The Pennsylvania State University

The Graduate School

College of Engineering

DESIGNING PROCESSING IN MEMORY ARCHITECTURES VIA STATIC ANALYSIS OF REAL PROGRAMS

A Thesis in

Computer Science and Engineering

by

Henry Francis Smith

Submitted in Partial Fulfillment
of the Requirements
for the degree of

Master of Science

May 2018

The thesis of Henry Francis Smith was reviewed and approved* by the following:

Vijaykrishnan Narayanan
Distinguished Professor of Computer Science and Engineering
Thesis Adviser

John Sampson
Assistant Professor of Computer Science and Engineering
Thesis Co-Adviser

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Graduate Program Chair

*Signatures are on file in the Graduate School.

Abstract

Processing in Memory (PIM) architectures have displayed a number of benefits over traditional processor architectures. By inserting compute elements nearer to—or inside—memory, data movement can be drastically reduced. This in turn reduces the time spent and the energy expended moving data, in addition to reducing memory contention and freeing processor cycles for more computation.

However, Processing in Memory architectures will not see popular adoption if there are not applications which can utilize them. Many PIM architectures present a high barrier to entry for software: to effectively utilize processing in memory, entire programs may need to be rewritten. Without promise of increased performance and decreased energy consumption for their application, software designers may not be willing to put in the effort to redesign their software.

In this thesis, we conduct a Processing in Memory design exploration entirely from the software perspective. With the above issues in mind, we focus on developing a PIM architecture which accelerates computation patterns found in real programs.

We first form a hypothesis about simple computation patterns which could be offloaded to memory, and define a basic PIM architecture to accelerate those patterns. Using static analysis tools, we then dig deep into applications and benchmarks to evaluate just how common these patterns are. Based on our results, we improve on our original architecture, finally proposing a new PIM architecture whose design is backed by computation patterns found in real programs.

The primary contribution of this thesis is to present and advocate for a new approach to Processing in Memory exploration. By building our PIM architecture around patterns extracted from real software, we can be more confident not only in our architecture's capability to improve performance and decrease energy, but also in our architecture's potential for adoption by software designers.

Table of Contents

List of Figures **v**

List of Tables **vi**

Acknowledgments **vii**

1 Introduction **1**

2 A Taxonomy of Processing in Memory Architectures **4**
2.1 In-Memory Processing for Applications 7
2.2 Processing Near Memory 8
2.3 Computing In-Place Within Memories 9
2.4 Processing in the Memory Hierarchy 10

3 Detecting Processing in Memory Sequences **13**
3.1 Offloading Load-Load-Op-Store Patterns 15
3.1.1 Our Initial Architecture 16
3.1.2 Finding Load-Load-Op-Store Patterns 17
3.1.3 Results 20
3.2 Moving Beyond Load-Load-Op-Store 26
3.2.1 A New Algorithm 28
3.2.2 Rethinking PIM Offload 29
3.2.3 Classifying PIM Sequences 31
3.2.4 A New PIM Architecture 32

4 Conclusions **36**
4.1 Future Work 37

Bibliography **39**

Appendix A: Code **43**

Appendix B: Interesting Subgraphs **45**

List of Figures

3.1 An example (in LLVM) of load-load-op-store patterns being replaced with an instruction from the new PIM ISA. 16
3.2 Our original pattern, versus the patterns which exist in real code. 26
3.3 Chaining cache instructions to offload a more complicated group of dependent instructions. 27
3.4 Examples of our new pattern. 27

List of Tables

2.1 A taxonomy of Processing in Memory architectures. 7

3.1 The applications and benchmarks used for testing. 15
3.2 Offloadable LLVM instructions. 19
3.3 Number of load-load-op-store patterns found in applications and benchmarks. . . . 20
3.4 Distribution of opcodes found in a load-load-op-store for each application. 21
3.5 Distribution of opcodes found in a load-load-op-store for each benchmark. 22
3.6 Number of op-store patterns. 23
3.7 If an operand isn't a load, what is it? (applications) 24
3.8 If an operand isn't a load, what is it? (benchmarks) 25
3.9 Number of subgraphs found within programs. 29
3.10 Number of unique classes of subgraphs. 32

Acknowledgments

I would like to acknowledge my advisers, Profs. Narayanan and Sampson, who have advised me on far more than just this thesis. I would be in a very different place had they not sparked my interest in research four years ago. I am excited to see where my future will take me, but I will never forget the department where I came from or the people who shaped my early years in research.

Thank you also to my friends and family—here in State College, at home in the Wyoming Valley, in Seattle, and elsewhere—who kept me sane during a challenging semester. Thanks especially to my roommates, who were always willing to provide a welcome distraction, and to Sam, whose unflagging confidence in me sometimes surpassed even my own.

Thank you to Craig Topper, Jason Lowe-Power, and Marek Chalupa, in addition to countless others (from llvm-dev, gem5-users, and elsewhere) who provided technical help. I owe a number of people a number of beers.

Lastly, and most importantly, thank you to my parents. My upbringing has let me achieve so much—for that, I will always be thankful.

This work was supported in part by NSF Expeditions in Computing Program: Visual Cortex on Silicon CCF 1317560 and NSF Award #1640081 E2CDA: Type I: EXtremely Energy Efficient Collective ELectronics (EXCEL).

The views expressed in this work are the author's alone and do not necessarily reflect the views of these sponsors.

Chapter 1

Introduction

For years, microarchitectural innovations have focused on improving the throughput of the processor itself, while hiding the growing processor–memory gap. However, the performance of the overall system is still limited by the need to bring every piece of data all the way to the processor. Often, data will be moved from the furthest reaches of memory, only to be transformed in the processor and immediately sent back to memory. This not only becomes a bottleneck on computation, but also devotes a significant portion of energy towards simply moving data.

Architectural trends have long pointed to the decentralization and parallelization as a solution to the computation bottleneck. Multiprocessor architectures, for example, provide multiple processors on which to compute, thus widening the bottleneck at the top of the hierarchy. Accelerators and GPUs are other great examples of this architectural trend. Accelerators and GPUs themselves are often highly parallel, internally; however, they also represent parallelization at another level up, as they provide another location within (or "next to") the memory hierarchy for computation to occur.

Though these approaches do reduce the compute bottleneck, they are still "moving data to the compute". For each computation, data must still be moved a great distance to and from the pro-

cessor. Processing in Memory (PIM) architectures challenge the traditional hierarchy by instead "moving compute to the data". In a PIM architecture, compute elements are inserted either nearby or within the memories themselves. Data then only has to travel to the nearest compute element. With enough compute elements distributed throughout memory, the bottleneck is loosened significantly, and data movement is drastically reduced.

Processing in Memory is far from a new concept; in choosing to pursue Processing in Memory, we are leaning on more than two decades' worth of research. For many years, research into PIM architectures slowed significantly as memory vendors chose to focus on improving capacity rather than researching the incorporation of processing into memory devices. However, with the growing commercial interest in 3D-stacked logic and memory devices like Micron's Hybrid Memory Cube, Processing in Memory is now becoming a viable concept, and we are seeing a resurgence of PIM research.

PIM architectures promise better performance and reduced power consumption. Due to emerging memory technology, now is also the perfect time to be researching PIM architectures. However, there is a deeper motivation behind Processing in Memory research. By studying such an unconventional architecture, we challenge current assumptions about the state-of-the-art. In pursuing Processing in Memory, it will be necessary to solve a number of significant design problems; graceful solutions to these problems may inspire new directions for our more standard architectures, preventing stagnation. Dataflow architectures are a great example of this knowledge transfer—though not a lasting commercial success, research into dataflow architectures helped to develop the concept of out-of-order processing, seen in nearly all modern processors.

The promised benefits of Processing in Memory, recent memory technology developments, and the broader benefits to architecture as a whole are the reasons which motivate us to explore Processing in Memory architectures. As mentioned above, though, research into PIM raises a significant number of design questions. Perhaps the most apparent problem with moving processing into memory is that we leave behind the familiar processor structures needed for basic operation: particularly, the Translation Lookaside Buffer (TLB) and page table walker needed for virtual-to-physical address translation. Yet even important design considerations such as these will not matter if there are not applications which can utilize the architecture. Thus, even before we tackle the low-level complexities and questions of PIM architectures, we claim there is first the funda-

mental problem of *application*—how can software utilize Processing in Memory, and what kinds of problems can PIM architectures be used to solve?

To truly motivate the study—and eventually, the adoption—of Processing in Memory architectures, we must begin our exploration with real applications. This has been addressed in a top-down method in the past. By starting with a large application category such as graph processing, the application can be broken down and certain parts can be accelerated in memory. Benchmarks are then used to evaluate the architecture.

In this thesis, we take the opposite, bottom-up approach. We begin by first defining a simple PIM architecture, though the specifics of the architecture itself will not be the focus of this thesis. Based on this architecture, we then characterize computation patterns which can be offloaded to memory, and implement a static analysis pass in LLVM to detect these patterns. In taking this bottom-up approach, we seek to investigate existing real-world software for potential patterns which can be offloaded into memory, and use these results to further drive our exploration.

Though we use the results of our exploration to develop a PIM architecture throughout the course of this thesis, this architecture is not the primary contribution of this thesis. Instead, our primary contribution is the explanation and exposition of a novel method for PIM exploration.

In the remainder of this chapter, we describe the layout of the rest of this thesis. Chapter 2 fully introduces the concept of Processing in Memory in the form of a taxonomy of Processing in Memory architectures.

We begin chapter 3 by describing, at a high level, what patterns of computations can be offloaded to our PIM architecture. We then formalize these patterns using the Program Dependence Graph. Once these patterns are formalized, we describe the process of detecting potential PIM-offloadable patterns in source code using LLVM, and present results of using these LLVM passes on real workloads. Using our results, we then develop a more complex model of PIM-offloadable computation.

