and SPEC FP's `sphinx3` and `povray` is not presented since these benchmarks either did not compile for the version of LLVM DragonEgg used or would not run for the version of gem5 used.

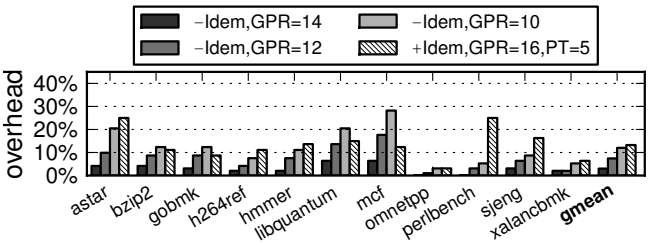


Figure 11. Execution time overhead for non-idempotent ARMv7 code assuming 10, 12, and 14 general purpose registers (-Idem,GPR=x) compared to a 16 GPR baseline. The overhead of idempotent ARMv7 code compiled with 16 GPRs using a pressure threshold of 5 (+Idem,GPR=16,PT=5) is shown for comparison.

5.4 Sensitivity to Control Flow Assumptions

Table 2 reports the increase in execution time for x86-64 compiling using the deterministic (Algorithm 1) and non-deterministic (Algorithm 2) code generation algorithms presented in Section 4. The table shows that overall, there is only a 0.5% overhead difference between the two algorithms. The data thus suggests that performance is largely unaffected by the distinction between determinstic and non-deterministic control flow assumptions.

While this result may seem counter-intuitive, its implication, which is that the opportunity to take advantage of determistic control flow is rare, is corroborated by inspection. We find that, while deterministic control flow allows statically live-in but dynamically dead registers to be re-used, the occurrence of this statically-live but dynamically-dead condition is relatively rare: it requires both an interval where a register is live but only used after control flow divergence (in which case the register is a good candidate for spilling) and register scarcity (if registers were indeed scarce, the live register would most likely already have been spilled).

For most benchmarks, the overheads are effectively unchanged. However, for two specific benchmarks, SPEC INT's `libquantum` and SPEC FP's `namd`, is there a very noticeable difference of roughly 10% (not shown in table).

6. Summary & Conclusions

Using idempotence for recovery is a novel and emerging paradigm with applicability in a variety of domains [7–11, 14, 18, 19]. However, its effectiveness is susceptible to the

	SPEC INT	SPEC FP	PARSEC	Parboil	Overall
A1	15.0%	14.1%	11.6%	9.3%	13.1%
A2	16.2%	14.7%	11.6%	9.1%	13.6%

Table 2. Geometric mean overheads using the deterministic (A1) and non-determinstic (A2) control flow algorithms.

whims of the compiler algorithms that generate idempotent code, the structure of applications that give rise to the code, and the architectures that run the code. An understanding of the resulting sensitivities is crucial to furthering the state of the art in applying the technique and its variants.

As a first result, this paper finds that it is possible to increase the performance of idempotent generated code both by increasing *and* decreasing idempotent region sizes. Although ideal region size depends on a variety of characteristics, the compiler-induced performance overheads for large regions quickly become negligible as regions grow beyond roughly 50 instructions. For small regions (typically 10-20 instructions in size), however, some overhead is unavoidable, and typical overheads of 10% are common, even with careful tuning. Yet with smaller idempotent region sizes, the instruction set plays an important role; we find that it influences performance typically by 2-3%, and sometimes by much more. Finally, we find that, in accounting for possible control flow side-effects, a more general idempotent code generation algorithm has only slightly worse performance than a more specific code generation algorithm.

In the end, we conclude that, when large idempotent regions are achievable and they make sense (i.e. recovery is infrequent, implying low re-execution costs), large regions are ideal in allowing the compiler to generate highly efficient idempotent code with effectively no overhead. In this mode, idempotence analysis is highly effective in simplifying recovery by allowing memory state to be freely overwritten and amortizing register preservation costs over large groups of instructions. Additionally, we find that a variety of program transformations and analyses can expose the inherent idempotence in loop-intensive applications to allow such large region sizes, suggesting a compelling fit with the expressiveness of functional languages and high-level, domain-specific languages. Yet even when idempotent regions are necessarily small, we find that keeping the performance overheads low is still possible with careful algorithmic tuning and instruction set co-design.

In the interest of enabling future development in each of these directions, we hope that the public release of our compiler [2] will enable future researchers and systems designers in building the efficient recovery systems of the future.

Acknowledgments

We thank the anonymous reviewers and the Vertical Research Group for comments and the Wisconsin Condor project and UW CSL for their assistance. Support for this research was provided by NSF under the grant CCF-0845751 and by a Google U.S./Canada PhD Fellowship.

References

[1] *DragonEgg - Using LLVM as a GCC backend*. http://dragonegg.llvm.org.

[2] *iCompiler – LLVM Idempotent Code Generation*. http://research.cs.wisc.edu/vertical/iCompiler.

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

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] 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.

[6] M. de Kruijf. *Compiler Construction of Idempotent Regions and Applications in Architecture Design*. PhD thesis, University of Wisconsin-Madison, 2012.

[7] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. MICRO '11.

[8] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. PLDI '12.

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

[10] M. Hampton. *Reducing Exception Management Overhead with Software Restart Markers*. PhD thesis, Massachusetts Institute of Technology, 2008.

[11] 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.

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

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05.

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

[15] S. P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool, 2012.

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

[17] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, and L.-W. Chang. Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report, University of Illinois at Urbana-Champaign IMPACT-12-01, March 2012.

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

[19] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *ASPLOS '13*.