COMPILER CONSTRUCTION OF IDEMPOTENT REGIONS AND APPLICATIONS IN ARCHITECTURE DESIGN

By

Marc A. de Kruijf

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 07/20/12

The dissertation is approved by the following members of the Final Oral Committee:
Karthikeyan Sankaralingam, Assistant Professor, Computer Sciences
Mark Hill, Professor, Computer Sciences
Gurindar Sohi, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Mikko Lipasti, Professor, Electrical and Computer Engineering

Acknowledgments

This research product owes many things to many people.

First is my advisor, Karu, who was instrumental in many ways. More than simply mentoring me in research, he has helped me to frame my own life—to strive to live happily, peacefully, and positively. His tireless work ethic in combination with his profound respect for life-balance is something that I admire greatly. Thanks almost entirely to him, my graduate career was rarely frustrating, full of interesting and exciting work, and very rewarding. I do not know what the future awaits, but thanks to you, Karu, I feel more prepared than ever before.

The other committee member who deserves special thanks is Mark, my other professional role model. When I arrived at UW-Madison, Mark was my initial mentor, my CS 552 instructor, and he was the one to invite me to my first Computer Architecture Affiliates meeting only a month after my arrival. Mark never wavered in his support or his willingness to offer guidance. He may not know it, but Mark made me a computer architect. Thank you, Mark.

Among the other members of committee, Guri forced me to think critically about my own ideas while remaining always supportive, Somesh was a crucial resource in developing key pieces of my research—he taught me to think in very precise terms, and his jovial and spirited nature was always an inspiration to me—and Mikko was an excellent resource for technical discussions in addition to being just an all-around great person. Thank you, Guri, Somesh, and Mikko.

There is no shortage of fellow students to thank. First, the Vertical group. From the days when I was the only student of the group (with an office all to myself), the group is now over ten students strong. Thanks to everyone, with special thanks to my officemates, Venkat and Tony, who always provided great fuel for discussion and distraction. Among the other members, Emily, Chen-han, Jai, Raghu, Zach, Ryan, and Chris also deserve special mention for their support and camaraderie.

Architecture lunch: I will miss you. Thanks to all those who attended in the past and those who will continue to attend. Thanks to past alumni for their sage advice: Mike, Philip, Andy, Natalie, Dan, Yasuko, Polina, Dana, Matthew, Jayaram, Luke, Kevin, James, and Derek (barely!). Rathijit, Arka, Somayeh, Hamid, Srinath, Gagan, Jayneel: good luck to you! You will do fine.

Thanks to Google for two awesome internships and now a full-time job! I am very much looking forward to joining the group again.

Thanks to my family in Holland for their support. Although I'm not sure they ever knew exactly what I was doing, the fact that I was traveling to conferences seemed to imply that everything was fine. Thanks, mom, for your emails and your conversation. I'm looking forward to seeing you again soon. Thanks, dad, for always believing in me and trusting me in everything. You may be gone, but I feel your influence carrying on in me in all the positive ways that I know you would have wanted.

Special thanks to the other half of my family, too. Graduate school with kids is never easy. Thank you, Barbara and Alan, for your continual support, both financially and otherwise, your faith, and your love. For all that, Laura and I are extremely grateful; without your support all this would have been much, much harder.

Laura, thank you for doing this with me. This is not the life we had planned. Life has been full of surprises. At times, it has been extremely difficult. Yet we have made it work, and our future together looks bright and full of opportunity. We are very lucky in so many ways, and although it may sound cliché, it is absolutely true that I love you more today than I ever have before... and I know it will only get better. (Forgive the perfectionist in me.)

Finally, Brynne and Reece, you're too young now, but maybe someday you'll be old enough that you might want to read this. Maybe it was some conversation we had that made you curious. Maybe you're in graduate school yourself and trying to figure it all out. Or maybe it's at a point where I'm no longer around. Regardless, thank you in your youth for being sweet, loving, and passionate in your own little ways, and giving me a good excuse to step back from my work. It is easy to burn your time on meaningless things; you are both far from meaningless. They say you can't predict the future, but I *know* that you both will be amazing adults and will grow to lead happy and successful lives. I sense it from you both, even now. Love you.

Contents

Abstract

In the field of computer architecture today, out-of-order execution is important to maximize architectural efficiency, the shadow of unreliable hardware is ever-looming, and, with the emergence of mainstream parallel hardware, programmability is once again an important and fundamental challenge. Traditionally, hardware checkpointing and buffering techniques are used to assist with each of these problems. However, these techniques introduce overheads, add complexity to the hardware, and often save more state than necessary. With today's renewed focus on energy efficiency, and with the commercial importance of reduced hardware complexity in today's processor market, the efficacy of these techniques is no longer absolute.

This thesis develops a novel compiler-based technique to efficiently support a range of hardware features without the need for checkpoints or buffers. The technique breaks programs into *idempotent regions*—regions that can be freely re-executed—to enable recovery by simple re-execution. The thesis observes that programs can be executed entirely as sequences of idempotent regions, and builds a classification framework to concretely reason about different interpretations of idempotence that apply in the context of computer architecture. It develops static analysis and compiler code generation algorithms and techniques to construct idempotent regions and subsequently demonstrates low overheads and potentially large region sizes for an LLVM-based compiler implementation. Finally, it demonstrates applicability across a range of modern architecture designs in addressing a variety of problems.

The thesis presents several findings. First, it finds that inherently large idempotent regions, in the range of tens to hundreds of instructions, exist across entire programs. It also finds that a compiler algorithm for constructing the largest possible regions, through careful allocation of function-local state, is capable of constructing regions close to these sizes. Various algorithms are demonstrated that are able to sub-divide these regions into smaller regions to optimize for specific constraints. In the end, however, code generation of small idempotent regions forces relatively high compiler-induced run-time overheads in the range of 10-20% (often increasing register pressure by over 50%), while, for larger regions, this overhead quickly approaches zero as region size grows beyond a few tens of instructions. Thus, the compiler-induced costs of constructing small regions are often out-weighed by any benefits, and optimization trade-offs thus generally favor constructing regions that are a few tens of instructions or more. This optimization goal tailors the suitability of idempotence-based recovery to specific architecture domains; this thesis considers specifically architecture design and evaluation for general exception support in GPUs, out-of-order retirement in general-purpose processors, and hardware fault tolerance in emerging processor designs.

1 Introduction

Recovery capability is a fundamental component of modern computer systems. Among other things, it is used to recover from hardware exceptions and interrupts [93, 95], branch mispredictions [92, 109], hardware faults [91, 96], speculative memory-reordering [37, 55], optimistic dynamic binary translation and code optimization [30, 40], and memory conflicts in transactional memory [48, 84].

Today, recovery capabiliity commonly involves the use of special-purpose hardware structures to buffer or checkpoint speculative state until it is safe to proceed. However, creating and maintaining these structures introduces processor design complexity and can have high resource overheads. This is at odds with recent studies that show that hardware complexity and processor overheads must reduce to meet future energy and reliability constraints [11, 19, 45].

Nevertheless, recovery support remains crucial to performance, usability, and correctness in modern microprocessors:

Performance: For acceptable single-thread performance, processors speculate on the direction of branches and also on the absence of exceptions. This is true even among highly power-constrained processors such as ARM mobile processors [14, 103] and throughput-oriented multimedia accelerator processors such as IBM's Cell SPEs [41].

Usability: Debugging and virtual memory are standard features of modern microprocessors. However, the sequential ordering semantics that they traditionally require complicate processor design [35, 61, 93, 95]. Additionally, to efficiently service long-latency exception conditions such as virtual memory page faults requires a mechanism to efficiently save and restore ("recover") program state to maximize utilization of computational resources and minimize user response time.

Correctness: As transistor technology continues to scale to lower feature sizes, hardware is becoming increasingly unreliabile [19]. To allow programs to continue to operate correctly even in the face of hardware transient or permanent faults, some form of recovery support is increasingly needed.

To reconcile the need to reduce processor overheads with the desire to support program recovery, this thesis develops *idempotence* to support efficient recovery by *re-execution*. Idempotence is the property that re-execution has no side-effects; that is, an operation can be executed multiple times with the same effect as executing it only once. At the coarsest granularity any application whose inputs do not change during execution is idempotent. At the finest granularity every instruction that does not modify its source operands is also idempotent. In both cases, re-executing the operation does not change the effect of the initial execution.

Why Idempotence?

An operation is idempotent if its inputs do not change over the course of its execution. Hence, idempotence can be thought of as implicitly forming a checkpoint with respect to the inputs of the operation. In this manner, idempotence over a region of code can render traditional hardware checkpointing techniques unnecessary; in the event of failure, idempotence can be used to correct the state of the system by simple re-execution.

Moving from explicit hardware checkpointing to an implicit checkpointing model built upon idempotence can benefit computer systems at multiple levels. First, at the microarchitecture level, the absence of hardware buffering and/or checkpointing reduces interdependencies between processor structures, reduces power and area, and allows existing hardware resources to be used more efficiently in the absence of contention. Second, at the circuit level, lower threshold voltages and tighter noise margins on transistors make hardware design and verification increasingly difficult; hence, less functionality in hardware implies substantially lower hardware design and verification effort. Finally, at the program level, checkpoints can be inflexible. This inflexibility is not only inconvenient, but it can also hurt overall efficiency if the checkpoints are overly conservative.

1.1 Contributions

This thesis observes that applications can fully decompose into idempotent regions of code, and that these regions can be used to recover from a range of failure scenarios. The size and arrangement of idempotent regions is configurable during compilation, and a compiler can construct idempotent regions that are usefully large at the expense of only a small amount of run-time overhead. This thesis makes the following specific contributions:

Idempotence in computer architecture: It presents the first comprehensive analysis of idempotence and its implications for architecture and compiler design. In particular:

1. it is the first work to observe that programs can decompose entirely into idempotent regions;
2. it observes that idempotence can can be used to recover from a variety of failure conditions by simple re-execution;
3. it identifies a variety of idempotence "models" that apply in the context of computer architecture and presents a taxonomy to concretely reason about them;
4. it analyzes the potential sizes of idempotent regions that arise from the space of idempotence models and finds that the regions can be large; and
5. it converges on two idempotence models that have the desirable properties of allowing (1) the decomposition of programs entirely into idempotent regions and (2) the construction of idempotent regions of maximal size with respect to the common case of data-race-free multi-threaded execution.

Compiler design: It develops a complete end-to-end compiler design and implementation for the automated formation and code generation of idempotent regions considering a range of design trade-offs. In particular:

1. it describes a static analysis algorithm to uncover the minimal set of idempotent regions in a function given semantic constraints;

2. it analyzes and proposes techniques to balance the size versus performance trade-off in compiling idempotent regions given application and environmental constraints; and
3. it details the operation of a fully working source-to-machine LLVM-based compiler implementation made publicly available [2] and the design considerations associated with its construction.

Architecture design: It explores the opportunity to apply idempotence to GPU, CPU, and emerging fault-tolerant architecture design. In particular:

1. for GPUs, it demonstrates how idempotence can provide general exception support and improve context switching efficiency;
2. for CPUs, it demonstrates how idempotence can alleviate the power and complexity burden of in-order retirement in aggressively-pipelined processors; and
3. for fault-tolerant architectures, it demonstrates how idempotence enables recovery from hardware faults in a manner that performs measurably better than other known fine-grained software recovery alternatives.

1.2 Summary of Findings

In terms of concrete evaluation, this thesis presents the following findings:

Idempotence in computer architecture: Given perfect run-time information, achievable idempotent region sizes range from 10s of instructions to 100s of instructions as an average across entire applications, depending on the idempotence model and application characteristics.

Compiler design – static analysis and transformations: Idempotent region sizes identified by a compiler static analysis algorithm range from 10 to 100 instructions as an average across entire applications. For loop-intensive applications, transformations that expose the inherent idempotence in applications allow for very large region sizes.

Compiler design – code generation: Code generation of small (semantically-constrained) idempotent regions commonly forces performance overheads of over 10%. For larger regions, this overhead approaches zero in the limit as region size grows beyond a few tens of instructions.

Compiler design – ISA sensitivity: Among three ways in which the ISA could affect the run-time overheads of idempotence-based compilation, none appear significant. Independent of the ISA, small (semantically-constrained) idempotent regions increase register pressure by approximately 60%. For larger regions, register pressure effects approach zero in the limit as region size grows beyond a few tens of instructions.

Architecture design: GPUs can support general exceptions cleanly using idempotence with run-time overheads of less than 2% (for traditional GPU workloads). CPUs can be simplified to support exceptions with out-of-order retirement with typical run-time overheads of 10%. Adding support for efficient branch-misprediction recovery using idempotence on CPUs increases the typical run-time overheads to 20%. Finally, architectures can use idempotence to support hardware fault recovery with run-time overheads of roughly 10%, assuming low-latency fault detection capability.

1.3 Organization

The core of the dissertation is organized into three parts: *idempotence models in computer architecture*, *compiler design & evaluation*, and *architecture design & evaluation*. These three parts span Chapters 2-6, with the closing Chapters 7-8 presenting related work and conclusions.

Idempotence Models in Computer Architecture

Chapter 2 explores and analyzes the concept of idempotence as it applies to computer architecture. As background, it presents examples of idempotence applied in computer science and subsequently develops a taxonomy to reason about idempotence specifically as it applies to computer architecture. Leveraging this taxonomy, it performs an empirical study of the sizes of idempotent regions that could be attained for different idempotence models arising from the taxonomy given semantic program constraints. Finally, it identifies the two idempotence models—*architectural* and *contextual* idempotence—that are developed in the remainder of the dissertation.

Compiler Design & Evaluation

Chapters 3-5 present the static analysis, code generation, and evaluation of a compiler design that constructs idempotent regions in programs, optimizing for architectural and contextual idempotence, across a range of application and environmental constraints. Chapter 3 develops a static analysis for identifying the largest idempotent regions given semantic program constraints. Chapter 4 develops support for sub-dividing regions and preserving the idempotence property of these regions as they are compiled down to machine instructions. Finally, Chapter 5 presents a comprehensive evaluation of a full, end-to-end compiler implementation.

Architecture Design & Evaluation

Chapter 6 motivates and develops the architecture support to utilize idempotence for recovery across a range of architecture designs. The overall architectural vision is one where the analysis of idempotence occurs in software (e.g. in a compiler), and the hardware consumes the output of this analysis to enable hardware design simplification and flexibility. Specifically, the applications to GPU, CPU, and emerging fault-tolerant architecture designs are explored and evaluated. In constrast to the rigorous compiler implementation evaluation of Chapter 5, the individual architecture evaluations are more abstract, using simulation-based evaluation. Detailed microarchitecture design and implementation is left as a topic for follow-on work.

1.4 A Note on Experimental Methodology

All three parts of the dissertation are empirically grounded with a largely common experimental methodology used throughout. However, there are differences as the experimental purpose varies. Table 1.1 highlights the primary differences.

Regarding benchmarks, the benchmark suites we study throughout are SPEC 2006 [99], a suite targeted at conventional single-threaded workloads, PARSEC [16], a suite targeted at emerging

Topic	Chapter/Section	Benchmark Suites	Simulation
Idempotence Analysis	Section 2.4	SPEC 2006, PARSEC, Parboil	Pin
Compiler Evaluation	Chapter 5	SPEC 2006, PARSEC, Parboil	Pin, gem5
GPU Evaluation	Section 6.1.3	Parboil	gem5
CPU Evaluation	Section 6.2.3	SPEC 2006, PARSEC	gem5
Fault Evaluation	Section 6.3.2	SPEC 2006, PARSEC, Parboil	gem5

Table 1.1: Differences in experimental methodology for different parts of the dissertation.

multi-threaded workloads[1], and Parboil [100], a suite targeted at massively parallel GPU-style workloads written for CPUs. For the GPU-specific evaluation of Section 6.1.3 we use only the GPU-target benchmark suite, Parboil, while for the CPU-specific evaluation of Section 6.2.3 we evaluate only the other two benchmark suites.

Regarding simulation, the empirical analysis of idempotence models in Section 2.4 uses Pin [65] to study x86 programs, the compiler evaluation of Chapter 5 uses both Pin and gem5 [17] to study both x86 and ARM[2] programs, while the evaluation sections 6.1.3, 6.2.3, and 6.3.2 all use gem5 to simulate different microarchitectural features using ARM.

1.5 Relation to Author's Prior Work

Table 1.2 highlights the influence of selected publications by the author on the chapters of this dissertation. A publication appearing in PLDI 2012 [29] has influences across Chapters 3, 5, and 6 primarily influencing the static analysis presented in Chapter 3. Another recent publication on the application of idempotence for GPUs, appearing in ISCA 2012 [70], influences the GPU architecture design presented in Chapter 6. A publication on the application of idempotence for CPUs, appearing in MICRO 2011 [28], similarly influences the CPU architecture design presented in Chapter 6, although it also has some ties to the compiler evaluation of Chapter 5.

Compared to all previous work, this thesis explores in greater detail the motivation behind building architectures that leverage idempotence and evaluates against a more mature and robust

[1] 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 setup phase.

[2] For ARM, data for SPEC INT's gcc and SPEC FP's sphinx3 and povray are omitted since these benchmarks either would not cross-compile or would not run for the version of gem5 used.

Prior Work	**Topic**	**Chapters**
PLDI 2012 [29]	Static analysis and compiler design	3, 5, 6
ISCA 2012 [70]	Application to GPU architecture	6
MICRO 2011 [28]	Application to CPU architecture	5, 6

Table 1.2: The relation of the author's prior work to the dissertation material.

compiler implementation that balances the execution overheads associated with smaller idempotent regions against those potentially associated with larger regions. Namely, Chapter 2, Chapter 4, and parts of Chapter 5 are largely unique to this thesis and are not part of previously published work.

2 Idempotence in Computer Architecture

This chapter analyzes idempotence and idempotence-based recovery specifically in the context of application programs executed as sequences of instructions. It develops a framework for the analysis of idempotence in this context and develops a taxonomy to reason about a spectrum of *idempotence models*. It subsequently offers empirical and qualitative analysis to identify two specific models—*architectural* and *contextual* idempotence—that are deemed meaningful for exploration in subsequent chapters.

Parts of this chapter are heavy on formalism; with an understanding of certain specific characteristics of architectural and contextual idempotence, the impatient reader is free to skip this chapter and continue on to the remaining chapters of this dissertation. The relevant characteristics are as follows. Both models allow the construction of idempotent regions of maximal size with respect to the common case of data-race-free multi-threaded execution. Importantly, both models specifically assume *invariable control flow* semantics upon re-execution with respect to non-local memory state. Where the two models differ is in what they assume with respect to other (local) state: while architectural idempotence again assumes invariable control flow, contextual idempotence allows for variable control flow semantics.

The chapter is organized as follows. Section 2.1 presents the intuition behind taxonomy it develops, presenting example idempotence models over sequences of instructions. Section 2.2 then formally defines key terms and Section 2.3 presents the taxonomy, identifying three axes of variation within the taxonomy. A permutation of the points along these axes forms an idempotence model. Section 2.4 analyzes the space of idempotence models and then distills the space to two models, architectural and contextual idempotence, that are deemed most meaningful. Section 2.5 presents a summary and conclusions.

2.1 Idempotence by Example

An operation is considered idempotent if the effect of executing it multiple times is identical to executing it only a single time. The concept of idempotence has many existing uses in the field of computer science:

- In the design of **database systems**, SQL SELECT queries do not modify any visible state and hence are idempotent [22]. Many UPDATE queries are also idempotent if they do not first read the state that they are modifying. Such queries can be safely retried in the event of failure.

- In the design of **distributed file systems**, the NFS protocol is stateless by design [88]; the server does not maintain protocol information and the client keeps track of all information required to send requests to the server. As a result, most NFS requests can be idempotent, allowing an NFS client to send the same request one or more times without any harmful side effects.

- In the design of **instruction set architectures**, RISC-style load and store instructions are idempotent. This allows a memory instruction causing a memory exception to be simply re-executed after the exception is serviced. Several processors take advantage of this property to simplify support for virtual memory [9, 33]. Other types of processors that cannot do so face significant challenges in implementing full virtual memory support [72].

- In the design of **network protocols**, application-layer HTTP GET, PUT, and DELETE requests are all idempotent according to the HTTP protocol standard [104]. This allows these requests to be safely retried in the event of a network outage, greatly simplifying the infrastructure support of the internet.

In these examples, idempotence applies specifically to the *externally-visible* state of the system. Importantly, idempotence over internally-visible state need not be preserved. For instance, re-executing a SQL SELECT query may update some internal state of the database, such as a transaction log or statistics maintained by a query optimizer, in a way that is not preserved by re-execution. Similarly, re-executing an NFS file read or HTTP GET request may affect the routing and network processing state of switches used to deliver the request over the network. Finally, a load or store instruction causing an exception may invoke an operating system service routine that updates some system-internal state (e.g. page table entries).

From this discussion, it is evident that the power of idempotence applied over a system lies in part with how that system is defined. Considering the architecture underlying the execution of an application program as the system, there are multiple definitions, or *models*, of idempotence that are meaningful. This chapter develops a formal taxonomy to concretely reason about these different models as they emerge from assumptions about the architecture environment. The discussion below presents intuition by presenting example models.

Example Idempotence Models

As stated earlier, an operation is idempotent if the effect of executing it multiple times is the same as the effect of executing it only once. This property is achieved if the operation's inputs are preserved throughout its execution; with the same inputs, the operation will produce the same outputs each time it executes. However, what it means to "preserve an input" is subject to interpretation, and many different interpretations make sense depending on the context. This section presents four different example interpretations (models) that are all meaningful in the context of programs executed as sequences of instructions. A *region* is considered the unit of operation, and the following definitions are assumed:

Region: A *region* is defined as a collection of instructions uniquely identified by the single instruction that forms its entry point. A region contains the set of instructions reachable by control flow from its entry point up to its exit points.

Live-in: A variable is *live-in* to a region if the variable may hold a value that is (a) defined (written) before entry to the region and (b) potentially used (read) after entry to the region.

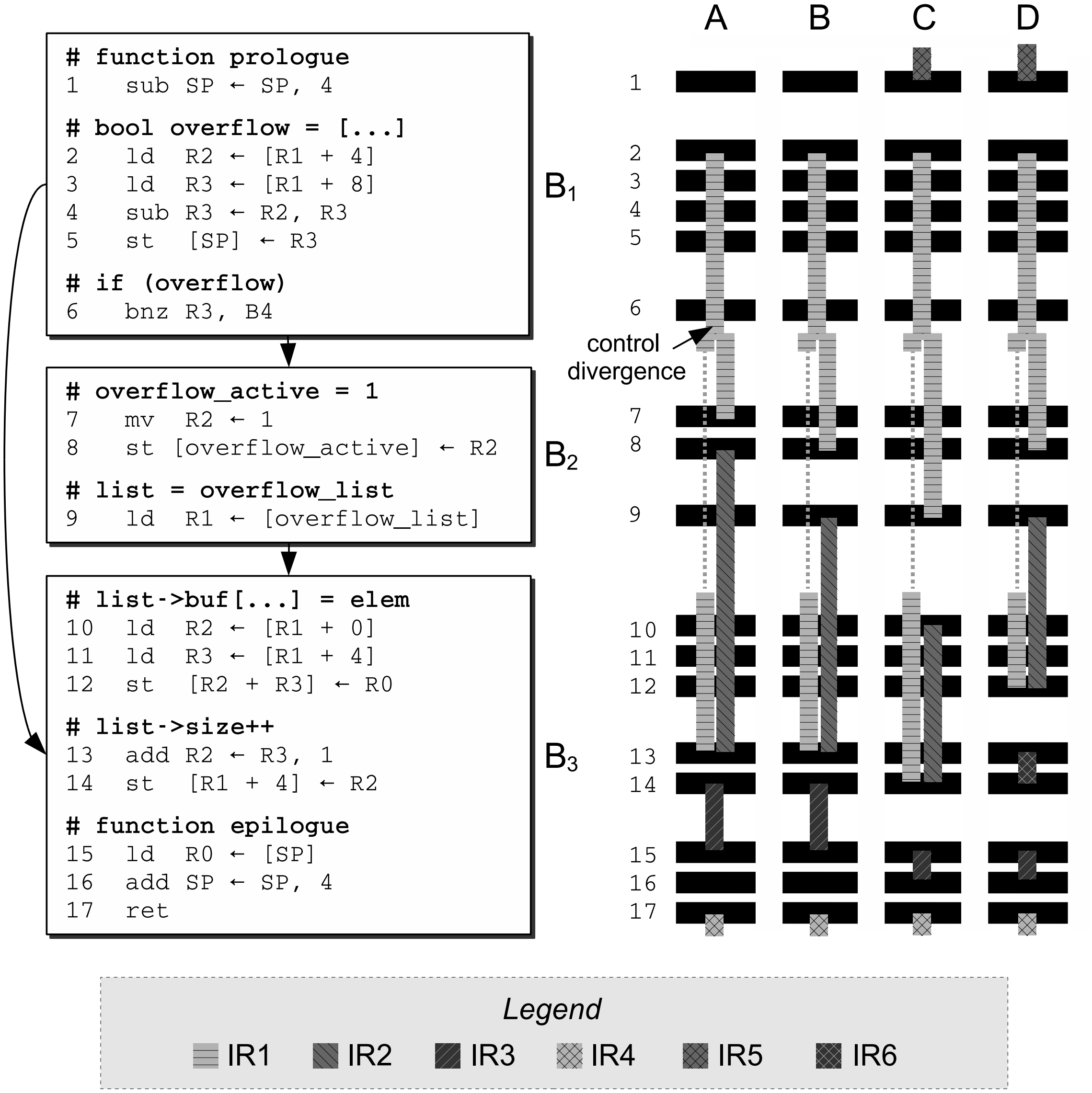
The code of the function shown in Figure 2.1, written in the C programming language, is used as an example of inherently non-idempotent code that can be divided into idempotent regions. The function, `list_push`, checks a list for overflow and then pushes an integer element onto the

```
typedef struct {
  int *buf;    // buffer
  int  size;   // num elements
  int  cap;    // capacity
} list_t;

list_t *overflow_list;
int     overflow_active;

int list_push(int     elem,
              list_t *list)
{
  // check for overflow
  int overflow =
      (list->size == list->cap);

  // if overflow use other list
  if (overflow) {
    overflow_active = 1;
    list = overflow_list;
  }

  // insert at end of list
  list->buf[list->size] = elem;
  list->size++;

  return overflow;
}
```

Figure 2.1: Example source code.

end of the list. The left side of Figure 2.2 shows the function compiled to a stylized assembly code organized into basic blocks, with arrows connecting the control flow between basic blocks. The code assumes four registers are available, R0-R3, with function arguments held in registers R0 and R1, and R0 also the return register.

In the discussion that follows, the effect of a given idempotence model is measured by forming the set of maximally-sized idempotent regions found by greedily scanning and incrementally adding instructions to a region until doing so would render the region non-idempotent, at which point a new idempotent region is formed starting at the next instruction that is itself idempotent. In practice, identifying idempotent regions—in particular, *semantically* idempotent regions—requires a more sophisticated analysis (see Chapter 3); this algorithm is assumed for illustration purposes only.



```
# function prologue
1   sub SP ← SP, 4

# bool overflow = [...]
2   ld  R2 ← [R1 + 4]
3   ld  R3 ← [R1 + 8]
4   sub R3 ← R2, R3
5   st  [SP] ← R3

# if (overflow)
6   bnz R3, B4
```

```
# overflow_active = 1
7   mv  R2 ← 1
8   st [overflow_active] ← R2

# list = overflow_list
9   ld  R1 ← [overflow_list]
```

```
# list->buf[...] = elem
10  ld  R2 ← [R1 + 0]
11  ld  R3 ← [R1 + 4]
12  st  [R2 + R3] ← R0

# list->size++
13  add R2 ← R3, 1
14  st  [R1 + 4] ← R2

# function epilogue
15  ld  R0 ← [SP]
16  add SP ← SP, 4
17  ret
```

Figure 2.2: The function of Figure 2.1 partitioned into idempotent regions.

For the example function, the effects of four different idempotence models are graphically illustrated using the labels A, B, C, and D on the right of Figure 2.2. In the figure, the black horizontal bars represent instructions and the vertical bars represent idempotent regions (labeled using the abbrevation "IR" in the legend). A region (vertical bar) overlaps an instruction (horizontal bar) if the region includes that instruction. In the case where two vertical bars overlap an instruction, the instruction is contained inside both regions, i.e. both regions contain the instruction but they have different entry instructions. The meaning of the four different idempotence models A, B, C, and D are presented below.

Model A

Definition 2.1 (Model A): *An input of a region is a variable that is live-in to the region. A region preserves an input if the input is not overwritten inside the region.*

The above model states simply that, for a region to be idempotent, a live-in variable may not be overwritten inside the region. This model produces the set of idempotent regions illustrated in Figure 2.2 under the label A. There are three regions fully contained inside the function.

The first region (IR1) spans instructions 2-7 and 10-13. Its control flow diverges at instruction 6 and it has two exit points. Its first exit point occurs immediately before instruction 8, which overwrites the memory location of the memory variable `overflow_active`. This variable is conservatively assumed live-in because it may be read after returning from the function (in which case the value read will be the value defined before entry to the function if basic block B_2 is not executed). Its second exit point occurs immediately before instruction 14, which overwrites the live-in memory location `R1 + 4` indexing into `list->buf`. The second region (IR2) spans instructions 8-13, also stopping before instruction 14 after re-convergence. Finally, the third region (IR3) spans instructions 14-15, stopping before instruction 16, which overwrites the live-in stack pointer register `SP`. A fourth region (IR4), only partially shown, begins at the return point of the function.

Model B

Model A makes some simplifying assumptions, both optimistic and pessimistic. One pessimistic assumption is that it assumes it is unsafe to overwrite the live-in memory location of the variable `overflow_active` at instruction 8. However, this memory location is *dynamically dead* at the point where it is written; if B_2 is entered, and assuming control flow does not vary upon re-execution, then the value in that memory location at the beginning of the function will never be read after entry to the function. If we can safely assume control flow will not vary when we re-execute, then the following definition more accurately captures idempotence constraints (differences in bold):

Definition 2.2 (Model B): *An input of a region is a variable that is live-in to the region. A region preserves an input if the input is not overwritten* **after it is read** *inside the region.*

If an input is written *before* it is read, then the input must be dynamically dead and thus overwriting is safe. Only in the case where an input has been read is it dynamically live. Hence, overwriting a live-in *after it is read* is the case that must be dis-allowed. Figure 2.2 shows the changes resulting from this model under the label B. The region that previously spanned instructions 2-7 and 10-13 now is allowed to span instruction 8 (IR1). However, it must stop before instruction 9, which overwrites the live-in register `R1` that is overwritten after it is read (in basic block B_1). All other regions are unchanged.

Model C

A second pessimistic assumption made in Model A is that the stack pointer adjustments of instructions 1 and 15 cannot be included inside any idempotent region because they overwrite the live-in register `SP` which they themselves also read. However, allowing such overwriting instructions to be included *at the end* of an idempotent region still renders any partial execution of the region idempotent, while affording certain conceptual and practical benefits. First, conceptually it allows instructions such as the `SP`-update instruction to both figuratively and logically "terminate" an idempotent region, and in the process it allows a function to be *fully* decomposed into idempotent regions. Second, for recovery, containing only the state written by the final instruction of a region often entails certain implementation conveniences. At most, it requires buffering (or otherwise duplicating) only one state element at the ordering point between regions. Hence, Model C is developed to allow for this specific exception:

Definition 2.3 (Model C): *An input of a region is a variable that is live-in to the region. A region preserves an input if the input is not overwritten after it is read inside the region* **by an instruction that is not a final instruction of the region**.

Figure 2.2 shows the regions constructed using this new definition under the label C. Instruction 1 is absorbed into the region that calls into the function (IR5), instruction 9 is absorbed into IR1, instruction 14 is absorbed into both IR1 and IR2, and instruction 16 is absorbed into IR3. All instructions are accounted for by some idempotent region.

Model D

Models A, B, and C optimistically ignore the impact of shared memory interactions between threads. Such interactions can affect whether or not a region is idempotent. For instance, a region may write to a memory location and a concurrently running region may read the memory location and subsequently modify it. Upon re-execution of the first region, the memory location will be written again to a now *incorrect* value, producing a side-effect that renders the region's execution non-idempotent. Model D captures this and similar effects relating to shared memory interactions:

Definition 2.4 (Model D): *An input of a region is a variable that* **(1)** *is live-in to the region* **or (2) while the region is executing, may be read in any concurrently executing region**. *A region preserves an input if the input is not overwritten after it is read inside the region by an instruction that is not a final instruction of the region.*

This model effectively disallows stores to any potentially shared memory locations inside an idempotent region (excepting the last instruction; see Model C). Re-executing such a store may cause inconsistent program effects as described above.

Figure 2.2 illustrates the impact of this new model on the construction of idempotent regions under the label D. The three shared memory stores in the example are instructions 8, 12, and 14. The other store, instruction 5 stores to the program stack, which is considered private to the running thread. IR1, which previously spanned instructions 2-9 and 10-14, now ends early to span only 2-8 and 10-12. IR2 grows backwards to include instruction 9. Finally, a new region (IR6) forms that spans instructions 13-14.

Summary and Implications

The remainder of this chapter presents a formal classification system (a taxonomy) to concretely reason about idempotence models such as Models A-D and others. Section 2.2 defines key terms and Section 2.3 identifies three different axes: the *control* axis, the *sequencing* axis, and the *isolation*

Model	**Control**	**Sequencing**	**Isolation**
A	All-variable	Free	Full
B	Invariable	Free	Full
C	Invariable	Commit	Full
D	Invariable	Commit	Private

Table 2.1: Idempotence configurations for Models A-D.

axis. The axis settings combine to form an idempotence model. As a preview, the axis settings for Models A-D are shown in Table 2.1.

Different idempotence models are more realistic or more desirable depending on the architecture environment. Section 2.4 presents an empirical analysis of the model space, presents findings, and concludes by identifying the two idempotence models that are carried through the rest of the dissertation.

2.2 Terms, Definitions, and Axioms

This section defines foundational terms and states the axioms that underlie the development of the taxonomy in Section 2.3. The architecture and compiler implications of this taxonomy on real systems are discussed and evaluated in Section 2.4.

Region: A *region* is as defined in Section 2.1.

Path: A *path* through a region is a sequence of instructions connected by control flow starting at the entry point of a region and ending at or before one of the region's exit points.

Live-in: A variable is *live-in* to a *region* as defined in Section 2.1. A variable is analogously *live-in* to a *path* if the same is true considering only the control flow executed along that path.

Live-out: A variable is *live-out* to a *region* if it may hold a value that is (a) defined (written) before exiting the region, and (b) potentially used (read) after exiting the region. A variable is analogously *live-out* to a *path* if the above is true considering only the control flow executed along that path.

State element: A *state element* is an architectural storage element such as a register or memory location.

Idempotence: Formally, let a program P consist of a set of regions $\{P_1, \cdots, P_n\}$ and for each region $P_i \in P$, let $P_i = \{X_{i_1}, \cdots, X_{i_m}\}$ consist of the set of all unique pairs $X_{i_j} = (I_{i_j}, O_{i_j})$ that each represent a set of possible paths through P_i, where I_{i_j} is that set of "input" state elements along those paths, and O_{i_j} the set of written "output" state elements along those paths (the precise contents of sets I_{i_j} and O_{i_j} depend on the idempotence model, which is the topic of Section 2.3). A region P_i is *idempotent* if:

$$\bigcup_{X_{i_j} \in P_i} I_{i_j} \cap O_{i_j} = \emptyset \tag{2.1}$$

By keeping the input and output sets distinct across all possible executions of itself, the region P_i is guaranteed to write the same output values to the same output state elements every time it executes. Thus, it will achieve the same effect each time it executes; hence, it will be idempotent.

Importantly, the set of input state elements I_{i_j} (for the paths represented by X_{i_j}) is used instead of the set of input state elements for the region P_i. These two sets are not the same, and the former yields a more precise formulation than the latter, as Lemma 2.2 shows.

Execution failure: An *execution failure* is an unexpected event in the program's execution that interrupts the normal program execution flow and requires some corrective action before execution can resume (i.e. an exception, hardware fault, mis-speculation, etc.) An execution failure may have side-effects associated with one or more specific instructions such that the *values* written by those instructions are incorrect, but that the target state elements (the *destinations*) of those values remain correct. For instructions that write specifically to the program counter (PC), any resulting change in control flow, while possibly incorrect, must still follow the static control flow edges of the program.