Finally, in chapter 4, we draw conclusions and expand on possible future directions.

Chapter 2

A Taxonomy of Processing in Memory Architectures

Modern processors are starved for data; a primary cause is the growing gap between processor and main memory speeds [22]. In the modern era, memory and logic manufacturers have two diverging goals—the former, to increase memory capacity, and the latter, to increase speed. As memory and logic pull in different directions, the interface between them becomes the primary source of data movement inefficiencies. Furthermore, the common solutions for hiding memory latency (prefetching, etc.) has been shown to *increase* memory bandwidth requirements [11]. As our applications become increasingly data-centric, this problem will become more and more obvious. At the same time, as applications become more data-centric, they expose more data parallelism. Our current architectures (e.g. modern vector processors) are still inherently sequential, however, and cannot exploit these levels of data parallelism effectively. Finally, as a consequence of our memory hierarchy design, a disproportionate amount of time and energy is also spent simply moving data, as compared with the amount of energy spent on actual computation [1, 6].

All of these issues can be wrapped up under the single problem of orchestrating data movement.

Processors can, in a simplistic way, be seen as systems which fetch data from memory, transform the data, and ultimately return the data back to memory. In this formulation, it is easy to see a number of the problems we have listed above. The lack of data parallelism manifests in the need for every piece of data to travel through the same processor, while excessive data movement is made obvious by the need to bring data to the processor, regardless of where data lives in memory.

The modern multiprocessor architecture attempts to solve some of these problems, but only exacerbates others. Multiprocessors were developed to tap into the blatantly underexploited parallelism of common programs. However, the more processors we add to the system, the more bandwidth is needed to keep the system fed. GPUs and accelerators are another great example of potential solutions to these problems; however, like multiprocessors, they have their own issues. GPUs and accelerators truly harness the data parallelism of common applications, making computation incredibly quick on these devices. However, there is still a cost of movement, as data needs to be moved beyond the bounds of the memory hierarchy.

Stream and dataflow processing (explored through notable projects like Imagine [12] and Merrimac [7]) were early examples of ideas that faced the memory bandwidth issue head-on, rather than simply attempting to cover it up. Stream and dataflow programs exhibit high memory locality; by adopting these models of computation and increasing the size of on-chip memory, these architectures were able to significantly decrease the bandwidth requirements to external memory. As we will see throughout this thesis, our discussion of offloadable computation for Processing in Memory architectures is inspired by stream and dataflow concepts.

Processing in Memory architectures present potential solutions to all of these problems. The problem of memory bandwidth arises because the processor is separated from the memory by a bus, or some other kind of network. This network, though designed for maximum bandwidth, is constrained and often overloaded. The internal bandwidth of a memory chip, on the other hand, is significantly higher [13, 22]. PIM architectures exploit this internal bandwidth—if data can move to a compute element entirely within the memory, then bandwidth should not be a limiting factor. In its simplest formulation, a PIM architecture simply places a large number of simple compute elements into the memory. With enough compute elements, a PIM architecture can potentially take advantage of massive data parallelism. Lastly, the most evident benefit of PIM is the reduction of data movement, when moving data to the compute element. Even placing compute elements at a

single level of the hierarchy—within the L2 cache, for example—total data movement could be drastically reduced.

Processing in Memory is far from a new concept; as we'll see, many of the most notable attempts at Processing in Memory architectures are nearly two decades old. However, we have recently seen a resurgence of interest in Processing in Memory—at the time of writing, Google Scholar returns 61 PIM architecture papers in 2018 alone [25]. We can partially attribute this resurgence in PIM to the manufacturers—memory manufacturers are now offering merged logic–memory devices, which can potentially make PIM architectures commercially viable. The availability of this new technology has sparked renewed interest in the subject; with support from commercial manufacturers, PIM architectures can potentially be manufactured at scale.

Before we begin our own exploration of Processing in Memory, we begin by first exploring previous works on Processing in Memory. The term "Processing in Memory" has been overloaded even since its first uses. In the rest of this chapter we clarify the different definitions of the term, while simultaneously presenting the foundational work which this thesis is built on. We present a taxonomy of various architectures which have, in the past, fallen under the broad label of "Processing in Memory". Some sections of this taxonomy will map more closely to the specific definition of Processing in Memory which we use in this thesis, while other definitions are presented simply to give context or explanation to a very overloaded phrase.

We identify four common definitions of Processing in Memory used in the literature. Note that these definitions are not necessarily orthogonal; a PIM architecture could fit into more than one of these categories.

We first briefly look at **in-memory processing for applications.** For applications and systems designers, "Processing in Memory" generally does not refer to physically adding new compute elements into the system; instead, it refers to operating on a working set that is stored completely within main memory, rather than on disk or other slower storage. Often motivated by further improving these in-memory applications, researchers began putting custom hardware near memory, to operate on data without needing to go to the CPU. These architectures fall into the second section of our taxonomy, which we label **Processing *Near* Memory**.

At the highest level, the multiple meanings of Processing in Memory can be broken down into hardware-based and software-based definitions. From the hardware perspective, the most literal

Category	Description	Examples
PIM for Applications	Moves applications into memory	Spark[27], SAP HANA [9]
Processing Near Memory	Accelerators near memory, e.g. on merged logic–DRAM	IMPICA[10], Tesseract[2]
Computing In-Place in Memories	Using memory device properties to do computation in-place	Compute Caches[1], PRIME[5]
Processing in the Memory Hierarchy	Giving memory compute semantics	Active Pages[21], Compute Caches[1]

Table 2.1: A taxonomy of Processing in Memory architectures.

interpretation of Processing in Memory is the act of **computing in place within memory devices,** which is the third section of our taxonomy.

The last and most important section of our taxonomy merges a number of these ideas together. It takes Processing Near Memory one step further, actually integrating the processing into the semantics of memory access. Often, these architectures will take advantage of compute-in-place hardware designs. We call this section of the taxonomy **Processing in the Memory Hierarchy.**

This taxonomy is crafted with the goal of presenting the literature which is foundational to this thesis. It is not meant to be comprehensive over the broad Processing in Memory field. Furthermore, the four sections of our taxonomy are crafted to best outline previous research in the context of this thesis. In other contexts, other taxonomies may make more sense. We refer the reader to Loh et al. [19] for another useful taxonomy of Processing in Memory architectures.

2.1 In-Memory Processing for Applications

Modern applications are more and more data-centric, and thus impose stricter demands on the memory system. As memory gets cheaper and larger, it becomes more feasible to store all of an application's data within main memory [28]. This "in-memory processing" is desirable, as previous disk-based systems were limited by long latencies to disk.

Though this definition of Processing in Memory is not what we will focus on in this thesis, it is important to mention for a number of reasons. The motivations behind Processing in Memory

at the application level and Processing in Memory at the architecture level are the same: modern applications need far more memory bandwidth. Databases were among the earliest motivations for Processing in Memory at the application level. Berkeley DB is an early example of a database which is stored completely in memory, and treats the disk as an archival storage device [20]. To this day, databases continue to be a hot area for this type of development, with SAP HANA and Apache Spark being two modern examples [9, 27]. However, a number of other applications have also developed an "in-memory" flavor, including streaming, graph, and key-value store applications.

Processing in Memory at the application and architecture levels originate from the same motivations. However, the connections between the two research areas go deeper. Much research into Processing in Memory at the architectural level is *motivated by* research into Processing in Memory at the application level. Memcached, a popular piece of software for caching in main memory, is a great example of in-memory software that has been a popular target for architectural acceleration [16, 17], especially considering its heavy usage at companies like Facebook and Google.

As this definition of Processing in Memory is not the focus of this thesis, we defer to Zhang et al. [28] for a thorough overview of in-memory data management for a number of application types.

2.2 Processing Near Memory

As discussed in the previous section, many in-memory applications became targets for *software* acceleration through data management strategies, such as keeping as much data as possible in main memory. At the same time, though, these applications can be searched for potential *hardware* acceleration. Applications such as relational databases have regular operations which can potentially be accelerated—Kung and Lehman [14] present a very early example which accelerates relational operations like union and intersection.

Recent advances in merged logic–DRAM fabrication—Micron's Hybrid Memory Cube (HMC) being a prime example—have opened the possibility for placing such accelerators near memory. Hybrid Memory Cube layers logic on top of DRAM, which is why these designs are referred to as 3D-stacked memories. The logic and memory layers are connected with high-bandwidth through-silicon vias (TSVs). Accelerators placed on the logic layers of 3D-stacked memories are often

designated as “in memory”, due to the high-bandwidth connections to memory through TSVs. We prefer to keep the “near memory” label for these designs, simply to separate them from “Processing in the Memory Hierarchy” architectures.

The simplest characterization of a Processing Near Memory design is simply as an accelerator. From the point of view of the programmer, a near-memory accelerator often appears just as a standard accelerator would—memory-mapped into a noncacheable region of memory. However, from the accelerator’s point of view, it has the benefit of increased bandwidth and decreased latency to memory. The IMPICA pointer-chasing accelerator [10] and the Tesseract graph processor [2] are two examples, both of which are designed using 3D-stacked memory.

Processing Near Memory is perhaps best viewed as a gray area between accelerators attached over a bus at one extreme, and Processing in the Memory Hierarchy (which will be presented in section 2.4) at the other. By moving an accelerator nearer to the data, these designs take advantage of increased bandwidth and decreased latency. By not incorporating the accelerators directly into the memories, these designs seek to avoid a number of issues, including fabrication and virtual-to-physical address translation [23].