With this definition, implicitly (1) a register-writing instruction always writes to the correct destination register, (2) a store commits its values to memory only if the address of that store is known to be correct, and (3) incorrect control flow may result only from the consumption of incorrect values or from incorrectly made control flow decisions. This definition furthermore intentionally excludes failures with side-effects that are, from the viewpoint of the program, spontaneous and uncorrelated with any instruction. Such failures include spontaneous register or memory corruptions (for example, due to a particle strike), faults in the cache coherence logic, or faults in the cache writeback logic. Additionally, control logic hardware faults and datapath faults that result in corrupted memory address lines or corrupted register destination fields are intentionally not covered.

Idempotence under execution failures: In the presence of potential side-effects due to execution failures, Equation 2.1 is insufficiently general: a region may generate incorrect values in its output state elements, and incorrect control flow may cause writes to incorrect output state elements as well.

Incorporating execution failures into the earlier definition of idempotence, a region P_i is idempotent if, for any execution with failures, that execution can be succeeded by a complete and failure-free execution such that the overall effect (with respect to the idempotence model) is as if only the final complete and failure-free execution occurred.

Given this context, we redefine X_{i_j} as a unique triple $X_{i_j} = (I_{i_j}, O_{i_j}, E_{i_j})$, where I_{i_j} and O_{i_j} are the set of input and output state elements for some set of failure-free path executions and $E_{i_j} \supseteq O_{i_j}$ is the set of all state elements that may be written to executing those paths in the presence of potential failures. P_i is *idempotent in the presence of execution failures* if:

$$\bigcup_{X_{i_j} \in P_i} I_{i_j} \cap E_{i_j} = \emptyset \tag{2.2}$$

(Although not used in this equation, O_{i_j} has applications in future sections.)

Recovery using idempotence: Using idempotence, recovery from an execution failure is by repeated re-execution until a failure-free execution occurs. Detection capability is assumed, and the detection of an execution failure is assumed to occur inside the same idempotent region as where the execution failure occurred. This is necessary because if failure occurrence and detection span multiple regions (that do not collectively also form an idempotent region), recovery by re-execution is not possible in general: a later region will eventually overwrite some input state element of the first region (otherwise they would form a larger idempotent region), and if the detection occurs in the later region, it is assumed unknowable whether the input element has yet been overwritten or not. Hence, having the effects of an execution failure span multiple idempotent regions is conservatively disallowed.

Examples, Lemmas, and Remarks

Before proceeding, we briefly present same examples, lemmas, and remarks regarding the above definitions. The reader can freely skip to the taxonomy presented in the next section if desired without loss of context.

Figure 2.3 shows the control flow graph and some psuedocode for a simple example region. For idempotence in the presence of execution failures, Example 2.1 uses Equation 2.2 to identify the idempotence of this region. Additionally, Lemmas 2.1 and 2.2 demonstrate two properties of Equation 2.2: Lemma 2.1 demonstrates that Equation 2.1 is a special case of Equation 2.2 where either no execution failures can occur or execution failures have no side-effects; and Lemma 2.2 shows that defining the set I_{i_j} as the input state elements of X_{i_j} is more precise than simply using the set of input state elements of region P_i. Finally, Remark 2.1 distills the impact of Lemma 2.2 on the idempotence potential for the example region of Figure 2.3.

Example 2.1. Let $P_x = \{X_{x_0}, X_{x_1}\}$ represent the region shown in Figure 2.3, with X_{x_0} representing the execution path exiting on the left and X_{x_1} representing the execution path exiting on the right. For $X_{i_j} = (I_{i_j}, O_{i_j}, E_{i_j})$, let I_{x_j} be defined as the set of state elements live-in to X_{x_j} and O_{x_j} the set of state elements that are written in X_{x_j}, and let E_{x_j} be the set of state elements potentially written in X_{x_j} allowing failures. P_x is idempotent in the presence of execution failures using Equation 2.2 as follows:

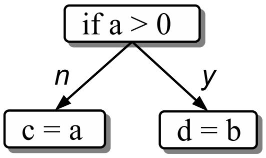


Figure 2.3: An example region with two control flow paths.

$$
\begin{array}{lll}
I_{x_0} = \{a\} & O_{x_0} = \{c\} & E_{x_0} = \{c, d\} \\
I_{x_1} = \{a, b\} & O_{x_1} = \{d\} & E_{x_1} = \{c, d\}
\end{array}
$$

$$
\begin{array}{ll}
\bigcup_{X_{i_j} \in P_x} I_{x_j} \cap E_{x_j} & = \emptyset \\
(I_{x_0} \cap E_{x_0}) \cup (I_{x_1} \cap E_{x_1}) & = \emptyset \\
(\{a\} \cap \{c, d\}) \cup (\{a, b\} \cap \{c, d\}) & = \emptyset \\
\emptyset \cup \emptyset & = \emptyset
\end{array}
$$

Lemma 2.1. *Equation 2.1 is a special case of Equation 2.2 where either no execution failures can occur or execution failures have no side-effects.*

Proof. In the case of no execution failures or failures with no side-effects, $E_{i_j} = O_{i_j}$. Substituting O_{i_j} for E_{i_j} in Equation 2.2 trivially reduces to Equation 2.1.

□

Lemma 2.2. *For the state that must not be overwritten in $X_{i_j} \in P_i$ to preserve the idempotence of P_i, the set of input elements I_{i_j} for X_{i_j} is is more precise than the set of input elements, L_i for the region P_i. That is, the former and the latter are not equivalent and the former is a subset of the latter, i.e. the following conditions are both true:*

- ***Condition 1:*** *There exists a P_i and an $X_{i_j} \in P_i$ such that $I_{i_j} \neq L_i$.*
- ***Condition 2:*** *For all P_i and all $X_{i_j} \in P_i$, $I_{i_j} \subseteq L_i$.*

Proof. Let $P_x = \{X_{x_0}, X_{x_1}\}$ represent the region shown in Figure 2.3 with X_{x_0} and X_{x_1} defined as in Example 2.1. Without loss of generality, let the input sets I_{i_j} and L_i be defined by the live-in

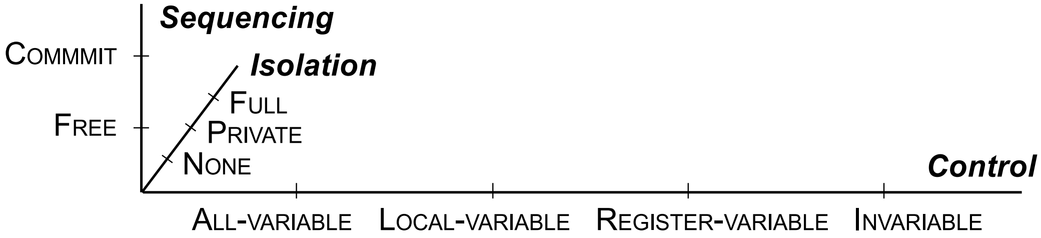


Figure 2.4: Axes in the idempotence taxonomy.

elements corresponding with those sets. Assume Condition 1 is not true and hence $I_{x_j} = L_x$ for all $X_{x_j} \in P_x$. This leads to a contradiction:

$$I_{x_0} = \{a\} \quad L_x = \{a, b\}$$

$$\begin{aligned} I_{x_0} &= L_x \\ \{a\} &= \{a, b\} \end{aligned}$$

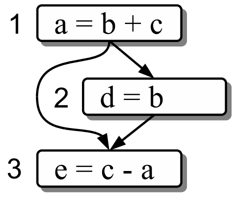
Hence, Condition 1 is true. Regarding Condition 2, the live-in set L_i of P_i is defined as the union of all I_{i_j} for $X_{i_j} \in P_i$. Hence, I_{i_j} is necessarily contained in L_i and thus Condition 2 is also true. □

Remark 2.1. It follows from Lemma 2.2 that the left-hand path of the region shown in Figure 2.3 may overwrite the variable b while maintaining the idempotence of the region. If preservation of the input state for the entire region were instead used as the basis for idempotence, this would not be the case.

2.3 A Taxonomy of Idempotence

The definitions of idempotence from the previous section, with and without failures, intentionally leave some things unspecified. In particular, what does it mean for an state element to be an "input" or an "output"? This section develops a taxonomy of idempotence that provides a spectrum of answers to this question. Illustrated in Figure 2.4, it proposes three "axes" of variation:

Control axis: The control axis (X axis) specifies how an execution failure manifests with respect to the control flow of a region.



Can control flow vary on re-execution?

No: Operation is idempotent.
d is trivially dead at start of basic block 2.

Yes: Operation is not idempotent.
d is potentially live at start of basic block 2.

Figure 2.5: The impact of variable control flow on idempotence.

Sequencing axis: The sequencing axis (Y axis) specifies how an execution failure manifests with respect to the sequencing of instructions in a region.

Isolation axis: The isolation axis (Z axis) specifies a region's isolation with respect to other regions runnning in a multi-threaded environment.

The next three subsections develop the three axes of the taxonomy.

2.3.1 The Control Axis

In the presence of execution failures, a region can experience potentially incorrect control flow. Figure 2.5 illustrates how incorrect control flow can make an otherwise idempotent region non-idempotent. In the case of the region shown, incorrect control flow can cause an execution intended to pass through basic blocks 1 and 3 to instead incorrectly pass through 2, in the process overwriting the state element d. If the original value held in d is needed for the remainder of the program to execute correctly, this overwriting will likely manifest as an error in the program. (Examples 2.2 and 2.3 at the end of this section use the notation of Section 2.2 to demonstrate how this region is idempotent without execution failures and is not idempotent with execution failures.)

Although the region shown in Figure 2.5 is not idempotent in the most general case, it can be idempotent if the write to element d generated through incorrect control flow is *contained*. For the purposes of this section, a write to particular state element is considered contained if it only becomes visible under correct control flow, with the degree of containment potentially varying depending on the type of state element.

Table 2.2 lists three different categories of state elements and the symbols used to represent them: R represents the set of all register elements, L represents the set of all function-local stack

State Elements	**Symbol**
All registers	R
All local memory (function-local stack memory)	L
All non-local memory (heap, global, and non-local stack memory)	N

Table 2.2: Spatial containment state element categories and their symbols.

Control Axis Setting	**Constraints**
All-variable	*None*
Local-variable	$E_{i_{j_N}} = O_{i_{j_N}}$
Register-variable	$E_{i_{j_L}} = O_{i_{j_L}}$ and $E_{i_{j_N}} = O_{i_{j_N}}$
Invariable	$E_{i_{j_R}} = O_{i_{j_R}}$, $E_{i_{j_L}} = O_{i_{j_L}}$, and $E_{i_{j_N}} = O_{i_{j_N}}$

Table 2.3: Output constraints for the different control axis settings.

memory elements, and N represents the set of all other types of memory elements. The output state elements of a region are comprised of elements belonging to these three different categories. For a collection of paths represented by $X_{i_j} = (I_{i_j}, O_{i_j}, E_{i_j})$ we denote this for the sets O_{i_j} and E_{i_j} as $O_{i_j} = O_{i_{j_R}} \cup O_{i_{j_L}} \cup O_{i_{j_N}}$ and $E_{i_j} = E_{i_{j_R}} \cup E_{i_{j_L}} \cup E_{i_{j_N}}$.

The *control axis setting* of X_{i_j} constrains the set of all potential output elements E_{i_j} as presented Table 2.3 and as discussed below:

All-variable: Under setting All-variable, control flow is potentially variable with respect to all state elements. Hence, there is *no containment* to account for variable control flow; all state elements can be written to as a result of incorrect control flow. With this control axis setting, the region from Figure 2.5 is not idempotent.

Local-variable: Under setting Local-variable, control flow is potentially variable with respect to function-local state elements only. Writes to non-local memory state elements (heap, global and non-local stack memory) are withheld using a mechanism such as a store buffer or by simply stalling execution of such a write until control flow is verified. With this control axis setting, the region from Figure 2.5 is idempotent if $d \in N$ and not idempotent otherwise.

Register-variable: Under setting Register-variable, control flow is potentially variable with respect to register state elements only. Writes to memory state elements (both local and non-local) are contained and hence do not become visible if they are generated through incorrect control flow. Again, these writes can be withheld using a store buffer or by stalling execution. With memory spatial containment, the region from Figure 2.5 is idempotent if $d \in (L \cup N)$ and not idempotent otherwise.

Invariable: Under setting Invariable, control flow is always effectively invariable. All writes to all architectural state elements are contained and do not become visible if they are generated through incorrect control flow. All such writes are withheld using, for example, a re-order buffer combined with a store buffer or by again simply stalling execution of the program in the presence of uncertain control flow. With full spatial containment, the region from Figure 2.5 is always idempotent.

Examples

Below we present some examples which the reader may skip as desired.

Example 2.2. Let P_x represent the region shown in Figure 2.5. Using Equation 2.2, P_x is idempotent assuming no execution failures as follows:

Let P_x consist of its two pairings $\{X_{x_0}, X_{x_1}\}$ with $X_{x_0} = (I_{x_0}, O_{x_0})$ representing the path that includes basic block 2 and $X_{x_1} = (I_{x_1}, O_{x_1})$ the one that does not. Let I_{x_j} be defined as the set of state elements live-in to X_{x_j} and O_{x_j} the set of state elements that are written in X_{x_j} (both I_{x_j} and O_{x_j} are expanded in future sections). For $X_{x_j} \in P_x$, the sets I_{x_j} and O_{x_j} are:

$$\begin{aligned} I_{x_0} &= \{b, c\} \quad O_{x_0} = \{a, d, e\} \\ I_{x_1} &= \{b, c\} \quad O_{x_1} = \{a, e\} \end{aligned}$$

For these sets, $I_{x_0} \cap O_{x_0} = \emptyset$ and $I_{x_1} \cap O_{x_1} = \emptyset$. Hence, $\cup_{(I_{x_j}, O_{x_j}) \in P_x} I_{x_j} \cap O_{x_j} = \emptyset$. Thus, region P_x is idempotent.

Example 2.3. Let P_x again represent the region shown in Figure 2.5. Using Equation 2.2, P_x is *not* idempotent in the presence of execution failures as follows:

With execution failures, let P_x again be defined as $P_x = \{X_{x_0}, X_{x_1}\}$, with X_{x_0} again representing the path that includes basic block 2 and X_{x_1} the one that does not. Given $X_{x_j} = (I_{x_j}, O_{x_j}, E_{x_j})$, the sets I_{x_j} and O_{x_j} are as defined above for the case with no execution failures and E_{x_j} is the set of state elements potentially written in X_{x_j} in the presence of execution failures. For $X_{x_j} \in P_x$ the sets I_{x_j}, O_{x_j}, and E_{x_j} are thus:

$$\begin{array}{lll} I_{x_0} = \{b, c\} & O_{x_0} = \{a, d, e\} & E_{x_0} = \{a, d, e\} \\ I_{x_1} = \{b, c, d\} & O_{x_1} = \{a, e\} & E_{x_1} = \{a, d, e\} \end{array}$$

Equation 2.2 unravels as follows:

$$\begin{array}{lll} I_{x_0} \cap E_{x_0} & = (\{b, c\} \cap \{a, d, e\} & = \emptyset \\ I_{x_1} \cap E_{x_1} & = (\{b, c, d\} \cap \{a, d, e\} & = \{d\} \end{array}$$

$$\textstyle\bigcup_{X_{i_j} \in P_x} I_{x_j} \cap E_{x_j} = \{d\}$$

Hence, P_x is not idempotent in the presence of execution failures.

2.3.2 The Sequencing Axis

Certain types of instructions are inherently non-idempotent. These include certain memory-mapped I/O instructions, some types of synchronization instructions (e.g. atomic increment), and instructions associated with the calling convention (e.g. stack pointer increment or decrement). Additionally, under certain thread-isolation assumptions (see next sub-section), any store instruction may overwrite an input to a concurrently executing region, and hence any region that contains a store must be considered non-idempotent. While the ability to fully partition programs into a collection of idempotent regions can be beneficial, with idempotence defined as in Equation 2.2 this is not generally possible as in the case of the examples mentioned.

Varying the *sequencing axis* allows this limitation to be overcome. In particular, the COMMIT setting allows program to be fully partitioned into idempotent regions by guaranteeing that, if the *final* instruction of a region executes successfully to completion, all instructions before it have also completed successfully and cannot experience an execution failure. Assuming instructions do not update any state if they experience an execution failure, this allows the final instruction to overwrite input elements of that region. Following from the definition of recovery from Section 2.2, an idempotent region must already wait for all execution failures to be detected before a subsequent region can begin execution and hence from an implementation standpoint Commit sequencing creating a natural ordering point on the final instruction that complements recovery requirements. Commit sequencing also allows programs to be fully decomposed into a collection of idempotent regions—at the most basic level each individual instruction is then itself an idempotent region. The Free setting, in contrast, does not place any constraints on the order of completion for the instructions of a region.

In the context of Equation 2.2, the effects of Commit sequencing can be accounted for by removing the set of state elements written by the final instruction of a region from that region's set of potential output elements *if* those elements are not also written by other instructions in the region. That is, for a collection of paths $X_{i_j} = (I_{i_j}, O_{i_j}, E_{i_j})$ let Z_{i_j} be the set of state elements potentially written in X_{i_j} (as in the definition of E_{i_j} assumed thus far), and let $Z_{i_{j_S}} \subseteq Z_{i_j}$ be the set of state elements contained by the sequencing axis setting in $X_{i,j}$. The sequencing axis affects the set of potential output state elements E_{i_j} as follows:

$$E_{i_j} = Z_{i_j} \setminus Z_{i_{j_S}} \tag{2.3}$$

Similarly, for O_{i_j} (the outputs under a failure-free execution of X_{i_j}), let W_{i_j} be the set of state elements written under a failure-free execution of X_{i_j} (as in the definition of O_{i_j} assumed thus far), and let $W_{i_{j_S}} \subseteq W_{i_j}$ be the set of state elements contained by the sequncing axis setting in $X_{i,j}$. The set O_{i_j} is re-defined as follows:

$$O_{i_j} = W_{i_j} \setminus W_{i_{j_S}} \tag{2.4}$$

Let $Z_{i_{j_F}} \subseteq Z_{i_j}$ be the set of output state elements for any final instruction in X_{i_j} and let $Z_{i_{j_O}} \subseteq Z_{i_j}$ be the set of potential output state elements for all other instructions in X_{i_j} in the presence of failures. Similarly, let $W_{i_{j_F}} \subseteq Z_{i_{j_F}}$ and $W_{i_{j_O}} \subseteq Z_{i_{j_F}}$ be the subsets that apply for a failure-free execution of X_{i_j}. By definition, $Z_{i_j} = Z_{i_{j_F}} \cup Z_{i_{j_O}}$ and $W_{i_j} = W_{i_{j_F}} \cup W_{i_{j_O}}$. Then, the sets $Z_{i_{j_S}}$ and $W_{i_{j_S}}$ for each of the two sequencing axis settings are defined as in Table 2.4 and as discussed below:

Sequencing Axis Setting	**Constraints**
Free	$Z_{i_{j_S}} = \emptyset$ and $W_{i_{j_S}} = \emptyset$
Commit	$Z_{i_{j_S}} = Z_{i_{j_F}} \setminus Z_{i_{j_O}}$ and $W_{i_{j_S}} = W_{i_{j_F}} \setminus W_{i_{j_O}}$

Table 2.4: Output constraints for the two sequencing axis settings.

Free: Under setting Free, the set of output state elements is simply the set of state elements written in the region.

Commit: Under setting Commit, the set of output state elements is the same as under Free excluding the output state elements of any final instruction that are not also output elements of other instructions in the region.

2.3.3 The Isolation Axis

The *isolation axis* affects whether or not the region is idempotent in the presence of data races. It can affect idempotence in the presence of both execution failures that do and do not have side-effects. With side-effects, a store from a failed execution can write a corrupted value that propogates to a concurrently running thread before it can be corrected by re-execution. Even without side-effects, a concurrently running thread may read a memory value written by a failed execution and subsequently modify the memory location before re-execution can occur. Then, upon re-execution, the memory location is written again back to the initially-written, now incorrect value. Both behaviors result in erroneous program execution.

In the context of Equation 2.2, the effects of thread isolation can be accounted for by considering a value that is written by a region as also read by the same region if it may be read by a concurrently executing thread. That is, for an execution $X_{i_j} = (I_{i_j}, O_{i_j}, E_{i_j})$ let L_{i_j} be the set of state elements live-in to X_{i_j} (as in the definition of I_{i_j} previously), and let C_{i_j} be the set of state elements that may be read by some region execution $X_{k,l} \in P_k$ for $P_k \in P$ that may execute concurrently with X_{i_j}. The isolation axis incorporates C_{i_j} to refine the definition for the set of input state elements I_{i_j} as follows:

$$I_{i_j} = L_{i_j} \cup C_{i_j} \tag{2.5}$$

Isolation Axis Setting	**Constraint**
None	$C_{i_j} = P \cup Z$
Private	$C_{i_j} = P$
Full	$C_{i_j} = \emptyset$

Table 2.5: Inter-thread visible state elements under different isolation axis settings.

Let P represent the set of thread-private memory state elements (thread-local storage and function-local stack) and Z represent all other types of memory state elements. The set C_{i_j} for each of three different levels of isolation is defined as in Table 2.5 and as discussed below:

None: Under setting None, all memory locations, even those considered thread-private, are potentially visible and modifiable by all threads. Even though sharing thread-private state generally gives undefined behavior, some programming languages provide *de facto* semantics for it. To accommodate these semantics, it must be assumed that a potential re-execution scenario exists such that a write to any memory location can "overwrite an input" of a concurrently executing region. Hence, any region with a memory write is non-idempotent.

Private: Under setting Private, thread-private memory can be safely considered isolated from other threads and hence a region may still be idempotent if it writes to thread-private memory. However, writes to potentially shared heap, global, or other memory may still cause undefined behavior.

Full: Under setting Full, all memory interactions inside a region are fully isolated. This applies to all single-threaded programs and properly synchronized programs where all regions can be serialized with respect to their shared memory interactions (i.e. there are no data races). It also applies to transactional systems where a region can execute as an atomic memory transaction [48] in order to prevent the visibility of intermediate state.

2.4 Measurement, Analysis, and Synthesis

The taxonomy developed in the previous section proposes three axes of idempotence with anywhere from two to four points on each axis for a total of twenty-four different combinations, or "idempotence models". In practice, however, only a few idempotence models are meaningful. In this section and the next, the model space is analyzed and pruned to those models that are explored in later chapters.

2.4.1 Measurement and Analysis: Model Characterization

The idempotence model primarily dictates two things: (1) achievable idempotent region sizes, and (2) the hardware and/or software support required to enable the model. In our analysis, we focus on the region size characteristic specifically and consider support mechanisms secondarily.

Trivially, idempotent regions can be as small as individual instructions. However, such regions are typically too small to be useful. In general, larger regions—in particular, longer *paths* executing though those region—are better because they allow for the greatest flexibility: when shorter path lengths are desirable, for instance, to minimize re-execution costs for recovery, paths can often be sub-divided into smaller paths as needed (more details in Chapter 4). The reverse, however, is typically not possible. This section presents empirical measurements and analysis on achievable idempotent path lengths for different idempotence models.

Experimental Method

We consider three benchmark suites: SPEC 2006 [99], a suite targeted at conventional single-threaded workloads, PARSEC [16], a suite targeted at emerging multi-threaded workloads, and Parboil [100], a suite targeted at massively parallel GPU-style workloads written for CPUs. All benchmarks are compiled using the LLVM with all optimizations enabled [62].

For each benchmark, we use the Pin dynamic instrumentation tool for x86-64 [65] to measure the idempotent path lengths that form at run-time inside individual functions. We monitor benchmark execution over a 10 billion instruction period starting after the initial 10 billion instructions of the application. For benchmarks with fewer than 20 billion instructions, the entire execution is monitored following the setup phase of the benchmark.

For analysis, we consider four representative idempotence models, three of which are fundamentally constrained by a single axis setting, and the fourth of which is unconstrained. For all models

Configuration	Control	Sequencing	Isolation	Region Formation
Store-Constrained	All-variable	Free	None	Ideal
Share-Constrained	All-variable	Free	Private	Ideal
Control-Constrained	All-variable	Free	Full	Ideal
Unconstrained	Invariable	Free	Full	Ideal
Obliviously-Formed	Invariable	Free	Full	Oblivious

Table 2.6: Experimental configurations, their idempotence axis settings, and assumptions.

we measure the largest possible path lengths achievable using perfect run-time information (more details below). For the unconstrained idempotence model we also consider path lengths formed compiling oblivious to the idempotence property (i.e. as might be generated by a conventional compiler) for comparison. In total we consider the five different experimental configurations shown in Table 2.6. From most constrained to least constrained, these configurations, and how idempotent path length is measured for each configuration, are explained as follows:

Store-Constrained: Independent of the Sequencing axis setting, which affects idempotent path lengths in a predictable fashion, this configuration is the most constrained configuration possible with control axis setting All-variable and isolation axis setting None. For this configuration, we measure idempotent path length simply as the distance between stores to memory. This measurement is ideal with the assumption that the region formation is able to avoid overwriting register live-ins by careful resource allocation and allocating temporary stack storage as necessary using techniques such as those described in Chapters 3 and 4.

Share-Constrained: This configuration raises the isolation axis setting from None to Private. Path lengths with any isolation setting other than Full are fundamentally limited by the occurrence of stores, regardless of whether those stores are encountered considering variable or invariable control flow; hence, the control axis setting is held at All-variable without interference. For this configuration we measure idempotent path length as the distance between stores to *non-local* memory, with local memory locations identified as those memory locations that are read or written relative to the stack pointer (these are universally register spill locations for the compiler we use).

Control-Constrained: This configuration further raises the isolation axis setting from Private to Full so that path lengths become constrained by control flow effects. Now, we measure idempotent path length as the distance between stores to non-local memory locations that are *live-in to the region*. The region distinction is important, since stores to non-local memory that do not overwrite live-ins of the active path may still overwrite live-ins of other paths that share the same region entry point. To measure the impact of this difference, a write to non-local memory is assumed to terminate the idempotence of the active path if that write follows a control decision since the start of the path. The existence of this control decision is used as a proxy for control divergence. It is slightly pessimistic since it assumes that the same non-local memory location is not also written along other paths through the containing region, either due to re-convergence or other behavior (in these cases the value would be known to be dead). This limitation is acknowledged, but deemed insignificant.

Unconstrained: This configuration assumes the most flexible control, sequencing, and isolation axis settings. As mentioned, a compiler is capable of avoiding overwritten local memory and register live-in state by allocating additional temporary stack storage. Hence, for the Local-variable, Register-variable, and Invariable control axis settings there are no differences in the ideal achievable path length measurement for any of these settings. (Differences in overhead are evaluated post-implementation in Chapter 5.) Hence, control axis setting Invariable is a representative setting for all control axis settings above All-variable. For this configuration, idempotent path length is measured as the number of instructions between overwriting of non-local memory locations that are live-in along the active path.

Obliviously-Formed: For this configuration, idempotent path length is measured as the length of the instruction sequence between overwriting of registers and memory live-in along the active path. In other words, these are the idempotent path lengths that exist in an actual program binary that is compiled oblivious to the idempotent path lengths achievable employing compiler renaming and allocation techniques.

The impact of the sequencing axis, which is not varied along any of the experimental configurations, is evaluated simply by measuring the percentage of instructions that are inherently

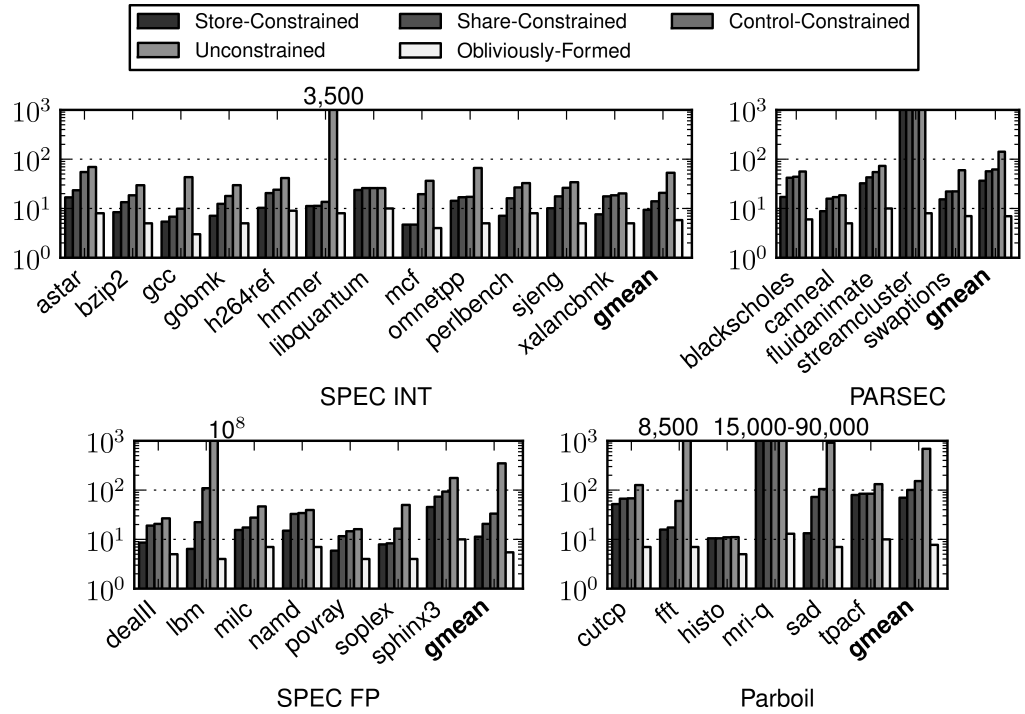


Figure 2.6: Mean idempotent path lengths.

non-idempotent during execution. These instructions overwrite a source operand that is neither a general-purpose register nor a local (stack) memory location (i.e. the conflict is unlikely to be under the control of the compiler). Under Commit sequencing, such instructions terminate the active idempotent path and a new idempotent path starts at the next instruction (i.e. the idempotent path length is simply shortened by one instruction).

Results

Figure 2.6 shows the mean idempotent path length for the five different experimental configurations with geometric mean values presented in Table 2.7.

Observation 2.1. *There is significant opportunity for constructing large idempotent regions by compiling specifically for idempotence.*

This first observation follows from comparing the overall geometric mean values on the far right column of Table 2.7. At a minimum, model Store-Constrained achieves more than 3x longer

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
Store-Constrained	8.4	10.3	35.5	69.0	**17.1**
Share-Constrained	13.0	19.6	55.8	99.6	**27.4**
Control-Constrained	19.7	32.3	60.8	151.2	**40.1**
Unconstrained	52.0	347.9	140.7	686.8	**160.2**
Obliviously-Formed	4.8	4.4	6.0	6.7	**5.2**

Table 2.7: Geometric mean idempotent path lengths.

path lengths than Obliviously-Formed, even with more constrained idempotence axis settings, while Unconstrained achieves more than 30x longer path lengths.

Observation 2.2. *Isolation axis setting* FULL *allows for roughly 1.5x longer path lengths than* PRIVATE *and more than 2x longer path lengths than* NONE.

This second observation follows from comparing Control-Constrained (with isolation setting FULL), to Share-Constrained (PRIVATE) and Store-Constrained (NONE). While seemingly the differences varying the isolation setting are small, isolation setting FULL is the only setting that enables even larger region sizes by allowing relaxed control flow assumptions. This effect is articulated by the following observation:

Observation 2.3. *On average, control axis settings* LOCAL-VARIABLE, REGISTER-VARIABLE, *and* INVARIABLE *allow for roughly 4x longer path lengths than* ALL-VARIABLE.

This third observation follows from comparing Unconstrained to Control-Constrained. This contrasts ALL-VARIABLE with the three settings that all assume invariable control flow interactions with respect to non-local memory (INVARIABLE is used as the representative setting). Unconstrained allows for substantially longer path lengths because only stores to dynamically live-in memory locations force termination of an idempotent region.

Observation 2.4. *For the sequencing axis,* COMMIT *is rarely needed, but is helpful in dealing with corner cases.*

Figure 2.7 shows the percentage of inherently non-idempotent instructions dynamically executed under the Unconstrained configuration. The figure contrasts the COMMIT versus FREE sequencing

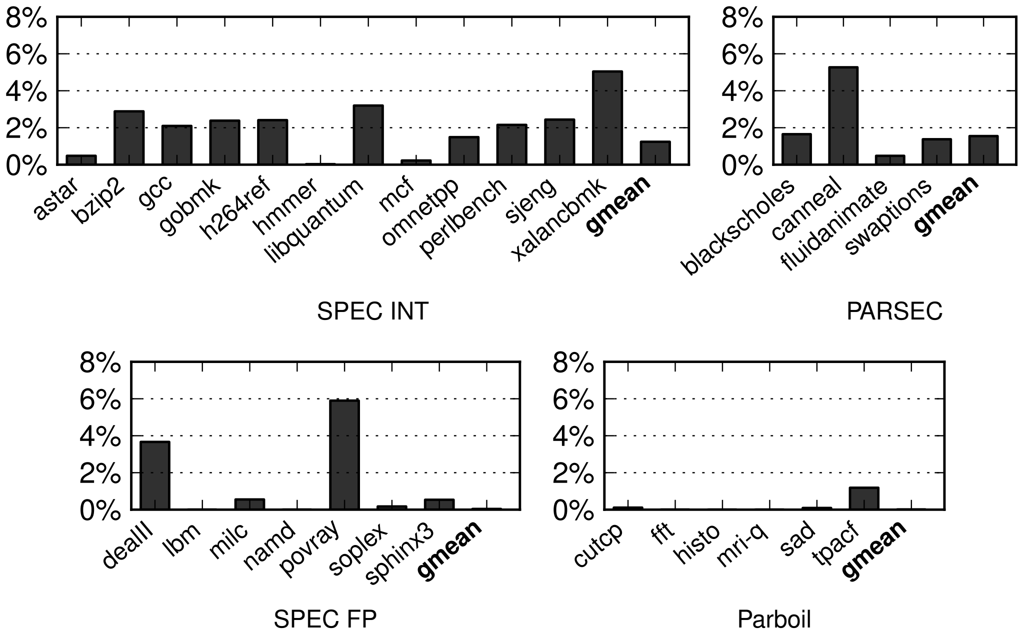


Figure 2.7: The percentage of non-idempotent instructions for model Unconstrained.

axis settings for models with Full isolation guarantees. While the percentage of non-idempotent instructions is in some cases over 5%, typically these instructions account for less than 1% of the program's execution. By far, most of the non-idempotent instructions are those associated with the calling convention (e.g. stack pointer increment/decrement). For models with weaker isolation guarantees, non-isolated stores would also be non-idempotent instructions (1-12% of all instructions depending on the benchmark; not shown in the figure).

2.4.2 Synthesis: Model Selection

The analysis from the previous section helps in identifying the set of most meaningful idempotence models. In particular, the findings can be summarized as: (1) the control axis settings that allow the greatest range of idempotent region sizes are Invariable, Register-variable, and Local-variable, (2) isolation axis settings other than Full severely limit the applicability of idempotence for recovery when used in combination with any of the above control axis settings. (3) support for Commit sequencing is desirable, although unnecessary to support for all cases, and hence should be supported

Idempotence Model	**Control**	**Sequencing**	**Isolation**
Architectural Idempotence	Invariable	Free*	Full
Contextual Idempotence	Local-variable	Free*	Full

Table 2.8: The two selected idempotence models. (* The sequencing axis setting for both configurations is nominally Free with non-idempotent instructions specially supported using Commit.)

only on an as-needed basis, if it is supported at all.

In the end, the two idempotence models shown in Table 2.8 are identified as the most meaningful. These are the two models explored in the remainder of this dissertation, where they are identified using the labels *architectural* and *contextual* idempotence.

Architectural Idempotence

Architectural idempotence is so named because it preserves the architectural state of the program between idempotent regions. It assumes the Invariable control axis setting, where an execution failure cannot affect state elements that are not also affected by a failure-free execution of the region ($E_{i_j} = O_{i_j}$).

With Invariable control, Equation 2.1 can be substituted for Equation 2.2 as shown by Lemma 2.1 of Section 2.2. Applying Lemma 2.1 to Equation 2.2 and also substituting in the effects of optional Commit sequencing (Equation 2.3) and Full isolation (Equation 2.5) produces the following equation for architectural idempotence with (top) and without (bottom) Commit sequencing:

$$\emptyset = \begin{cases} \bigcup_{X_{i_j} \in P_i} L_{i_j} \cap (W_{i_j} \setminus (W_{i_{j_F}} \setminus W_{i_{j_O}})) & \text{if } P_i \text{ has } \textsc{Commit} \text{ sequencing} \\ \bigcup_{X_{i_j} \in P_i} L_{i_j} \cap W_{i_j} & \text{otherwise} \end{cases}$$

From an end-user perspective, architectural idempotence is the most intuitive category of idempotence, and it is the one required to observe an architecturally-precise, sequentially-ordered snapshot of the program at region boundaries. Such an ordered snapshot guarantees a consistently optimal view of the program's state for the purposes of debugging, and supports the execution of interrupt handlers that require visibility to possibly arbitrary precise state.

In terms of recovery, architectural idempotence can be used to cleanly recover from page faults in the manner described in the beginning of this chapter. More generally, it can be used to recover from all execution failures that have no side effects (e.g. traditional hardware exceptions) as well as execution failures where the effects of the failure can be spatially contained to the output elements of a failure-free execution of the region. In practice, such types of failures tend to be the kind that occur rarely, and Full isolation in combination with Invariable control allows architecturally idempotent regions to be very large. Large regions are ideally suited to handling infrequent failures as explained in more detail in Chapter 4.

Contextual Idempotence

Contextual idempotence provides a looser definition of idempotence than architectural idempotence, relaxing control flow constraints and thus supporting a weaker level of spatial containment. In particular, contextual idempotence employs control axis setting Local-variable, allowing an idempotent region to write to function-local storage resources that may not be written to given an actual failure-free execution of the program.

The adverse consequence is that, whereas architectural idempotence preserves all architectural state between regions, contextual idempotence preserves only the architectural state that is *live* with respect to the running program (the program "context") between regions. The key tradeoff is thus that contextual idempotence allows corrupted state to be generated and held in *dead* state (rather than in otherwise special-purpose buffering structures as would be required in the case of architectural idempotence) at the loss of sequential precision in the architectural state of the program. Contextual idempotence still employs Full isolation to preserve architecturally correct state with respect to non-local memory. This allows idempotent path lengths to remain large as noted combining Observations 2.2 and 2.3 (although this may not always be necessary or desirable).

Substituting the effects contextual idempotence into Equation 2.2 produces the following equation with (top) and without (bottom) Commit sequencing (Local-variable control constraints omitted):

$$\emptyset = \begin{cases} \bigcup_{X_{i_j} \in P_i} L_{i_j} \cap (Z_{i_j} \setminus (Z_{i_{j_F}} \setminus Z_{i_{j_O}})) & \text{if } P_i \text{ has Commit sequencing} \\ \bigcup_{X_{i_j} \in P_i} L_{i_j} \cap Z_{i_j} & \text{otherwise} \end{cases}$$

In terms of recovery, using the program's dead state elements to hold corrupted state contextual idempotence can be leveraged to more flexibly recover from execution failures such as branch mispredictions and hardware faults. If an execution is correct, the dead state written by a region will eventually become live over the course of the region's execution. Similarly, if the execution is incorrect and the region must be re-executed, either the dead state is overwritten with a correct value before becoming live, or it will remain dead throughout the re-execution.

Other Idempotence Models

While only architectural and contextual idempotence are explored in the remainder of this dissertation, it is worth noting that other idempotence models may have applicability in specific domains. For instance, full isolation may be an unrealistic assumption in some cases, while the data from Section 2.4 suggests that usefully large idempotent region sizes may still be achievable even with weaker isolation guarantees. A central aim of this dissertation, however, is to explore what is achievable in the limit, and both architectural and contextual idempotence allow the construction of the largest possible idempotent regions subject to semantic program constraints. Thus, henceforth we only focus on these specific models in order to fully understand what they can achieve.

2.5 Summary and Conclusions

This chapter explored and analyzed the concept of idempotence as it applies specifically in the context of computer architecture. As background, it presented examples of idempotence applied in computer science and presented different interpretations of idempotence that were all valid in the context of computer architecture. It then developed a formal taxonomy to reason about these different interpretations, and others, in the architecture context. Leveraging this taxonomy, it presented an empirical study of the sizes of idempotent regions that could be attained for different idempotence models arising from the taxonomy. Finally, it identified the two idempotence models, *architectural* and *contextual* idempotence, that are to be developed in subsequent chapters.

3 Static Analysis of Idempotent Regions

This chapter describes an algorithm for identifying the largest *semantically* idempotent regions in a program, for both contextual and architectural idempotence, using intra-procedural static analysis. Section 3.1 presents fundamental concepts and provides an overview of the algorithm which operates by "cutting" antidependences. Section 3.2 describes the transformations that allow the problem of constructing idempotent regions to be cast and solved efficiently in these terms. Section 3.3 then describes the core static analysis technique for cutting the antidependences and Section 3.4 describes optimizations for dynamic behavior. Finally, Section 3.5 presents a summary and conclusions.

3.1 Static Analysis Overview

This section develops a foundation for the analysis of idempotence using data dependence information. Section 3.1.1 develops and defines two key terms: *clobber dependence* and *clobber antidependence* (a type of clobber dependence). It also distinguishes between between *semantic* (required) and *artificial* (not required) clobber dependences. Section 3.1.2 then presents an overview of the static analysis algorithm that attempts to partition program functions into the largest possible semantically idempotent regions.

3.1.1 Idempotence Analysis Using Data Dependence Information

In the discussion that follows, the term *flow dependence* is used to refer to a read-after-write (RAW) dependence, the term *antidependence* is used to refer to a write-after-read (WAR) dependence, and the term *output dependence* is used to refer to a write-after-write (WAW) dependence. Applying

Term	Storage Resources
Pseudoregisters	Registers and local stack memory
Non-local memory	Heap, global, non-local stack memory

Table 3.1: Terms used to refer to different types of architectural storage resources.

these terms to the definition of live-in from Chapter 2, a variable is live-in to a region (or path) simply if it has a flow dependence spanning the entry point of the region (or path).

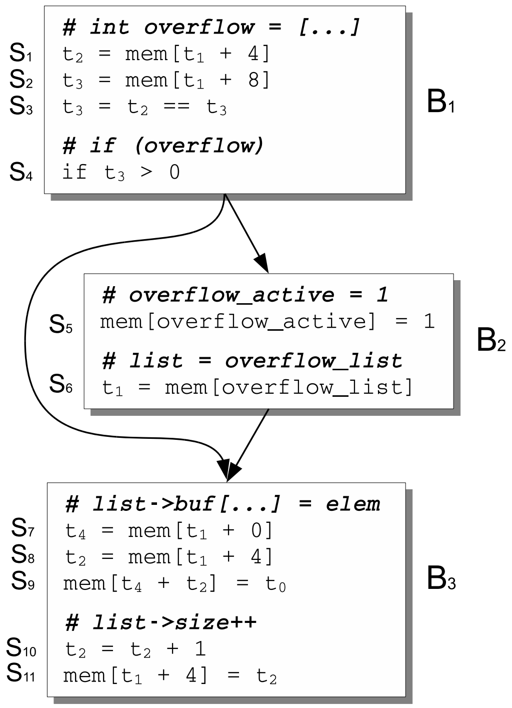
Table 3.1 additionally distinguishes among two different categories of architectural storage resource. The term *pseudoregister* is used to refer to a function-local, private storage elements such as registers or stack memory locations. The term *non-local memory* is used to refer to to all other types of storage elements; namely heap, global, and non-local stack memory.

Clobber Antidependences

Both architectural and contextual idempotence allow overwriting non-local live-in variables if they are dynamically dead along a given path through a region. A live-in variable is known to be dynamically dead beyond some point in the region 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. The act of overwriting implicitly guarantees (b), hence, with respect to the point of the write, a live-in is dynamically dead simply if it has not yet been read. Overwriting a live-in that has been read is thus the case that must be dis-allowed. The write-after-read relationship is an antidependence, and hence we infer that, to preserve idempotence, a live-in variable may not be overwritten as part of an antidependence relationship that follows the live-in's flow dependence spanning the region entry point. The existence of such an antidependence that follows a flow dependence spanning the region entry we call a *clobber antidependence*.

Clobber Dependences

Architectural idempotence can be reasoned about entirely in terms of clobber antidependences. Under contextual idempotence, however, pseudoregister live-in variables may *never* be overwritten inside a region, irrespective of whether the write occurs before or after a read, due to the possibility



```
# int overflow = [...]
t2 = mem[t1 + 4]
t3 = mem[t1 + 8]
t3 = t2 == t3

# if (overflow)
if t3 > 0
```

```
# overflow_active = 1
mem[overflow_active] = 1

# list = overflow_list
t1 = mem[overflow_list]
```

```
# list->buf[...] = elem
t4 = mem[t1 + 0]
t2 = mem[t1 + 4]
mem[t4 + t2] = t0

# list->size++
t2 = t2 + 1
mem[t1 + 4] = t2
```

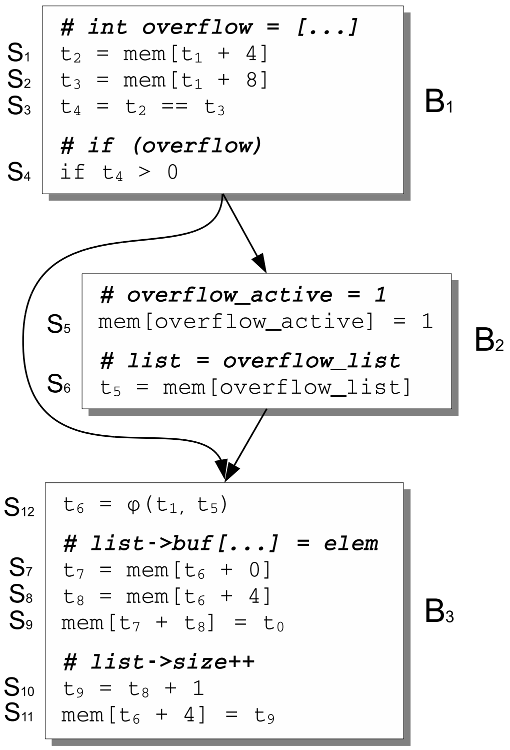
Figure 3.1: Control flow and compiler intermediate code (not in SSA form).

of variable control flow upon re-execution. With this stricter requirement, the absence of clobber antidependences is insufficient—a write need not follow a read to break idempotence. For this case, a live-in variable may not be overwritten to form a write-after-write (output dependence) that spans the entry point of the region, where the first write is the write of the live-in's flow dependence that spans the entry point. The dual occurence of a flow dependence and output dependence both spanning the entry point of a region we call a *clobber dependence*. Note that a clobber antidependence is a special type of clobber dependence.

Semantic and Artificial Clobber Dependences

Some clobber dependences (both clobber antidependences and other types of clobber dependences) are strictly necessary according to program semantics. These clobber dependences we label *semantic* clobber dependences. The other clobber dependences we label *artificial* clobber dependences. The following example demonstrates semantic and artificial clobber dependences.

We revisit the function from Figure 2.1 as a running example. Figure 3.1 shows the function



```
# int overflow = [...]
t2 = mem[t1 + 4]
t3 = mem[t1 + 8]
t4 = t2 == t3

# if (overflow)
if t4 > 0
```

```
# overflow_active = 1
mem[overflow_active] = 1

# list = overflow_list
t5 = mem[overflow_list]
```

```
t6 = φ(t1, t5)

# list->buf[...] = elem
t7 = mem[t6 + 0]
t8 = mem[t6 + 4]
mem[t7 + t8] = t0

# list->size++
t9 = t8 + 1
mem[t6 + 4] = t9
```

Figure 3.2: Control flow and compiler intermediate code in SSA form.

compiled to a load-store intermediate representation. The figure shows the control flow graph of the function, which contains three basic blocks B_1, B_2, and B_3. Inside each block are shown the basic operations, S_i, and the use of pseudoregisters, t_i, to hold operands and read and write values to and from memory. The clobber dependences on pseudoregisters are *artificial*, while other clobber dependences are *semantic* as explained below.

As an enabling transformation, Figure 3.2 shows the effect of converting all pseudoregister assignments to static single assignment (SSA) form [25]. Note that, under SSA, all pseudoregister output dependences and antidependences on pseudoregisters disappear[1]. Such dependences are removable and hence unnecessary: all clobber dependences that arise from them are *artificial*. After transforming the code into SSA (or similar) form, a compiler can permanently eliminate clobber dependences by ensuring that, during register and stack slot allocation, pseudoregisters live at an idempotence boundary are not overwritten by the assignments of other pseudoregisters sharing

[1] In general all such types of dependences disappear. However, they may remain across loop iterations. For overview purposes this issue is ignored here and is revisited in Section 3.3.2.

Type of Clobber Dependence	Storage Resources
Artifial clobber dependence/antidependence	Pseudoregisters
Semantic clobber antidependence	Non-local memory

Table 3.2: Semantic and artificial clobber dependences and associated resources.

the same storage resource (in the case of architectural idempotence, such an assignment may occur, but only if it is not preceeded by a read). This resource allocation constraint enables idempotence in exchange for additional register pressure on the region. Assignments to non-local memory storage resources, however, cannot be renamed by a compiler. They are bound to fixed locations by the semantics of the program and hence clobber dependences over these locations are *semantic*.

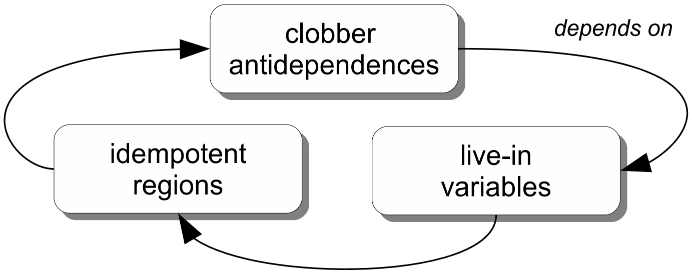
Summary and Implications

Table 3.2 summarizes the differences between semantic and artificial clobber dependences. 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, compiling for idempotence does not grow the size of the stack significantly and we have no size-related difficulties compiling any benchmarks.

Semantic clobber dependences, in contrast, act on non-local memory locations: heap, global, and non-local stack memory. These memory locations are not under the control of the compiler; they are specified in the program itself and cannot be re-assigned. Furthermore, since both architectural and contextual idempotence specify that live-in non-local memory may be assigned inside a region if the assignment occurs before a read, at this point we need no longer concern ourselves with the general case of clobber dependences over non-local memory; only clobber antidependences apply over non-local memory.

3.1.2 The Partitioning Algorithm

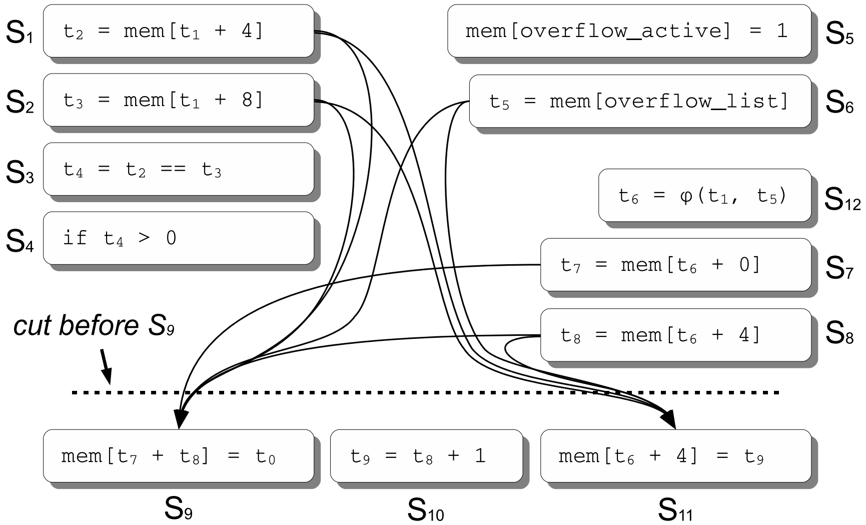
Assuming it is possible to eliminate all artificial clobber dependences, the idempotent regions that exist in application programs are potentially large as suggested by the data from Section 2.4. However, the problem of statically constructing large idempotent regions remains surprisingly non-trivial. 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 dependences requires identification of region live-in variables, which in turn requires identification of the regions. This circular dependence is illustrated below:



The static analysis algorithm of this chapter resolves this circular dependence problem. The first step of the algorithm (Section 3.2) transforms the function so that, with the exception of self-dependent pseudoregister dependences[2], *all antidependences are necessarily semantic clobber antidependences*. This enables the construction of idempotent regions simply in terms of antidependence information. Antidependence information does not depend on region live-in information, and hence the circular dependence chain is broken. During the transformation process, self-dependent pseudoregister dependences are optimistically assumed not to emerge as clobber dependences; those that would emerge as clobber dependences after the region construction are patched in a subsequent refinement step.

Considering only antidependence information, the problem of partitioning the program into idempotent regions is reduced to the problem of "cutting" the antidependences, such that a cut before statement S starts a new region at S (Section 3.3). In this manner, no single region contains both ends of an antidependence and hence the regions are idempotent. To maximize the region sizes, the problem is cast in terms of the NP-complete vertex multicut problem and an approximation algorithm is used to find the minimum set of cuts, which finds the minimum set of regions. Intuitively, this produces large static region sizes averaged across a function. Finally, heuristics are introduced to maximize the sizes of regions as they occur dynamically at runtime (Section 3.4).

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



```
t2 = mem[t1 + 4]
t3 = mem[t1 + 8]
t4 = t2 == t3
if t4 > 0
mem[overflow_active] = 1
t5 = mem[overflow_list]
t6 = φ(t1, t5)
t7 = mem[t6 + 0]
t8 = mem[t6 + 4]
mem[t7 + t8] = t0
t9 = t8 + 1
mem[t6 + 4] = t9
```

Figure 3.3: The data dependence graph of Figure 3.2 showing only non-local memory clobber antidependences using solid lines. Cut placement shown using dashed line.

Example 3.1. Figure 3.3 shows the algorithm applied to our running example. The initial step is the conversion of all pseudoregister assignments to SSA form as previously illustrated in Figure 3.2. Figure 3.3 shows the dependence graph of Figure 3.2 with non-local memory may-alias clobber antidependences shown using solid black lines. Under SSA, the artificial clobber dependences disappear and the semantic ones remain. 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). (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).

All semantic clobber antidependences involve a write to either memory location mem[t_7 + t_8] or memory location mem[t_6 + 4]. 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 all antidependences exactly between the statements S_8 and S_9 as shown in Figure 3.3. With this cut in place, the function is ultimately divided into two semantically 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.

```
Before:               After:
1.  mem[x] = a        1.  mem[x] = a
2.  b = mem[x]        2.  b = a
3.  mem[x] = c        3.  mem[x] = c
```

Figure 3.4: Eliminating non-clobber memory antidependences.

3.2 Program Transformation

In the static analysis algorithm, two code transformations precede any idempotence analysis. The two transformations are (1) the conversion of all pseudoregister assignments to static single assignment (SSA) form, and (2) the elimination of all non-local memory antidependences that are not clobber antidependences. The details on why and how are given below.

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

To resolve this ambiguity, after the SSA transformation we employ a simple redundancy-elimination transformation illustrated by Figure 3.4. The sequence on the left has an antidependence on non-local 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 is eliminated as shown on the right of the figure: the use of memory location x is replaced by the use of pseudoregister a and the antidependence disappears.

Unfortunately, there is no program transformation that resolves whether any remaining self-dependent pseudoregister antidependences or output dependences will manifest as clobber dependences. In the following section, we initially assume that these dependences can be register allocated such that they do not become clobber dependences (i.e. we can precede an antidependence or output dependence in a region with a flow dependence in the same region). Hence, we construct regions around them, considering only the known, non-local memory clobber antidependences. After the construction is complete, we check to see if our assumption holds. If not, we insert additional region cuts as necessary.

3.3 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 non-local memory 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.

3.3.1 Cutting Non-Local Memory Antidependences

To ensure that a non-local memory 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. This goal of creating smallest number of regions allows us to specify our problem constraints in simple terms, while intuitively it is an effective proxy for enabling the construction of the largest regions in a function. This section derives our algorithm as follows:

1. We define our problem as a graph decomposition that must satisfy certain conditions.

2. We reduce the problem of finding an optimal graph decomposition to the minimum vertex multicut problem.

3. To generate a solution, we formulate the problem in terms of the hitting set problem.

4. We observe that a near-optimal hitting set can be found efficiently using an approximation algorithm.

Problem Definition

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

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

Our problem is to decompose G into the smallest set of sub-graphs $\{G_1, \cdots, G_k\}$. Figure 3.5 gives an example. Figure 3.5(a) shows a control flow graph G and 3.5(b) shows the set of antidependence edges in G. Figure 3.5(c) shows a possible region decomposition for G. The shown region decomposition happens to be optimal; that is, it contains the fewest possible number of regions.

Reduction to Vertex Multicut

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

[3] Note that, while we use the term *header node*, we do not require that a header node h_i dominates all nodes in V_i as defined in other contexts [4].

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

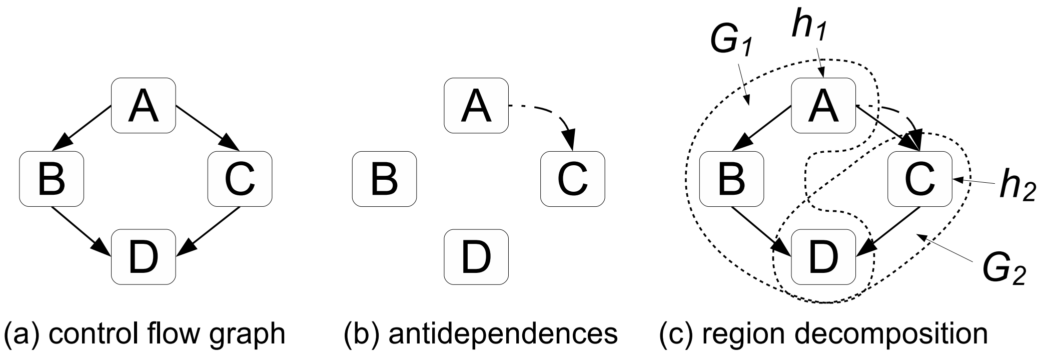


Figure 3.5: An example region decomposition.

Definition 3.1 (Vertex Multicut): *Let $G = (V, E)$ be a directed graph with set of vertices V and edges E. Assume that we are given pairs of vertices $A \subseteq V \times V$. A subset of vertices $H \subseteq V$ is called a* vertex multicut *for A if in the subgraph G' of G where the vertices from H are removed, for all ordered pairs $(a, b) \in A$ there does not exist a path from a to b in G'.*

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

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

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

Solution Using Hitting Set

The vertex multicut problem is NP-complete for general directed graphs [42]. To solve it, we reduce it to the hitting set problem, which is also NP-complete, but for which good approximation algorithms are known [23].

Definition 3.2 (Hitting Set): *Given a collection of sets $C = \{S_1, \cdots, S_m\}$, a* minimum hitting set *for C is the smallest set H such that, for all $S_i \in C$, $H \cap S_i \neq \emptyset$.*

Note that we seek a set $H \subseteq V$ such that, for all $(a_i, b_i) \in A$, all paths π from a_i to b_i have a vertex in H (in other words, H is a "hitting set" of $\Pi = \cup_{(a_i,b_i)\in A}\pi_i$, where π_i is the set of paths from a_i to b_i). This formulation is not computationally tractable, however, as the number of paths between any pair (a_i, b_i) can be exponential in the size of the graph. Instead, for each $(a_i, b_i) \in A$, we associate a single set $S_i \subseteq V$ that consists of the set of nodes that dominate b_i but do not dominate a_i. We then compute a hitting set H over $C = \{S_i | S_i \text{ for } (a_i, b_i) \in A\}$. Using Lemma 3.1 it is easy to see that for all antidependence edges $(a_i, b_i) \in A$, every path from a_i to b_i passes through a vertex in H. Hence, H is both a hitting set for C and a vertex multicut for A.

We use a greedy approximation algorithm for the hitting set problem that runs in time $O(\sum_{S_i \in C} |S_i|)$. This algorithm chooses at each stage the vertex that intersects the most sets not already intersected. This simple greedy heuristic has a logarithmic approximation ratio [23] and is known to produce good quality results.

Lemma 3.1. *Let $G = (V, E, s)$ be a directed graph with entry node $s \in V$ and (a, b) be a pair of vertices. If $x \in V$ dominates b but does not dominate a, then every path from a to b passes through x.*

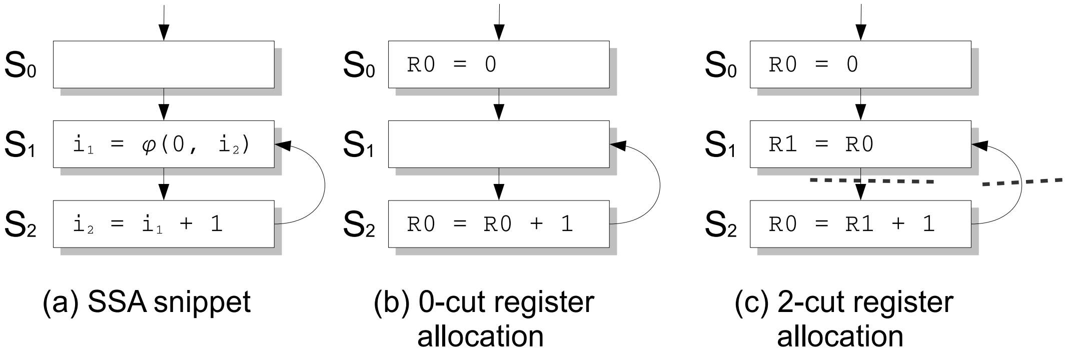
Proof. We assume that a pair of vertices (a, b) are both reachable from the entry node s. Let the following conditions be true.

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

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

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

□



(a) SSA snippet (b) 0-cut register allocation (c) 2-cut register allocation

Figure 3.6: Clobber-free allocation of self-dependent pseudoregister dependences.

3.3.2 Cutting Self-Dependent Pseudoregister Dependences

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

In SSA form, a self-dependent pseudoregister dependence manifests as a write occurring at the point of a ϕ-node assignment, with one of the ϕ-node's arguments data-dependent on the assigned pseudoregister itself. Due to SSA's dominance properties, such self-dependent pseudoregister assignments alway occur at the head of loops (for natural loops). Figure 3.6(a) provides a very simple example. Note that in the example the self-dependent "dependence" show, is actually two antidependences, $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_1$ (in the general case, it might also form an output dependence along some path through the loop where the pseudoregister is not read before it is updated). We refer to it as a single "dependence" for ease of explanation.

To prevent self-dependent pseudoregister dependences from emerging as clobber dependences, the invariant we must enforce is that a loop containing such a dependence 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 dependence shown in Figure 3.6(a). For a loop that contains no cuts, this dependence can be trivially register allocated as shown in Figure 3.6(b). In the figure, we define the register (which could also be a stack slot if registers are scarce) of the dependence outside the loop and hence across all loop iterations all instances of the dependence are preceded by a flow dependence.

Case 2: A Loop With At Least 2 Cuts

A self-dependent dependence for a loop that contains at least two cuts can also be trivially-register allocated as shown in Figure 3.6(c). Here the dependence 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 3.6(c) but in the general case it may not be.

Case 3: Neither Case 1 Or 2

In the remaining case, the self-dependent dependence is in a loop that contains at least one cut but there exist one or more paths through the loop body that do not cross at least two cuts. In this case, we know of no way to register allocate the dependence such that it does not emerge as a clobber dependence. Hence, we resign ourselves to cutting the dependence edge 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 3.6(c).

3.4 Optimizing for Dynamic Behavior

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

We account for loop information by incorporating a simple heuristic into the hitting set algorithm from Section 3.3.1. In particular, we adjust the algorithm to greedily choose cuts at nodes from the outermost loop nesting depth first. We then break ties by choosing a node with the most sets not already intersected as normal. This improves the path lengths substantially in general, although there are cases where it reduces them. A better heuristic most likely weighs both loop nesting depth and intersecting set information more evenly, rather than unilaterally favoring one. Better heuristics are a topic for future work.

3.5 Summary and Conclusions

This chapter described an algorithm for identifying the largest idempotent regions in a function given semantic program constraints. It presented a framework for the analysis idempotence in terms of data dependence information, and introduced the terms *clobber dependence* and *clobber antidependence* (a special type of clobber dependence) to describe the data dependence patterns that inhibit idempotence. It then presented an algorithm to partition functions into idempotent regions by first removing program artifacts that inhibit effective idempotence analysis and subsequently cutting the antidependences that remain. Finally, it described optimizations for dynamic behavior.

Overall, the algorithm operates at a high level and effectively abstracts out the fundamental constraints on idempotence to deliver a clean and modular solution to the partitioning problem. This abstract modularity allows it to be easily extended to incorporate various custom heuristics, or to operate at an inter-procedural level, as desired in future pursuits.

4 Code Generation of Idempotent Regions

The static analysis algorithm developed in Chapter 3 approximates the largest semantically idempotent regions in a function. These regions are a rough upper bound on the sizes of the regions that can constructed by a source code transformation tool such as a compiler, which must observe programmer- and language-specified semantics. The primary purpose in identifying these largest possible regions is that it is generally straight-forward to take these large idempotent region and partition them into smaller regions as desired, whereas the reverse problem of taking small idempotent regions and making them larger is a much harder problem.

This chapter considers opportunities to sub-divide our large idempotent regions and describes a code generation flow to map idempotent regions to machine code. Specifically, Section 4.1 considers the opportunity to tailor the static analysis region construction to accommodate the constraints of the application and the environment in which the regions are to be used for recovery. Section 4.2 then describes the algorithms used to compile the final set of idempotent regions such that the idempotence property is preserved through the compiler's register and stack allocation process. Section 4.3 presents end-to-end code generation examples. Finally Section 4.4 presents a summary and conclusions. (Specific details regarding compiler implementation are also discussed in Appendix A.)

4.1 Tailoring Idempotent Region Sizes

Even in the context of generating code for idempotence-based recovery, large idempotent regions often remain best for two reasons: they minimize the *detection stall latency* at idempotent region boundaries, and they minimize the *register pressure* overhead of preserving live register state across

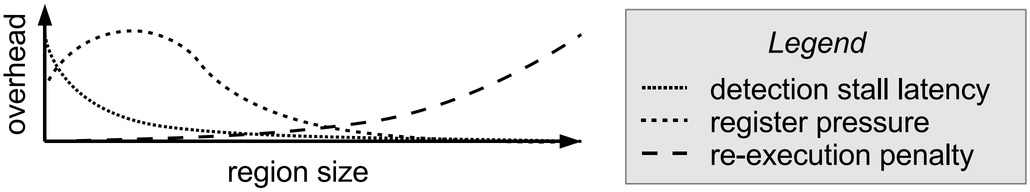


Figure 4.1: The runtime overhead in relation to idempotent region size for three different contributing factors (approximate; not to scale).

idempotent regions. However, it is not always possible to construct idempotent regions large enough to minimize register pressure over a sufficiently large group of instructions. Additionally, minimizing the *re-execution penalty* in the event of recovery favors smaller region sizes.

The trade-off among these three factors is sketched in the graph of Figure 4.1. The shapes of the curves are explained as follows:

Detection stall latency: To support idempotence-based recovery, program execution may not proceed beyond the end of an idempotent region until that region is verified to be free of failures. Otherwise, the following region could overwrite one of the earlier region's inputs, which would render recovery impossible. Hence, larger region sizes allow execution to proceed "speculatively" over longer periods of time while potential execution failures remain undetected.

Given a type of execution failure with detection latency d instructions, let r represent the number of instructions to execute a region. Equation 4.1 computes the detection stall latency, l, as a fraction of r. The latency l is inversely proportional to r, approaching zero as r approaches infinity, irrespective of d. This inverse relationship is what is shown for the corresponding curve of Figure 4.1.

$$l = \frac{d}{r + d} \tag{4.1}$$

Register pressure: Rather than present an equation that models the effect of register pressure, the curve shown in Figure 4.1 can be explained more easily by appealing to intuition. 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.

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[1], and hence this configuration introduces only modest additional register pressure and hence incurs little runtime overhead as illustrated at the far left point of the curve on Figure 4.1.

As region size grows beyond one instruction, however, the registers that were live at that initial instruction must be preserved by idempotence, even if they were previously killed[2] by that instruction or subsequent instructions. Hence, quickly register pressure grows and registers need to be spilled to make room. Initially, the rate of growth can be as high as one register spill for each register-writing instruction added to a region, and control divergence only exacerbates the rate at which the list of live-ins accumulates. Eventually, however, the number of registers live at the entry point of the region starts to diminish as the values in those registers are forcibly spilled, allowing those registers to be re-used. Eventually the worst case is reached, where all of a region's live-in 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 due to additional register pressure approaches zero.

Re-execution penalty: When an execution failure occurs, recovery is performed by re-executing the containing idempotent region. Assuming a fixed per-instruction execution failure probability p and a region of length r instructions, the average number of times a region must be re-executed before it completes successfully, e, is exponentially proportional to r according to the following equation:

[1] It is rare generally speaking, although it is not true for instruction sets with predominantly two-address instructions (e.g. x86). See Section 5.4.

[2] An instruction *kills* a register if is the last instruction to use the value held in that register.

Factor	**Example**	**Optimal Size**
Long detection latencies	Page faults in GPUs	Large
Limited semantic idempotence	Object-oriented programs	Lowest pressure
Frequent execution failures	Branch mis-predictions	Small

Table 4.1: Example factors that affect optimal idempotent region size.

$$e = \left(\frac{1}{1-p}\right)^r \tag{4.2}$$

We derive this equation as follows. Let the random variable X denote the number of instructions executed before a failure occurs. X has a geometric distribution with $P(X = k) = (1-p)^{k-1}p$. Let p_{succ} denote the probability of a successful (failure-free) execution of a region. $p_{succ} = P(X > r) = (1-p)^r$. It follows that the number of attempts to execute over the region before success, denoted by the random variable Y, has a geometric distribution with $P(Y = k) = (1-p_{succ})^{k-1}p_{succ}$ and expected number of executions:

$$e = E(Y) = \frac{1}{p_{succ}} = \frac{1}{(1-p)^r} = \left(\frac{1}{1-p}\right)^r$$

If we allow the simplifying assumption that the number of "wasted" instructions w executed before failure during a failed execution grows proportionally[3] to r, then the overall overhead of re-execution grows exponentially with r. This growth is illustrated by the corresponding curve of Figure 4.1.

Given the interplay of the various overhead factors illustrated by Figure 4.1, it is evident that optimal region size is dependent on a number of external factors. Table 4.1 lists some of these factors, along with examples of where they come into play using idempotence for recovery and how they impact the optimal region size.

[3] In practice, w grows slightly sub-linearly with respect to r, intuitively because a failure is more likely to occur earlier rather than later during the execution of a region (assuming independent failure). However, this effect is not strong enough to counter-act the overall exponential growth in overhead with respect to r.

Strategy	Goal
Max-Size	Maximize region size
Min-Pressure	Minimize register pressure
Min-Recovery	Minimize re-execution time

Table 4.2: Code generation strategies and the goal of each.

4.1.1 Code Generation Strategies

To balance the size versus performance trade-off for application and environmental factors such as those in Table 4.1, we propose the three recovery code generation strategies shown in Table 4.2. The strategies are labeled Max-Size, Min-Pressure, and Min-Recovery. They are explained below.

Max-Size: Constructing the maximum possible idempotent region sizes is ideal for recovering from execution failures such as exceptions or hardware faults that (a) are infrequent, (b) involve potentially long detection latencies, and (c) occur in programs that have inherently large idempotent regions (over which register pressure can be minimized). These are the types of execution failures ideally targeted by the compilation strategy Max-Size.

A compelling example use-case for this strategy is in supporting infrequent exceptions with long detection latencies, such as page faults under a virtual caching implementation, on architectures that execute programs with inherently large idempotent regions, such as GPUs. Supporting page faults to enable demand-paged virtual memory on GPUs would ordinarily require large checkpointing or buffering structures, which are difficult to support without excessively compromising throughput or energy-efficiency. Since GPU programs have inherently large idempotent program regions (see Chapter 5), idempotent regions could be used for recovery with comparatively little hardware and only minimal performance overhead.

Min-Pressure: GPU programs and other computationally-intensive applications tend to have inherently 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. As seen in the static analysis example of Section 3.1.2, these programs often contain many ambiguous "may-alias" memory references of the same variety as those that inhibit automatic parallelization opportunities in compilers [71].

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. If execution failure detection latencies are not particularly long, then programs are better served by judiciously shortening region sizes to minimize overheads resulting from register pressure. This is the approach of compilation strategy MIN-PRESSURE.

Strategy MIN-PRESSURE takes as argument a *pressure threshold*, which is a value from 1 to 10. It 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 equals 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. An example that illustrates this is provided in Section 4.3.