2.3 Computing In-Place Within Memories

At a hardware design level, the term “Processing in Memory” describes a very broad class of designs which compute in place, with little to no movement of data. This definition of Processing in Memory is far from the definition we saw in section 2.1; however, much like section 2.1, this section of the taxonomy is important in the larger context, as the development of in-place computation at the hardware level encourages architects to examine how these designs might fit into the larger system.

Often, Processing in Memory at the hardware level is achieved using the inherent properties of memory technologies to compute. ReRAM is a current example; this memory technology can be used to implement a multiply-accumulate function in-place. PRIME [5] is just one example of an architecture which utilizes this property, in this case to create main memory arrays for accelerating neural networks. Aga et al. [1] is another example which computes in place within caches; we will look at this example in more detail in the next section.

2.4 Processing in the Memory Hierarchy

Architectures which fall into this section of the taxonomy are distinguished from the previous section by a number of concurrent features.

Structure of the memory hierarchy. These architectures change the structure of the memory hierarchy itself. Some of these architectures place new compute elements or accelerators within memory, as seen to a certain extent in the Processing Near Memory architectures in section 2.2, while others completely reorganize the hierarchy to bring compute closer to the data.

Semantics of memory. These architectures change the semantics of memory beyond simply accessing an accelerator via a memory-mapped region of data. In the context of this thesis, in which we will be looking at instruction sequences which can be offloaded to memory, this is the most interesting and distinctive feature of architectures in this section of the taxonomy. With the change in the hierarchy, certain aspects of processor design might have to change, but at a higher level, programs may largely be able to stay the same. On the other hand, when we change the semantics of memory access—for example, by adding new instructions for computing in memory—the entire programming model needs to change, too.

Graphics processing has long motivated architecture research. Thus, it isn't surprising that many of the earliest forms of Processing in Memory were centered around graphics and video processing. We present FBRAM, first introduced by Deering et al. [8], as FBRAM strongly displays both of these features listed above.

In their 1994 paper, Deering et al. [8] point out the inefficient data movement that occurs in many graphics processing algorithms; often, data for a single frame would be moved to and from the processor multiple times. Noting the high internal bandwidth of DRAM memories, the authors present FBRAM, which accelerates the Z-buffering algorithm in memory, thus reducing data movement and improving the frame throughput for graphics processing applications. However, of equal interest to this thesis is how FBRAM changes the semantics of memory access for graphics processing. A core step of the Z-buffering algorithm is one in which old data is read from memory, compared with new data in the CPU, modified, and finally written back to memory. The authors

note that, by moving compute into the memory, they are able to simplify this step from the software perspective from read-modify-write to a single write into FBRAM with an *implied* read and modify. This is an early example of exploring the software-level changes that accompany PIM architectures, which we make the primary thrust of our exploration.

Ideas and ambitions for putting processing within the memory hierarchy started small and focused, but became more and more generalized and ambitious. Both the EXECUBE project and the Intelligent RAM (or IRAM) project introduced a radically different, more generalized approach to putting processing inside the memory hierarchy. Both projects focused on integrating compute directly into the structure of memory. EXECUBE achieved this by building an array of CPUs connected in a hypercube. Each CPU had high bandwidth to a nearby memory, and the semantics of the CPU ISA exposed a way to reason about and take advantage of this bandwidth [13]. For IRAM, the vision was a little more daring: no longer should we manufacture DRAM and processor separately; instead, we should incorporate them on a single die [22]. With the massive internal bandwidth of DRAM, and short wire lengths, memory latencies near the speed of standard SRAM L2 caches could be achieved, thus implying that chip designers wouldn't have to focus so much effort into larger caches, and could instead increase the capacity of on-chip DRAM itself. This is a primary selling point of IRAM, as DRAM is significantly denser than SRAM. IRAM's approach to processing within the memory hierarchy is to fundamentally redesign the memory hierarchy itself. Instead of offloading specific computation into memory, as we will in our simple PIM architecture, IRAM pushes this one step further, by moving the entire CPU into DRAM.

The Active Pages project from UC Davis, originally presented by Oskin et al. [21], is perhaps the best example of what this thesis is labeling "Computing Within the Memory Hierarchy". Active Pages brings general compute capability to the memory. Regions of memory are dynamically allocated as "active pages", which export some high-level interface, beyond just loads and stores. For example, in an application which does sparse matrix multiplication, a region might export a function which gathers the operands for the matrix multiplications and places them in a region of memory for the CPU to read.

Active Pages cites many of the same motivations for its existence as other PIM architectures—attacking the processor–memory gap, harnessing internal memory bandwidth—but separates itself from earlier works by focusing on the higher-level programming model and the affects of Active

Pages on applications of varying kinds. Effectively utilizing Active Pages requires programs to be partitioned into processor and memory computations. In Oskin et al. [21], the authors focus heavily on how this new programming model affects applications; specifically, the authors implement this partitioning for a number of different applications. Our goals in this thesis are similar: we would like to define an architecture, and explore how software can be restructured to fit the architecture. However, we take a fundamentally different approach, by defining at a high level what types of computations can be offloaded to memory, and then statically detecting these patterns within binaries. Furthermore, we don't just shape software to the architecture; we also let our observations from real software shape our architecture.

A more recent example of processing in the memory hierarchy comes from Aga et al. [1], which introduces the Compute Cache. Computing is done at the bit-line level, within the sub-banks of the cache itself. Within a sub-bank, their compute cache provides a number of basic operations, from basic logic operations to search and copy operations. Our exploration in this thesis will use a PIM architecture inspired by Aga et al. [1]. However, while the authors of [1] explore more of the intricacies involved in effectively utilizing compute elements in the cache—for example, the locality of the operands to be processed—we will make some simplifying assumptions that will eliminate these complexities.

Chapter 3

Detecting Processing in Memory Sequences

In this chapter, we will discuss the process of developing a Processing in Memory architecture by starting from real software. Recall our motivation: for a PIM architecture to be adopted, it must be useful to software. If it is not useful, software designers will have no motivation to redesign their software for the new architecture—and without software to run, the architecture might as well not exist. Thus, to design a PIM architecture which will be useful to real software, we must analyze the software itself, with the goal of identifying patterns which can be offloaded to memory.

Steering architecture design by looking at real workloads has proven successful in the past. Rengasamy et al. [24] analyze real application traces to find instruction patterns which can be offloaded to hardware accelerators on mobile devices. Their results showed that accelerators designed to target patterns seen in real workloads resulted in overall speedups of up to twelve percent. In this thesis, we choose to statically analyze benchmark code, rather than analyzing their dynamic traces as seen in [24]. Inspired by [24], we cite the analysis of instruction traces as a prime area of future research.

We begin our design process by defining an initial simple instruction pattern which we expect to find in software, and imagining a corresponding PIM architecture which could accelerate these

patterns. We then build static analysis tools to to find these patterns, and report on our observations. After observing these patterns in the wild, we correct some of our original assumptions and define more complex instruction patterns to search for. Finally, based on these results, we suggest a new PIM architecture.

As mentioned above, we choose to perform static analysis on our applications and benchmarks. Static analysis refers to the process of analyzing code without executing it. On the other hand, *dynamic* analysis generally involves analyzing an actual execution of a program, either through instrumenting the binary, or via gathering an instruction trace (log) and post-processing it. Through static analysis, this work shows the existence of patterns of computation, and proposes a PIM architecture to accelerate them. However, to quantify exactly how much benefit such a PIM architecture would have, we would need to know just how often these patterns are executed in practice. Thus, one of the primary next steps for the work conducted in this thesis is to perform a dynamic analysis of the same workloads, and search for the same patterns in the instruction traces.

Our static analysis will be built on top of LLVM, a compiler infrastructure project [15]. LLVM is an extensive project with many applications, but the primary draw of LLVM for the purpose of this thesis is the LLVM Intermediate Representation (LLVM IR) language. LLVM IR provides a universal language on which to perform analysis; regardless of the source language, we can compile our benchmarks to LLVM IR and perform the same analyses. The semantics of LLVM IR are low-level enough that we will be able to detect patterns of instructions which can be realistically offloaded in memory; however, it is not so low-level as to completely lose the meaning of the original code.

Furthermore, LLVM provides framework for creating compiler optimization passes which we can leverage. During an optimization pass, we are able to inspect the code function-by-function or even instruction-by-instruction. For each instruction, data and control dependency information is also provided; this is the main information we will leverage while searching for patterns.

We will be running our analysis on a group of applications and benchmarks from LLVM's test-suite project [18]. This collection represents a diverse set of program functions: from encoders, to databases, to design automation tools. These benchmarks and applications are listed in table 3.1.

All analyses in this work are performed on unoptimized versions of the programs. Our assumption is that optimization will reduce the number of patterns we identify in the code; thus, we work

application	description
lencod	Video encoder
ldecod	Video decoder
clamscan	Antivirus
sqlite3	Database
SPASS	Theorem prover

benchmark	description
7zip-benchmark	Multithreaded compression
smg2000	Equation solver
CLAMR	Adaptive mesh refinement
espresso	PLA logic minimizer
pairlocalalign	Sequence alignment

Table 3.1: The applications and benchmarks used for testing.

with unoptimized code with the intention of capturing as many patterns as possible.

3.1 Offloading Load-Load-Op-Store Patterns

To begin our search for computational patterns, we will start as small as possible: we will consider the case of offloading only single instructions. However, which single instructions can be offloaded?

To compute in memory, an operation needs both of its operands in memory; furthermore, the result of the operation should also be destined for memory. Within code, this manifests as loading two operands from memory, performing an operation, and storing the result. We call this basic pattern a load-load-op-store. Note that we also consider the common case where one of the operands is an immediate; for our purposes, we can also consider this a load-load-op-store.