MIN-RECOVERY: Even when semantically large idempotent regions are abundant, for types of execution failures that occur frequently, these regions may be unsuitably large. For instance, if we assume a region length of 100 instructions ($r = 100$), a per-instruction failure probability of 1% ($p = 0.01$) and approximate the wasted instructions of a failed execution at half of the instructions in a region ($r/2$), using Equation 4.2 the re-execution overhead computes to:

$$\left(\frac{1}{(1-p)}\right)^r \times \frac{r}{2} = \left(\frac{1}{1-0.01}\right)^{100} \times \frac{100}{2} = 136.6$$

Hence, with these parameters more than half of the time attempting to execute the region is spent in wasted (unproductive) execution cycles.

The compilation strategy MIN-RECOVERY attempts to minimize the re-execution cost by placing region boundaries as close to the site of an execution failure as possible. It assumes that only specific instructions, such as branches, loads, or stores, can cause execution failures, and that the compiler potentially has some information that allows it to predict with some accuracy how likely such an instruction is to cause an execution failure.

To illustrate the benefit, consider a low-confidence branch instruction with a 30% mis-prediction probability ($p = 0.3$). Under MIN-RECOVERY, the compiler might construct a 10-instruction region starting immediately before that instruction ($r = 10$). Assume that once a misprediction occurs it does not recur, and that a 2-instruction processing delay exists before misprediction is detected and re-execution can be initiated. The average re-execution overhead for this branch is then $2 \times 0.3 = 0.6$ instructions compared to $(2 + 10) \times 0.3 = 4$ (almost half of r) had the branch been allowed to occur in the middle of a similarly-sized region.

4.2 Idempotence State Preservation

After the idempotent region boundary locations are finalized by applying one of the compilation strategies from the previous section, the job of the compiler is to generate code for these regions in such a way that their semantic idempotence is preserved throughout the process of register and stack allocation. This section covers the algorithms involved in this task specifically.

Live Intervals, Region Intervals, and Shadow Intervals

To preserve the idempotence of semantically idempotent regions, we introduce the concepts of a *region interval* and a *shadow interval*. These concepts are analogous to the existing concept of a *live interval*, which is a 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 [4]. Figure 4.2 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. Since the live

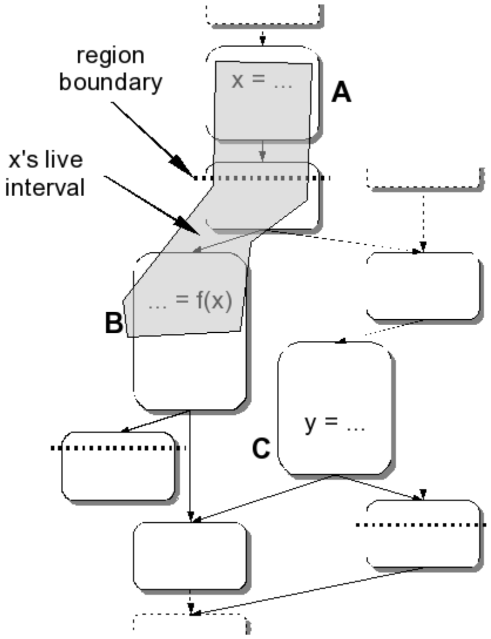


Figure 4.2: An example CFG subset showing the live interval for a variable x.

interval already prevents overwriting where a shadow interval and live interval might otherwise overlap, for clarity the shadow interval is kept disjoint from the live interval.

The shadow interval is computed in terms of live intervals and region intervals using an algorithm that is specific to the idempotence model. Below, the algorithms for contextual idempotence and architectural idempotence are presented in Sections 4.2.1 and 4.2.2, respectively. We consider the state preservation algorithm for contextual idempotence first since it is simpler than for architectural idempotence.

4.2.1 State Preservation for Contextual Idempotence

For contextual idempotence, within an idempotent region *no* variable write may be co-allocated with a variable that is live-in to the region, lest the write will create a clobber dependence on the allocated resource. For this idempotence model, the shadow interval of the 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 as inputs a variable v and the set of region intervals R in v's function, the algorithm is as presented in Algorithm 1.

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

```
s ← ∅
l ← COMPUTE-LIVE-INTERVAL(v)
for r ∈ R such that (the entry point of r) ∈ l do
    s ← s ∪ r
end for
return s \ l
```

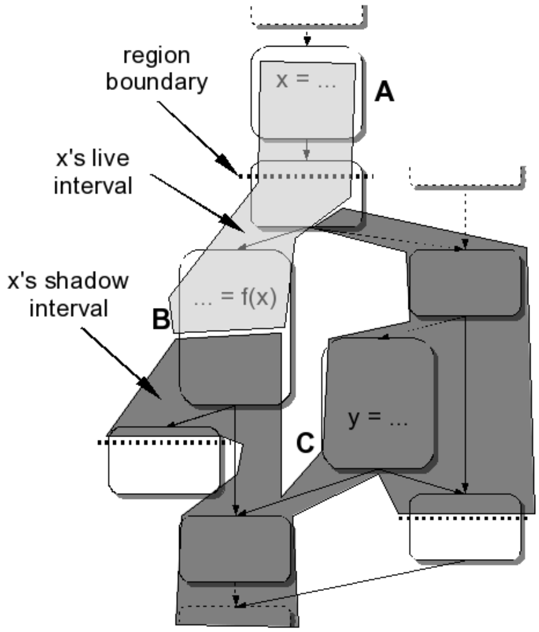


Figure 4.3: Shadow intervals in the case of contextual idempotence.

The result of applying this algorithm to the example CFG of Figure 4.2 (with $v = \text{x}$) is illustrated in Figure 4.3, 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.

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 $\sum_{r \in R} |r|$, which 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. While s is theoretically bounded by n—implying worst case complexity $O(n^2)$—in practice s is small with $s \ll n$ and s can moreover be statically bounded if needed. Hence, in practice the complexity is only $O(n)$. To provide contrast, the complexity of the live interval computation is also $O(n)$.

4.2.2 State Preservation for Architectural Idempotence

The constraints for architectural idempotence are more specific than for contextual idempotence. Hence, the resulting algorithm is slightly more complex. For architectural idempotence, a variable write *may be* co-allocated with a variable that is live-in to the region, as long as the live-in variable is dynamically dead at the entry point of the region with respect to the point of the write. As identified in Chapter 3, a live-in variable is known to be dynamically dead beyond some point in the region 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 (b); hence, the shadow interval concept need only additionally prevent (a). The shadow interval of a live-in variable under architectural idempotence is thus the portion of a region interval that is reachable from a read inside the region (with all live interval ranges again removed).

The algorithm for computing this shadow interval is shown in Algorithm 2. Similarly to Algorithm 1, it takes as input a variable v and the set of region intervals R in v's function. The key difference is that it accumulates on the shadow interval s only those ranges in a region r that are reachable from the use points of v (the points where v is read). The result of applying this algorithm to the example CFG of Figure 4.2 (again, with $v = \mathsf{x}$) is illustrated in Figure 4.4. 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. Simply extending the live intervals would not allow x and y to be co-allocated. With the shadow interval concept, however, a live interval may

Algorithm 2 COMPUTE-ARCHITECTURAL-IDEMPOTENCE-SHADOW-INTERVAL(v, R)

```
$s \leftarrow \emptyset$
$l \leftarrow$ COMPUTE-LIVE-INTERVAL($v$)
$U \leftarrow$ GET-USE-POINTS($v$)
for all $r \in R$ such that (the entry point of $r$) $\in l$ do
    for all $u \in U$ such that $u \in r$ do
        $i \leftarrow$ COMPUTE-REACHING-INTERVAL-IN-REGION($u$, $r$)
        $s \leftarrow s \cup i$
    end for
end for
return $s \setminus l$
```

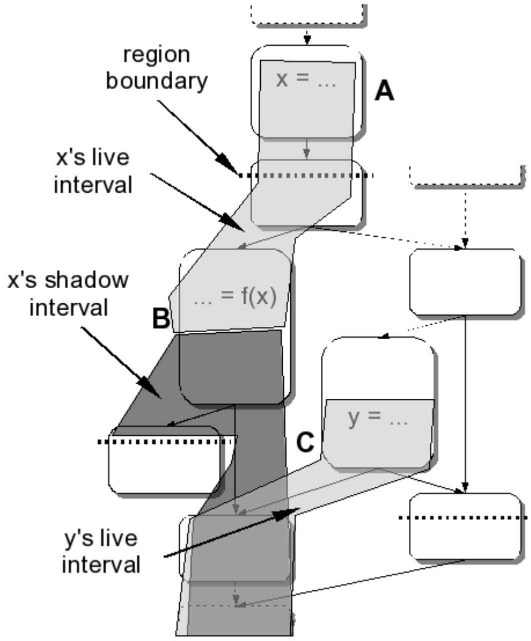


Figure 4.4: Shadow intervals in the case of architectural idempotence.

overlap a shadow interval as long as the definition point(s) of the live interval do not overlap the shadow interval.

Algorithmic Complexity

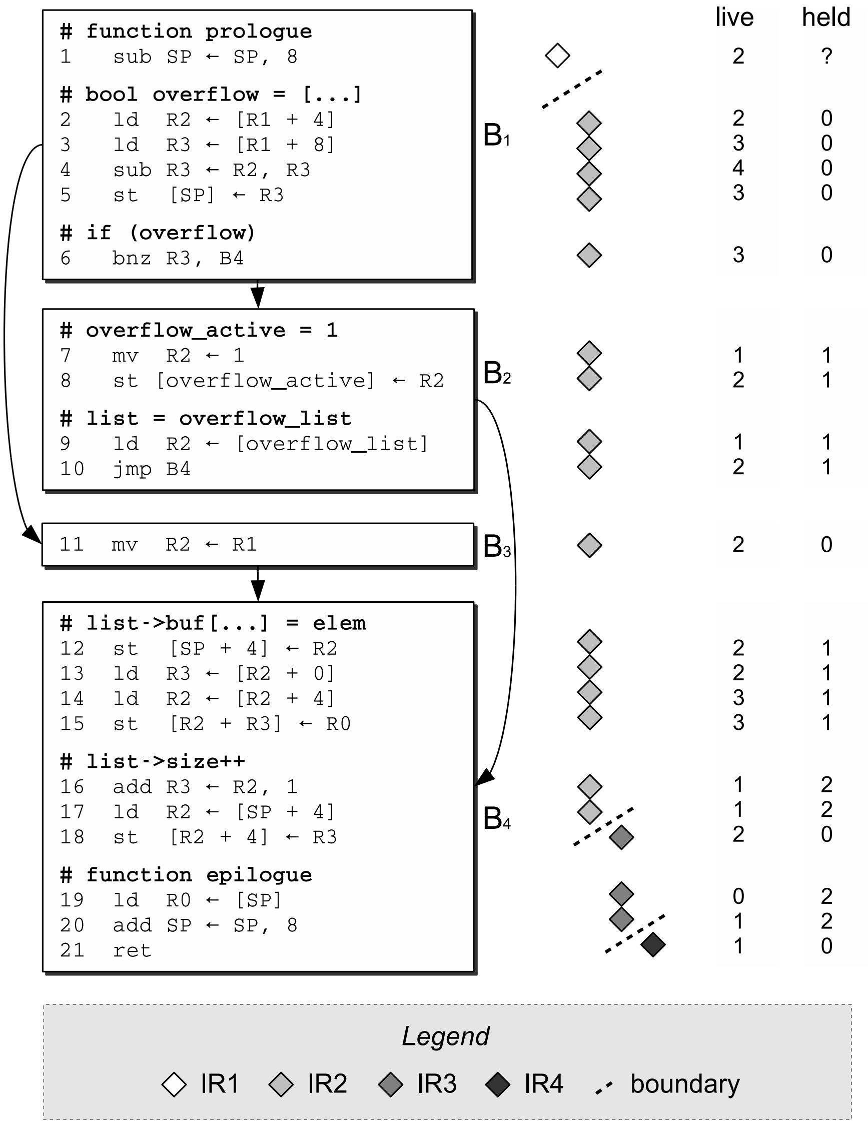
Recall that Algorithm 1 has worst case complexity bounded by $n \cdot s$. Algorithm 2 adds $|U|$ as an additional factor and is thus bounded by $n \cdot s \cdot |U|$. While, relative to Algorithm 1, some additional work is performed in the inner loop of Algorithm 2 to compute the reaching interval of v with respect to r, the overall work of the inner loop is still bounded overall by $|r|$ just as in Algorithm 1, and hence only the newly introduced iteration over U is of significance. While $|U|$, similarly to s, is also theoretically bounded by the number of instructions in the function, n—implying worst case complexity $O(n^2)$ even with s held constant—in practice $|U|$ is small with $|U| \ll n$ and the mean $\overline{|U|}$ across all variable instances v in the function is a constant. The latter is because, intuitively, the total number of uses must grow proportionally to the number of variables. Hence, in practice the complexity remains $O(n)$.

4.3 Code Generation Examples

Examples 4.1, 4.2, and 4.3, in combination with Figures 4.5, 4.6, and 4.7, show sample compiler output for the running example of Figure 2.1. The examples consider each of the Max-Size, Min-Pressure, and Min-Recovery strategies and use the contextual idempotence state preservation algorithm of the previous section. (The differences between architectural and contextual idempotence are omitted since there are no differences for the running example. In general the differences are not substantial; see evaluation Section 5.3.) We consider the Min-Recovery strategy for recovery from branch misprediction specifically.

In each figure, the left side shows the function compiled to assembly code with arrows connecting the control flow between basic blocks. As in Section 2.1, the code assumes four registers are available, `R0`-`R3`, with function arguments held in registers `R0` and `R1`, and `R0` also the return register. 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 class of non-idempotent instruction allowed to terminate a region as allowed by the Commit sequencing model of Section 2.3.) For the Min-Recovery strategy, boundaries may also appear immediately before branch instructions (again, a free bit is assumed).

The middle of each figure marks the region(s) each instruction belongs to, according to the legend shown at the bottom of the figure ("IR" stands for Idempotent Region). The right of each



```
# function prologue
sub SP ← SP, 8
# bool overflow = [...]
ld  R2 ← [R1 + 4]
ld  R3 ← [R1 + 8]
sub R3 ← R2, R3
st  [SP] ← R3
# if (overflow)
bnz R3, B4
```

```
# overflow_active = 1
mv  R2 ← 1
st [overflow_active] ← R2
# list = overflow_list
ld  R2 ← [overflow_list]
jmp B4
```

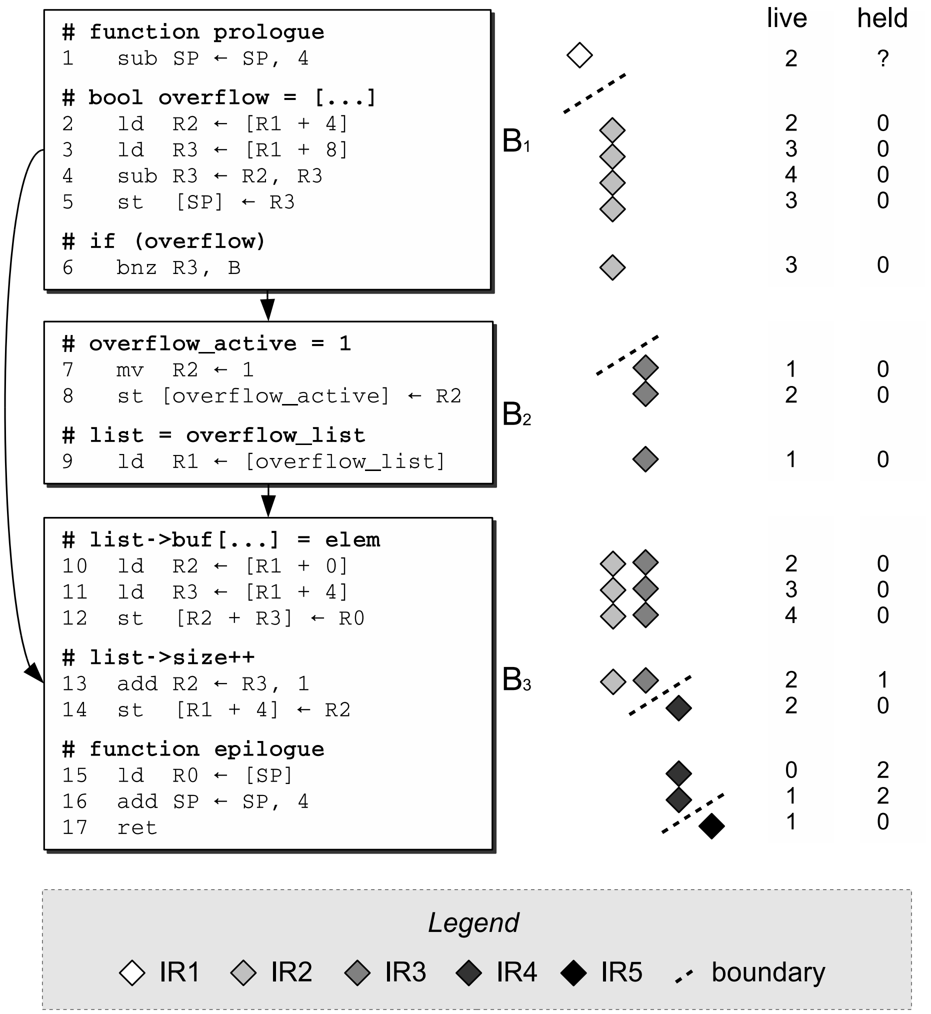
```
mv  R2 ← R1
```

```
# list->buf[...] = elem
st  [SP + 4] ← R2
ld  R3 ← [R2 + 0]
ld  R2 ← [R2 + 4]
st  [R2 + R3] ← R0
# list->size++
add R3 ← R2, 1
ld  R2 ← [SP + 4]
st  [R2 + 4] ← R3
# function epilogue
ld  R0 ← [SP]
add SP ← SP, 8
ret
```

Figure 4.5: Compiler output with code generation strategy Max-Size.

figure shows two columns with the column headers “live” and “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 registers immediately before each instruction that are dead but not re-usable because they are live at a containing region’s idempotence boundary. These “held” registers are those that contribute register pressure due to idempotence.

Example 4.1 (Strategy Max-Size): The result of compiling for strategy Max-Size closely resembles the output of the static analysis presented in Section 3.1.2, since the sole responsibility of the



```
# function prologue
1   sub SP ← SP, 4
# bool overflow = [...]
2   ld  R2 ← [R1 + 4]
3   ld  R3 ← [R1 + 8]
4   sub R3 ← R2, R3
5   st  [SP] ← R3
# if (overflow)
6   bnz R3, B
```

```
# overflow_active = 1
7   mv  R2 ← 1
8   st [overflow_active] ← R2
# list = overflow_list
9   ld  R1 ← [overflow_list]
```

```
# list->buf[...] = elem
10  ld  R2 ← [R1 + 0]
11  ld  R3 ← [R1 + 4]
12  st  [R2 + R3] ← R0
# list->size++
13  add R2 ← R3, 1
14  st  [R1 + 4] ← R2
# function epilogue
15  ld  R0 ← [SP]
16  add SP ← SP, 4
17  ret
```

Figure 4.6: Compiling for strategy Min-Pressure.

compiler given this compilation strategy is simply to preserve live state across the semantically idempotent regions identified in that analysis.

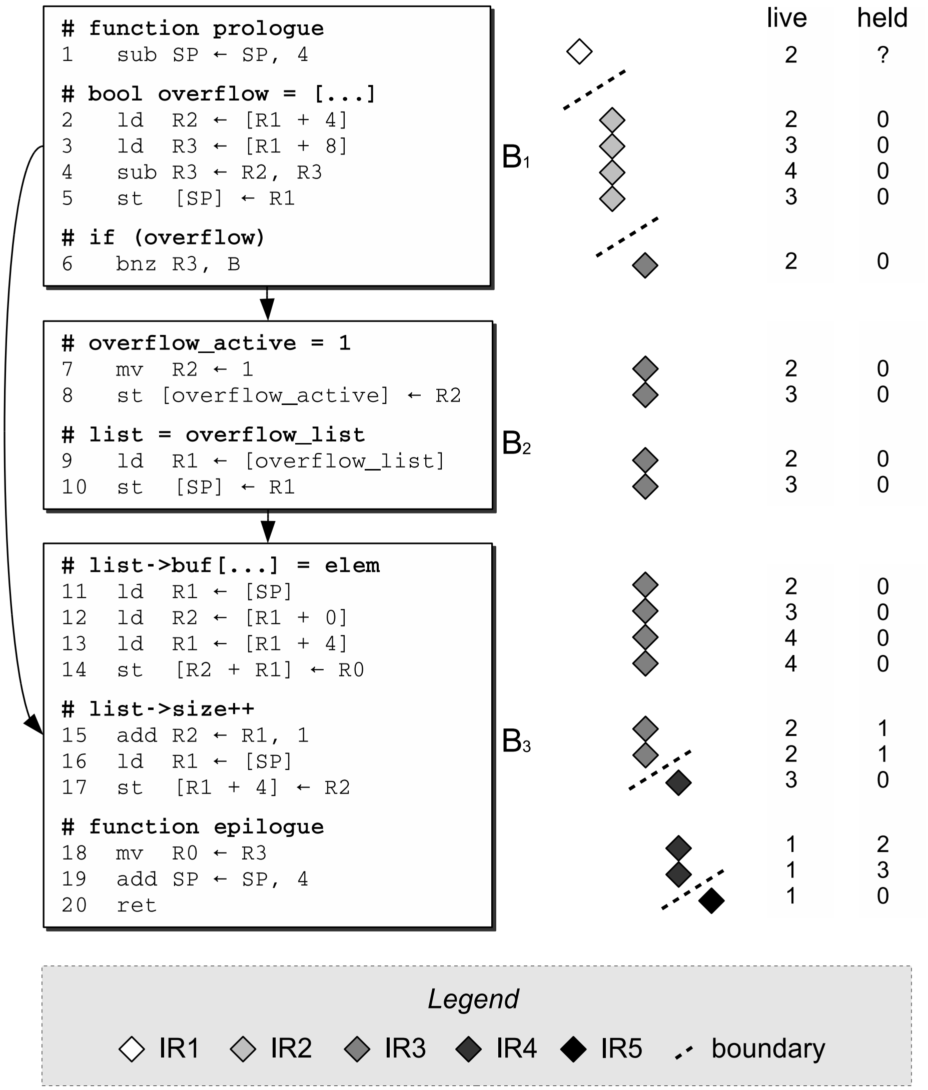
Figure 4.5 shows the result. The cut placed before S_9 in Figure 3.3 translates to a cut before the store instruction on line 18. The compiler preserves the function argument `list` that is live at the entry of region IR2, initially in `R1`, by re-assigning it to `R2` in lines 9 and 11. To accomodate the increase in register pressure in block B_4 where `R1` is held (dead but unavailable), `R2` is spilled on line 12 and then re-loaded on line 17. In total, 21 instructions are generated by the compiler.

Example 4.2 (Strategy Min-Pressure): Strategy Min-Pressure attempts to minimize register pressure by sub-dividing the semantically idempotent regions into smaller regions when idempotence-induced register pressure is particularly high. It does this specifically by examining the entry point of each basic block and placing an idempotence boundary at that point if the number of held registers exceeds the threshold value.

The algorithm examines the entry point of basic blocks because that is where held registers typically emerge as a result of control divergence. This can be seen examining the output of strategy MAX-SIZE in Figure 4.5. At the start of B_2 register R1 is a held register because it is dead along that control flow path. Given a pressure threshold value less than 2, the algorithm places a region boundary at the very start of B_2 as shown in Figure 4.6. The register pressure exerted by R0 along that control flow path is 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, a reduction of 4 instructions over strategy MAX-SIZE. This total number of instructions is in fact ideal—this code would also contain 17 instructions if idempotence was not considered.

Example 4.3 (Strategy MIN-RECOVERY): Strategy MIN-RECOVERY minimizes the recovery re-execution cost by placing region boundaries close to likely points of failure. For the specific case of branch misprediction recovery, it places region boundaries immediately before the site of low-confidence branches, such as the branch instruction on line 6 of Figure 4.5. The result is shown in Figure 4.7.

Unfortunately, the effect of placing region boundaries at the end of basic blocks under this strategy (in contrast to at the beginning of basic blocks under strategy MIN-PRESSURE) aggravates the register pressure effects of idempotence due to control flow divergence. Figure 4.7 shows this specifically for the branch condition value stored in R3. Immediately after the branch, this value is effectively dead—it is not used again until the return at the very end of the function and hence it is a great candidate for spilling—however, the register is not profitably spilled because its value is live at the start of region IR3 and thus spilling it would require a spill and a reload immediately before and after the start of IR3. In the end, the compiler determines that the value is best pinned down throughout IR3, along both control flow paths, forcing a spill and refill of register R1 instead. This way, the total number of instructions is 20. While this is 3 instructions more than than strategy MIN-PRESSURE, spilling R3 instead of R1 would produce code identical to the MAX-SIZE case of



```
# function prologue
sub SP ← SP, 4

# bool overflow = [...]
ld  R2 ← [R1 + 4]
ld  R3 ← [R1 + 8]
sub R3 ← R2, R3
st  [SP] ← R1

# if (overflow)
bnz R3, B

# overflow_active = 1
mv  R2 ← 1
st [overflow_active] ← R2

# list = overflow_list
ld  R1 ← [overflow_list]
st  [SP] ← R1

# list->buf[...] = elem
ld  R1 ← [SP]
ld  R2 ← [R1 + 0]
ld  R1 ← [R1 + 4]
st  [R2 + R1] ← R0

# list->size++
add R2 ← R1, 1
ld  R1 ← [SP]
st  [R1 + 4] ← R2

# function epilogue
mv  R0 ← R3
add SP ← SP, 4
ret
```

Figure 4.7: Compiling for strategy Min-Recovery.

Figure 4.5, which has even more instructions with 21 instructions in total.

These three examples illustrate how control flow divergence tends to aggravate register pressure for the strategies Max-Size and Min-Recovery, and how strategy Min-Pressure tends to alleviate this pressure. It also shows that the compiler-induced overheads for strategy Max-Size can be high when region sizes are necessarily small, as is the case for small functions such as the example function.

4.4 Summary and Conclusions

This chapter presented three different code generation strategies to tailor idempotent region sizes for factors relating to (a) the inherent degree of idempotence in applications and (b) the effects of using idempotence to recover from different types of execution failures. It also presented two algorithms—one for contextual idempotence and one for architectural idempotence—for preserving the idempotence property of semantically idempotent regions during code generation using the concept of *shadow intervals*. The contextual idempotence algorithm is simpler than the one for architectural idempotence, but their computational complexity is effectively the same—effectively linear with respect to the size of the function.

Finally, several examples were shown to demonstrate the code generation flow when compiling for idempotence, and to illustrate the trade-offs in compiling for the three different code generation strategies Max-Size, Min-Pressure, and Min-Recovery. While Max-Size produces the largest idempotent regions, Min-Pressure is able to reduce the compiler-induced runtime overhead by judiciously sub-dividing regions when semantically idempotent region sizes are small and register pressure is high. Finally, Min-Recovery sacrifices region size and suffers high runtime overhead by aggravating control divergence effects, but successfully reduces the recovery re-execution time to support branch mis-prediction recovery.

5 Compiler Evaluation

This chapter evaluates the results of the static analysis algorithm presented in Chapter 3 and the code generation strategies presented in Chapter 4. Section 5.1 presents the experimental methodology and results are subsequently presented in three parts. First, Section 5.2 presents results for the static analysis described in Chapter 3. Second, Section 5.3 presents results for the code generation flow of Chapter 4. Third, Section 5.4 presents results on ISA sensitivity, comparing performance across two different ISAs, x86 and ARM, and across compilations varying the number of available general purpose registers. Finally, Section 5.5 presents a summary and conclusions.

5.1 Experimental Method

Below we present the compiler, benchmarks, measurements, and sampling method used in the evaluation of this chapter.

Compiler

The static analysis and code generation algorithms from Chapters 3 and 4 are implemented on top of a modified version of LLVM 3.0 [62]. Source code is compiled to LLVM IR using GCC with the LLVM DragonEgg plug-in [1]. The static analysis is performed on the IR, and the code generator takes the output of this analysis and forms the regions as the IR is gradually lowered to machine code. The compiler targets both the x86 (x86-64) and ARM (ARMv7) instruction sets.

Binary version	Generated using...
ORIGINAL	Regular optimized LLVM compiler flow
MAX-SIZE	Strategy MAX-SIZE
MIN-PRESSURE-5	Strategy MIN-PRESSURE with threshold value 5
MIN-PRESSURE-2	Strategy MIN-PRESSURE with threshold value 2
MIN-RECOVERY	Strategy MIN-RECOVERY cutting before every conditional branch

Table 5.1: Binary versions and how they are generated.

Benchmarks

For benchmarks, we consider the same three benchmark suites as in the analysis of Chapter 2: SPEC 2006 [99], a suite targeted at conventional single-threaded workloads, PARSEC [16], a suite targeted at emerging multi-threaded workloads, and Parboil [100], a suite targeted at massively parallel GPU-style workloads written for CPUs. Using our compiler, the benchmarks are compiled with full optimizations (-O3, etc.) to the five different binary versions shown in Table 5.1. These different binary versions are used in our evaluation as described below.

Measurements

Section 5.2 evaluates the static analysis algorithm of Chapter 3 by measuring the sizes of the idempotent regions formed by the algorithm. Specifically, we use Pin [65] to measure the idempotent path lengths that execute through the idempotent regions in terms of the number of x86-64 instructions for binary version MAX-SIZE. For specific cases where the region formation falls short of what is possible in the ideal case (comparing against the dynamic empirical measurements from Chapter 2) we analyze and measure the extent to which more sophisticated compiler analysis or source code restructuring code improve the algorithm's performance.

Next, Section 5.3 evaluates the effectiveness of the code generation strategies developed in Chapter 4 considering all binary versions in Table 5.1[1]. Here, we measure performance degradation in terms of increase in dynamic instruction count, again using Pin. For justification, Figure 5.1 plots the increase in cycle count as a result of increase in dynamic instruction count compiling for MAX-SIZE and simulating a wide range of benchmarks for 10 billion instructions using the gem5

[1] As explained in Section 4.1, a PRESSURE threshold value of x implies that a region is cut if $x/10$ of the "heaviest" live or held intervals at a given point in the region are held.

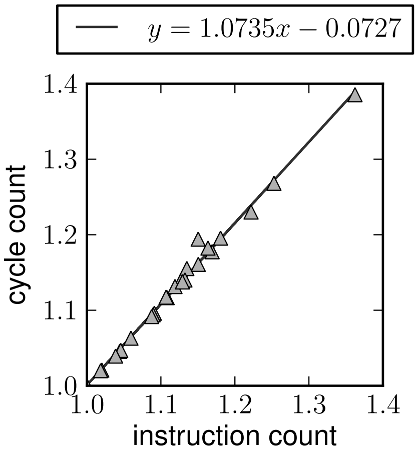


Figure 5.1: Mean idempotent path lengths.

simulator [17]. As shown, increase in cycle count correlates very closely with increase in dynamic instruction count. While the processor core modeled in the simulator is simple—a two-issue in-order core that can complete most instructions in a single cycle—this core design is representative of the types of architectures considered later in Chapter 6, while dynamic instruction count furthermore provides an architecture-neutral platform for compiler evaluation.

Finally, Section 5.4 evaluates ISA senstivity by isolating ISA-specific sources of impact between the x86 and ARM instruction sets. For this analysis, gem5 [17] is used to gather the same measurements for the ARMv7 instruction set as are gathered for x86-64 using Pin.

Sampling Method

To account for the differences in instruction count between the different ISAs and binary versions, simulation/monitoring time in Pin (x86-64) and gem5 (ARMv7) is measured in terms of the number of functions executed, which is constant between all versions compiled for the same instruction set. Initially, all benchmarks execute unmonitored for the number of function calls needed to execute at least 10 billion instructions on the Original binary. Execution is then monitored for the number of function calls needed to execute 10 billion additional instructions on the Original binary. 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.

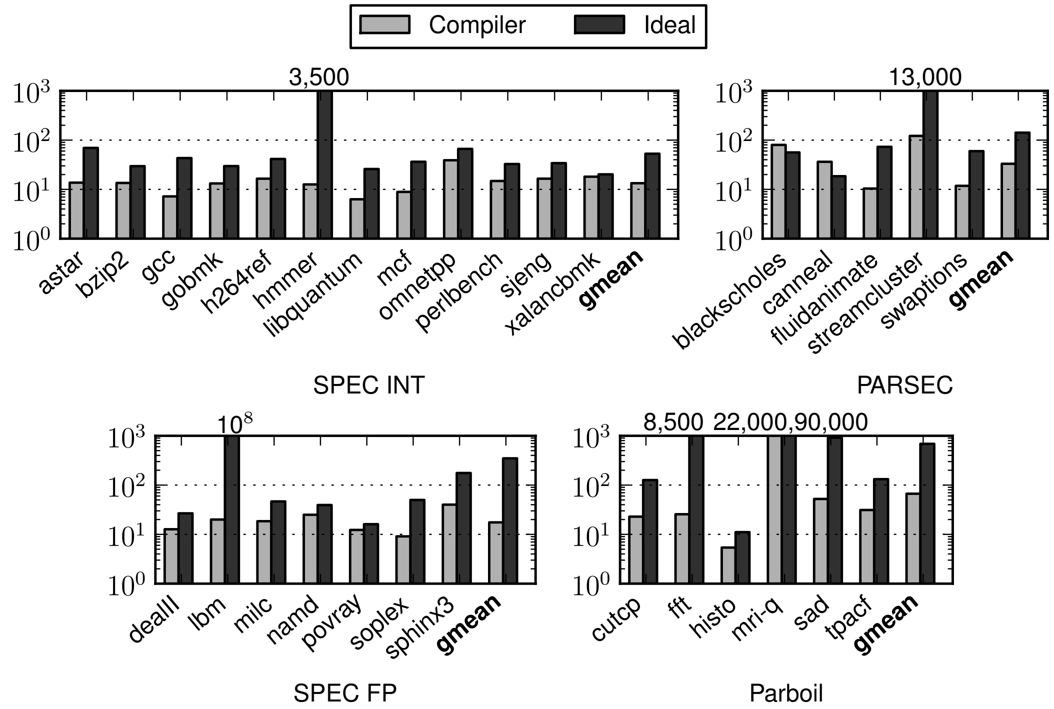


Figure 5.2: Mean idempotent path lengths.

5.2 Static Analysis Results

A key characterstic of the static analysis algorithm of Chapter 3 is the size of the idempotent regions it produces. In particular, we care about the length of the paths that dynamically execute through these idempotent regions at runtime, since those are the distances that are actually covered dynamically executing real programs. In general, longer path lengths are better because they allow for the greatest flexibility: when shorter path lengths are desirable to minimize re-execution costs or reduce register pressure, the static analysis does not get in the way of sub-dividing paths into smaller sub-paths if needed.

Unmodified Benchmark Path Lengths

Figure 5.2 evaluates how close the static analysis algorithm is able to approximate the ideal idempotent path lengths measured in the empirical analysis study of Chapter 2 for unmodified benchmarks. It shows the (arithmetic) mean idempotent path length produced by the static analysis (Compiler)

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
Compiler	12.4	16.5	32.1	65.8	**21.6**
Ideal	52.0	347.9	140.7	686.8	**160.2**
Ideal excluding outliers	35.4	42.4	45.0	415.0	**61.5**

Table 5.2: Geometric mean idempotent path lengths.

compared to configuration Unconstrained from Section 2.4.1 (Ideal)[2]. Geometric mean values across benchmarks are presented in Table 5.2. For Compiler the path length is the distance between idempotence boundaries in binary version Max-Size.

Table 5.2 shows numerically that the overall difference between Compiler and Ideal is quite large—roughly 7x (20.4 vs. 137.5). Four benchmarks have much longer path lengths (over 100x) in the ideal case: hmmer (3,500 vs. 11.6), lbm (100,000,000 vs. 19.0), streamcluster (13,000 vs. 120.7), and fft (8,500 vs. 24.6). In all cases, these large discrepancies are primarily due to limited aliasing information in the compiler (more details in the next sub-section). With more sophisticated alias analysis or small modifications to the source code that improve aliasing knowledge, longer path lengths can be achieved. If we ignore these four outliers, the difference narrows to roughly 3x comparing the top and bottom rows of Table 5.2.

Figure 5.3 shows the median idempotent path lengths across the benchmark suites with the 25th to 75th percentile range shown using vertical error bars. Across the vast majority of benchmarks, most idempotent path lengths are consistently in the range of 8-16 instructions. Some benchmarks, such as mri-q and blackscholes have median path lengths over 100 instructions. However, these types of benchmarks are not common. The following observation summarizes our analysis of Figures 5.2 and 5.3:

Observation 5.1. *Median idempotent path lengths produced by the static analysis algorithm are typically 8-16 instructions long, which is typically 3-8x lower than what can be achieved in the ideal case. Compute-intensive applications tend to have more variance, with a heavier tail, suggesting that these applications present the greatest flexibility and opportunity.*

[2] In the case of blackscholes and canneal, the Compiler measurement is higher than the Ideal measurement. This is due to inlined assembly synchronization that ends a region in the Ideal case but not in the Compiler case; the compiler is not programmed to detect inlined assembly interactions.

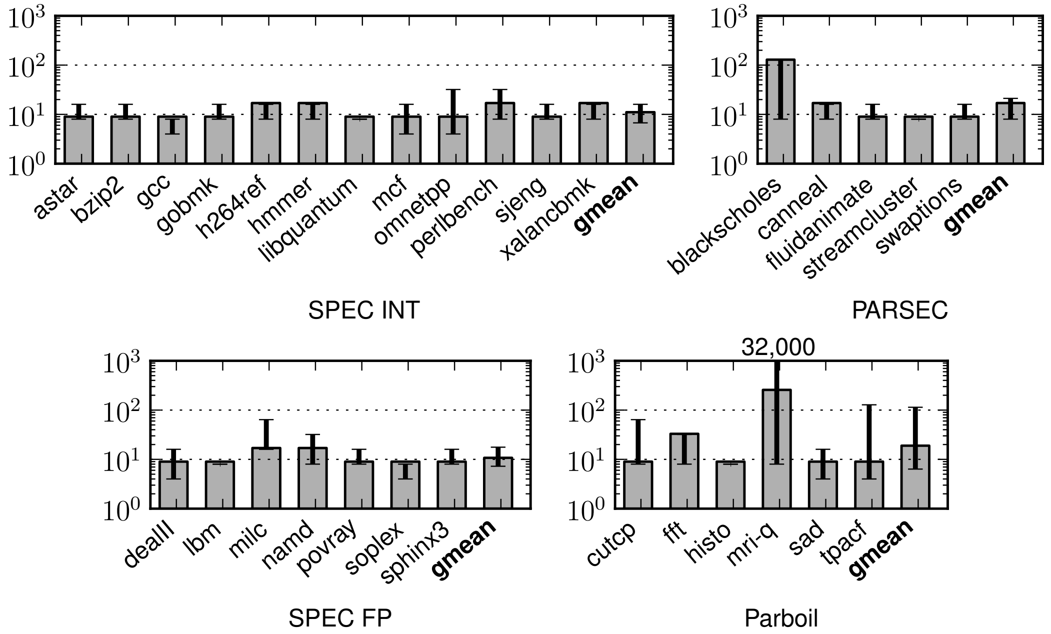


Figure 5.3: Idempotent path lengths for Compiler with unmodified benchmarks. Bars plot median values and the error bars plot the 25th to 75th percentile range.

Analysis of Limiting Factors

For several benchmarks, idempotent path lengths are much shorter than they could be. Limited aliasing information in the compiler is in part responsible for the reduced path lengths for specific benchmarks such as hmmer, lbm, and fft. However, there are at least two other reasons why path lengths may be artificially small: overlooked loop optimization opportunities and large storage arrays. Additionally, as a fourth reason, our analysis is only intra-procedural while some inter-procedural information can often improve idempotent path lengths. In the discussion that follows, these four total problem sources are identified using the labels Aliasing, Loop-Opt, Arrays, and Scope respectively.

Aliasing: With limited aliasing information, a pair of load and store instructions may be believed to potentially 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

Before:

```
for (int i = 0; i < end; ++i) {
  *result += compute(i);
}
```

After:

```
float accum = 0.0;
for (int i = 0; i < end; ++i) {
  accum += compute(i);
}
*result += accum;
```

Figure 5.4: Example showing scalarization of a loop accumulation variable.

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 loop iteration (i.e. the aliasing dependence relationship is strictly a flow dependence). Such loop-level aliasing information is a common feature of auto-parallelizing compilers [71], and although forthcoming in LLVM [26], this information is not supported in LLVM at the time of writing. For data-parallel applications in particular, the lack of loop-dependence information results in pessimistic aliasing assumptions that limit idempotent path lengths.

Loop-Opt: Certain loop optimizations, such as loop fission/fusion, loop interchange, loop peeling, loop unrolling, scalarization, etc. allow the construction of larger idempotent regions because they allow clobber antidependences to span longer distances. Scalarization is an example of an optimization that can be particularly helpful. The optimization is illustrated by Figure 5.4 and involves a memory variable being temporarily held and updated inside a register before being written back to memory. The version of LLVM we used (in combination with the GCC DragonEgg plug-in) does not provide automatic support for this optimization, although it is legal even under strict sequential consistency semantics provided that there are no other memory operations inside the loop.

Scalarization of the loop accumulation variable in Figure 5.4 allows the non-local memory antidependence inside the loop to be moved outside the loop, consequently allowing an idempotent path to contain the entire loop execution. In contrast, without scalarization, at least *two* idempotent paths must form per loop iteration to allow proper allocation of the loop-dependent variable `i` as described in Section 3.3.2. The latter can significantly degrade code generation flexibility and hence performance suffers as well.

Arrays: Chapter 3 (Section 3.2) showed how non-clobber antidependences (write-read-write patterns) on non-local memory could be eliminated by using local storage to hold the result of the first write and replacing the read to use the local storage. This technique can be impractical, however, particularly 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 local stack. In addition to stack memory often being bounded, the state duplication can reduce cache locality and hurt performance.

To support writes to such large or potentially unbounded arrays efficiently, our compiler does not duplicate large structures or arrays in local memory, and hence must conservatively cut the potentially non-clobber antidependences that they create on the assumption that their occurrence is rare (see Appendix A, Section **??**). However, there is a specific pattern, the *initialization-accumulation* pattern, that is common among certain data-parallel applications 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.

To support the initialization-accumulation pattern and similar patterns, the compiler analysis can be augmented to identify and ignore potentially non-clobber antidependences in the antidependence cutting phase of the static analysis, and then handle these antidependences in a separate post-cutting phase "as-needed" in the same manner that potentially non-clobber self-dependent pseudoregister dependences are handled as a post-pass (see Section 3.3.2).

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 current idempotence analysis furthermore forces cuts at function boundaries to contain inter-procedural effects.

Before our static analysis, a simple top-down interprocedural analysis that identifies whether a function is wholly idempotent is relatively cheap and could serve as an input to the static

Benchmark	Limiting Factor(s)	Length Before	Length After
blackscholes	Aliasing, Scope	78.9	>10,000,000
canneal	Scope	35.3	187.3
fluidanimate	Arrays, Loop-Opt, Scope	9.4	>10,000,000
streamcluster	Aliasing	120.7	4,928
swaptions	Aliasing, Arrays	10.8	210,674
cutcp	Loop-Opt	21.9	612.4
fft	Aliasing	24.7	2,450
histo	Arrays, Scope	4.4	4,640,000
mri-q	—	22,100	22,100
sad	Aliasing	51.3	90,000
tpacf	Arrays, Scope	30.2	107,000

Table 5.3: The benchmarks from the PARSEC and Parboil benchmark suites, their limiting factors, and the mean path lengths produced by the static analysis before and after addressing these limiting factors.

analysis. With such simple inter-procedural information, along with the capability to save the stack pointer value along with the program counter, idempotence could be used to safely recover back up the program call stack in a number of cases. Future work Section 8.2 covers in more detail on how to cheaply integrate inter-procedural information.

Modified Benchmark Path Lengths

We now evaluate possible idempotent path lengths given the capability to overcome each of the four limiting factors identified above. We consider as a case study the PARSEC and Parboil benchmarks, which are relatively small and well-contained, and thus suitable for manual analysis and modification. They are also limited by each of the four factors in the manner indicated by the second column of Table 5.3.

To address the Aliasing problem, we assist the compiler in identifying no-alias memory relationships by manually providing source code annotations using the C *restrict* keyword and/or performing code refactoring. To address the Loop-Opt problem, we manually modify the program source code for scalarization of non-local memory variables. To address the Arrays problem, we manually modify the benchmark binary to simulate the behavior of the compiler analysis extension we described. Finally, to address the Scope problem, we perform manual inlining.

Table 5.3 shows that very large path length improvements can be achieved with these modifications. In all cases, the mean path length is at least in the hundreds of instructions and in the vast majority of cases path lengths of many thousands of instructions are possible. In all cases, path lengths can be effectively arbitrarily reduced employing loop blocking and other similar techniques. Unfortunately, LLVM does not yet have the features of an automatic parallelizing compiler to realize the above manually implemented transformations easily. Hence, automating them through a compiler or other code rewriting tool remains a topic for future work. The following observation summarizes our analysis and modification results:

Observation 5.2. *Factors including limited aliasing information, overlooked loop optimizations, array initialization patterns, and inter-procedural scoping effects often obscure the inherent idempotence of programs and limit idempotent path lengths. With the capability to account for and mitigate these factors, many benchmarks—particularly "emerging" benchmarks such as those from the PARSEC and Parboil suites—have large regions of code that are inherently idempotent.*

5.3 Code Generation Results

During code generation, forcing the register allocator to preserve live-in state across an idempotent region adds overhead because the allocator may not re-use live-in register or stack memory resources. Code generation strategy Min-Pressure attempts to minimize this source of overhead. Additionally, both code generation strategies Min-Pressure and Min-Recovery trade off path length to reduce register pressure and re-execution time costs, respectively.

In this section, we first consider register pressure overheads in compiling specifically for each of the Max-Size, Min-Pressure-5, Min-Pressure-2, and Min-Recovery binary versions, considering the effects of both architectural idempotence and contextual idempotence. We then explore the impact of these strategies on idempotent path lengths. Finally, we consider the overhead effects of the modifications to the PARSEC and Parboil benchmarks described in Section 5.2.

Unmodified Benchmark Overheads

Figures 5.5 and 5.6 report the increase in dynamic instruction count executing the four binary versions compiling for architectural idempotence (Figure 5.5) and contextual idempotence (Fig-

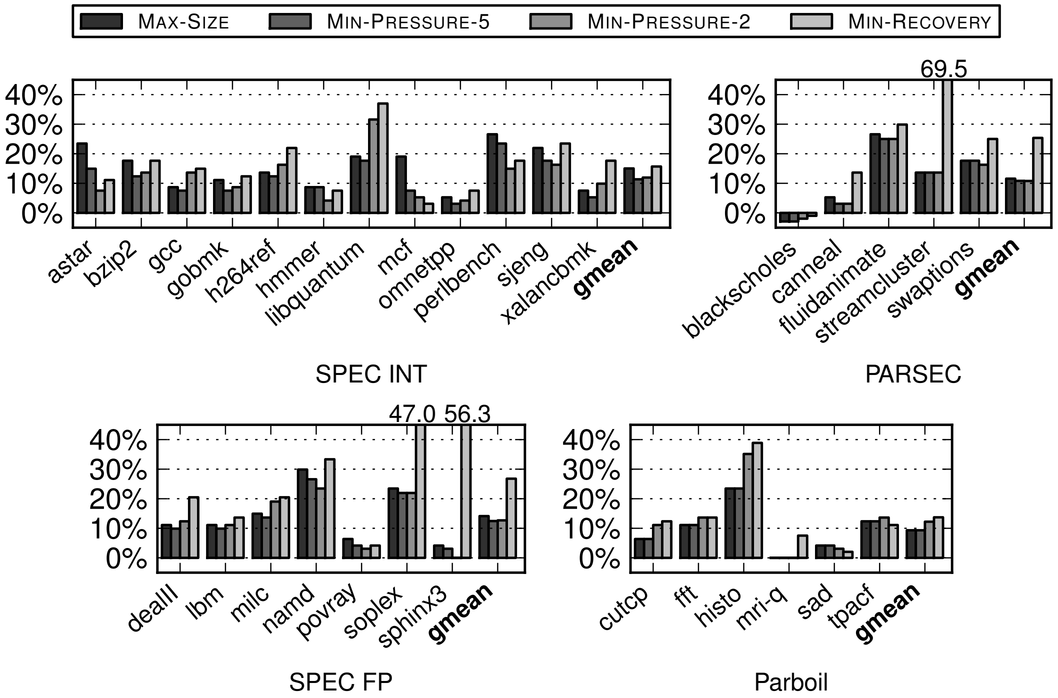


Figure 5.5: Instruction count overheads compiling for architectural idempotence.

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
Max-Size	15.0	14.1	11.6	9.3	**13.1**
Min-Pressure-5	11.4	12.5	10.8	9.3	**11.1**
Min-Pressure-2	11.9	12.7	10.8	12.2	**12.0**
Min-Recovery	15.7	26.8	25.4	13.7	**19.4**

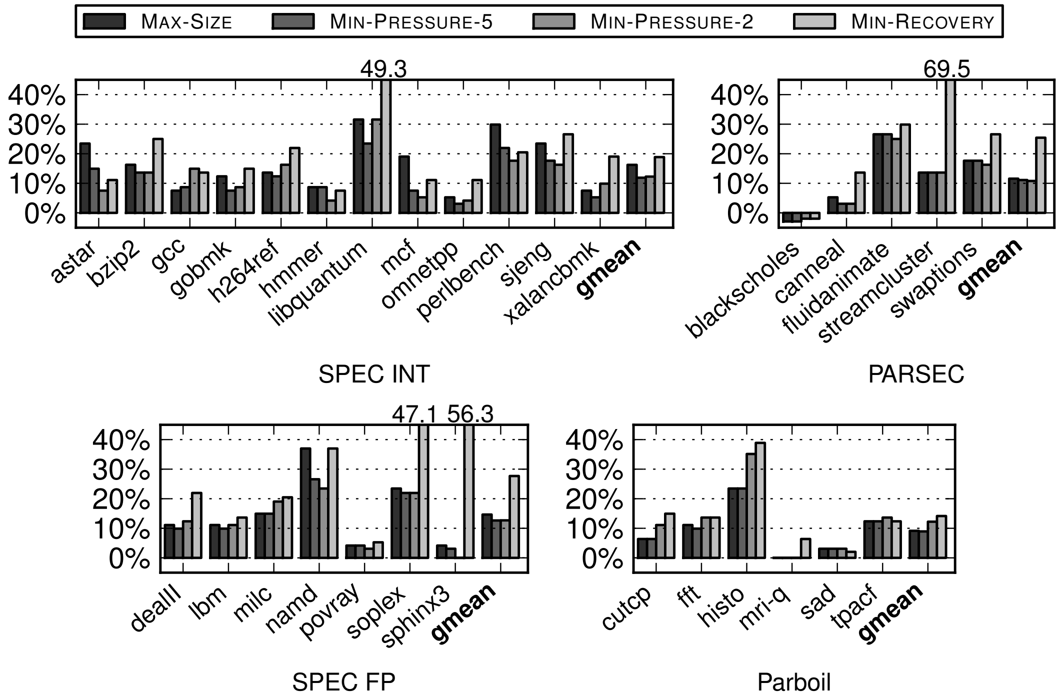
Table 5.4: Geometric mean percentage overheads under architectural idempotence.

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
Max-Size	16.2	14.7	11.6	9.1	**13.6**
Min-Pressure-5	11.9	12.6	11.1	8.9	**11.3**
Min-Pressure-2	12.3	12.7	10.8	12.2	**12.1**
Min-Recovery	18.9	27.7	25.4	14.2	**21.0**

Table 5.5: Geometric mean percentage overheads under contextual idempotence.

ure 5.6). Tables 5.4 and 5.5 show the geometric mean values across benchmarks and overall as well. A first observation regarding the difference between architectural and contextual idempotence immediately presents itself:

Observation 5.3. *Dynamic instruction count overheads are largely unaffected by the distinction between*



SPEC INT

PARSEC

SPEC FP

Parboil

Figure 5.6: Instruction count overheads compiling for contextual idempotence.

architectural and contextual idempotence.

The overheads of strategy Max-Size for architectural idempotence range from roughly 9 to 15%, while for contextual idempotence the range is only slightly higher from roughly 9 to 16%. The differences for the other three strategies are similarly not significant. The intuitive reason for this small difference is captured in the code generation example considered in Section 4.3. In particular, architectural idempotence allows statically live-in but dynamically dead registers to be re-used . However, the occurrence of this statically-live and dynamically-dead condition—and the opportunity to take advantage of it—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).

A second observation regarding the difference between strategies Max-Size and Min-Pressure is the following:

Observation 5.4. *Strategy* Min-Pressure *reduces register pressure overheads relative to* Max-Size*, but* *not substantially, even across a range of pressure threshold values.*

Here, while strategy Max-Size yields runtime overheads of roughly 9-16%, Min-Pressure-5 and Min-Pressure-2 yield reduced overheads of 9-13% and 10-13%, respectively—a significant 3% reduction at the high end. However, the differences between Min-Pressure-5 and Min-Pressure-2 (and neighboring threshold values; not shown but independently measured) are not substantial.

Finally, we make a third observation regarding the difference between strategies Min-Recovery and the other strategies:

Observation 5.5. *Strategy* Min-Recovery *incurs high register pressure overheads relative to the other strategies.*

Strategy Min-Recovery yields high, 13-28%, overheads. The reason for this is primarily that placing region boundaries immediately before conditional branches aggravates the effects of control divergence on register pressure (as also identified in the code generation example from Section 4.3). However, this alone does not explain why, in the specific cases of libquantum, soplex, sphinx3, and streamcluster, the overheads of Min-Recovery are *really* high—over 45%. In these cases the bodies of inner loops that are otherwise idempotent are being cut at the branch points. This cut forces a second cut in the loop for the reasons given in Section 3.3.2. These two cuts together significantly expand the register pressure in the loop and cause very high overheads. Incidentally, this pair of cuts in a loop is also what causes the high overheads of Parboil benchmark histo. However, for that benchmark the second loop cut is required because of the clobber antidependence on the memory variable (the histogram bucket) being conditionally updated inside the loop.

While, for Min-Recovery, the overheads of architectural idempotence are not as high for contextual idempotence, architectural idempotence requires that the hardware contain the effects of control flow divergence over pseudoregister state, and hence it is not a useful idempotence model for the purposes of branch misprediction recovery.

Path Lengths

We now briefly examine the effects of the code generation strategies on idempotent path lengths. Figure 5.7 show the mean idempotent path lengths and geometric mean values are given in Table 5.6.

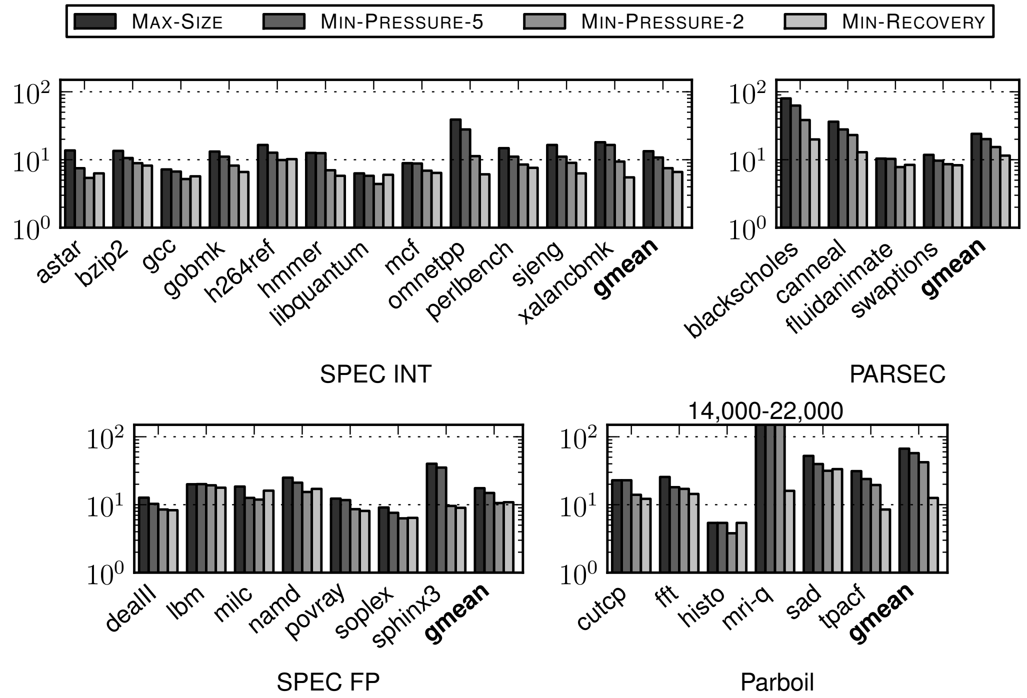


Figure 5.7: Mean idempotent path lengths.

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
Max-Size	12.3	16.5	23.1	65.8	**20.4**
Min-Pressure-5	9.8	13.9	19.1	56.4	**16.8**
Min-Pressure-2	6.5	9.6	14.4	41.2	**11.7**
Min-Recovery	5.6	9.9	10.5	11.6	**8.1**

Table 5.6: Geometric mean idempotent path lengths.

Table 5.6 shows that strategy Max-Size produces the largest idempotent path lengths (these are the same lengths as in Section 5.6). Binary versions Min-Pressure-5 and Min-Pressure-2 results in shorter path lengths by roughly 10-25% and 35-50%, respectively. Strategy Min-Recovery reduces path length the most, by 45-80% overall.

Modified Benchmark Overheads

Finally, we consider the register pressure overheads resulting from the benchmark modifications described in Section 5.2. Table 5.7 shows the decrease in dynamic instruction count compiling with strategy Max-Size before and after modification. In most cases, runtime overhead is effectively

Benchmark	% Overhead Before	% Overhead After
blackscholes	-2.9	-0.05
canneal	5.3	1.3
fluidanimate	26.6	-0.62
streamcluster	13.6	0.00
swaptions	17.6	0.00
cutcp	6.38	-0.01
fft	11.11	0.00
histo	23.46	0.00
mri-q	0.00	0.00
sad	4.17	0.00
tpacf	12.36	-0.02

Table 5.7: Percentage overheads for the six benchmarks from the Parboil benchmark suite before and after addressing the limiting factors identified in Section 5.2.

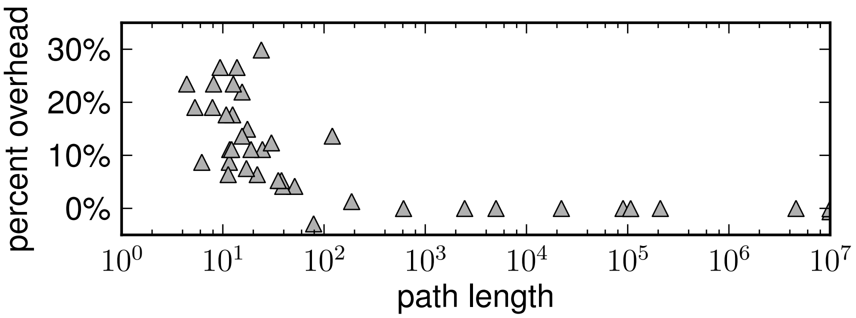


Figure 5.8: Idempotent path length plotted against dynamic instruction count overhead.

eliminated. The data strongly supports constructing larger idempotent regions to reduce runtime overheads. The reason is intuitive: larger regions allow amortizing the cost of preserving region live-in state over longer distances.

Observation 5.6. *The best way to reduce the runtime overhead of idempotent region construction is to enable the construction of idempotent regions with path lengths over 50 instruction.*

Figure 5.8 plots the correlation between path length and overhead for both the modified and unmodified benchmarks compiled with Max-Size. The data suggests that path lengths of 50 instructions and more are sufficient and yield negligible overhead overall; benchmarks omnetpp, blackscholes, canneal, sphinx3, and sad all have average path lengths in the range of 30-80 instruction under strategy Max-Size, 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 without modification. For this benchmark, the compiler compiles the most critical loop of the benchmark differently for idempotence, inserting one extra instruction under Max-Size compared to the original and growing the size of the loop from 7 instructions to 8 instructions. The reason is not clear—the extra instruction appears redundant. Indeed, comparing against the same code compiled for ARMv7 (more detail in Section 5.4), the loop is compiled without any extra instructions and the overall overhead becomes negligible (see Figure 5.9). We thus conclude that this difference is purely due to noise resulting from the way our code generation algorithms interact with the the compiler's register allocation flow.

Of course, we note that large idempotent regions (e.g. over 50 instructions) are inappropriate when failures are frequent (e.g. over 1% probability per instruction). In some scenarios, they also present problems relating to potential livelock. The issue of livelock is explored as an architecture-specific detail in Chapter 6.

5.4 ISA Sensitivity Results

Instruction sets vary in a number of ways. When compiling regions for idempotence, the following three ways in which they differ can be of significance:

Register-memory vs. register-register: Whether an instruction set is a RISC-like load/store instruction set or not can affect the performance overhead of idempotence particularly in terms of any resulting increase in the 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 by converting the instruction from a register-register instruction to a register-memory instruction. A load/store instruction set such as MIPS, SPARC, 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 the source operands with the result. In contrast, instruction sets such as MIPS, SPARC, or ARM instead implement "three-address" instructions where a third destination operand can be specified that can be made distinct from the source operand locations. Two-address instructions are self-antidependent and hence *not idempotent*. Therefore, when contained inside idempotent regions they must be preceded by some instruction that defines the overwritten source register before it is again re-defined in order to avoid a clobber dependence. When no such instruction already exists, a (logically redundant) move or copy instruction must be inserted to satisfy this requirement. This can artificially boost idempotent performance overheads particularly when idempotent region sizes are small. Three-address instructions 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 due to idempotence more easily by simply allocating more available registers rather than spilling to the function stack. However, as is the case without idempotence, more registers are only beneficial as long as they are 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 may exert extra register pressure.

Unfortunately, there is no practical way to evaluate the positive impact of register-to-memory instructions independent of the negative impact of two-address instructions given that no mainstream register-to-register two-address instruction set (nor register-to-memory three-address instruction set) exists. Instead, 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 (compiling for architectural idempotence using strategy Max-Size). Although not a perfectly controlled experiment, both instruction sets have the same number of general purpose integer registers (16) and we compile then with the same floating point support (x86-64 has 16 single-precision floating point registers with SSE2 and we compile ARMv7 to similarly use only 16 single-precision floating point registers, even though 32 are available, with NEON).

Figure 5.9 plots per-benchmark performance numbers and Table 5.8 presents overall geometric mean values. Overall, Table 5.8 indicates a lower geometric mean overhead for ARMv7, at 12.6%,

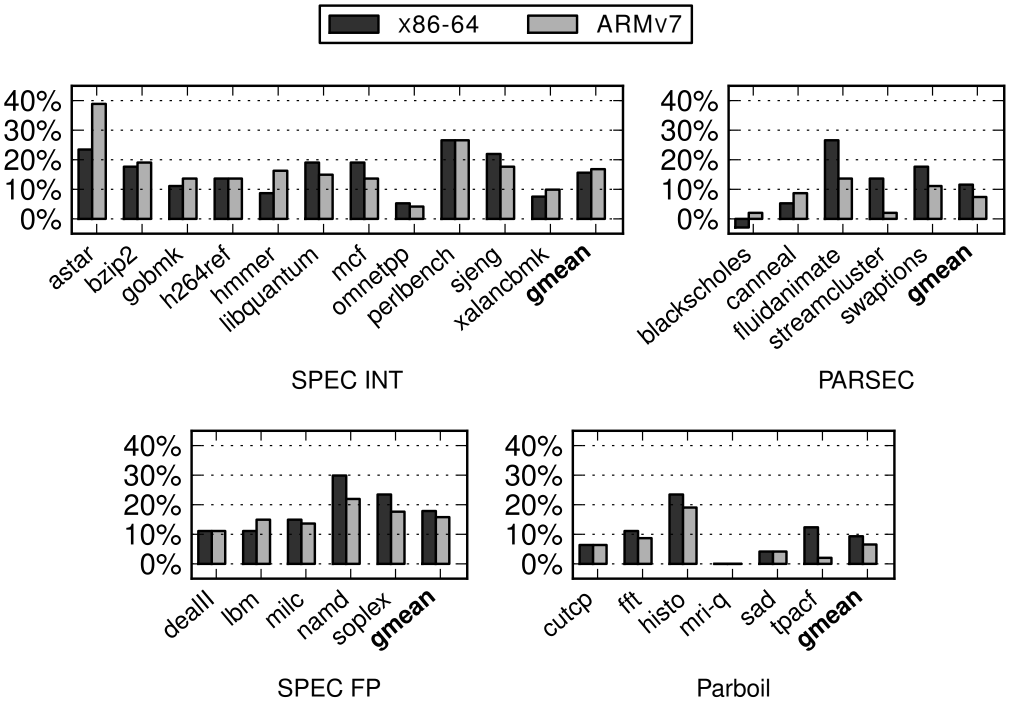
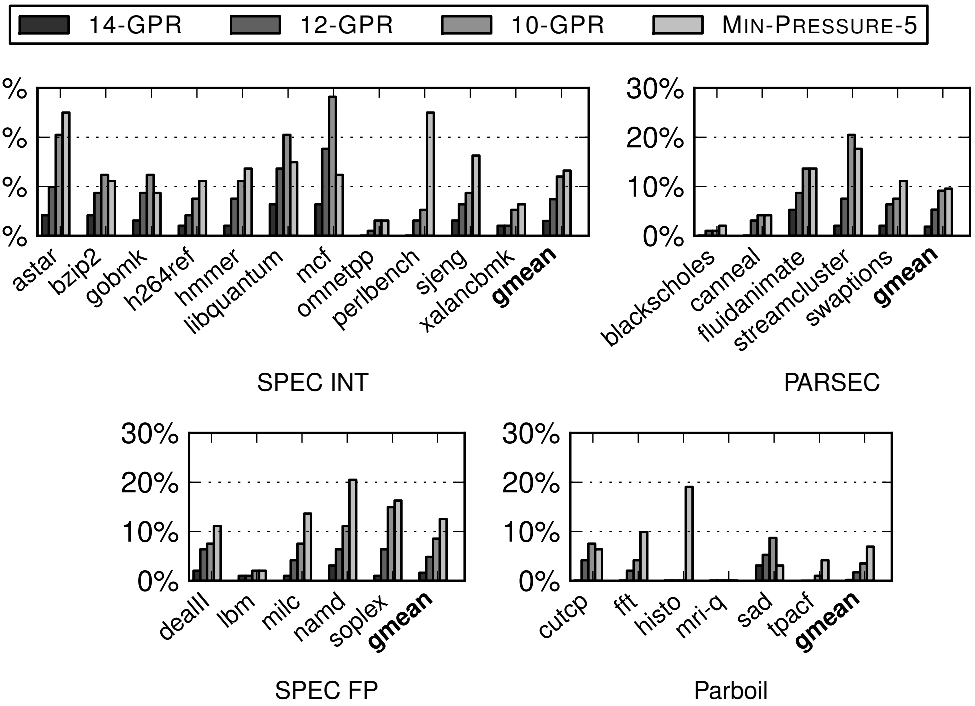


Figure 5.9: Instruction count percentage overhead for x86-64 vs. ARMv7.

	SPEC INT	SPEC FP	PARSEC	Parboil	**Overall**
x86-64	15.6	17.9	11.1	9.3	**13.9**
ARMv7	16.8	13.4	9.7	6.7	**12.6**

Table 5.8: Geometric mean percentage overhead for x86-64 vs. ARMv7.

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 performs better by a relatively large margin 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 performs slightly better 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, however, the differences are not strong—both architectures support the construction of idempotent regions with a comparable degree of overhead. Thus, we make the following summary observation regarding the memory-register and three-address instruction ISA effects:



SPEC INT

PARSEC

SPEC FP

Parboil

Figure 5.10: Performance overhead for ARMv7 assuming 10, 12, and 14 general purpose registers (GPRs) compared to Original (which has 16 GPRs) as the baseline. The overhead of strategy Min-Pressure-5 is shown for comparison.

Observation 5.7. *In general, memory-register support (in x86) and three-address instruction support (in ARM) do not appear to significantly impact performance in terms of raw instruction count, and intuitively as regions grow larger the effects of such ISA features approach insignificance.*

To compare the effect of the number of available registers, Figure 5.10 compares the increase in dynamic instruction for ARMv7 reducing the number of general purpose registers (GPRs) from 16 to 14, 12, and 10. Examining the data leads to the following observation:

Observation 5.8. *For integer benchmarks, the performance of having only 10 (removing 6 out of 16) general purpose integer registers roughly corresponds with the performance of code generation strategy* Min-Pressure-5*. The implicication 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.*

The effective 60% increase in register pressure due to idempotence seems remarkably high. However, it is fairly intuitive that idempotent region sizes in the range of 5 to 10 instructions have a roughly proportional number of live-in registers that must be preserved for idempotence.

5.5 Summary and Conclusions

This section evaluated an LLVM-based compiler implementation employing the static analysis and code generation algorithms of Chapters 3 and 4. Analysis of experimental data yielded eight distinct observations in total (Observations 5.1-5.8) summarized as follows:

Static Analysis: Across most benchmarks, the compiler static analysis produces idempotent region sizes that are typically around 10-20 instructions in size. As expected, the memory- and control-intensive SPEC benchmarks have significantly smaller idempotent region sizes than the more compute-intensive PARSEC and Parboil benchmarks. Program transformations that help to extract the inherent idempotence in applications, manually applied (although automatable), showed that, for several benchmarks, much larger idempotent region sizes are achievable.

Code Generation: The state preservation of small idempotent region sizes introduces runtime overheads in the range of 10-20% percent. While seemingly quite high, this overhead approaches zero in the limit as region size grows beyond a few tens of instructions. Distinguishing between architectural and contextual idempotence affects performance only minimally, and while greater differences arise distinguishing between code generation strategies Max-Size, Min-Pressure, and Min-Recovery, overall the differences are also not particularly significant.

ISA Sensitivity: Three ways in which the ISA might affect the overheads of idempotence-based compilation were identified. Of the three, the number of available registers was determined to be the most significant; naturally, more registers allows the register pressure effects of the idempotence state preservation to be reduced. Overall, idempotence was measured as increasing the register pressure of general purpose integer registers by approximately 60% when region sizes were constrained and register pressure effects were already initially high.

Based on these results, the evidence thus strongly suggests that the principal advantage of idempotence is not in how it simplifies the preservation of register state—as the 60% increase in register pressure over small regions clearly indicates—but rather in how it simplifies the preservation of memory state by allowing such state to be freely overwritten *without* preservation over large groups of instructions. Assuming recovery is infrequent, architectures designed with idempotence in mind thus might benefit most from programs with extractably large idempotent regions written in domain-specific or other high-level functional or declarative languages, since straight C/C++, while manageable, evidently imposes difficulties and requires advanced compiler technologies to enable low software overheads.

6 Architecture Design and Evaluation

Compiler-constructed idempotent regions have applications in computer architecture design, and this chapter considers three important application categories: Section 6.1 considers idempotence for efficient exception and context switching support on GPUs; Section 6.2 considers idempotence for out-of-order execution in general-purpose CPUs; and Section 6.3 considers idempotence for hardware fault recovery in emerging architectures. The three sections each present background, a design, and a high-level evaluation in support of each of the application scenarios using the compiler evaluated in Chapter 5. Finally, Section 6.4 presents a summary and conclusions.

6.1 General Exception Support in GPU Architectures

Since the introduction of fully programmable vertex shader hardware, GPU computing has made tremendous advances. To improve the effectiveness of GPUs as general-purpose computing devices, GPU programming models and architectures continue to evolve, and exception support—particularly demand-paged virtual memory support—appears a crucial next step to further expand their scope and usability. Unfortunately, traditional mechanisms to support exceptions and speculative execution are intrusive to GPU hardware design.

Legacy Challenges

Implementing general exception support on GPUs presents two key challenges. The discussion below discusses these two challenges in the context of exception support on CPUs, to elucidate how CPU mechanisms are problematic to apply directly to GPUs.

For CPUs, the problem of exception support was solved relatively early on [93, 95]. Instrumental at that time was the definition of *precise exception handling*, where an exception is handled precisely if, with respect to the excepting instruction, the exception is handled and the process resumed at a point consistent with the sequential architectural model [93]. With support for precise exceptions, all types of exceptions could be handled using a universal mechanism such as the re-order buffer. Unfortunately, precise exception support has historically been difficult to implement for architectures that execute parallel SIMD or vector instructions, where precise state with respect to an individual instruction is not natural to the hardware. High fan-out control signals to maintain sequential ordering in a vector pipeline are challenging to implement, and while buffering and register renaming approaches have been proposed [35, 61, 93], they are costly in terms of power, area, and/or performance. Hence, a key challenge is in supporting *consistent exception state* by exposing sequentially-ordered program state to an exception handler and enabling program restart from a self-consistent point in the program.