It is unlikely that offloading single instructions into memory will be advantageous, especially in the case where the data exhibits high locality [3]. So, why are we pursuing such a simple pattern? Well, we need to start somewhere. As we will see, the results of this initial analysis will guide our subsequent steps; though we do not intend to keep this simplistic model, it will provide valuable initial direction. Furthermore, these simple patterns are not completely irrelevant; in fact, we will see them crop up in the more complex patterns we uncover later. Thus, understanding these simple patterns will help us reason about more complex patterns down the road.

```
%1 = load i32, i32* %ptr1
%2 = load i32, i32* %ptr2
%3 = add i32 %1, %2
store i32 %3, i32* %ptr3
```

```
cache_add i32* %ptr3, %ptr1, %ptr2
```

```
%1 = load i32, i32* %ptr1
%2 = add i32 %1, 23
store i32 %2, i32* %ptr3
```

```
cache_add i32* %ptr3, %ptr1, i32 23
```

Figure 3.1: An example (in LLVM) of load-load-op-store patterns being replaced with an instruction from the new PIM ISA.

3.1.1 Our Initial Architecture

Now that we understand the types of instruction patterns we are interested in offloading, we will describe, at a high level, a basic architecture centered around these instructions.

We model our architecture off of previous PIM work which puts compute elements in the cache—namely, the Compute Cache idea set out by Aga et al. [1]. Our compute-enabled cache will be able to handle single instructions of the form described above. Furthermore, we assume that this offload will happen at the ISA level. There will be new instructions added into our ISA for performing operations in the cache—thus, load-load-op-store patterns will be replaced with single "cache instructions" as shown in figure 3.1.

Simplifications in this Model

This is far from a full description of an architecture. However, this high-level description is appropriate for our needs in this thesis. Currently, we are primarily concerned with examining software; thus, to conduct our exploration, we need only to define the architecture's interface to software. Of course, excessive simplification is always cause for skepticism; in this section, I will briefly address the simplifications and assumptions of this model, and cite examples which give us reason to believe the simplifications are justified.

Once computation is no longer concentrated in one place, it becomes necessary to deal with the problem of *data locality*. This is not just the case in Processing in Memory—many multiprocessor architectures must deal with the same issue. However, a number of solutions to the locality problem have been demonstrated. Notably, Ahn et al. [3] developed hardware for determining operand locality at runtime, and only offloading when it is advantageous.

For many PIM architectures, there's another type of operand locality which must be considered: locality within the memory itself. Depending on how the PIM memory is designed, operands cannot be operated on in arbitrary locations; generally, they must be near each other. Though this is a limitation, the Compute Cache presented by Aga et al. [1] ameliorates the issue, guaranteeing that operands can be operated on as long as they are aligned to word boundaries.

When we leave the processor behind in favor of computing in memory, we leave behind a number of important structures—most notably, the Translation Lookaside Buffer and the Page Table Walker used to do virtual to physical address translation. However, there have been a number of projects which have brought address translation into the memory—see [2, 10].

We have also failed to mention a fairly basic detail of our architecture—that being, *which instructions will it actually support in memory?* However, this is a great example of a design question which our static analysis will help us answer. That is, rather than deciding upon a limited set of instructions now, we will observe which patterns actually occur in real programs, and use that to inform our decision. So, without any more hesitation, let's begin our static analysis!

3.1.2 Finding Load-Load-Op-Store Patterns

We have our architecture in mind, and we have an idea of the types of instruction patterns it will support. In this section, we will describe our method of finding those instruction patterns in software.

Finding these patterns is relatively straightforward. The algorithm is described in detail in algorithm 1, but we describe it here in plain English. We scan through the entire program (i.e. we look at each LLVM IR instruction[1]one-by-one) until we find an instruction which we consider to be "offloadable" to memory. We will describe the instructions we consider to be offloadable in a moment; however, each offloadable instruction is a unary or binary operation with no side effects.

Once we find an offloadable instruction, we must check whether it can actually be offloaded. To do this, we need to ensure two things:

[1]We actually iterate over `llvm::Value` objects, which can represent not only instructions, but a number of other program constructs. In the code that accompanies this thesis, the `llvm::Value` objects will only ever be instructions or immediates; however, for the purposes of this thesis itself, it is an appropriate simplification to think only about instructions.

Algorithm 1 Find load-load-op-store patterns.

```
1: Initialize list of found patterns ℓ
2: for instruction i in the program do
3:     if IsLoadLoadOpStore(i) then
4:         append (operands of i, i, users of i) to ℓ
5:     end if
6: end for
7: return ℓ

1: procedure IsLoadLoadOpStore(i)
2:     if i is not in the list of offloadable instructions then
3:         return false
4:     end if
5:     for operand o of i do
6:         if o is not a load or immediate then
7:             return false
8:         end if
9:     end for
10:    for user u of i do
11:        if u is not a store then
12:            return false
13:        end if
14:    end for
15:    return true
16: end procedure
```

1. The operands of the instruction are already in memory.

2. The result of the instruction is destined for memory, and not used by any other instructions.

If these two requirements are met, we save the instruction (including its operands and its users) as an offloadable instruction.

To achieve the first requirement, we ensure that its operands come only from loads and immediates. We are assuming for now that immediates will be sent to the cache along with the cache compute instruction. We will observe later how this can be avoided—i.e., how immediates can be "baked in" to our architecture. Note that it would still be possible to offload an instruction to memory if one or both of its operands weren't already in memory—the operands would simply need to be stored into memory first, and then the cache compute instruction could be run. For now, we consider only patterns where the operands are in memory; later, we will discuss the possibilities of "breaking up" patterns like this.

instruction	description	instruction	description
arithmetic		bitwise	
add		shl	left shift
sub		lshr	logical right
mul		ashr	arithmetic right
udiv, sdiv	integer division	and	
urem, srem	modulo	or	
floating point		xor	
fadd		conversion	
fsub		trunc, fptrunc	truncate
fmul		zext, sext	zero/sign extend
fdiv	float division	fptoui, fptosi	float to integer
frem	float remainder	uitofp, sitofp	integer to float
		ptrtoint, inttoptr	pointer conversion
		bitcast	static-width cast

Table 3.2: Offloadable LLVM instructions.

To ensure the second requirement, we check that anything using the result of the offloadable instruction is a store. Much like the first requirement, this requirement can be worked around: if we offload an instruction to memory whose result is needed in the CPU, we would simply need to load the result. Again, we will not consider these complex cases for now, and will defer that discussion until later.

Before running this algorithm over a program, we first define a list of offloadable instructions. To populate this list, we choose all instructions which plausibly could be done in memory. The exact list is shown in table 3.2. Not all of these instructions take two operands; for single-operand instructions, the pattern would simply be "load-op-store".

Notice that these instructions vary in complexity. Some, such as the logic operations, are easy to imagine implementing in memory; others, such as floating point multiplication, are harder to imagine. Though we will not go into detail about implementing these instructions, we will justify our choices by pointing out that we have not assumed anything about the internal implementation of our PIM architecture. Our compute-enabled cache might vary wildly in complexity; it might do simple logic operations at the bit-line level, as in [1], or it might be imbued with more complex ALUs as in the WaveScalar architecture [26]. Furthermore, given 3D-stacked logic-memory, we can imagine putting even more complex operations near to our cache. Implementation is not im-

application	num patterns	% of all instrs
lencod	1505	0.7
clamscan	1075	0.6
ldecod	519	0.5
sqlite3	465	0.4
SPASS	155	0.1

benchmark	num patterns	% of all instrs
7zip-benchmark	1199	0.5
pairlocalalign	703	0.7
smg2000	543	0.5
CLAMR	407	0.3
espresso	223	0.5

Table 3.3: Number of load-load-op-store patterns found in applications and benchmarks.

portant to us at this stage; as long as an instruction could plausibly be implemented, we include it on our list.

This list is not comprehensive—there are other LLVM instructions which could be included on this list in the future. For example, while `getelementptr` is grouped with the memory instructions within LLVM, it does not access memory; it simply computes the location of a field within a struct, vector, or array. This is another operation that could plausibly be done in memory.

3.1.3 Results

In this section, we present the results of running our analysis on the applications and benchmarks listed in table 3.1.

The results in table 3.3 show that such patterns do in fact exist. However, these results should not indicate anything about the potential performance gains of implementing a PIM architecture. Without dynamic analysis of these workloads, we have no clue how often these load-load-op-store patterns are actually run. Nonetheless, for the purpose of this thesis, these results serve an important role: they confirm our hypothesis that some code will exhibit potentially offloadable patterns. These results motivate us to proceed.

Next, we want to know a number of things about the load-load-op-store patterns themselves. By observing the patterns which occur in real programs, we can begin to imagine how our PIM architecture should be designed.

If our PIM architecture were being designed and fabricated in earnest, one of the first decisions we would be faced with is: which operations do we want to support? The easiest way to answer this question is to look at the programs themselves. Tables 3.4 and 3.5 present details about the most (and least) common instructions to find within load-load-op-store patterns. Recall that the only

	clamscan	ldecod	lencod	SPASS	sqlite3
add	30.0	35.6	48.7	16.1	6.5
sub	21.0	17.0	19.9	9.0	6.2
mul	0.4	6.4	3.2		0.2
udiv		1.0	0.5	1.3	
sdiv		0.4	0.4		0.6
urem		0.6	0.2		
srem			0.1		0.9
fadd			0.3		0.6
fsub			0.4		0.4
fmul			0.1		0.4
fdiv			0.1		0.9
shl	1.0		0.1		0.2
lshr	0.6				
ashr	0.1	0.8	1.3		0.2
and	0.3	4.2	0.7		0.6
or	1.1		0.4	15.5	1.9
xor	2.6				
trunc	13.7	12.1	10.6	5.8	28.4
fptrunc			0.1		
zext	16.1	15.6	5.0	7.7	23.0
sext	1.5	6.4	6.6	14.8	9.0
fptoui	0.1				
fptosi			0.1		1.7
sitofp			1.1		1.7
ptrtoint				0.6	0.4
bitcast	11.6		0.3	29.0	15.9

Table 3.4: Distribution of opcodes found in a load-load-op-store for each application.

	7zip-benchmark	CLAMR	espresso	pairlocalalign	smg2000
add	30.2	16.7	23.8	10.7	40.1
sub	22.9	16.7	4.5	16.4	8.5
mul	1.3	11.5	2.2		1.8
udiv	0.7				
sdiv	0.2	2.0	0.4	0.1	13.4
urem	0.1	7.4			
srem		0.7			13.6
fadd		3.2		44.2	1.3
fsub		4.4		3.6	0.6
fmul		3.4		7.7	1.1
fdiv		3.7		2.7	1.3
shl	0.2				
lshr	0.7				
and	0.7		18.4		
or	0.2	1.0	40.4		
xor	1.9		0.9		
trunc	13.3	7.4	0.9	1.3	
fptrunc				4.8	
zext	19.3	0.5			
sext	0.8	2.9	4.0	1.4	
fptosi				0.7	
sitofp		0.2		6.4	
bitcast	7.8	18.2	4.5		18.2

Table 3.5: Distribution of opcodes found in a load-load-op-store for each benchmark.

application	num op-st	benchmark	num op-st
lencod	2960	7zip-benchmark	2466
clamscan	2314	smg2000	1861
SPASS	1145	CLAMR	1135
ldecod	1121	pairlocalalign	1083
sqlite3	1097	espresso	261

Table 3.6: Number of op-store patterns.

instructions that can appear here are those which we listed in table 3.2. We generated table 3.2 by filtering the total list of instructions to just those instructions which can plausibly be implemented in memory; the data presented in these tables can help us filter even further, by indicating which instructions would be *worthwhile* to implement in memory.

Tables 3.4 and 3.5 paint a clear portrait of which instructions appear most frequently in these load-load-op-store patterns. Arithmetic operations—mainly adds and subtracts—are the most frequently-occurring instructions by far. This is unsurprising; arithmetic operations are incredibly common in code in general. In addition, as we pointed out earlier, our load-load-op-store pattern captures the common case where a variable is loaded, incremented, and stored; thus, most loops will yield a pattern.

While the dominance of arithmetic operations is to be expected, it is surprising how infrequently some other operations appear in these patterns. Of particular note are the logic operations; other than a few logic-based benchmarks and applications (e.g. espresso), logic operations are largely absent within load-load-op-stores. Designs such as the Compute Cache show that logic operations are straightforward to implement in memory, and can be implemented directly onto the wordlines of memory banks; however, if offloadable logic operations do not frequently appear within code, then we may have to focus on supporting more complex operations.

Up until now, we have discussed results showing where the load-load-op-store model works. We have used these results to great effect, to discuss features of our potential PIM architecture. The results have shown us, for example, which instructions may be most worthwhile to support in memory. However, the most important results from these load-load-op-store analyses are those which indicate where the model fails; it is these results which will push us forward in our exploration.

	clamscan	ldecod	lencod	SPASS	sqlite3
add	12.9	15.7	9.6	8.7	11.2
alloca				0.2	
and	9.0	0.9	1.3		5.7
ashr	3.2	21.7	11.7		1.7
bitcast	0.2				
call	9.9	7.8	18.6	64.0	22.3
extractvalue	0.1				
fadd		0.2	1.6		0.9
fcmp					0.1
fdiv			0.7		0.9
fmul		0.2	2.4		0.9
fpext			0.1		0.2
fptosi			0.3		0.1
fptoui			0.0	0.2	
fptrunc			0.0		
fsub			0.3		0.3
getelementptr	2.7		0.1	4.3	8.1
icmp	0.9	2.2	1.3	1.6	4.5
lshr	5.8	0.2	0.1		0.5
mul	2.6	13.5	9.5	0.4	2.1
or	7.0	0.5	1.1		8.6
phi	3.2	8.0	6.8	4.7	2.1
ptrtoint	1.9				
sdiv	0.2	2.1	1.6	0.2	1.1
select	0.3	0.7	0.9	0.7	1.7
sext	1.9	3.6	8.7	4.5	3.8
shl	14.3	5.2	6.8	0.3	5.6
sitofp		0.2	1.3		0.9
srem	0.1	1.1	0.3		0.1
sub	7.7	7.3	5.2	8.6	5.0
trunc	1.1	0.1	0.0	0.4	
udiv	0.2		0.0	0.2	0.1
uitofp	0.0		0.2		
urem	0.0		0.0		0.6
xor	4.3	0.5	0.3	0.1	0.9
zext	10.5	8.3	8.8	1.0	10.1

Table 3.7: If an operand isn't a load, what is it? (applications)

	7zip-benchmark	CLAMR	espresso	pairlocalalign	smg2000
add	9.3	6.2	6.8	4.1	7.8
and	2.7		4.5		
ashr	1.1		2.3		
bitcast	8.8	1.1			
call	7.1	23.4	27.9	31.7	7.6
fadd		7.7		10.3	4.1
fcmp		0.5			
fdiv		2.6	0.4	2.7	0.2
fmul		13.1		20.8	4.5
fpext				1.4	
fptrunc				0.3	
fsub		4.1		4.9	0.1
getelementptr	11.7	0.2			0.1
icmp	3.7	0.3	1.9	3.2	
invoke	3.6	5.0			
lshr	6.8		0.4		
mul	1.2	10.3	5.3	1.0	28.5
or	3.0	0.5			
phi	1.7	0.7	2.3	3.9	19.7
ptrtoint	0.4	1.2			
sdiv	0.9	2.2	2.3	0.2	0.3
select	0.4	0.2	0.8		7.9
sext	0.7	0.7		0.6	
shl	9.0	2.3	22.6		
sitofp		0.6	1.1	7.9	0.1
srem	0.1	0.1			0.0
sub	6.5	6.5	5.7	5.8	19.0
trunc	5.8	2.9		0.7	
udiv	0.4	1.1			
uitofp		0.1			
urem	0.1	5.3			
xor	3.1	1.0	13.2	0.1	
zext	12.1	0.1	2.6	0.3	0.1

Table 3.8: If an operand isn't a load, what is it? (benchmarks)

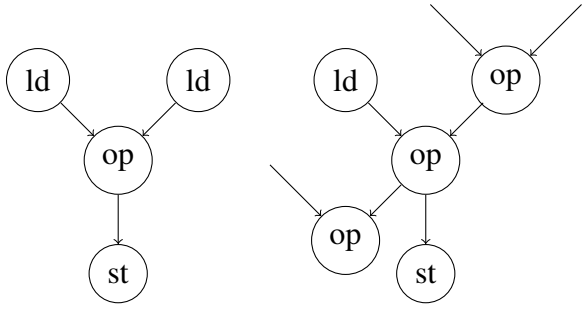

Figure 3.2: Our original pattern, versus the patterns which exist in real code.

Table 3.6 is best viewed as a table of missed opportunities. It shows the total number of so-called "op-store" patterns—patterns which were very nearly load-load-op-store patterns, except that one of both of their operands did not come from a load or immediates. In nearly all cases, there were far more missed op-stores than there were load-load-op-stores found (shown in table 3.3). Tables 3.7 and 3.8 further detail exactly which opcodes appeared in the operands, in these cases.

These tables clearly show us that, while we may have been able to capture a number of load-load-op-store patterns in these programs, there are a huge number of cases which these patterns do not capture. In the next section, we will discuss how we can improve our current model to capture more complex offloadable patterns.

3.2 Moving Beyond Load-Load-Op-Store

While table 3.3 shows that load-load-op-store patterns *do* exist in real programs, there are also a huge number of "missed opportunity" cases, as described in table 3.6. These are the direct result of using the overly simplistic load-load-op-store model for the computation which can be offloaded.

Figure 3.2 illustrates the patterns which our exploration so far has hinted at. As described in our previous results, we often see groups of instructions which are very nearly a load-load-op-store, with one of the elements missing; for example, an operation whose result is immediately stored but with an operand that is not a load, or an operation with two loads as operands, but whose result is then used by another instruction.

This is not surprising; in fact, this is consistent with our expectations. Other than counters, it is difficult to imagine frequently-occurring situations where we need to load data, perform a

```
%1 = load i32, i32* %ptr1
%2 = load i32, i32* %ptr2
%3 = mul i32 %1, %2
%4 = load i32, i32* %ptr3
%5 = add i32, %3, %4
store i32 %5, i32* %ptr3
```

```
cache_mul i32* %tmp, %ptr1, %ptr2
cache_add i32* %ptr3, %tmp, %ptr3
```

Figure 3.3: Chaining cache instructions to offload a more complicated group of dependent instructions.

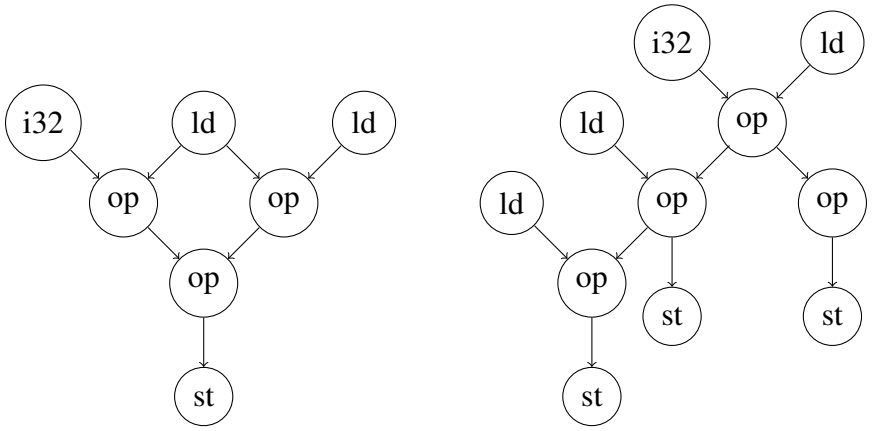

Figure 3.4: Examples of our new pattern.

single operation on it, and store it back. Though a program may use a number of counters, it is unlikely that the bulk of its computation is made up of incrementing counters. More likely, we will see groups of dependent instructions which load a number of pieces of data, perform multiple operations on them, and store their results.