A second reason for the widespread adoption of precise exception support in CPUs was that it enabled support for demand paging in virtual memory systems: to overlap processor execution with the long latency of paging I/O, the state of a faulting process could be cleanly saved away and another process restored in its place. Simply borrowing techniques from the CPU space to implement context switching on GPUs, however, is difficult. In particular, saving GPU state and then context switching to another process while a page fault is handled imposes a monumental undertaking: while on a conventional CPU core a context switch requires little more than saving and restoring a few tens of registers, for a GPU it can require saving and restoring *hundreds of thousands* of registers. Thus, a second key challenge is supporting *efficient context switching* by minimizing the amount of state that must be saved and restored to switch among running processes.

Exception Support Using Idempotent Regions

The above two challenges of enabling general exception recovery on GPUs are fundamentally bounded by the ability to (i) restart from a consistent program state, and (ii) minimize the amount of program state to be preserved. When cast in these terms, idempotence provides a natural fit because it (i) allows restart from a consistent program state at configurable program points, and (ii) allows

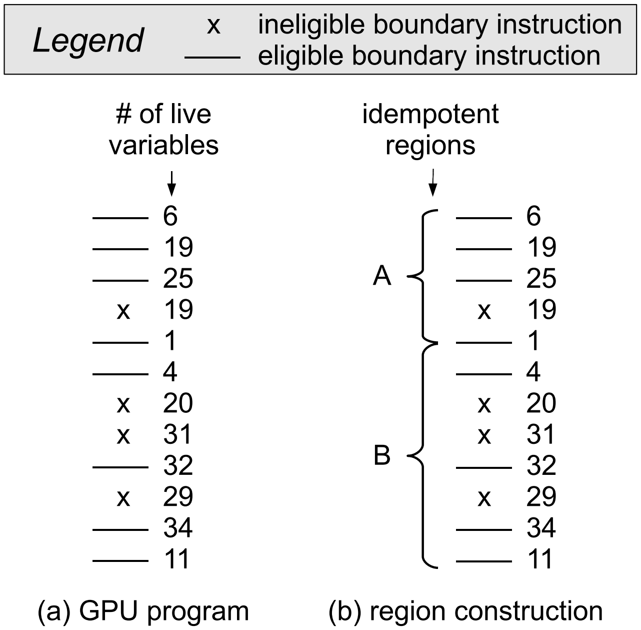
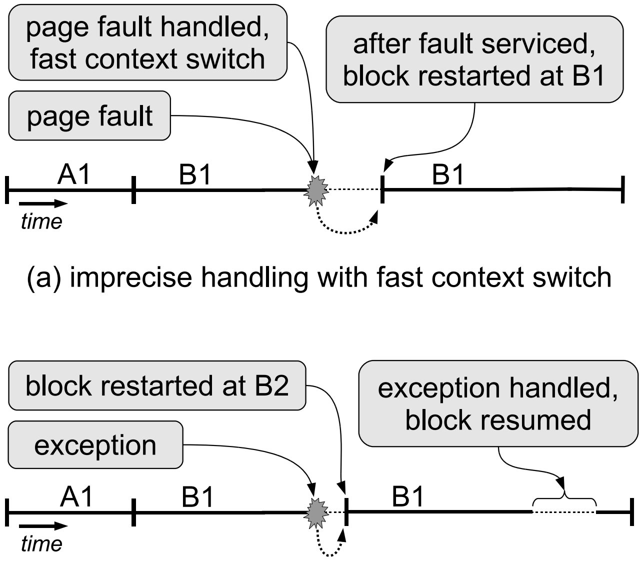


Figure 6.1: GPU support for exceptions and context switching using idempotence.

these points can be chosen intelligently such that the amount of program state to be preserved from that point forward is minimized. This observation, combined with the insight that GPU workloads typically have large regions of code that are idempotent [70], allows the implementation of exception and context switching support using idempotent regions with very low overhead.

Figure 6.1 illustrates how idempotence can provide efficient exception and fast context switching support. Figure 6.1(a) shows that at each point in a program's execution there are differing amounts of live state, and Figure 6.1(b) shows how programs can be decomposed into idempotent regions, with the boundaries between regions preferentially chosen at locations that contain relatively little amounts of live state. As previously mentioned, GPU application programs tend to have large regions of code that are idempotent and hence these regions can be very large.

Figure 6.2(a) shows how these regions can be used to efficiently recover from exception conditions requiring a context switch. Suppose that in the midst of executing region B2 a page fault occurs. Suppose also that the running program pushes and pops only a small number of live registers from the program stack at the boundary points between idempotent regions. Then, the page fault can be serviced and a context switch can occur effectively instantaneously. After the fault is serviced,



(a) imprecise handling with fast context switch

(b) handling with consistent exception state

Figure 6.2: GPU support for fast context switching and exceptions using idempotence.

the original process can be switched back in at a convenient time, restarting from region B1. In this scenario, the exception handling need not be “precise” with respect to the faulting instruction, and hence the exception can be both handled and a context switch performed immediately without concern for the program’s state at the point of the fault.

Figure 6.2(b) shows how the idempotent regions can be used to recover from general exception conditions where precise state is important as well. Suppose that an arithmetic exception occurs executing region B2 and that the architectural state at the point where it is detected is not sequentially consistent with respect to the excepting instruction. The GPU recovers by re-executing the short region precisely to the point of the exception, handles the exception, and then recovers by resuming execution from the immediately following instruction. The precise re-execution allows the exception handler to see a consistent live program state with respect to the point of the exception, with forward progress ensured when augmented with some support for avoiding live-lock, using mechanisms such as those described in Section 6.1.2.

Term used	NVIDIA term	OpenCL term
SIMD processor	Streaming Multiprocessor	Compute Unit
SIMD instruction	PTX instruction	FSAIL (AMD) instruction
SIMD thread	Warp	Wavefront
SIMD thread lane	Thread	Work item
SIMD thread group	Thread block	Work group

Table 6.1: GPU terms used in this paper (adapted from Hennessy and Patterson [49]).

Section Organization

The remainder of this section (Section 6.1) presents a GPU design that supports the mechanisms shown in Figure 6.2. Section 6.1.1 first presents background on GPUs and Section 6.1.2 subsequently describes a GPU architecture for idempotence-based recovery with supporting mechanisms for exception recovery and fast context switching. Finally, Section 6.1.3 presents an evaluation.

6.1.1 GPU Background

The discussion below gives background on the memory and register architecture of current-generation GPUs, the state of exception support in those GPUs, and how idempotence manifests strongly particularly in GPU workloads.

Terminology

Table 6.1 shows GPU terms used as well as the equivalent NVIDIA and OpenCL terms. A GPU consists of a number of *SIMD processors* that execute *SIMD instructions* sequentially ordered into *SIMD threads*. A vertical cut of a SIMD thread, which corresponds with one element of a SIMD lane, we call a *SIMD thread lane*. Finally, identical SIMD threads that run on the same processor form a *SIMD thread group*.

GPU Memory and Register Architecture

Figure 6.3 shows the high-level architecture of a modern GPU that supports virtual address translation. Each SIMD processor has a hardware-managed L1 cache, we assume each processor has a TLB, and all processors share an L2 cache. NVIDIA's most recent GPU architecture, Fermi, and AMD's

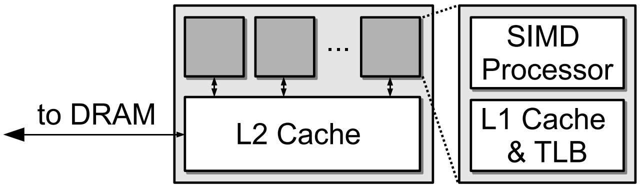


Figure 6.3: The organization of a modern GPU that supports address translation.

GPU architecture	L2 State	L1 State	Register State
AMD HD 6550D	128KB	40KB	1.28MB
NVIDIA GTX 580	768KB	1MB	1.92MB

Table 6.2: The memory and register size characteristics of two commodity GPUs.

recent Llano Fusion architecture both resemble this description [7, 8, 77]. The size characteristics for an integrated AMD GPU part and a discrete NVIDIA GPU part are shown in Table 6.2. The table shows that GPU register state is more than both the L1 and L2 cache state combined.

GPU Exception Support

Current GPUs do not implement general exception support, although Fermi supports timer interrupts used for application time-slicing [77], which can be scheduled at convenient points in time. Demand paging, in contrast, requires support for page faults, which present much more onerous timing and restartability constraints.

While both Fermi and Llano support virtual addressing on some level [7, 77], neither supports all the features of virtual memory, such as demand paging and complete support for execution of processes only partially resident in memory. Specifically, Fermi does not support paging at all, while Llano supports a limited form of paging at kernel boundaries [7]. Llano requires the memory accesses to be known in advance, which allows paging to be orchestrated by the GPU driver software at scheduling time. Although demand paging is not supported, it is on the AMD Fusion roadmap for the future [6]. However, this support must find a way to overcome the vast amount of process-local state that must be saved and restored to support context switches on page faults. As is evident, saving and restoring all register state is likely to be both slow and power intensive.

```
1 __global__ void MatrixMultiplyAccumulate(Matrix A, Matrix B, Matrix C) {
2   int row = blockIdx.y * blockDim.y + threadIdx.y;
3   int col = blockIdx.x * blockDim.x + threadIdx.x;
4   float Cvalue = C.data[row * C.width + col];
5   for (int i = 0; i < A.width; ++i)
6     Cvalue += A.data[row * A.width + i] * B.data[i * B.width + col];
7   C.data[row * C.width + col] = Cvalue;
8 }
```

Figure 6.4: A simple matrix multiplication CUDA kernel.

Idempotence in GPU Workloads

Figure 6.4 illustrates how idempotence exists in abundance in traditional GPU workloads by presenting example GPU code. It shows a simple GPU kernel written in C for CUDA that is representative of the types of workloads typically run on GPUs—workloads that have a high degree of data parallelism and have regular streaming memory interactions. A common byproduct of these characteristics is distinct read and write data sets, which implies a lack of antidependences, which leads to the property of idempotence.

This specific kernel computes the matrix multiplication of matrices `A` and `B` and accumulates the result onto matrix `C`. The accumulation is not standard for matrix multiplication. However, it makes the example interesting since the accumulation forms a clobber antidependence across lines `4`→`7`, and hence the kernel is not completely idempotent, allowing us to revisit this kernel as a running example in the next section.

6.1.2 GPU Design

This section develops the architecture and mechanisms of a GPU leveraging idempotence for exception recovery. Figure 6.5 shows the modifications over a conventional GPU architecture. The compiler, ISA, and hardware extensions are described in this section and are marked in the figure using black boxes.

Compiler: The device code generator of a traditional GPU is modified as shown at the top of Figure 6.5. The code generator generates device code from an intermediate representation (IR) and with our modifications it identifies the idempotent regions in the IR GPUs already

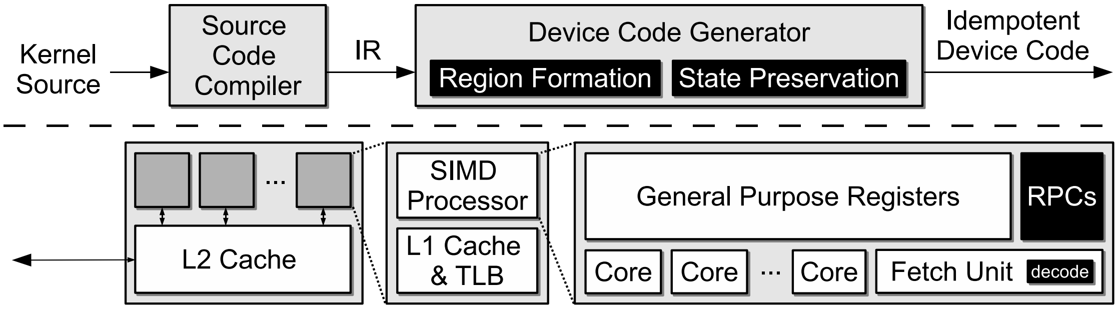


Figure 6.5: GPU software (top) and hardware (bottom) with modifications in black.

use an SSA-like IR (e.g. NVIDIA's PTX) with infinite registers, so IR antidependences already occur only among non-local variables or across loop iterations, assisting in the static analysis of Chapter 3. The code generator constructs idempotent regions by cutting them as described in that chapter, with the key difference that it prefers to place cuts before instructions with the minimum amount of live state. This process of placing idempotent region boundaries is the *region formation* phase of the code generator.

After the regions are formed, the code generator then enters a second phase, the *state preservation* phase. During this phase, the code generator prevents new clobber antidependences from arising during register and stack memory allocation by allocating local variables in such a way that those variables live at idempotence boundaries are not overwritten using the techniques detailed in Chapter 4. Figure 6.6 illustrates the overall flow for the example kernel from Figure 6.4 using a stylized device code representation. Recall that the kernel contains one clobber antidependence. For region formation, the clobber antidependence is cut at the point with the least live state, which occurs immediately before the loop entry at the initialization of loop variable `i` in register `r2`. The idempotence boundary is placed as shown in the lower half of Figure 6.6. During state preservation, the code generator then logically inserts a move instruction from register `r1` to a freshly allocated register, `r3`, after the boundary instruction. From that point on, it accumulates the `CValue` variable onto `r3` instead of `r1`, preserving the value in the live register `r1` at the expense of some additional register pressure on the kernel. The other live variables at the idempotence boundary are not subsequently overwritten and hence require no action to be preserved.

Before:

```
 ...   ($r0 holds C.data offset)
 add.u32 $r0, param[C_data], $r0;
 mov.f32 $r1, global[$r0];
 mov.u32 $r2, 0x00000000;
LOOP:
 ...
```

After:

```
 ...   ($r0 holds C.data offset)
 add.u32 $r0, param[C_data], $r0;
 mov.f32 $r1, global[$r0];
 idem.boundary;
 mov.f32 $r3, $r1;
 mov.u32 $r2, 0x00000000;
LOOP:
 ...   (uses of $r1 replaced by $r3)
```

Figure 6.6: GPU state preservation.

ISA: The ISA is extended with a special instruction to mark the boundaries between idempotent regions as shown in Figure 6.6, When the boundary instruction is executed, the PC value associated with the immediately following instruction is saved away in a special RPC (restart PC) register. The boundary instruction also acts as an instruction barrier such that a SIMD thread's in-flight instructions must retire before proceeding. This ensures that a region remains recoverable while an exception or a mis-speculation remains undetected for the region's in-flight instructions.

Hardware: To support idempotence-based recovery in the hardware, one RPC register is associated with each SIMD thread and some decode logic is added to process boundary instructions. These two changes are illustrated on the right side of Figure 6.5. NVIDIA's Fermi architecture allows a maximum of 48 SIMD threads per SIMD processor, so for this case the RPC state would amount to a 192-byte register file (assuming 4-byte RPC values) physically removed from the hardware critical path.

To support the possibility of thread divergence causing thread lanes of the same thread to enter different idempotent regions (for which multiple RPC values would be required to maintain correct execution), we assume thread splitting techniques such as the one proposed by Meng *et al.* are employed when needed to maintain a single RPC per thread [69]. This allows divergent paths to be treated as separate logical SIMD threads, each maintaining its own RPC value. Reconvergence of divergent SIMD threads to saturate the available SIMD width is allowed as well, with the additional restriction that thread reconvergence may occur only *after* encountering the first boundary instruction following the path reconvergence point.

Overall, the hardware changes are small, especially considering the many thousands of registers and tens of functional units already resident on the SIMD processors of modern GPUs. Alternative CPU-like mechanisms to achieve both exception and speculation support would require much more hardware. Additionally, at the circuit level, the timing of exception and mis-speculation control signals can be relaxed compared to traditional hardware pipeline-based approaches (such as those covered in Section 6.2).

General Exception Support Mechanisms

General exception support on GPUs requires exposing consistent exception state to an exception handler and mechanisms to prevent exception live-lock. The problem of supporting consistent state is that on a GPU multiple exceptions can occur executing a single SIMD instruction, and determining in an exception handler which lanes of the instruction experience an exception can be difficult. The problem of exception live-lock is that multiple recurring exception conditions inside a single idempotent region can lead to live-lock. The discussion below presents software and hardware solutions to both problems.

Consistent exception state: To service SIMD exceptions, an exception handler must determine which lanes in a SIMD instruction experience an exception. We initially propose to achieve this without hardware modification as follows. First, prior to re-execution, a software routine patches the excepting instruction with a trap to an emulation routine, similarly to the way a debugger inserts a breakpoint into a running program. Upon re-execution, the emulation routine then emulates the instruction in its entirety, servicing all exception conditions. It then "unpatches" the instruction and resumes execution at the immediately following instruction, as before. Other solutions that require hardware support, such as adding exception status bits to record the excepting lanes of an instruction and the associated circuitry, are also possible. With this solution, the exception handler would require that an excepting instruction execute all lanes of the instruction to completion, updating all result registers except those whose lanes experience an exception.

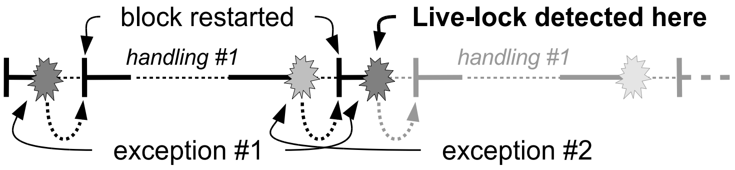


Figure 6.7: Live-lock under multiple recurring exceptions.

Exception live-lock: A potential live-lock condition is indicated if, during re-execution, an instruction experiences an exception and its PC value does not match the PC of the instruction that initiated the re-execution. Consider, for instance, a region that experiences two arithmetic exceptions as shown in Figure 6.7. After the first exception is handled, the second exception is encountered shortly afterwards. However, upon subsequent re-execution, the first exception is encountered again, leading to live-lock.

One way to detect live-lock is to save the PC value of an excepting instruction to a dedicated register for comparison during re-execution. Live-lock is then detected as soon as an exception recurs a second time, as shown in Figure 6.7.

To resolve the live-lock condition after it has been detected, single-stepped execution at the granularity of individual SIMD lanes allows precise servicing of individual exceptions. However, GPUs do not natively support this type of execution and adding it requires non-trivial modification to the hardware. Instead, an alternative option is to leverage the virtualized nature of modern GPU instruction sets and employ dynamic re-compilation to prevent live-lock. Rather than the hardware entering a single-stepping mode upon live-lock detection, a dynamic compiler instead recompiles the code such that the two excepting instructions causing the live-lock condition are placed in separate idempotent regions. Afterwards, idempotence-based re-execution can be retried, and the re-compilation effort ensures forward progress. Alternatively, single-stepped lane execution can be implemented in the hardware. Both solutions would be slow to execute; however, it is important to note that potential live-lock should arise only in rare circumstances for the vast majority of possible exception conditions.

Efficient Context Switching Mechanisms

Idempotence-based recovery can substantially reduce the overheads of context switching on GPUs. This is particularly valuable to the implementation of demand paging, for which a context switch would traditionally require saving and restoring vast amounts of register state. The key insight is that, utilizing idempotence, context switch state need only be saved and restored with respect to the boundaries between idempotent regions, and that the locations of these boundaries is moreover configurable. Hence, boundary locations can be chosen such that this state is minimized, enabling much more efficient context switching.

To minimize live state on context switches, antidependences are simply cut at instructions where there is the least amount of live state, as demonstrated earlier. To illustrate how this helps, suppose that a page fault occurs executing the statement on line `6` of the kernel in Figure 6.4. Using traditional state-minimization techniques [79, 89], a minimum of 8 live registers would need to be saved to memory[1]. However, restarting from our idempotence boundary at the head of the loop, only 3 live registers need to be saved. This is a greater than two-fold reduction in context switch state.

6.1.3 GPU Evaluation

This section presents an abstract evaluation of the GPU design of the previous section. A more detailed evaluation is covered in the co-authored work on the iGPU architecture [70]. While the iGPU architecture evaluation evaluates GPU CUDA workloads using Ocelot [31] and GPGPUSim [12], in this dissertation we focus on the CPU versions of the Parboil benchmarks, targeted at GPUs, which allows us to use the LLVM-based compiler implementation evaluated in Chapter 5.

Experimental Method

As mentioned above, we consider the Parboil benchmark suite [100] specifically, which consists of compute kernels that have been ported to both GPUs using "C for CUDA" and CPUs using

[1] Although the `A`, `B`, and `C` matrix objects (with member variables `width` and `data`) are live on entry, along with built-in variables `blockIdx`, `blockDim`, and `threadIdx`, these are read-only kernel input variables that are backed in memory. They are assumed to be loaded on first use inside a region and hence do not need to be saved on a context switch.

GPU Hardware	Configuration
Pipeline	fetch/decode/issue/commit: 2-wide
L1 caches	1KB per thread context, 2-way set-associative, 64-byte line size
L2 cache	1MB 8-way set-associative, 64-byte line size, 10-cycle hit latency
Memory	200-cycle access latency

Table 6.3: GPU-like configuration for evaluation. Latencies in cycles are "effective cycle" latencies that account for the latency-hiding effects of multi-threading on GPUs.

traditional C/C++. We use the CPU versions of the benchmarks modified as described in Section 5.2 and subsequently further modified to model a partitioning among threads[2]. We then compile this CPU code using our LLVM compiler employing code generation strategy Max-Size. To minimize context switch overhead, we observe that compiling for the largest idempotent regions produces region boundaries contained in outer-most loops, and that live variables at these points are often spilled to the stack anyway to free the registers for use inside the idempotent regions, and hence Max-Size is used unmodified. Although incorporating liveness information into the LLVM region construction is possible as a future extension, LLVM's target-neutral IR makes this challenging.

After compilation, we simulate the benchmarks, which are fairly short, to completion using gem5 [17] for ARM. While executing ARM instructions does not allow us to model GPU-specific effects such as branch divergence/re-convergence of parallel threads, stalls for thread barrier synchronization, or special functional unit (SFU) operations, the compiler behavior and the resulting execution is only negligibly impacted, if at all, by these effects. The approach of simulating a GPU ISA using a CPU ISA is moreover an accepted method for GPU evaluation [76], and it allows to faithfully capture the compiler aspect of the architecture using our fully-working LLVM implementation, which does not support either compiling CUDA code or producing device-specific GPU machine code. The GPU-like configuration modeled in gem5 is shown in Table 6.3[3].

For measurements, idempotent region sizes are measured in terms of instructions and runtime performance overhead is measured in terms of the percentage of additional execution cycles simulating a simple two-issue in-order core that completes most instructions in a single cycle (as

[2] We partition to optimize for path lengths on the order of thousands of instructions. We note that this may yield sub-optimal throughput for certain GPU device characteristics.

[3] NVIDIA's Fermi similarly supports dispatching 2 warp instructions per cycle [77].

Parboil Benchmark	Mean Path Length	% Run-time Overhead
cutcp	612.4	1.42
fft	2,452.1	0.38
histo	4,683.3	0.29
mri-q	5,825.9	0.19
sad	3,021.2	0.42
tpacf	3,215.8	0.39

Table 6.4: GPU evaluation results.

is effectively the case for a GPU with multi-threading). At the boundaries between regions, we conservatively model a 10-cycle stall latency to ensure the absence of any exception conditions before proceeding. Finally, context switch efficiency is qualitatively measured by manually inspecting the compiler assembly output and observing that a majority of live registers are spilled to stack memory as a side-effect of preserving the idempotence property.

Results

Column 1 of Table 6.4 shows the mean idempotent path length of each of the six Parboil benchmarks after all code modification compiling with code generation strategy Max-Size. Column 2 shows the overhead of the resulting code in terms of execution cycles compared to the same code compiled using the normal optimized LLVM compiler flow. While the overhead is low, we acknowledge the possibility of under-estimating this overhead given that GPUs often statically constrain the number of available registers at compile-time based on resource requirements, while ARM CPUs, in contrast, have a (relatively high) fixed number of available registers. Hence, if a program would otherwise not be able to use all the available registers on a CPU, no overhead may be recorded for the CPU, whereas on the GPU this might alter the number of registers statically allocated to a thread, potentially affecting overall throughput. We assume that this explains the larger overhead numbers reported in the iGPU work [70], which also has higher overhead due to a higher modeled spill/re-fill latency. However, even in that work, the overheads do not exceed 4%. The evident conclusion is thus that compiling for idempotence introduces only small amounts of run-time overhead even in the worst case.

6.2 Out-of-Order Execution in CPU Architectures

Emerging challenges in technology scaling present diminishing opportunity to improve the energy efficiency of (general-purpose) CPU processors at the transistor level. As a result, the task of improving CPU energy efficiency is increasingly falling to computer architects. This section explores how idempotence-based execution can alleviate CPU overheads associated with in-order retirement of instructions, a feature widely assumed necessary in CPUs to ensure that processor state is consistent with respect to a running program at all times. In-order retirement simplifies program debugging and enables seamless support of page faults and other exceptions in software. However, special-purpose hardware structures, such as a reorder buffer or speculative register file, are often required, and the resulting hardware complexity and power consumption can be high [52, 93, 95, 105].

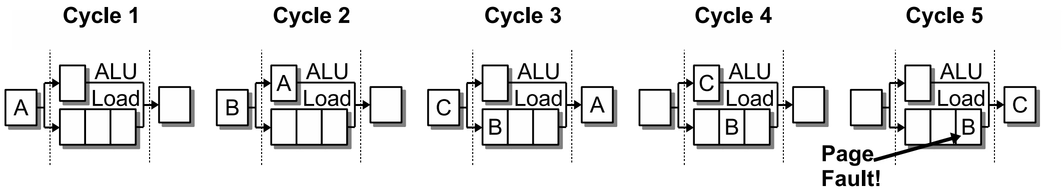
Enabling out-of-order retirement in CPUs is in many ways the complement of providing exception support in GPUs—for CPUs idempotence can preserve exception support while enabling simpler hardware, while for GPUs it can enable exception support while preserving simple hardware. As for GPUs, idempotent regions can be used to recover sequentially correct state for CPUs by jumping back to the beginning of a region and re-executing precisely up to the point of a specific instruction and no further. In the CPU context, the approach is similar to previous proposals for speculative out-of-order retirement using hardware checkpoints [5, 24, 52, 67]. However, it is software co-designed to incur substantially less hardware power, area, and complexity overhead leveraging the property of idempotence.

Out-of-Order Execution Using Idempotent Regions

Figure 6.8 illustrates how sequentially-correct state can be recovered despite out-of-order retirement on a CPU for a simple sequence of three instructions using idempotence. It assumes a very simple single-issue CPU design that issues instruction in program order but retires them potentially out of order. Figure 6.8(a) shows the instruction sequence along with the issue cycle, execute latency, and writeback cycle of each instruction, and Figure 6.8(b) shows the cycle-by-cycle state of the processor pipeline as instructions move through it.

Instruction	Issue Cycle	Execute Latency	WB Cycle
`A. R2 ← add R0, R1`	1	1	3
`B. R3 ← ld  [R2 + 4]`	2	3	6
`C. R2 ← add R2, R4`	3	1	5

(a) A simple three-instruction code sequence and execution characteristics.



(b) Execution through the issue, execute, and write-back pipeline stages.

Figure 6.8: Out-of-order retirement over a simple instruction sequence.

In the example, a page fault occurs during the execution of instruction B in cycle 5. Because instruction C retires in that same cycle, normally speaking a processor is unable to cleanly resume execution after handling the page fault because the program state at the end of cycle 5 is not consistent with respect to the start of either instruction B or C. However, the program state *is* consistent with respect to the start of instruction A; after servicing the page fault, it is possible to resume execution from instruction A. Alternatively, the processor can first issue and execute from instruction A precisely to instruction B, service the page fault, and then resume execution from instruction C. This is possible because instructions A-C form an idempotent region. The same idea can be applied to provide recovery support for other out-of-order techniques such as branch prediction and memory dependence prediction as well.

Opportunity Analysis

Within an idempotent region, a CPU may execute and retire instructions out of order. This enables three key simplification opportunities in the processor design. First, it allows the results of low latency operations to be used immediately after they are produced, without the need for complex staging and bypassing as in conventional processors. Second, it simplifies the implementation of exception support, particularly for long latency operations such as floating point operations. Finally, it enables instructions to retire out of order with respect to instructions with unpredictable latencies,

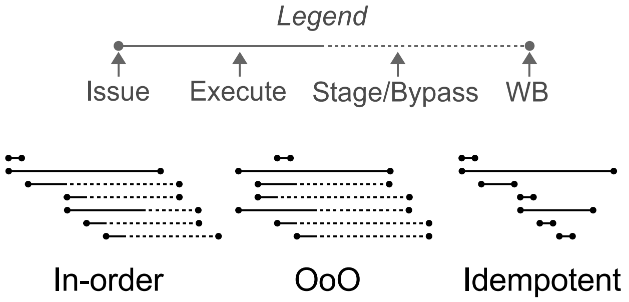


Figure 6.9: A comparison between idempotent and traditional CPU designs.

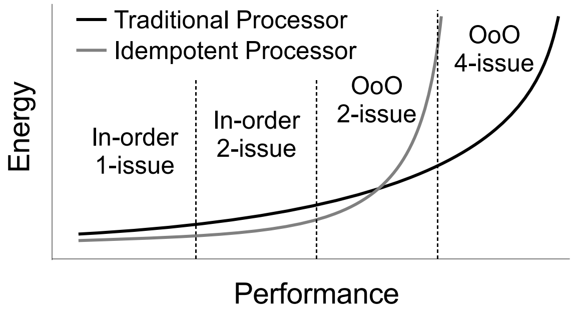


Figure 6.10: The CPU design space.

such as loads.

Figure 6.9 compares the resulting execution behavior of an a processor using idempotent regions for recovery to that of an in-order and out-of-order processor. While the in-order and out-of-order processor both stage and bypass instruction results until they are able to retire in order, the processor executing idempotent regions does not.

To understand the energy impact of this idea across a range of CPU designs, the discussion below uses the Pareto-optimal energy-performance frontier as a framework for analysis. Azizi *et al.* measured this frontier for microprocessors using existing techniques [11], and Figure 6.10 approximates their findings. The black line shows the energy-performance frontier curve with annotations for the four processors designs that Azizi *et al.* determine perform optimally over different ranges on the curve. The gray line projects this curve for processors executing idempotent regions: for some ranges on the curve, performance and energy improve; for other ranges, there is a loss. The rationale for the projected shifts are given below.

Among in-order processors, single-issue in-order processors benefit from the ability to retire instructions ahead of a cache miss: normally, a miss forces a stall. However, with only modest pipeline depths and only one instruction issued per cycle, there is limited apparent benefit. As a result, the overall gain for these processors is relatively small. In contrast, dual-issue in-order processors are often designed with much deeper pipelines, have a range of execution unit latencies, and utilize extra logic to track in-flight instructions and ensure they commit in order. Hence, these processors benefit from each of three simplification opportunities mentioned earlier and thus the curve shifts more drastically.

Dual-issue out-of-order processors can complete many instructions out of order, and idempotent processing enables those instructions to also retire out of order. However, relative to an in-order processor, the processor energy budget is substantially higher to implement out-of-order issue. Additionally, these processors may re-use some of the retirement ordering logic to also resolve data hazards and recover from branch mispredictions perhaps more efficiently than can be achieved utilizing idempotence, diminishing the overall benefit. For small instruction windows, there may still be an overall benefit. However, as the window size increases, there is an eventual loss in efficiency as the idempotent region sizes begin to limit the capability and efficiency of out-of-order issue.

Section Organization

Based on the analysis of Figure 6.10, the remainder of this section (Section 6.2) explores the opportunity for dual-issue in-order processor designs. Section 6.2.1 first presents background on the complexities arising from in-order retirement in modern dual-issue in-order processors, giving supporting references to commercially available processor designs. Section 6.2.2 then shows how out-of-order retirement enables power and complexity savings. Finally, Section 6.2.3 presents an evaluation.

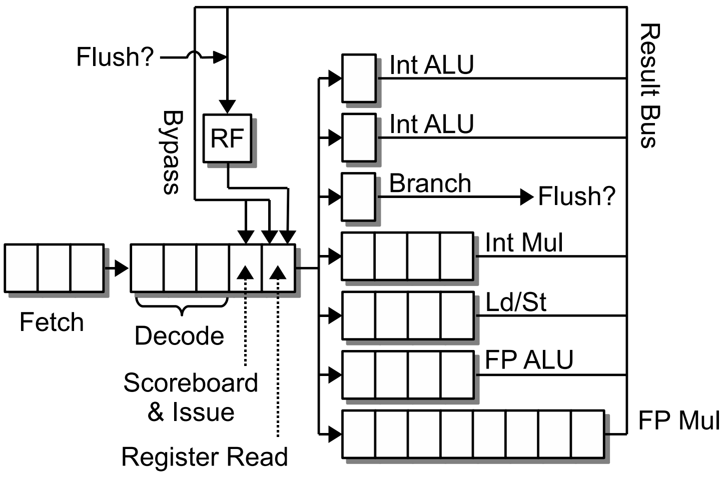


Figure 6.11: Baseline processor configuration.

6.2.1 CPU Background

Idempotence enables safe out-of-order retirement within a region. This enables valuable pipeline simplifications for processors that employ special-purpose hardware to manage in-order retirement. This section shows the ways in which in-order retirement complicates a conventional processor design.

As a representative processor, we consider an aggressively-designed two-issue in-order processor, loosely based on the energy-efficient ARM Cortex-A8 processor core [14]. However, we also make reference to two other widely-used two-issue in-order processor implementations—the Cell SPE [57], and the Intel x86 Atom [44]—when their design choices differ significantly from the Cortex-A8.

Figure 6.11 shows an initial, baseline processor configuration with features similar to those of the Cortex-A8, but without any high-performance optimizations to overcome the inefficiencies introduced by in-order retirement. In the following sections, we incrementally add in these optimizations. The processor is aggressively pipelined with 3 cycles for fetch, 5 for decode and issue, and the following execution unit latencies: 1 cycle for integer ALU operations and branches; 4 cycles for integer multiply, load, store, and floating point ALU operations; and 8 cycles for floating point multiply and divide. Since the processor is dual-issue, the processor has two integer ALU units. The pipeline supports bypassing results over the result bus immediately before writeback.

The Complexities of In-Order Retirement

In-order retirement complicates the design of (1) the integer processing pipeline, (2) the floating point processing pipeline, and (3) support for cache miss handling, as described below.

Staging and bypassing in the integer pipeline: In Figure 6.11, integer ALU operations complete before integer multiply and memory operations. Hence, the processor cannot immediately issue ALU instructions behind these other instruction types because it will lead to out-of-order retirement. To improve performance, the pipeline can be modified to retire the ALU instructions in order using a technique called *staging*. Using staging, retirement of ALU instructions is delayed so that they write back to the register file in the same cycle as the multiply and memory operations. This technique is used to achieve in-order retirement of integer operations on each of the Cortex-A8, Cell SPE, and Atom processors. The Cortex-A8 has up to 3 cycles [14], Atom has up to 5 cycles [44], and the Cell SPE has up to 6 cycles of staging [57].

A basic implementation of staging involves the insertion of staging latches to hold in-flight results as they progress down the pipeline. Results are then bypassed to executing instructions by adding a unique data bus for each staging cycle and destination pair. Unfortunately, this implementation results in combinatorial growth in the complexity of the bypass network. For this reason, a sometimes preferred solution is instead to implement a special register-file structure to hold results until they are allowed to retire. Results are then bypassed out of this structure in a manner similar to how results are read from the ROB in an out-of-order processor. González *et al.* describe this structure in more detail, calling it the staging register file (SRF) [39].

Figure 6.12 shows the addition of an SRF to our processor to temporarily hold completed results. To determine whether values should be read from the register file (RF) or SRF after issue, it also includes an extra rename stage in the pipeline to map registers to their location in either the RF or SRF (the Cortex-A8 includes such a rename stage after issue [14]). Finally, the pipeline flush logic is added to flush the pipeline in the event of an exception or branch misprediction in the integer pipeline. It is worth noting that support for result bypassing is

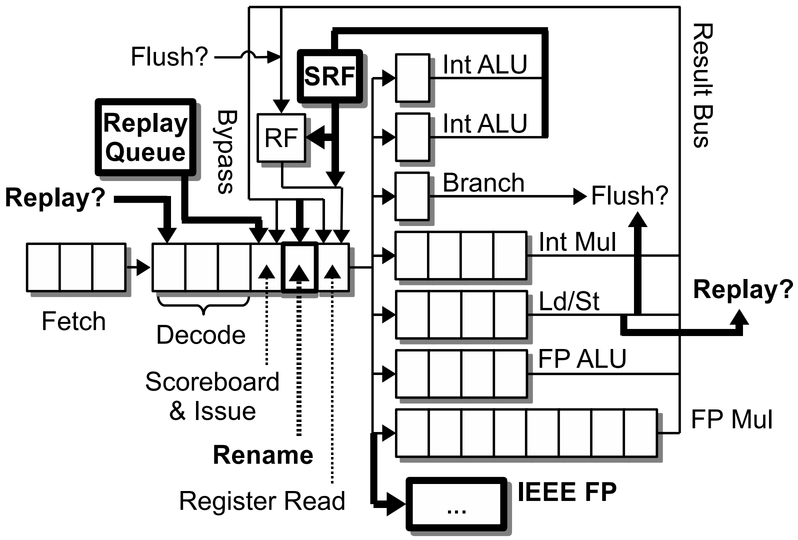


Figure 6.12: The complexities introduced by in-order retirement.

more complicated than depicted in Figure 6.12 for processors with execution unit pipelines that consume and produce values at multiple stages, such as the actual Cortex-A8 and the Atom, and for processors with very deep pipelines, such as the Cell SPE.