This realization should not cause dismay; we can still use our simple PIM architecture to handle these groups of dependent instructions. This is done by chaining together multiple cache operations; figure 3.3 shows an example of replacing a multiply-accumulate series of instructions with two cache instructions.

Knowing this, it is clear we need to push beyond single-instruction patterns. To do this, we generalize our load-load-op-store pattern to an entire *subgraph* of offloadable instructions. Instead of a single operation, we will search for a subgraph of dependent operations; furthermore, we will look for this subgraph to be fed by loads or immediates, and to feed into stores. Figure 3.4 shows examples of the pattern we are describing.

In the load-load-op-store case, where we are only offloading a single instruction, we were able to ignore the issue of control dependencies. However, if we are now searching for arbitrary-

sized subgraphs, we must be mindful of not only data flow, but control flow through a program. In this thesis, we will assume any PIM architecture we are considering cannot handle control dependencies; thus, we will constrain our subgraphs to being entirely within the same basic block.

Next, we will introduce a more complicated algorithm to seek out these new patterns.

3.2.1 A New Algorithm

Algorithm 2 shows our new algorithm for finding subgraphs within programs. This algorithm begins much like algorithm 1: it loops over every instruction, searching for an offloadable instruction; when one is found, it takes some additional steps, and then logs a result. The primary difference is evident when we examine what happens to an offloadable instruction i when it is first encountered.

When this algorithm encounters an offloadable instruction i, its primary action is to run the PROBE procedure. PROBE's goal is to recursively spread "upwards" (through the operands) and "downwards" (through the users) until it hits one of two obstacles:

1. An instruction which can't be offloaded to memory, or
2. An instruction which is in a different basic block.

It is important to highlight obstacle 2. We only spread as far as the basic block boundary; this is to ensure that we do not accidentally involve any control dependencies in our offloadable subgraph.

Each recursive call to PROBE will result in more recursive calls upwards and downwards, effectively capturing all possible dependent offloadable instructions. The result of this algorithm is a subgraph s with three main components:

- The **rear frontier**, which comprises the "sources" of the subgraph—the instructions that feed the subgraph with data.
- The core of the subgraph itself, which is made up of dependent offloadable instructions.
- The **frontier**, which comprises the "sinks" of the subgraph.

Our previous algorithm, algorithm 1, returns only load-load-op-store patterns which can be immediately offloaded—that is, the operation's operands come from loads or immediates, and its

application	num subgraphs	benchmark	num subgraphs
lencod	24761	7zip-benchmark	22470
clamscan	14591	CLAMR	14076
ldecod	10961	pairlocalalign	12390
sqlite3	8135	smg2000	9534
SPASS	7722	espresso	4154

Table 3.9: Number of subgraphs found within programs.

results only go to stores. It is important to note that this is not the case for algorithm 2; each subgraph s returned by this algorithm is not necessarily immediately offloadable. Ideally, the rear frontier is comprised entirely of loads, and the frontier, entirely of stores; however, in practice, either could also contain non-offloadable instructions and instructions in other basic blocks.

The decision to collect subgraphs which are not immediately offloadable was intentional, for two reasons. First, though these patterns are not immediately offloadable, they will give us valuable insight into the structure of common subgraphs. Much like our analysis of the previous section, we will also be able to view these as "missed opportunity" cases, which can drive our exploration in the future. Second, though these subgraphs are not immediately offloadable, they can still be offloaded. The results of the non-loads and non-immediates in the rear frontier would simply need to be stored first; likewise, the results of the offloaded instruction would need to be loaded for the non-stores in the frontier. Details on how exactly to handle this situation are out of the scope of this thesis; however, we detail future directions in this space in section 4.1.

Table 3.9 presents our initial findings. Much like table 3.3, we should not assume this table reflects the performance improvements which would arise from offloading these computations to memory. Instead, we should simply interpret this table as indicating that these patterns do exist, and that we should continue with our exploration. However, as we can see, we capture an order of magnitude more potential patterns with our new approach. In the next sections, we will attempt to get a grasp on exactly what these patterns look like.

3.2.2 Rethinking PIM Offload

Potentially offloadable subgraphs do exist in real programs; however, we have not discussed exactly how these subgraphs will be offloaded. As we will see, this decision is important—in fact,

this decision is what shapes the next steps of our exploration. In this section, we will explore the the possible improvements over our original PIM architecture.

As explained in figure 3.3, if we stick with our current architecture which offloads single instructions, we can chain together multiple cache instructions to implement increasingly complex subgraphs. However, there is a fundamental inefficiency with this approach: as our PIM architecture is computing operations in memory, our processor is moving in lockstep, commanding it at each step of the way. In reality, a modern out-of-order processor would be able to process other instructions at the same time, and thus "lockstep" is perhaps an extreme term. Nonetheless, the inefficiency remains: the processor must pass each instruction to the memory. Though we eliminate the data movement to and from the processor, we would like to decouple the offload even more.

The next logical step, then, is to revise our original assumption about offloading single instructions. If we could instead offload multiple instructions at once, without explicitly needing to pass each and every instruction from processor to memory, we could further decouple the processor and memory. Specifically, instead of offloading single instructions, we would like to focus on offloading entire subgraphs.

There are likely a number of differing approaches to offloading entire subgraphs of computation to memory, but each approach will first need to answer the same question: *when* will the patterns be programmed into the PIM device? When the memory needs to execute the subgraph, it needs to know what it should be executing. As with many architectural decisions, the answers to this question are on a spectrum. Our initial PIM architecture is at an extreme end of this spectrum; the subgraphs are programmed at the instant they are needed—the subgraph is programmed one computation at a time. At the opposite end of the spectrum is a PIM architecture which can handle only a static set of subgraphs in memory, and thus, programming is done when the architecture is designed.

In section 3.2.4 we will discuss this question in greater detail. For now, however, we will make the following assumption: if we would like to offload subgraphs of computation, these subgraphs will need to be programmed into the PIM device at some point before execution.

Knowing that we would like to explore an architecture into which subgraphs will be programmed, we need to know which subgraphs should be included. However, before that, we need to know whether there will be enough instances of identical subgraphs to make such an architec-

ture worthwhile. Realizing this, in the next section we will develop an algorithm for classifying identical subgraphs, and report on our findings.

3.2.3 Classifying PIM Sequences

Algorithm 2 returns a list of subgraphs in a program. While each subgraph has a distinct location in the code, each subgraph may not necessarily be different. Two subgraphs may share the same underlying structure and operations; in this case, both of these subgraphs could be handled using the same hardware in a PIM architecture.

This is a problem of *graph isomorphism*. To check for isomorphism in subgraphs, we developed algorithm 3. We will explain it here in plain English.

We will be using the graph terminology "nodes" and "edges" to describe this algorithm. Nodes in the graph are always instructions or immediates, while edges are data dependencies between them.

We first use some simple checks to determine if subgraphs s_1 and s_2 are similar. We check that the sizes of the subgraphs are equal; if they are, we check that the distribution of opcodes and immediate types are equal. If these checks pass, we revert to a brute-force graph algorithm.

The brute-force graph algorithm walks from each rear frontier node in s_1 downward through the subgraph, until it hits a frontier node. Along the way, it matches nodes in s_1 to nodes in s_2. We use a worklist to keep track of the next nodes in s_1 to process; initially, the worklist is filled with the rear frontier nodes of s_1.

The algorithm is recursive; the correctness of the algorithm is based on the following recurrence relation: There is a compatible mapping of nodes in the worklist to nodes in s_2 if and only if, for a node n in the worklist, there is a compatible mapping between n and some n' in s_2 and there is a compatible mapping of nodes in $(worklist - \{n\}) \cup \{\text{users of } n \text{ in } s_1\}$.

A compatible mapping is one in which, if node n is mapped to node n', then n and n' are compatible. Two nodes are compatible if they represent the same operation (e.g. they are both adds) or the same immediate type.

We use algorithm 3 during our analysis to categorize subgraphs we find into classes, where each subgraph in a class is isomorphic. Table 3.10 shows our results, which send a resounding message: for each benchmark or application, there are an order of magnitude less *unique* classes

applications	subgraphs	unique	benchmark	subgraphs	unique
lencod	24761	1006	7zip-benchmark	22470	239
clamscan	14591	1044	CLAMR	14076	1143
ldecod	10961	579	pairlocalalign	12390	527
sqlite3	8135	610	smg2000	9534	291
SPASS	7722	234	espresso	4154	151

Table 3.10: Number of unique classes of subgraphs.

of subgraphs than there are *total* subgraphs; in the case of 7zip-benchmark, there is two orders of magnitude difference.

This is significant, as it gives us a look at the composition of real programs in terms of potentially offloadable subgraphs. What we see is that only a few unique subgraphs compose programs, and can be found numerous times throughout the code. This is still static analysis, and we do not know anything about how the code behaves at runtime. However, the fact that there are so few unique subgraphs in the code means we can confidently upper-bound the number of unique subgraphs which could appear at runtime.

This is a powerful realization, as it indicates that we may not have to support a large number of different subgraphs in memory. Using this fact, we will drive the final step of this thesis: the envisioning of a new PIM architecture.