Exception support in the floating point pipeline: In Figure 6.12, integer operations retire after all possible exception points in the integer pipeline, but the floating point pipeline may experience exceptions as well. Although floating point exceptions are typically rare, the IEEE floating point standard requires that five specific floating point exception conditions be minimally detected and signaled [3]. The IEEE standard also recommends precise floating point exception support for trapping to software.

Regarding the latter, supporting precise floating point exceptions in hardware is difficult. Hence, many processors, including the Cortex-A8 and Cell SPE, do not support it [9, 53]. Atom does support it [56]. Unfortunately, the specific details on Atom's implementation are not publicly available. The remaining aspects of the IEEE standard are implemented in the Cortex-A8 using a second, much slower, non-pipelined floating point unit that handles IEEE compliant floating point operations, with exceptions handled entirely in hardware. Figure 6.12 shows this support added to our representative processor.

Cache miss handling: Occasionally, a load or store will miss in the L1 cache. In this case, an in-order processor must typically wait until the cache is populated with the missing data before it can resume execution. In the case of a simple in-order pipeline, the pipeline often simply asserts a global stall signal that prevents instructions from advancing down the pipeline. However, for deeper pipelines with many stages this stall signal must propagate over long distances to many latches or flip flops and thus often forms one or more critical paths [18, 33].

The Cortex-A8 has such a deep pipeline. Hence, it uses a different approach, which is to issue a replay trap—re-use the exception logic to flush the pipeline and re-execute—in the event of a cache miss [14]. The pipeline is then restarted to coincide precisely with the point where the cache line is filled. To enable this coordination of events, the Cortex-A8 employs a replay queue to hold in-flight instructions for re-issue. Figure 6.12 shows the addition of the replay queue to our representative processor. Compared to the Cortex-A8, the Cell SPE does not implement special support for cache misses since the SPE cache is software managed, while specific details on Atom's implementation are unavailable.

Overall, while conceptually simple, modern in-order processors are quite complex and exhibit multiple sources of overhead relating to in-order retirement as demonstrated by the differences between Figures 6.11 and 6.12. This includes the extra register state to hold bypass values in the SRF, the additional rename pipeline stage to map operand values into the SRF, additional circuitry to flush the pipeline for exceptions and cache misses, additional floating point resources, a special replay queue to hold in-flight instructions, and associated circuitry to issue from the replay queue and force replay traps. As we show in the next section, this additional complexity can be eliminated in a processor using idempotent regions for recovery.

6.2.2 CPU Design

For a processor executing idempotent regions, out-of-order retirement of integer and floating point operations are not problematic within the confines of a region, floating point exceptions are easily supported precisely, and the processor pipeline must neither stall nor flush in the presence of a cache miss. It must use only slightly modified scoreboarding logic, and the only additional hardware requirement is the ability to track the currently active idempotent region and only issue instructions from that region. In particular, if a potentially excepting instruction from the currently active region is still executing in the processor pipeline, an instruction from a subsequent region may not issue if it might retire ahead of the exception signal. Constraining issue in this way ensures that the contents of regions retire in order with respect to the contents of other regions. In our CPU design, we propose a special counter to track the number of potentially excepting instructions executing in the pipeline (up to 4). The processor only advances the active region and begins issuing instructions from the next region when the counter is at zero.

Power and Complexity

Intuitively, significant power and complexity savings are enabled by the opportunity to eliminate staging, bypassing, and support for replay traps. In particular, we eliminate all of the following: a 6-entry SRF and a 8-entry replay queue (both dual-ported circular buffers), the *entire* rename pipeline stage including the rename table, the pipeline flush and issue logic for exceptions and replay traps (likely to form one or more critical paths), an entire IEEE-compliant floating point execution unit, and all of the control logic associated with each of these various pieces.

While arguably these savings are specific to this particular dual-issue core design, similar core designs have been repeatedly built over the years [33, 51] and similar savings would be achieved even for single-issue in-order designs such as those used in embedded processors. The latter claim is supported by Wang and Emmett, who evaluate VLSI implementations of a range of in-order retirement buffering strategies for a simple pipelined RISC processor and find that area and performance overheads are both in excess of 15% [105].

ISA Support

ISA support is similar to the ISA support for the GPU design from Section 6.1.2. Rather than create a special dedicated instruction, however, we propose to divide regions at specially marked store instructions, repurposing a single bit in the immediate offset field of the store to indicate whether it marks an idempotent region boundary. As an example, store instructions for the ARM ISA use a 12 bit immediate field for immediate offset addressing. We shorten this to 11 bits and use the freed bit to mark region boundaries.

In the event of mis-speculated out-of-order retirement (i.e. an exception), the CPU takes the following corrective action: (1) it stores the PC value of the excepting instruction to a register (e.g. the exception register); (2) it sets the PC to the PC of the most recent idempotent region boundary point; and (3) it resumes the processor pipeline, issuing instructions only up to the PC of the excepting instruction. When the excepting instruction executes and causes the exception a second time, the CPU traps to software precisely with respect to that point in the program. After the exception is handled, execution is resumed from the PC of the excepting instruction. As a fallback, live-lock is handled by single-stepped re-execution with in-order retirement as described for the GPU hardware case.

6.2.3 CPU Evaluation

This section presents an abstract evaluation of the CPU design of the previous section. A more detailed evaluation is covered in co-authored work on Idempotent Processors, which also augments the design with some low-complexity out-of-order issue support [28].

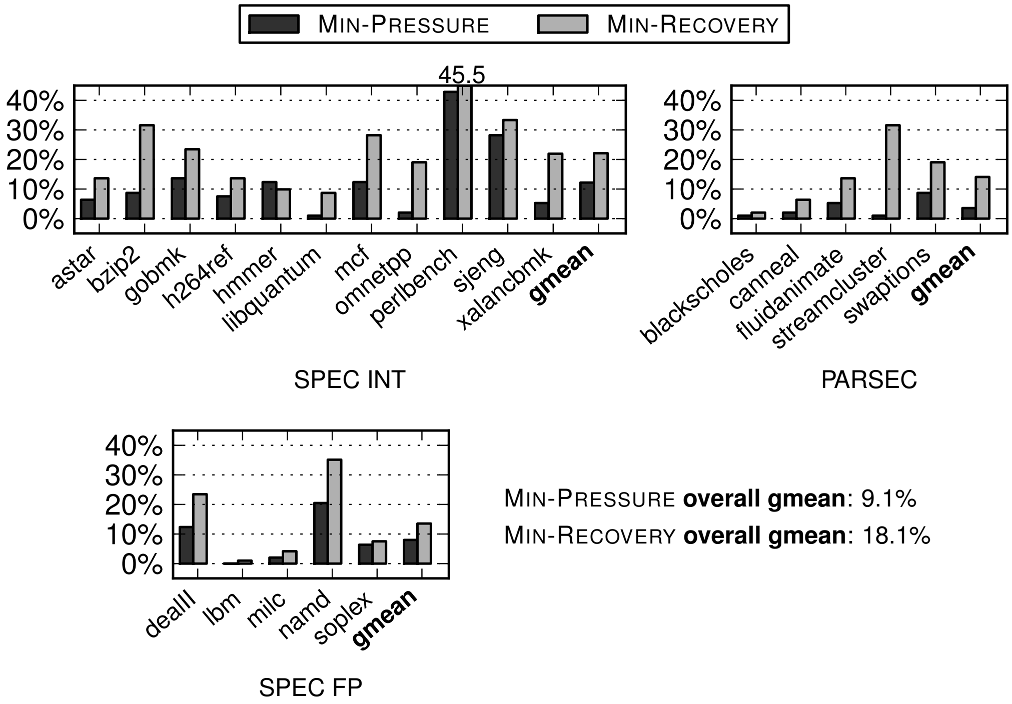
Experimental Method

To evaluate our CPU design, we consider the benchmarks from the SPEC 2006 [99] and PARSEC [16] suites and simulate them using gem5 [17] for ARM. We measure runtime overheads for two compiler/hardware configurations. For the first configuration, we assume branch mis-prediction recovery in the hardware. We compile using code generation strategy Min-Pressure with threshold value 5 and assume architectural idempotence. For this configuration, we stall in the simulator only to clear potential exception conditions. For the second configuration, we assume branch mis-prediction recovery using idempotence. We compile using code generation strategy Min-Recovery and assume contextual idempotence. For this configuration, we stall in the simulator to clear both potential exception conditions and branch mis-predictions.

The CPU hardware configuration we model in gem5 is shown in Table 6.5. The hardware

CPU Hardware	Configuration
Pipeline	fetch/decode/issue/commit: 2-wide
L1 caches	32kB DCache, 2-way set-associative, 64-byte line size
L2 cache	1MB 8-way set-associative, 64-byte line size, 10-cycle hit latency
Memory	200-cycle access latency

Table 6.5: CPU hardware configuration for evaluation.



MIN-PRESSURE **overall gmean**: 9.1%
MIN-RECOVERY **overall gmean**: 18.1%

Figure 6.13: The percentage run-time overheads of idempotence for recovery on a CPU.

configuration is similar to the GPU configuration from Section 6.1.3, except that (a) the L1 cache size is much larger given that the cache is not shared among many thread contexts as on a GPU, and (b) we model the longer functional unit execution latencies of the baseline CPU shown in Figure 6.11. Run-time overheads are measured in terms of simulated execution cycles using the same methodology as in the compiler evaluation methodology of Section 5.1, with the single difference that benchmarks are only simulated for one billion instructions to accommodate increased simulator complexity in modeling a variable latency pipeline.

Results

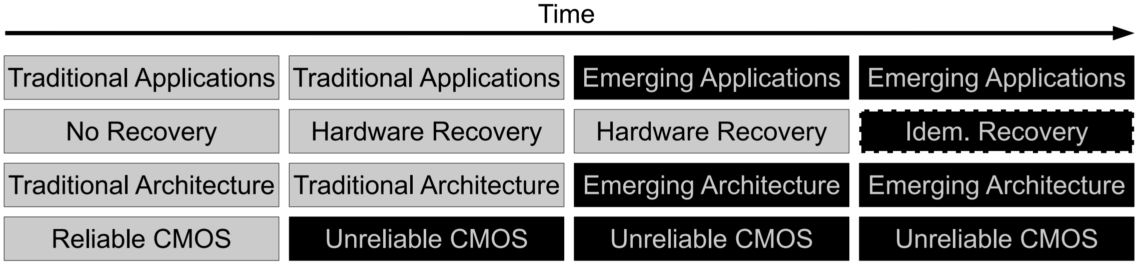
Figure 6.13 shows the overheads considering separately code generation strategies Min-Pressure and Min-Recovery. Across all benchmarks the geometric mean overhead for Min-Pressure is 9.1%, whereas for Min-Recovery it is roughly double at 18.1%. The higher overheads for Min-Recovery are predominantly due to the increased compiler-induced overheads in compiling for this strategy. However, the overheads of contextual idempotence and also of stalling for branch resolution each play a small factor as well.

Overall, it appears likely that using idempotence to achieve recovery from mis-predicted branches is not profitable, although the potential improvements from allowing branch-dependent instructions to issue early were not evaluated. Using idempotence simply to enable out-of-order retirement, however, appears like it could enable overall efficiency savings. In general, the overhead of strategy Min-Pressure is less than 10%, and the overheads are particularly low for floating point benchmarks, where the relative cost of additional register-to-register move and stack spill/reload operations is low compared to the latency of the floating point operations.

6.3 Hardware Fault Recovery in Emerging Architectures

As CMOS technology scales, individual transistor components will soon consist of only a handful of atoms. At these sizes, transistors are extremely difficult to control in terms of their individual power and performance characteristics, their susceptibility to soft errors caused by particle strikes, the rate at which their performance degrades over time, and their manufacturability—concerns commonly referred to as variability, soft errors, wear-out, and yield, respectively.

Checkpoints provide a conceptually simple solution to these problems. However, in the context of hardware faults, checkpoints are problematic for several reasons. First, software checkpoints often have high performance overhead and hence, to maintain reasonable performance, hardware support is often necessary. This hardware support, however, forces interdependencies between processor structures, occupies space on the chip, and entails recurring energy expenditure regardless of failure occurrence. Particularly for emerging massively parallel and mobile processor designs, the per-core hardware support comes at a premium, while the recovery support may be desirable only



Traditional Applications	Traditional Applications	Emerging Applications	Emerging Applications
No Recovery	Hardware Recovery	Hardware Recovery	Idem. Recovery
Traditional Architecture	Traditional Architecture	Emerging Architecture	Emerging Architecture
Reliable CMOS	Unreliable CMOS	Unreliable CMOS	Unreliable CMOS

Figure 6.14: The evolution of hardware, architecture, and applications in the context of hardware fault recovery.

under specific or rare circumstances. Hardware checkpointing resources are also rarely exposed to software, and are even less often configurable in terms of their checkpointing granularity, limiting their wider applicability. Finally, checkpoints have limited application visibility and are often overly aggressive in saving more state than is required by the application.

Hardware Fault Recovery Using Idempotent Regions

Compared to checkpoints, idempotence-based recovery of hardware faults presents a low-overhead, more configurable alternative. When hardware detection and recovery support is unavailable, such as on current-generation GPUs or mobile CPUs, idempotence-based recovery can be opportunistically combined with software detection techniques for recovery. And even when hardware detection and recovery support is available, idempotence can still provide a more efficient alternative to hardware checkpoints when the inherently idempotent regions in programs are large and thus the overheads of executing idempotent regions is low.

Figure 6.14 shows the evolutionary path to idempotence-based recovery considering recent trends in hardware, architecture, and applications. Historically, traditional applications running on traditional superscalar processor architectures built on top of perfect CMOS technology required no recovery as indicated on the left of the figure. Even with imperfect CMOS, these applications still work best utilizing hardware recovery when running on traditional processor architectures as shown center-left. However, with emerging data-parallel applications containing large inherently idempotent regions and running on emerging architectures, hardware recovery introduces the inefficiencies described above, as indicated center-right. In the future, while hardware substrates will be unreliable, mechanisms that provide flexibility to software and keep the architecture simple are desireable. An architecture that exposes the occurrence of hardware faults to allow software recovery using idempotence enables synergy between applications and architectures as shown on the right.

Section Organization

The remainder of this section (Section 6.3) presents a range of architecture designs that support idempotence-based recovery of hardware faults. Section 6.3.1 describes the architecture design issues, ranging from fault detection support to the hardware organization, and Section 6.3.2 presents an evaluation.

6.3.1 Fault-Tolerant Design

This section presents the design of emerging architectures that implement fault tolerance using idempotence. The section first discusses the hardware benefits of using idempotence for fault recovery. It then discusses the necessary ISA support, detection support, and possible hardware organizations.

Hardware Simplification

Idempotence-based fault recovery provides several hardware benefits. First, the hardware need not provide support for buffering, checkpointing, or rollback for software-recoverable errors. Second, idempotence reduces hardware design complexity because design margins to account for silicon uncertainity can be relaxed. This also potentially improves energy efficiency, as it allows hardware to be designed for correct and efficient operation under common case conditions, but with possible failures under dynamically worst case conditions. The overall result is hardware that is error-prone, but is easier to design and potentially more energy efficient.

ISA Semantics

Recovery using idempotence allows instructions to commit potentially erroneous state, while the compiler ensures that this state is either discarded or overwritten after the fault is discovered and recovery is initiated. However, for the compiler to ensure recovery from the fault, the resulting error must be a *locally correctable error*, as defined by Sridharan et al. [98], i.e. the error must be spatially and temporally contained, forcing the ISA constraints listed below. These constraints were previously implicitly enumerated in the idempotence taxonomy's definition of execution failure from Section 2.2, and are re-iterated here only for completeness:

1. Errors must be spatially contained to the target resources of an idempotent region's execution. In other words, an instruction must not commit corrupted state to a register or memory location not written to by other instructions in the idempotent region. For stores, this means that a store must not commit if its destination address is corrupt, or if the store is reached through erroneous control flow. A simple (but high overhead) way to handle this is to stall on the error detection logic prior to committing a store. For other instructions that write only to registers, a tight coupling between the detection logic of the destination register datapath and the instruction commit logic enables rapid resolution of writes to incorrect destination registers.

2. The contents of memory locations must not spontaneously change, e.g. due to a particle strike. Idempotence-based recovery depends on traditional mechanisms such as ECC to protect memories, caches, and registers from soft errors. Other errors that cannot be temporally contained to the scope of a idempotent region, such as most faults in the cache coherence or cache writeback logic, are also not recoverable using idempotence.

3. Arbitrary control flow is not allowed. Control flow must follow the program's static control flow edges. Note that under contextual idempotence faulty control *decisions* are still acceptable since the static control flow is not violated.

4. Hardware exceptions must not trigger until hardware detection ensures that the exception is not the result of an undetected hardware fault.

Execution may leave an idempotent region once the hardware detection guarantees error-free execution. In the event of an error, the hardware must trigger recovery at some point before execution leaves the idempotent region.

Fault Detection Support

Idempotence-based recovery requires support for low-latency fault detection in either software or hardware. Two viable hardware alternatives are Argus [68] and redundant multi-threading (RMT) [74]. Argus provides comprehensive error detection specifically targeted at simple cores, and RMT runs two copies of a program on separate hardware threads and compares their outputs to detect faults. Software detection techniques such as instruction-level dual-modular redundancy (DMR) [86, 78] are also viable techniques when hardware support is not available but resiliency is important. For example, commodity GPUs are able to provide order-of-magnitude performance improvements, but due to their commodity nature, are typically unreliable. Software DMR allows GPUs to still offer substantial performance improvements while simultaneously providing reliable execution for applications that need it, such as scientific computing applications.

Hardware Organization

While hardware that uses idempotence everywhere and has no recovery support at all is possible, it is disruptively different from existing hardware and may not be optimal considering that (a) architectures are becoming increasingly heterogeneous [20], and (b) the performance of compiling for idempotence is heavily dependent on the workload (see Chapter 5). In particular, workloads that have large idempotent regions are the best candidates for idempotence-based fault recovery because hardware faults are presumably rare. In this case the re-execution penalty is outweighed by the performance savings of constructing large idempotent regions.

Configurations that partially implement idempotence, such as the ones shown in Figure 6.15, can be incrementally built into existing hardware organizations and can opportunistically exploit idempotence where it is most beneficial. Below, we consider three such organizations with both simplified hardware and normal hardware, where idempotent regions execute on the simplified hardware and other code executes on the normal hardware, and where this partitioning can be configured either statically, at manufacturing time, or dynamically, at runtime.

A first alternative is a statically configured heterogeneous architecture with support for relatively fine-grained parallelism, where idempotent regions are enqueued on a neighboring, unreliable

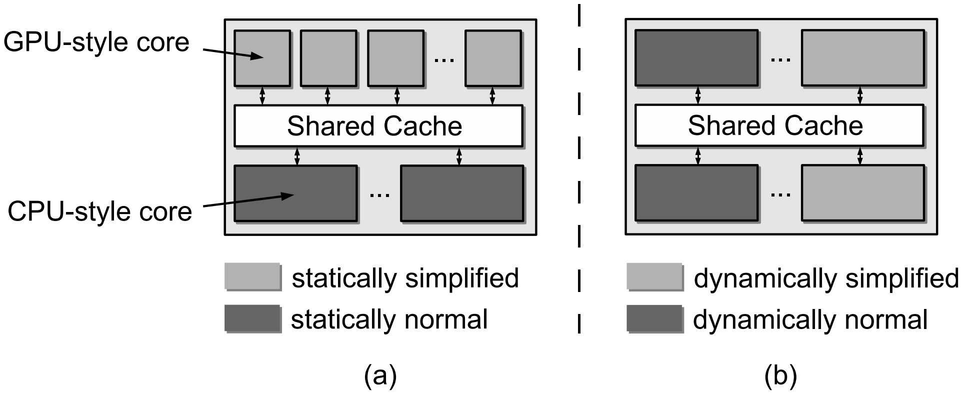


Figure 6.15: Hardware organizations that uses idempotence opportunistically.

core (such as a GPU-like SIMD core or Cell-like SPE core). This is the configuration shown in Figure 6.15(a). A second alternative is an organization where both hardware detection and recovery support exist, but where the recovery support is adaptively disabled to save power and energy when executing low-overhead idempotent regions. This is one example of the configuration shown in Figure 6.15(b). Finally, a third alternative is a dynamically configured architecture that does not have hardware recovery support but has efficient detection support and uses dynamic frequency and voltage scaling (DVFS) inside idempotent regions to run more efficiently in the presence of faults when the trade-off is favorable. This is another example of the configuration shown in Figure 6.15(b).

6.3.2 Fault-Tolerance Evaluation

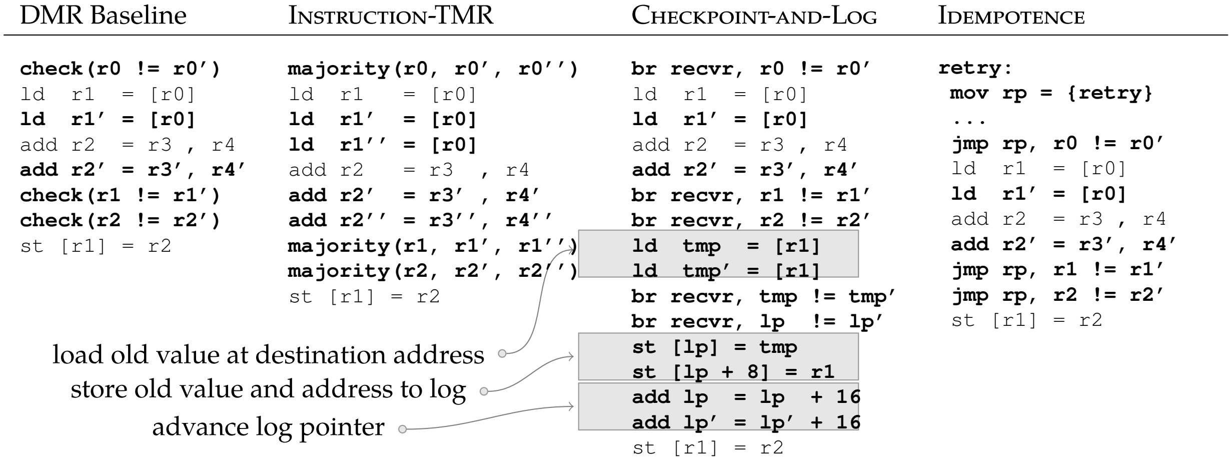
This section presents an abstract evaluation of the fault-tolerant architecture design of the previous section. A more detailed evaluation is covered in co-authored work on demonstrating the application of the static analysis algorithm proposed in this thesis [29] as well as in work on the Relax framework [27].

Experimental Method

To evaluate our fault-tolerant architecture design, we consider the benchmarks from the SPEC 2006 [99], PARSEC [16], and Parboil [100] suites. Using our LLVM compiler, we compile these benchmarks using code generation strategy Pressure with threshold value 5 and assume contextual idempotence to allow divergent control flow effects over pseudoregisters.

We consider as a fault-tolerance scenario the case for recovery from transient hardware faults (soft errors) and evaluate against two other compiler-based recovery techniques. For all techniques we assume compiler-based error detection using instruction-level dual-modular redundancy (DMR), with detection at load, store, and control flow boundaries, as previously proposed by Reis *et al.* [86] and Oh *et al.* [78]. In our experiments, the DMR is modeled by enabling only half of the available registers in the compiler and duplicating instructions to use the other set of available registers during simulation in the instruction stream. We note that while the overhead of such DMR techniques can be high (typically 1.5x [86]), we assume that this overhead is more than offset by any performance improvements running on a commodity architecture that is not inherently fault-tolerant. An example of where this might be true is in running highly sensitive scientific compute kernels on GPUs, where no fault-tolerant alternative to a GPU exists that provides comparable performance.

Figure 6.16 illustrates the behavior of the three recovery alternatives we consider. The first recovery technique, Instruction-TMR, implements TMR at the instruction level. Our implementation attempts to replicate the work of Chang *et al.* [21], which adds a third copy of each non-memory instruction and use majority voting before load and store instructions to detect and correct failures. We support the majority voting as a single-cycle operation. The second technique, Checkpoint-and-Log, is an implementation of software logging similar to logging in software transactional memory systems [48]. Before every store instruction, the value to be overwritten is loaded and written to a log along with the store address, and the pointer into the log (assigned a dedicated register, `lp`) is incremented. In our implementation, as the log fills, the log is reset and a register checkpoint is taken, which starts a new checkpointing interval. We assume a 16KB log size (1K stores per checkpoint interval) with intelligent periodic polling for log overflow using a technique similar to that proposed by Li and Fuchs [64]. In our simulations, all log traffic writes through the L1 cache, and we optimistically assume that both the register checkpointing and periodic polling contribute no runtime overhead. The final technique, Idempotence, is our idempotence-based recovery technique. Here, as each idempotent region boundary is encountered, its address is written to the register `rp`. In the event that a fault is detected, execution jumps to the address contained in `rp` (the use of


DMR Baseline

```
check(r0 != r0')
ld  r1  = [r0]
ld  r1' = [r0]
add r2  = r3 , r4
add r2' = r3', r4'
check(r1 != r1')
check(r2 != r2')
st [r1] = r2
```

Instruction-TMR

```
majority(r0, r0', r0'')
ld  r1   = [r0]
ld  r1'  = [r0]
ld  r1'' = [r0]
add r2   = r3  , r4
add r2'  = r3' , r4'
add r2'' = r3'', r4''
majority(r1, r1', r1'')
majority(r2, r2', r2'')
st [r1] = r2
```

Checkpoint-and-Log

```
br recvr, r0 != r0'
ld  r1  = [r0]
ld  r1' = [r0]
add r2  = r3 , r4
add r2' = r3', r4'
br recvr, r1 != r1'
br recvr, r2 != r2'
ld  tmp  = [r1]
ld  tmp' = [r1]
br recvr, tmp != tmp'
br recvr, lp  != lp'
st [lp] = tmp
st [lp + 8] = r1
add lp  = lp  + 16
add lp' = lp' + 16
st [r1] = r2
```

Idempotence

```
retry:
 mov rp = {retry}
 ...
 jmp rp, r0 != r0'
 ld  r1  = [r0]
 ld  r1' = [r0]
 add r2  = r3 , r4
 add r2' = r3', r4'
 jmp rp, r1 != r1'
 jmp rp, r2 != r2'
 st [r1] = r2
```

load old value at destination address
store old value and address to log
advance log pointer

Figure 6.16: Three software recovery techniques on top of instruction-level DMR. Changes over original load-add-store sequence in bold.

a register to hold the restart address is necessary to handle potentially overlapping control flow between regions).

We simulate our benchmarks using gem5 [17] for ARM to measure the overheads of the three techniques, assuming a simple two-issue in-order core that completes most instructions in a single cycle. We do not inject faults on the assumption that fault occurrence is rare and that subsequently the act of recovery does not signficantly impact performance. The hardware configuration we model in gem5 and the simulation methodology is otherwise the same as for the CPU design evaluation of Section 6.2.3.

Results

Figure 6.17 presents results comparing the overhead of the three techniques relative to performance of the underlying DMR. Across all benchmarks, Instruction-TMR performs worst with 29.3% geometric mean performance overhead, Checkpoint-and-log has 22.2% overhead, and Idempotence performs best with only 9.0% overhead.

Compared to Instruction-TMR, Checkpoint-and-log performs worse for applications with frequent memory interactions, such as several SPEC INT applications, but better for all other applications where its per-instruction overheads are lower. Overall, Idempotence outperforms both

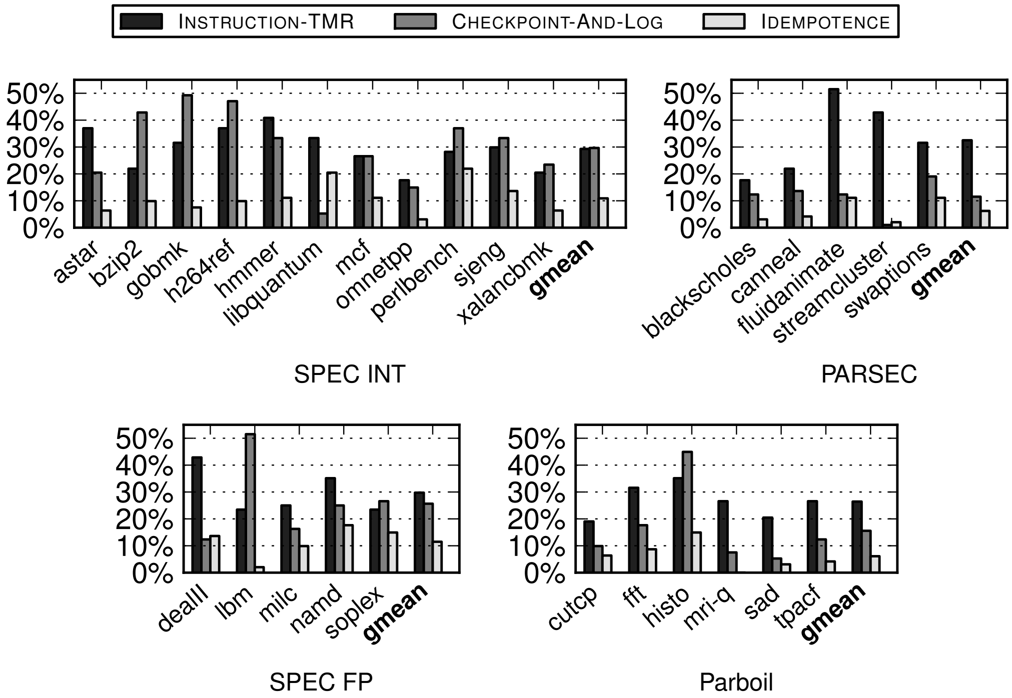


Figure 6.17: Overhead of the three software recovery techniques relative to the DMR baseline.

techniques by a significant margin. It avoids the redundant operations added by Instruction-TMR to correct values in-place, and avoids the overheads associated with unnecessary logging in Checkpoint-and-log. In particular, under Checkpoint-and-log, only the first memory value written to a particular memory location *must* be logged. However, the occurrence of this first write is not statically known and cannot be efficiently computed at runtime. Additionally, Idempotence understands the concept of a store to memory being idempotent, which precludes the necessity for logging, and it also preserves local stack memory more efficiently with its fully-integrated compile-time approach.

6.4 Summary and Conclusions

This chapter explored several applications of idempotence in architecture design. It considered specifically its application in the GPU (high-throughput), CPU (general-purpose), and fault-tolerant architecture space.

GPUs: Modern GPUs do not support exceptions, severely limiting their usability and programmability. Specifically, they do not provide demand-paged virtual memory support or provide efficient means to switch between running contexts. Supporting these features using traditional techniques is not economical for GPUs—the power and area resources are typically better spent elsewhere. However, we observe that GPU workloads contain large regions of code that are idempotent and that these regions can be formed in such a way that contain little live register state at their entry point, enabling both features. The hardware support is minimal and the construction of idempotent regions is moreover fully transparent under the typical dynamic compilation framework of GPUs. Our evaluation of traditional GPU workloads demonstrates very low overheads for recovery from page faults using idempotence.

CPUs: For CPUs, energy efficiency is today's primary design constraint. While CPU workloads tend to have more traditional characteristics than those of GPUs—i.e., they have less inherent idempotence—incremental hardware checkpointing additions remain expensive and idempotence can enable better energy efficiency with simpler hardware. We specifically demonstrated how idempotent processing simplifies the design of in-order processors, which conventionally suffer from significant complexities to support the execution of variable latency instructions and enforce precise exceptions. Idempotence can eliminate much of this complexity and allow instructions to retire out of order. Our evaluation demonstrates that idempotence can enable fundamentally simpler hardware with typical compiler-induced runtime overheads in the range of 5-20%.

Fault-Tolerance: As hardware devices continue to shrink, reliability is an ever-growing concern. To allow programs to operate correctly even in the face of hardware transient or permanent faults, some form of recovery support is increasingly needed. However, emerging hardware such as GPUs and embedded processors often employ simple, in-order cores to maximize throughput and energy efficiency with little or no support for buffering or checkpointing state. Idempotence-based recovery can enable flexible hardawre fault recovery support with very little hardware. We describe ISA-level requirements of how faults can manifest to allow recovery using idempotence, and describe possible hardware organizations and supporting detection mechanisms. Our evaluation shows that idempotence-based recovery is more than twice as efficient as (has less than half the overhead of) two competing state-of-the-art compiler-automated recovery techniques.

7 Related Work

The idea of leveraging idempotence for recovery is fairly new and has previously received only a modest amount of attention from researchers. With respect to the compiler work of Chapters 3-5, the only related work is the work of Hampton [46] and Feng *et al.* [36], and with respect to the idempotence taxonomy of Chapter 2 there has been no prior work—the closest related work we are aware of would be the work of Sridharan *et al.* on their taxonomy on error recovery and correction in software [98].

There has, however, been some broader work on the applications of idempotence for recovery in architectures and programming languages. Below, Section 7.1 expands on related work in this area, with a discussion of related compiler work included therein. Section 7.2 then ends by reviewing alternative, classical approaches to recovery.

7.1 Idempotence-Based Recovery

Over the years, idempotence has been both explicitly and implicitly employed as a recovery alternative to checkpoints. Tables 7.1 and 7.2 classifies known prior work in terms of its application domain and the level at which idempotence is used and identified. As the tables show, prior work on idempotence (also referred to as “restartable program regions”) has been sporadic and applied only under specific domains such as speculation and multi-threading (Table 7.1), and often only under restricted program scope using only limited or no static analysis (Table 7.2).

Technique name	Year	Application domain
Sentinel scheduling [66]	1992	Speculative memory re-ordering
Fast mutual exclusion [15]	1992	Uniprocessor mutual exclusion
Multi-instruction retry [63]	1995	Branch and hardware fault recovery
Atomic heap transactions [90]	1999	Memory allocation
Reference idempotency [59]	2006	Reducing speculative storage
Restart markers [47, 46]	2006	Virtual memory for vector machines
Relax [27]	2010	Hardware fault recovery
Data-triggered threads [102]	2011	Speculative multi-threading
Encore [36]	2011	Hardware fault recovery

Table 7.1: Previous applications of idempotence in architecture design.

Technique name	Program scope	Idempotence analysis
Sentinel scheduling [66]	Speculative regions	Register-level analysis
Fast mutual exclusion [15]	Atomicity primitives	Programmer inspection
Multi-instruction retry [63]	Whole program	Antidependence analysis
Atomic heap transactions [90]	Garbage collector	Programmer inspection
Reference idempotency [59]	Non-parallel code	Memory-level analysis
Restart markers [47]	Loop regions	Loop analysis
Relax [27]	Selected regions	Programmer inspection
Data-triggered threads [102]	Selected regions	Programmer inspection
Encore [36]	Selected regions	Interval analysis

Table 7.2: Program scope and idempotence analysis in prior work.

Idempotence in CPU Design

Specifically for CPUs, one of the earliest uses of idempotence is the work of Mahlke *et al.* on *sentinel scheduling* (1992), which uses restartable instruction sequences for exception recovery to enable compiler speculative memory re-ordering in VLIW and superscalar processors [66]. Whenever a potentially-excepting instruction is speculatively re-ordered, the compiler inserts a non-speculative sentinel instruction at the instruction's logical point of occurrence to check if an exception occurred. The technique to recover from such an exception is to simply re-execute from the point of the speculatively re-ordered instruction after handling the exception, with the program compiled such that the path between the speculatively re-ordered instruction and the sentinel is idempotent. Although this technique is similar in spirit to the techniques explored in this dissertation, our techniques are far more general and can be applied across a broader range of circumstances.