In what is left of this thesis, we will not go into detail about the actual subgraphs found, as it would be tangential to our final goal. However, we present a selection of our interesting findings in an appendix.

3.2.4 A New PIM Architecture

In previous sections, we discussed the need to offload entire subgraphs to a PIM architecture. Furthermore, at the end of the previous section, we presented evidence which indicated that we may be able to offload just a few different classes of subgraphs and still see benefit. We will now discuss updates to our original Processing in Memory architecture, in light of all we have seen. We will do no further exploration in this chapter; rather, this final section serves as a reflection on our discoveries. This reflection will lead us to an idea for a new PIM architecture.

Let us first recall our original architecture. Our PIM architecture incorporated compute el-

ements into the cache. We assumed we could offload only single instructions to this compute-enabled cache. Furthermore, our offload occurred at the ISA level.

The primary change we would like to explore is that of offloading entire subgraphs to the PIM architecture. In the previous section, we observed that we could find a significant number of these offloadable subgraphs in memory. Furthermore, we saw that the subgraphs in a program fall into a number of classes, where the number of classes is much smaller than the total number of patterns. This information suggested that it may be possible to support just a few patterns in memory.

With this in mind, we propose a new architecture which leverages reconfigurable hardware to implement a set of subgraphs in memory. Much like our original architecture used new instructions to perform operations in the cache, our new architecture would use single instructions to execute entire subgraphs. These subgraphs might be explicitly encoded by the programmer during the software design process, or they could be dynamically chosen at compile-time using the analysis methods we have presented in this thesis. Furthermore, programming may happen at OS boot, or at program load, or even throughout the runtime of the program.

Reconfigurability within memory has been proposed in the past. Active Pages, an earlier PIM architecture, proposed the inclusion of FPGAs into memory, to enable the creation of program-specific accelerators within memory [21]. Emerging memory devices are a popular inspiration for reconfigurable memories; Chi et al. [5] present PRIME, which uses ReRAM both as storage and to implement matrix–vector multiplication. This same strategy could be adopted in our architecture to support offloading matrix–vector-related subgraphs.

As with our first architecture, this proposed design is clearly underspecified, and does not answer to a number of potential issues—in addition to the issues presented in section 3.1.1. For example, even if we reduce the number of cache instructions which need to be processed by programming an entire subgraph of computation into the memory, we have not discussed the issue of feeding the initial addresses into the subgraph. Solutions to these interesting challenges are, unfortunately, out of the scope of this thesis.

Algorithm 2 Recursively find subgraphs of offloadable instructions.

```
1:  Initialize set ℓ                                          ▷ set of subgraphs found
2:  Initialize map m                          ▷ maps instructions to the subgraph they belong to
3:  for instruction i in the program do
4:      if i is in the list of offloadable instructions and i is not in m then
5:          Allocate a new subgraph s
6:          b ← i's basic block
7:          PROBE(s, b, i, INIT)
8:          Append s to ℓ
9:          for instruction i′ in s do
10:             m[i′] ← s
11:         end for
12:     end if
13: end for
14: return ℓ
```

```
1:  procedure PROBE(s, b, i, direction)          ▷ direction: direction of recursion
2:      if i is already in s then                              ▷ prevent infinite recursion
3:          return
4:      end if
5:      if i is in the list of offloadable instructions and i is in b then
6:          Insert i into s
7:      else if direction = UPWARD then
8:          Insert i into s; mark i as being in the "rear frontier"
9:          return
10:     else if direction = DOWNWARD then
11:         Insert i into s; mark i as being in the "frontier"
12:         return
13:     end if
14:     for operand o of i do
15:         PROBE(s, b, o, UPWARD)
16:     end for
17:     for user u of i do
18:         PROBE(s, b, u, DOWNWARD)
19:     end for
20: end procedure
```

Algorithm 3 Check for isomorphism between two subgraphs s_1 and s_2.

```
Run simple checks to determine if s_1 and s_2 are similar; if not, return false.
return RunBruteForceCheck(s_1,s_2)
procedure RunBruteForceCheck(s_1, s_2)
    Initialize m                                   ▷ Maps nodes in s_1 to nodes in s_2
    Initialize worklist                            ▷ List of (n, parent_node) tuples
    for rear frontier node n in s_1 do
        Insert (n, NULL) into worklist
    end for
    return FindMatchingWalks(s_1, s_2, worklist, m)
end procedure
procedure FindMatchingWalks(s_1, s_2, worklist, m)
    if worklist is empty then
        return true
    end if
    n, parent_node ← any item from worklist
    if n is matched to a node n' in m then
        parent_node' ← m[parent_node]
        if there is an edge between n' and parent_node' then
            return true
        else
            return false
        end if
    else
        if parent_node is not NULL then
            potential_matches ← users of m[parent_node]
        else
            potential_matches ← rear frontier of s_2
        end if
        for n' in potential_matches do
            if n' is not matched and n is compatible with n' then
                m[n] ← n'
                for user child of n do
                    Add (child, n) to worklist
                end for
                if FindMatchingWalks(s_1, s_2, worklist, m) then
                    return true
                else
                    Remove mapping from n to n' in m
                    Delete added items from worklist
                end if
            end if
        end for
        if n is a rear frontier node, add (n, NULL) back to worklist
        return false
    end if
end procedure
```

Chapter 4

Conclusions

In this thesis, we conducted an exploration of Processing in Memory using a bottom-up approach. This approach focused on finding computation in real software which might be offloadable to a Processing in Memory architecture; when we observed these patterns in software, we used our observations to guide the design of the architecture.

We first fixed a simple hardware architecture: a cache which could handle single instructions. This gave us a model to work with. Using this model, we then envisioned the load-load-op-store instruction sequences which we could offload to memory. We implemented a static analysis pass in LLVM to gather information about the presence of load-load-op-store patterns in real programs.

Our results from this initial analysis suggested that we needed to rethink what patterns we should be searching for. We generalized the concept of a load-load-op-store into an entire dependent subgraph of computation. We performed a similar static analysis, which, among other things, confirmed the presence of these patterns. Motivated by the results of this final analysis, we improved upon our original PIM architecture, proposing a reconfigurable PIM architecture to which these subgraphs could be offloaded.

Throughout the course of this thesis, we have aimed to contribute more than just tables of data

pertaining to a single Processing in Memory architecture. In fact, the data presented is the least important of our contributions.

The first and most basic of the contributions of this thesis is the technical documentation of our exploratory process. Our use of LLVM and the implementation of the necessary algorithms came only after days and weeks of research, design, and trial and error. The documentation in this thesis will hopefully save someone this same effort.

This thesis also contributes a method of finding PIM instructions by searching for patterns in the Program Dependence Graph. Though using the Program Dependence Graph to find patterns in code is co-opted from the field of compilers and is far from a novel concept, this thesis hopes to show that static analysis of this type can be a powerful tool in Processing in Memory exploration.

Lastly, this thesis advocates for more software-based exploration of Processing in Memory architectures. The field of Processing in Memory is, unsurprisingly, dominated by those interested in designing the architectures themselves. However, how we will write programs for Processing in Memory architectures is equally important. In the past, PIM explorations have focused on the architecture being designed, and left the programming model as an afterthought; this thesis advocates for the opposite approach. By starting with software, we can discover patterns which can be offloaded to memory, which can then inform the design of our Processing in Memory architectures.

4.1 Future Work

In this final section of the thesis, we detail future directions.

Dynamic analysis. As mentioned a number of times throughout this thesis, our entire exploration was conducted using static analysis. Though static analysis gives us useful results—for example, our realization that the number of unique offloadable subgraphs in a program is actually rather low—it does not give us the whole picture. The 90/10 rule tells us that a program spends 90% of its time executing 10% of its code; if we take this to be true, then the distribution of unique subgraphs executed may differ substantially from the distribution of subgraphs we see in the code. Thus, the next logical step from this thesis would be to perform the same steps, but using dynamic

analysis methods. There are a number of possible ways to do this; for example, instrumenting the applications and benchmarks with code that will count the number of times each pattern is run.

Advanced static analysis techniques. In this thesis, though we applied static analysis techniques in novel ways (namely, for the exploration of PIM architectures), the static analysis techniques we used were fairly simple. Coming off of the success of our analyses, we could attempt to apply more complex static analysis methods to the same problem. For example, points-to analysis and static slicing methods may make it possible to identify much larger offloadable patterns throughout the program. One possible library we could leverage is dg, a static slicing library first presented in Chalupa [4].

PIM simulation. In this thesis, we implement LLVM passes to detect potential PIM offload. If we take this one step further and replace those patterns with new PIM instructions, these modified binaries could be run through a simulator (e.g. gem5).

Breaking up large subgraphs. Our current analysis greedily finds the biggest subgraphs possible. However, this may not always be preferable; that is, we may want to break up a subgraph and compute only part of it in memory, passing data between memory and the processor with loads and stores. The decision to break up large subgraphs can be made in myriad ways—at compile time, or dynamically at runtime, for example—and exploration of this question may be best left until after this architecture can be simulated.

Bibliography

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, Feb 2017. doi: 10.1109/HPCA.2017.21.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015. doi: 10.1145/2749469.2750386.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348, June 2015. doi: 10.1145/2749469.2750385.

[4] Marek Chalupa. Slicing of llvm bitcode.

[5] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 27–39, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8947-1. doi: 10.1109/ISCA.2016.13. URL `https://doi.org/10.1109/ISCA.2016.13`.

[6] W. J. Dally. The end of denial architecture and the rise of throughput computing. In *2009 46th ACM/IEEE Design Automation Conference*, pages xv–xv, July 2009.

[7] William J Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth

Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003.