In an ostensibly unrelated domain, the *atomic heap transactions* work of Shivers *et al.* (1999) [90]

and the *fast mutual exclusion* work of Bershad *et al.* (1992) [15] both explore using idempotence to achieve atomicity on uniprocessors. Both observe that, when a process is de-scheduled in the midst of an atomic operation that logically commits only upon execution of the final instruction, restarting the operation from the beginning of the operation maintains the perception of atomicity in a uniprocessor environment. Shivers *et al.* apply this observation to provide memory allocator atomicity by advancing a single pointer value at the end of an atomic operation, which then absorbs the operation's work into the state of the program. Similarly, Bershad *et al.* effectively propose to implement the load-linked store-conditional synchronization sequence for uniprocessors as a simple load and store sequence that is restarted if pre-empted.

The work of Li *et al.* on *multi-instruction retry* (1995) develops compiler-driven re-execution as a technique for branch misprediction recovery [63]. Similar to the work of this dissertation, they break antidependences to create recoverable code regions. However, they do so over a sliding window of the last *N* instructions rather than over static program regions. As such, they never explicitly refer to the more general concept of idempotence and 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. The use of static program regions in this dissertation work further allows for the construction of much larger recoverable regions with relatively low overheads.

Finally, Tseng and Tullsen's *data-triggered threads* (2011) [102] and Kim *et al.*'s *reference idempotency* (2006) [59] work apply idempotence in the context automatic parallelization and thread-level speculation. Tseng and Tullsen manually identify and construct idempotent code regions in their programs to run on top of their architecture, which requires parallel tasks to be idempotent because they may be spuriously executed or asynchronously aborted as a result of data modifications. In contrast, Kim *et al.* track idempotent memory references that need not be tracked in the thread-speculative storage, and can instead directly access non-speculative storage, which substantially reduce the occurrence of speculative storage overflow.

Idempotence in GPU Design

This dissertation work is the first to explore idempotence applied specifically to GPU architectures. However, in a very related domain, Hampton and Asanović apply idempotence explicitly to implement virtual memory for CPUs with tightly-coupled vector co-processors [46, 47]. There are several differences, however. First, they explore only simple compute kernels executed by a vector co-processor, consider only loop regions, and do not develop mechanisms to tolerate multiple recurring exceptions. Second, their work also does not tie the benefits of idempotence-based exception recovery with the benefits of the large idempotent regions and the reduced live register state at region boundaries on data-parallel architectures. Finally, their technique does not support exceptions that require visibility to precise program state, whereas this dissertation demonstrates how idempotence provides a unified mechanism for both speculation and general exception support.

Regarding optimizations for context switching specifically, others have developed techniques to reduce the overhead but have failed to synergistically exploit the property of idempotence. Snyder *et al.* describe a compiler technique where each instruction is accompanied by a bit that indicates whether that instruction is a "fast context switch point" [94]. Zhou and Petrov also demonstrate how to pick low-overhead context switch points where there are few live registers [110]. Although both approaches work well for servicing external interrupts, neither approach can be used to service exception conditions associated with a specific instruction such as page faults. Additional examples include the work of Saulsbury and Rice on compiler annotations to mark whether registers are live or dead [89], and the the IBM System/370 vector facility's use of "in-use" and "changed" bits for vector registers to avoid unnecessary saves and restores on a context switch [79]. While these latter two approaches can be used to reduce switching overheads for page faults, the swapped state must still be sequentially precise with respect to the faulting instruction. Idempotence allows us to side-step this requirement for even lower context switch overheads.

Idempotence for Fault Tolerance

Relax (a work of the author that is not otherwise mentioned in this dissertation) was the first to propose leveraging the property of idempotence for hardware fault recovery [27]. It did so over programmer-identified code regions. Encore later leveraged the concepts of Relax to provide low-cost hardware fault recovery for selected code regions automatically using a compiler [36]. In contrast to this dissertation work, Encore builds idempotent code regions using a different type of compiler analysis and using a hybrid checkpointing/idempotence approach. Idempotent regions are formed using compiler interval analysis, where each region corresponds with an interval that, by definition, includes a dominating header node. As a result, their applications do not cleanly decompose into entirely idempotent regions and several non-idempotent regions remain after the interval analysis completes. These regions are selectively made "idempotent" by checkpointing live-in state that they overwrite when the overhead is modest. The Encore authors claim their best-effort approach achieves 97% coverage with an average of 14% overhead across an assortment of SPEC 2000 and MediaBench benchmarks.

Compared to Encore, the approach of this dissertation is fundamentally different. In Encore, intervals are checked for idempotence in a hierarchical manner with idempotent regions greedily formed. In contrast, the static analysis presented in Chapter 3 identifies fundamental data dependence constraints and constructs regions by formulating the region construction problem as a graph cutting problem subject to these data dependence constraints. Among other things, this allows (a) applications to be decomposed solely into idempotent regions, and supports (b) a variety of code generation strategies, such as the Pressure and Recovery strategies of Chapter 4, cleanly and modularly integrated into the region formation process.

7.2 Classical Recovery

There are a wide variety of classical recovery techniques that have been proposed for exception recovery, branch misprediction recovery, and hardware fault recovery, particularly in the general-purpose CPU space, but also, although to a lesser extent, in the throughput-oriented GPU space.

Classical Recovery in CPU Design

There is a long history of prior work on recovery techniques targeted specifically at general-purpose processors. With respect to the complexities of out-of-order execution and support for precise program state in hardware, seminal works include the work of Hwu and Patt on processor checkpoint repair [52], the work of Smith and Plezkun on the re-order buffer, history buffer, and future file approaches for maintaining a sequential execution abstraction [93], and the work of Sohi and Vajapeyam on the unified RUU structure for handling precise exceptions, renaming, and out-of-order execution [95]. These works have had strong influence and variations on them have been integrated into a variety of commercial processor designs [30, 50, 58, 80, 91, 108]. Recent follow-on work in hardware recovery has focused specifically on support for large instruction windows and speculative out-of-order retirement using checkpoints [5, 24, 67].

As a contrast to hardware techniques, software techniques include Gschwind and Altman's exploration of precise exception support in a dynamic optimization environment using repair functions invoked prior to exception handling [40], a technique used in Daisy [32]. Other software techniques for architecture recovery are few and far between, perhaps in part because well-established hardware techniques generally work well enough, while have the side-benefit of enabling support for dynamic out-of-order execution, a standard feature in modern high-performance processors.

With the growth of mobile and GPU architectures and the emergence of energy-efficiency as the primary design constraint, however, software techniques may have renewed potential. In particular, the idempotence technique proposed in this dissertation enables recovery without explicit checkpoints or buffers, instead identifying and constructing implicit, low-overhead checkpoint locations in the application itself. The power and complexity overheads of supporting re-execution from these locations in the hardware are demonstrably small, and results show that the software performance overheads are low enough that the technique is practical and can be used to improve energy efficiency. When used in combination with selective hardware buffering or checkpointing techniques, it can potentially provide unprecedented levels of energy efficiency.

Classical Recovery in GPU Design

Smith and Plezkun discuss the complications of precise exception support in vector architectures (among which the GPU is an example), and suggest duplicated vector register state to achieve it [93]. They argue that such duplicated state is preferable to the performance complications arising from instead buffering completed results. For a processor implementing short-width SIMD processing capability, duplicated register state may be practical, but for a GPU it is not.

The approach of using a unified register file structure to hold speculative state alongside renaming hardware, first proposed by Moudgill *et al.* [73], has been proposed for GPU-like vector architectures as well [34, 35, 61]. However, compared to general-purpose CPUs, the benefits to GPU-like processors are less clear as out-of-order execution (a side benefit of register renaming) is typically not necessary to achieve high performance. Hence, the additional hardware complexity, additional register pressure, and resulting performance loss (the CODE project measures the performance loss due to register pressure at roughly 5% [61] and Espasa *et al.* measure it as even higher [35]) only to support precise exceptions are hard to justify for GPUs.

Lastly, Imprecise exception support mechanisms that involve snapshotting some amount of microarchitectural state have also been proposed. These include the "length counter" technique of the IBM System/370 [79], the "invisible exchange package" of the CDC STAR-100, the "instruction window" approach of Torng and Day [101], the WISQ "exposed re-order buffer" approach of Pleszkun *et al.* [82], and the "replay buffer" approach of Rudd [87]. While these mechanisms provide full exception support, they expose microarchitectural details and are likely too complex for GPUs.

Today, GPUs do not support exceptions, and hence no exception recovery mechanism has proven sufficiently efficient and straightforward that it can be incorporated into modern GPUs, despite the fact that the desire for full virtual memory support on GPUs has been widely recognized. In particular, id Software has demonstrated a need to support multi-gigabyte textures larger than can be resident on GPU physical memory, and develop a technique called *virtual texturing* to stream texture pages from disk [54]. Unfortunately, the technique requires careful scheduling effort on the part of the programmer. GPU virtual memory paging would simplify its implementation and the implementation of similar techniques. Gelado *et al.* recognize the GPU productivity issue and explore asymmetric distributed shared memory (ADSM) for heterogeneous computing architectures. ADSM allows CPUs direct access to objects in GPU physical memory but not vice-versa [38]. This improves programmability at low hardware cost, but falls short of supporting a complete virtual memory abstraction on the GPU.

Classical Recovery for Fault Tolerance

Substantial prior work has explored recovery specifically in the context of fault tolerance and Sorin provides a complete treatment of fault recovery solutions [96]. He describes two primary approaches to error recovery: *backward error recovery* (BER) and *forward error recovery* (FER). Examples of each that are notable with respect to idempotence are considered below.

While idempotence-based recovery is itself a BER technique, it is distinct from other mechanisms primarily in that it does not involve any restorative action (e.g. restoring a checkpoint). It also notably is software based and has a small sphere of recoverability, which is relatively unique: other software approaches have larger spheres of recoverability [64, 81, 106] which comes at a substantial cost to performance, while hardware approaches have both large [83, 75, 97] and small [10, 43, 85] spheres of recoverability. Among hardware approaches, ReVive I/O is particularly notable for its use of idempotence to recover certain I/O operations efficiently. One challenge with hardware approaches in the fault-recovery context, however, is that large hardware checkpointing structures may not be feasible when dealing with highly error-prone environments where the checkpoints themselves cannot be made relatively immune to faults. For this reason, the fine-grained software recovery enabled by idempotence may be a good fit for a possible future with high fault rate systems.

On the FER side, the main competing approach is triple-modular redundancy (TMR). However, even TMR is not commonly employed since it can entail substantial overheads. The most noteworthy application of TMR for fault recovery is Chang *et al.*'s compiler work on transient fault recovery at the granularity of single instructions using TMR on top of DMR [21]. Chang *et al.* also explore two partial recovery techniques in addition to TMR. However, for full recovery functionality, the overheads remain effectively the same as TMR, and this dissertation's results show that idempotence-based recovery typically has better performance than TMR.

7.3 Summary and Conclusions

Compared to prior work on idempotence, the work of this dissertation is unique in many respects: it develops a compiler analysis to uncover the minimal set of semantically idempotent regions across *entire* programs, it uniquely describes both the algorithmic and implementation challenges and techniques for compiling these regions, and explores how the regions can be used to recover from several different types of execution failures considering a range of architecture designs.

8 Conclusions and Future Work

The aim of this dissertation was to explore the concept of idempotence as it applies to computer architecture, and to evaluate its opportunity as a recovery tool in compiler and architecture design. The work arose from the observation, first articulated in the author's work on Relax [27], that checkpoint-based recovery approaches are often overly aggressive, particularly in the context of emerging media-intensive applications, and that re-execution is often a simple and effective means of recovery given the right application behavior. The subsequent follow-on work, presented in this dissertation, identified how applications could be fully and automatically decomposed into safely re-executable (idempotent) regions, and that this decomposition had applications in GPU exception recovery, CPU design simplification, and hardware fault recovery. This chapter reviews the technical contributions of the dissertation, presents possible directions for future work, and offers some final, concluding remarks.

8.1 Summary of Contributions

This dissertation is the first comprehensive study of idempotence in the computer architecture space, its applications therein, and the role of the compiler in potentially identifying and constructing idempotent regions for architecture consumption. Contributions are established in three broad areas:

Idempotence in Computer Architecture: We examined a spectrum of idempotence interpretations ("models"), formulated a taxonomy, performed an empirical workload analysis, and identified two important and practical idempotence interpretations that were further developed, explored, and evaluated.

Compiler Design: We detailed a complete end-to-end compiler design and implementation for the automated formation and code generation of idempotent regions. We developed techniques to uncover the coarsest semantically idempotent regions and to balance a range of design trade-offs in the compilation of idempotent regions.

Architecture Design: We identified and explored in detail the opportunity to apply idempotence to achieve general exception support and efficient context switching on GPUs, simplified processor pipeline design in CPUs, and hardware fault recovery support in emerging architecture designs.

All proposed techniques were evaluated with experimental studies across a diverse suite of benchmark applications. These experimental studies demonstrated the applicability and effectiveness of the proposed techniques, both generally, and in considering specific application domains.

8.2 Future Work

The work presented in this thesis is complete in that it has exhaustively explored the opportunity to use compiler *static intra-procedural* analysis and code generation techniques for recovery using *pure* idempotence. However, the work opens up several interesting avenues for future work in the areas of inter-procedural, dynamic, or hybrid idempotence-checkpointing techniques. Below, Section 8.2.1 considers future work in using inter-procedural and dynamic analysis techniques to enlarging regions and reduce runtime overheads, and Section 8.2.2 considers the opportunity of hybrid checkpointing-idempotence techniques.

8.2.1 Enlarging Regions to Reduce Pressure and Stall Latencies

Enlarging idempotent region sizes has the potential to reduce the register pressure and detection stall latency overheads identified in Chapter 4. Below, two specific means for achieving larger regions are discussed: augmenting the static analysis with interprocedural information, and compiling with optimistic aliasing assumptions.

Interprocedural Analysis, Dynamic Analysis. and Language-Level Constructs

While this dissertation showed that idempotent regions can be large, limited program knowledge sometimes inhibits region sizes unnecessarily. However, there are opportunities to mitigate this problem. First, as shown in previous work by the author (in material not covered in this dissertation), allowing idempotent regions to cross function call boundaries potentially yields region sizes that are larger by a factor of 10 (geometric mean exclusing outliers yielded a factor of 4) [29]. While extending the static analysis of Chapter 3 across procedure call boundaries (beyond run-of-the-mill inlining) is likely to be computationally prohibitive, a simple top-down preliminary interprocedural analysis that identifies whether functions that are wholly idempotent, and annotates them as such, is relatively cheap and can help a great deal. Additionally, incorporating a dynamic analysis that tracks common-case behavior and performs adaptive recompilation, or programmer annotations that effectively say “this function’s execution is behaviorally idempotent”, can also help.

With inter-procedural analysis capability and the capability to save the stack pointer value along with the program counter, recovering back “up” the program stack is always safe since, in the process of resetting the stack pointer, all local variables below the stack pointer will necessarily be regenerated. (Recovering “down” the program stack is unsafe due to the possible occurrence of intervening function calls that can clobber the call stack, and hence this behavior remains undefined.) In this manner, better aliasing information (either dynamically informed or programmer directed) and/or the use of more declarative programming styles that allow effective inter-procedural optimization may allow the construction of much larger idempotent regions.

Compiling with Optimistic Aliasing Assumptions

As described in Chapter 5 (Section 5.2), the reduced size of the compiler-generated regions compared to the Pin-measured ideal regions is primarily due to conservative memory aliasing assumptions in the compiler; if the load and store of a memory clobber antidependence may potentially alias, the compiler conservatively assumes they do alias, and does not allow a region to contain the antidependence. The accompanying discussion proposes and evaluates the results of applying more sophisticated alias analysis and/or source code annotations to assist the compiler in confirming no-alias relationships. However, such alias analysis and source annotations are not always a practical. Not only can they substantially increase compile-time and/or excessively burden the programmer, but there are cases where it is impossible to determine with 100% accuracy that two memory locations never alias under all circumstances.

With hardware support, the compiler could make optimistic aliasing assumptions instead of conservative ones. Leveraging existing memory disambiguation hardware support, the compiler could construct two versions of an optimistically idempotent region—one version with the single region constructed assuming no aliasing, and the other version with multiple regions assuming potential aliasing. At the head of the optimistic region, the compiler encodes for the hardware the address of the conservative code version. The hardware then checks for load-to-store aliasing for memory operations inside the region and then, if such aliasing exists, redirects the program to the conservative code region. Assuming the likelihood that potentially aliasing memory operations actually alias is very low, this optimization will allow noticeable increases in constructed region sizes, which is likely to reduce runtime overheads as a result.

8.2.2 Hybrid Checkpointing-Idempotence Techniques

Hybrid checkpointing techniques combining both idempotence with buffering/checkpointing have the potential to perform more efficiently than either pure buffering/checkpointing or pure idempotence alone. Such hybrid techniques can be employed either statically, dynamically, or through some combination of the two.

Static Hybrid Checkpointing with K-Idempotence

Let k-idempotence be the property that a region overwrites no more than k inputs. This dissertation has described how to identify 0-idempotent regions. We can generalize the techniques of Chapter 3 for k-idempotent regions by cutting antidependences such that no region contains more than k clobber antidependences.

At first glance, k-idempotence appears a sensible and attractive extension of this work. Indeed, Pin-based experiments (results omitted) suggest that each incrememntal value of k should in principle allow a steady and noticable increase in average region size. However, when considering k-idempotence it is important to distinguish between statically and dynamically occurring clobber antidependences. For $k = 0$, this distinction is not important since no static clobber antidependences implies no dynamic ones as well. However for $k > 0$, the k value specifically pertains to the number of *dynamic* clobber antidependences. Unfortunately, initial results suggest that determining the number of dynamic occurrences of each static antidependence in a program is hard; in most cases the number of dynamic occurrences is unknown.

To overcome the lack of dynamic information, techniques such as loop unrolling, loop blocking, loop interchange, etc. can help. For example, ordinarily with k-idempotence a potentially unbounded loop containing a clobber antidependence must be cut at least once. However, utilizing loop unrolling, for instance, to unroll the loop k times, a region boundary needs to appear in only one unrolled iteration of the loop[1]. Then, instead of crossing a region boundary at each loop iteration, execution only crosses at each k^{th} loop iteration. These and other ideas relating to k-idempotence are interesting for exploration in future work.

Dynamic Hybrid Checkpointing with Buffering or Logging

For the reasons given above, bounding the value of k in k-idempotence is pragmatically challenging while the transformations required to usefully exploit it may be overly invasive to the standard compiler code generation and optimization flow. Hence, a perhaps preferable alternative is to augment idempotence with selective *dynamic* checkpointing capability; when semantic idempotent regions are small, performance overheads can be high, and allowing almost-idempotent regions in combination with some dynamic checkpointing can improve overall efficiency. As an example, using software checkpointing to log the address and old value of an antidependent memory location can allow an otherwise idempotent region to perhaps span a loop, potentially alleviating register pressure overheads enough to generate an overall performance gain.

It is possible, however, that even such partial idempotence techniques—even with hardware support—are likely to remain unsuitable in a number of cases. For fine-grained recovery from out-of-order execution, for instance, the standard register renaming and re-order buffer approaches of

[1] Although a second boundary is probably still also needed for the reasons given in Section 3.3.2.

modern out-of-order processors are efficient and are synergistic with out-of-order issue techniques. Even among high-performance in-order processors, these techniques work fairly well, and register renaming is a fairly standard and low-overhead technique for these processor designs [14, 39, 44]. While idempotence information could allow some efficiency improvements in in-order designs beyond those demonstrated in Section 6.2, e.g. by informing the processor when saving of an overwritten register is unnecessary because its value is dead, the overall benefit remains unclear considering the ISA modifications required. Nevertheless, this topic remains a largely unexplored area with ample opportunity for deeper exploration and research work.

8.3 Reflections

This dissertation has demonstrated idempotence as a powerful recovery primitive with key applications specifically in the field of computer architecture. Through the exercise of compiler design and implementation, and through the various empirical analyses presented in this dissertation, we have also made several observations and insights, on which we now take the opportunity to reflect. For those who might wish to follow in our footsteps, such reflections may be among the most valuable contributions of this work.

Fitting a Square Peg into a Round Hole

Following our work on Relax, it seemed obvious that the restartability property afforded by idempotence could be applied across whole programs, and could also be applied in a variety of domains beyond simply fault recovery. It was, however, not obvious what those domains were, and in many ways it felt as though we had a solution that, despite its inherent conceptual interest, remained in search of a relevant problem.

One early idea was to apply idempotence to assist recovery in out-of-order issue processors. However, from the very beginning there were doubts. First, recovery from out-of-order issue was a very old problem that many intelligent and far more experienced people had worked on for many, many years. The notion of supplementing out-of-order issue with software support was moreover not new, and had been previously explored in depth, particularly in VLIW and binary translation contexts. Indeed, Transmeta, Dynamo, and Daisy, among others, all implemented various sophisticated hardware/software co-designed techniques to achieve efficient out-of-order execution and recovery [13, 30, 32, 40].

Beyond this obstacle, there were many practical complications as well. We knew intuitively that idempotent regions had to be quite large to keep the compiler-induced run-time overheads low. However, large regions were not always possible, we knew, and even if they were, there was the inescapable truth that large regions forced high re-execution costs. As such, in recovery from frequent mis-speculations such as branch mis-predictions, it seemed that most of the execution time would be spent re-executing. For the case of branch mis-prediction, however, the hope was that by placing idempotent regions next to mispredicted branches the re-execution penalty could be nullified. It turns out, though, that the act of doing so aggravates register pressure effects due to control divergence (see Section 5.3), and while opportunity to reduce these effects may exist through dynamic analysis and selective checkpointing techniques, these techniques seemingly complicate an otherwise fairly straight-forward compilation flow. In the end, it is not at all clear that these techniques would allow idempotence to compete with hardware renaming-based techniques that are generally efficient, operate seamlessly underneath the ISA, and moreover already assist in dynamically resolving data dependences to enable out-of-order issue.

From there, there was some conjecture that we could use idempotence to enable fundamentally new speculative hardware mechanisms of the variety that optimize for common-case conditions, but may fail under worst-case conditions. However, existing out-of-order architectures—which effectively had equivalent (or better) levels of recovery support—did not already do this. So while in principle we could claim to decouple detection from recovery support, in practice there was nothing we could provide that couldn't already be handled by raising some sort of exception signal in combination with existing recovery support in out-of-order issue processors. One promising idea we had was that idempotence could freely enable speculative memory re-ordering coupled with hardware memory disambiguation support using re-execution for recovery. It was not one of these fundamentally new mechanisms, but hiding memory latencies remained an important problem even for simpler architectures. However, that idea, in turned out, was already two decades old [37, 66].

The Elusive Square Hole

It was evident then, fairly early on, that this concept of idempotence was likely to be valuable only for relatively simple architectures that could not afford extensive buffering or checkpointing support. Embedded systems seemed like a great target, but the benefit would have been hard to quantify since the simplification would be at the circuit level (relaxed signaling constraints to reduce power and improve timing) rather at the microarchitecture level. Not only would circuit level simplifications be a hard sell for the architecture community, but we were ill equiped (and I personally was entirely unmotivated) to take on the implementation and evaluation effort.

Our MICRO 2011 paper was thus a gamble, eschewing a detailed power and circuit-level evaluation and instead appealing to intuition by illustrating high-level inefficiencies specifically in high-performance in-order architectures. Without a detailed hardware prototype, however, the work remains inconclusive, and I personally remain unconvinced one way or the other, but the potential for the idea was thus demonstrated. The key challenge that remains is that idempotent region sizes in conventional workloads remain small, and these small sizes result in high register pressure overheads (recall the 60% number from Section 5.4). In the end, the key advantage of idempotence is not in how it simplifies the preservation of register state (after all, increasing the number of general purpose registers by 60% to achieve comparable performance—to the extent that that is even possible—is a daunting proposition), but rather in how it simplifies the preservation of memory state by allowing such state to be freely overwritten *without* preservation. And the larger the idempotent regions, the greater the savings due to this particular aspect. Assuming recovery is infrequent, architectures designed with idempotence in mind thus might benefit most from large idempotent regions, and thus would work best running programs written only in functional, mostly functional [60], domain-specific [107], or other high-level or declarative language styles, since straight C/C++ evidently imposes difficulties.

This of course is what led to the ISCA 2012 paper on applying idempotence to CUDA programs on GPUs. In retrospect, it was perhaps obvious that GPUs were an excellent candidate, since they can be very complex in their non-sequential nature, and tracking the microarchitectural state associated with this complexity to support page faults, even without the need to support backward compatibility in the physical ISA, is difficult for a multitude of reasons. On top of this, we were able to identify a secondary advantage of idempotence in enabling context switching GPU program state with relatively very low overhead. The excellent fit of GPU demand-paged virtual memory can thus be attributed to it meeting the three efficiency requirements for idempotence perfectly; namely that, (1) the type of execution failure (page fault) is infrequent, (2) GPU programs have inherently large regions of code that are idempotent, and (3) detection latencies may be large or difficult to bound, but large detection latencies are easily tolerated with large regions.

8.4 Closing Remarks

During the second half of the 20th century, computer architects primarily focused on the single objective of improving single-thread single-chip performance, and decade after decade they succeeded in continually furthering this objective. However, today the computing landscape and the constraints driving computer architecture research are vastly different from even just ten years ago. Performance is no longer the primary driver of computer architecture research. Instead, power and energy are primary design constraints, the shadow of fundamentally unreliable hardware is everlooming, and, with the emergence of mainstream parallel hardware, usability and programmability are once again important fundamental challenges.

Given this modern context, idempotence is a conceptually powerful tool with many prevailing qualities. In the context of energy efficiency, idempotence can enable fundamentally simpler and more energy efficient processor designs. Additionally, it provides a powerful primitive for supporting fault tolerance and unreliablility, and it can further the usability of massively parallel hardware by enabling key programmer abstractions such as virtual memory. In this dissertation, each of these applications was demonstrated, with a focus on the compiler support to transparently enable them. Although the future is never easy to predict, it is our hope that this work will inspire new insights, motivate future researchers, and enable continued innovation in computer architecture into the rest of the 21st century and beyond.

A Code Generator Specifics

This chapter discusses compiler implementation specifics in compiling loops from SSA to machine code (Section A.1), supporting large structures and arrays (Section A.2), supporting architecture-specific calling conventions (Section A.3), and statically verifying code generation output (Section A.4). Additional details on each of these aspects, and others, are available in the publicly-released source-code compiler documentation [2].

A.1 Out-of-SSA Translation and Back-Edge Clobbers

The application of static single assignment (SSA) form in the static analysis of Chapter 3 eliminated all antidependences and output dependences on pseudoregister locations *except* for those that appear across loop iterations. In Section 3.3.2, loops were cut to enable the clobber dependences that could subsequently still arise to be removed. However, these clobber dependences were not actually removed at that time; the capability was merely provided. In transitioning out of SSA in preparation for register and stack allocation—a process commonly referred to as *out-of-SSA translation*—the code generator must remove this final source of clobber dependences to set the necessary invariant in place.

The problem that needs to be solved we refer to as the "back-edge clobber" problem, and it is illustrated using an example in Figure A.1. While a minimum of two boundary crossings exist for all paths through the loop body, as is required, a clobber dependence on variable x still exists—the assignment to x at the head of the loop clobbers the use of x in the region that crosses the back-edge of the loop. In the pre-allocation phase of the code generator, we insert copies to avoid this. In the case of Figure A.1, we insert a copy from pseudoregister x to a newly-allocated pseudoregister

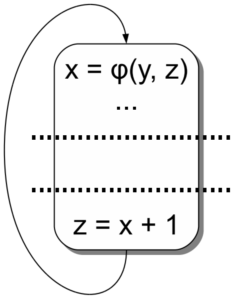


Figure A.1: The "back-edge clobber" problem in out-of-SSA translation.

x' before the back-edge region of the loop, and then replace all *uses* (only the uses) of x in this back-edge region with x'. The solution specific to this example is fairly straightforward. However, a general solution to this problem requires a complex algorithm. In our implementation, we employ an iterative algorithm that converges on a stable solution [2].

We note that migrating the assignment of z before the back-edge region boundary can avoid the need to insert a copy. The general case where this opportunity most often arises is much more complex than in Figure A.1, however. Hence, for simplicity, we do not implement or otherwise explore this optimization as it is not required for correctness.

A.2 Scalar Replacement of Aggregates

After SSA, the second transformation applied in Chapter 3 showed how non-clobber antidependences (write-read-write patterns) on non-local memory could be eliminated by using local storage to hold the result of the first write and replacing the read to use the local storage. This technique can be impractical, however, particularly 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 local stack. In addition to stack memory often being bounded, the state duplication can reduce cache locality and hurt performance.

To support writes to such large or potentially unbounded structures efficiently, our compiler implementation does not duplicate large aggregate structures or arrays in local memory, and hence the static analysis conservatively cuts the potentially non-clobber antidependences that they create on the assumption that their occurrence is rare. For most benchmarks, they are indeed rare, with any additional cuts that they would require masked by inter-procedural effects. Augmented with inter-procedural analysis capability, however, it would be beneficial to incorporate this aspect into the static analysis algorithm. One possible solution is presented in the static analysis evaluation of Section 5.2.

A.3 Support for Calling Conventions and Condition Codes

For most purposes, function calls and returns are non-idempotent since they minimally read-modify-write the stack pointer register. To allow these read-modify-write patterns, we assume that stack pointer register updates are operations that either complete successfully and start a new idempotent region, or fail with no side-effects (see Commit sequencing from Chapter 2). Additionally, the preservation of callee-saved registers at call return sites creates further complexities that are further documented elsewhere [2].

Finally, with respect to condition code registers (e.g. x86 EFLAGS), we make the simplifying assumption that condition code register state is saved and restored prior to re-execution using idempotence. This is because the architecture-specific information necessary to place boundaries to avoid clobber dependences over condition code registers is not available at the LLVM IR level. While this information could be incorporated, it was not done in this work.

A.4 Static Output Verification

The idempotence property can be statically verified in the compiler during code generation. Under contextual idempotence, the simple invariant that must be maintained is that a storage resource live at the start of an idempotent region must not be overwritten anywhere inside that region. The constraint for architectural idempotence is more complex, however. Here, a live storage resource may not be overwritten at some point only if it might have been used before that point (i.e. it was *dynamically* live on entry). We employ an iterative data-flow analysis to determine the latter condition and check that no such storage resources are overwritten. Hence, we are able to statically verify both architectural and contextual idempotence at each step in the code generation process [2].

Bibliography

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

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

[3] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 2008.

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

[5] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, MICRO '03.

[6] AMD. *Evolution of AMD Graphics*. http://developer.amd.com/afds/assets/keynotes/6-Demers-FINAL.pdf.

[7] AMD. *Memory System on Fusion APUs*. http://goo.gl/r72cp.

[8] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide, Rev. 1.3f*. 2011.

[9] ARM. *Cortex-A8 Technical Reference Manual, Rev. r2p1*.

[10] T. Austin. DIVA: A Reliable Substrate for Deep Submicron MicroarchitectureDesign. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, MICRO '99.

[11] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10.

[12] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis (ISPASS)*, ISPASS '09.

[13] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '00.

[14] M. Baron. Cortex-A8: High speed, low power. *Microprocessor Report*, 2005.

[15] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '92.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '08, pages 72–81.

[17] 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.

[18] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

[19] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[20] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12.

[21] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks*, DSN '06, pages 83–92.

[22] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

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

[24] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of The International Symposium on High-Performance Computer Architecture*, HPCA '04.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89.

[26] S. Das. *Loop Dependence Analysis Patch for LLVM*. https://github.com/sanjoy/llvm/tree/lda.

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

[28] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '11.

[29] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler implementation for idempotent processing. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[30] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International ACM/IEEE Symposium on Code Generation and Optimization*, CGO '03.

[31] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '10.

[32] K. Ebcioğlu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97.

[33] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro*, 15(2):33–43, 1995.

[34] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. thew Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 281–292, May 2002.

[35] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '97.

[36] S. Feng, S. Gupta, A. Ansari, S. Mahlke, and D. August. Encore: Low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '11.

[37] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '94, pages 183–193.

[38] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10.

[39] A. González, F. Latorre, and G. Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan & Claypool, 2010.

[40] M. Gschwind and E. R. Altman. Precise exception semantics in dynamic compilation. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 95–110.

[41] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[42] J. Guo, F. HÃ¼ffner, 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.

[43] M. Gupta, K. Rangan, M. Smith, G.-Y. Wei, and D. Brooks. Decor: A delayed commit and roll-back mechanism for handling inductive noise in processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA '08.

[44] T. R. Halfill. Intel's tiny Atom. *Microprocessor Report*, April 2008.

[45] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the Annual Symposium on Computer Architecture*, ISCA '10.

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

[47] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06.

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

[49] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 5th edition, 2011.

[50] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, February 2001.

[51] T. Horel and G. Lauterbach. UltraSPARC-III: designing third-generation 64-bit performance. *Micro, IEEE*, 19(3):73 –85, May/Jun 1999.

[52] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the Annual Symposium on Computer Architecture*, ISCA '87, pages 18–26.

[53] IBM. *SPU Instruction Set Architecture, Rev. 1.2*. 2007.

[54] id. id tech 5 challenges: From texture virtualization to massive parallelization. In *SIGGRAPH '09*.

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

[56] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2011.

[57] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), September 2005.

[58] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[59] 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.

[60] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112.

[61] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the Annual Symposium on Computer Architecture*, ISCA '03.

[62] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International ACM/IEEE Symposium on Code Generation and Optimization*, CGO '04, pages 75–88.

[63] 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.

[64] C.-C. J. Li and W. K. Fuchs. CATCH – Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing (FTCS-20)*, FTCS '90, pages 74 –81.

[65] 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. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200.

[66] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '92, pages 238–247.

[67] J. Martinez, J. Renau, M. Huang, and M. Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '02.

[68] 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.

[69] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10.

[70] J. Menon, M. de Kruijf, and K. Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12.

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

[72] Motorola. *M68000 Family Programmer's Reference Manual*. http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf.

[73] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '93.

[74] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA '02*, pages 99–110.

[75] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. Revivei/o: Efficient handling of i/o in highlyavailable rollback-recovery servers. In *HPCA '06*, pages 203–214.

[76] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '11, pages 308–317.

[77] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Ver. 1.1.* 2009.

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

[79] A. Padegs, B. Moore, R. Smith, and W. Buchholz. The IBM System/370 vector architecture: design considerations. *Computers, IEEE Transactions on*, 37(5):509–520, May 1988.

[80] D. Papworth. Tuning the pentium pro microarchitecture. *Micro, IEEE*, 16(2):8 –15, Apr. 1996.

[81] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[82] A. R. Pleszkun, J. R. Goodman, W. C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter. Wisq: a restartable architecture using queues. In *Proceedings of the Annual Symposium on Computer Architecture*, ISCA '87, pages 290–299.

[83] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA '02*, pages 111–122.

[84] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '01, pages 294–305.

[85] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '01.

[86] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. Swift: software implemented fault tolerance. In *International ACM/IEEE Symposium on Code Generation and Optimization*, CGO '05.

[87] K. W. Rudd. Efficient exception handling techniques for high-performance processor architectures. Departments of Electrical Engineering and Computer Science, Stanford University, Technical Report CSL-TR-97-732, August 1997.

[88] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation or the sun network filesystem, 1985.

[89] A. Saulsbury and D. Rice. Microprocessor with reduced context switching and overhead and corresponding method. United States Patent 6,314,510, November 2001.

[90] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the International Conference on Functional Programming*, ICFP '99.

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

[92] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 135–148, 1981.

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

[94] J. S. Snyder, D. B. Whalley, and T. P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995.

[95] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the Annual Symposium on Computer Architecture*, ISCA '87.

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

[97] 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*, pages 123–134.

[98] V. Sridharan, D. A. Liberty, and D. R. Kaeli. A taxonomy to enable error recovery and correction in software. In *Workshop on Quality-Aware Design*, 2008.

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

[100] 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, 2012.

[101] H. Torng and M. Day. Interrupt handling for out-of-order execution processors. *Computers, IEEE Transactions on*, 42(1), 1993.

[102] H.-W. Tseng and D. Tullsen. Data-triggered threads: Eliminating redundant computation. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA '11.

[103] J. Turley. ARM launches Cortex-A15. *Microprocessor Report*, September 2010.

[104] W3C. *Hypertext Transfer Protocol – HTTP/1.1*. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[105] C.-J. Wang and F. Emnett. Implementing precise interruptions in pipelined risc processors. *Micro, IEEE*, 13(4):36 –43, Aug. 1993.

[106] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS '95*, page 22.

[107] Wikipedia: Domain-specific language, http://en.wikipedia.org/wiki/domain-specific_language.

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

[109] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual ACM/IEEE international symposium on Microarchitecture*, MICRO '91, pages 51–61.

[110] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the Annual Conference on Design Automation*, DAC '06.