[8] Michael F. Deering, Stephen A. Schlapp, and Michael G. Lavelle. Fbram: A new form of memory optimized for 3d graphics. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 167–174, New York, NY, USA, 1994. ACM. ISBN 0-89791-667-0. doi: 10.1145/192161.192194. URL `http://doi.acm.org/10.1145/192161.192194`.

[9] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[10] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32, Oct 2016. doi: 10.1109/ICCD.2016.7753257.

[11] A. Kagi, J. R. Goodman, and D. Burger. Memory bandwidth limitations of future microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 78–78, May 1996. doi: 10.1109/ISCA.1996.10002.

[12] Ujval J Kapasi, William J Dally, Scott Rixner, John D Owens, and Brucek Khailany. The imagine stream processor. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 282–288. IEEE, 2002.

[13] P. M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined dram and logic chip for massively parallel systems. In *Proceedings Sixteenth Conference on Advanced Research in VLSI*, pages 4–16, Mar 1995. doi: 10.1109/ARVLSI.1995.515607.

[14] H. T. Kung and Philip L. Lehman. Systolic (vlsi) arrays for relational database operations. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*,

SIGMOD '80, pages 105–116, New York, NY, USA, 1980. ACM. ISBN 0-89791-018-4. doi: 10.1145/582250.582267. URL http://doi.acm.org/10.1145/582250.582267.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[16] M. Lavasani, H. Angepat, and D. Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, July 2014. ISSN 1556-6056. doi: 10.1109/L-CA.2013.17.

[17] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 36–47. ACM, 2013.

[18] LLVM. Llvm test-suite. https://github.com/llvm-mirror/test-suite.

[19] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, 2013.

[20] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[21] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. *SIGARCH Comput. Archit. News*, 26(3):192–203, April 1998. ISSN 0163-5964. doi: 10.1145/279361.279387. URL http://doi.acm.org/10.1145/279361.279387.

[22] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, Mar 1997. ISSN 0272-1732. doi: 10.1109/40.592312.

[23] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 190–200, March 2014. doi: 10.1109/ISPASS.2014.6844483.

[24] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. Characterizing diverse handheld apps for customized hardware acceleration. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, volume 00, pages 187–196, Oct. 2017. doi: 10.1109/IISWC.2017.8167776. URL `doi.ieeecomputersociety.org/10.1109/IISWC.2017.8167776`.

[25] Google Scholar. Google scholar results for "processing in memory architecture". URL `https://scholar.google.com/`. Accessed April 9, 2018.

[26] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL `http://dl.acm.org/citation.cfm?id=956417.956546`.

[27] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59 (11):56–65, 2016.

[28] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7): 1920–1948, July 2015. ISSN 1041-4347. doi: 10.1109/TKDE.2015.2427795.

Appendix A: Code

name	link
external projects	
LLVM	https://github.com/llvm-mirror/llvm
LLVM test-suite	https://github.com/llvm-mirror/test-suite
	https://github.com/gussmith23/test-suite/tree/compile-monolithic-ir-files
dg	https://github.com/mchalupa/dg
	https://github.com/gussmith23/dg/tree/export-llvmdependencegraph
our code	
analysis passes	https://github.com/gussmith23/llvm-pim-passes
results generator	https://github.com/gussmith23/masters-thesis-data-generation
gem5 modifications	https://github.com/gussmith23/gem5
this thesis	https://github.com/gussmith23/masters-thesis

Table 1

In this section, we list all of the code needed to reproduce our work. Instructions for building each of the projects can be found within the project documentation, unless otherwise noted.

LLVM. LLVM is the infrastructure on which our static analysis passes are built. Instructions for building LLVM can be found here: https://llvm.org/docs/CMake.html.

LLVM test-suite. The test-suite project is LLVM's repository of tests and benchmarks for LLVM, used to verify LLVM's correctness. In this thesis, we use a number of applications and benchmarks from the test-suite project. In table 1 we have listed both the main repository and a specific branch on our fork of test-suite. Our fork modifies the build process so that test-suite outputs monolithic

LLVM IR files, in addition to binaries. Our analysis passes are built to be run on these monolithic IR files. LLVM's instructions for building the test-suite can be used to build our fork; these instructions can be found here: `https://llvm.org/docs/TestSuiteMakefileGuide.html#running-the-test-suite-via-cmake`

dg. dg is a static slicing library mentioned in the "future work" section. Though we did not end up using it in this thesis, we contributed some development to dg related to this thesis, and wrote an analysis pass which utilizes dg. We have included links both to the original repository and our fork.

Analysis passes. This repository contains the analysis passes for this thesis. The bulk of the work of the thesis is contained in this repository; namely, the implementations of all algorithms presented.

Results generator. This repository is a tool for generating the results used in this thesis. It includes a standardized way of declaring results to generate, in addition to parsers for parsing the results into LaTeX tables.

gem5 modifications. As stated in the future work section, building a system-level simulator for PIM architectures would be a valuable next step. This fork of gem5 contains modifications for decoding custom PIM instructions, and could be used as a basis for such an effort.

This thesis. The LaTeX code for this thesis.

Appendix B: Interesting Subgraphs

Here, we present a number of interesting subgraphs found in real code. This appendix is provided as much for the entertainment as for the edification of the reader; it is not meant to be a deep analysis of the subgraphs found. We consider this kind of deep analysis to be out of the scope of this thesis; however, it is a potentially valuable future direction. If you would like to perform your own analysis of these subgraphs, we suggest downloading and running the code for this thesis yourself. Please refer to the previous appendix for instructions on running the code.

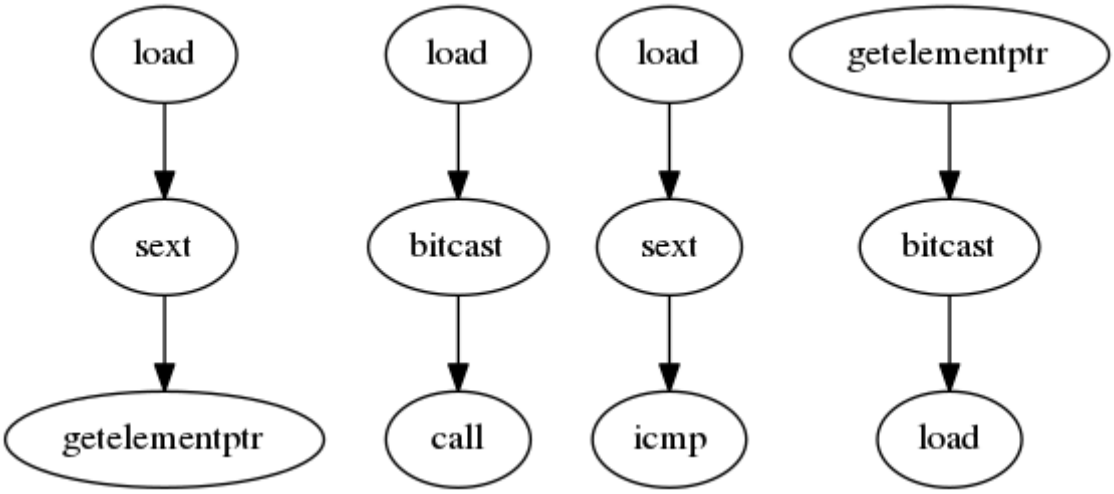

Figure 1

First, in figure 1 we present a selection of "noise" picked up by our algorithm. Though these subgraphs are picked up by our algorithm, they are far from being offloadable. In general, offloadable subgraphs load data, transform the data, and store the data back to memory; these subgraphs, on the other hand, are achieving much different goals (for example, getting specific fields within structs). These subgraphs are picked up due to the wide variety of instructions we consider offloadable. Filtering out this "noise" would be a useful improvement for our analysis algorithms.

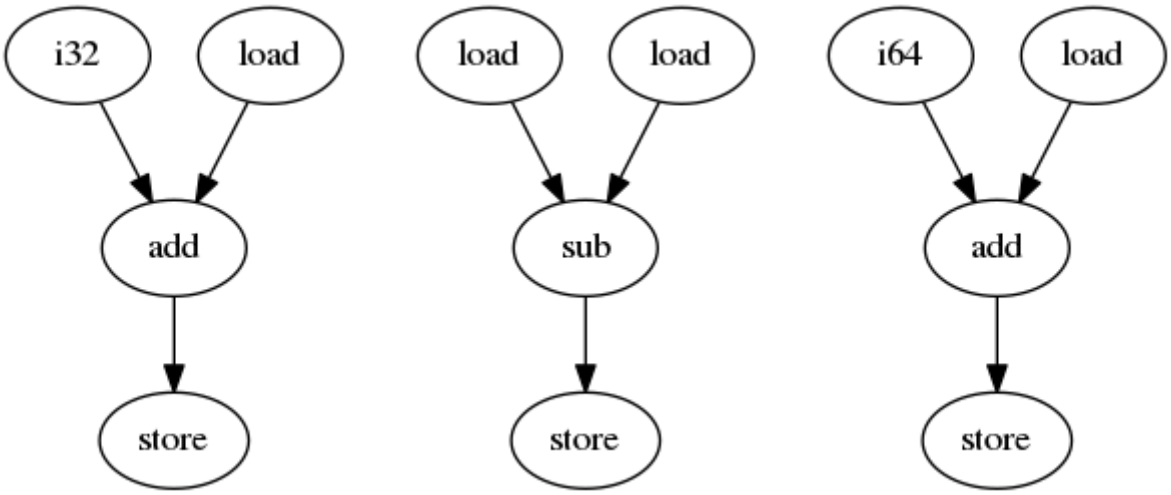

Figure 2

Figure 2 shows a selection of our "traditional" offloadable subgraphs—the load-load-op-stores we initially searched for. Unsurprisingly, though our analysis finds significantly more complex subgraphs, these load-load-op-stores are always among the most frequently occurring subgraphs in programs.

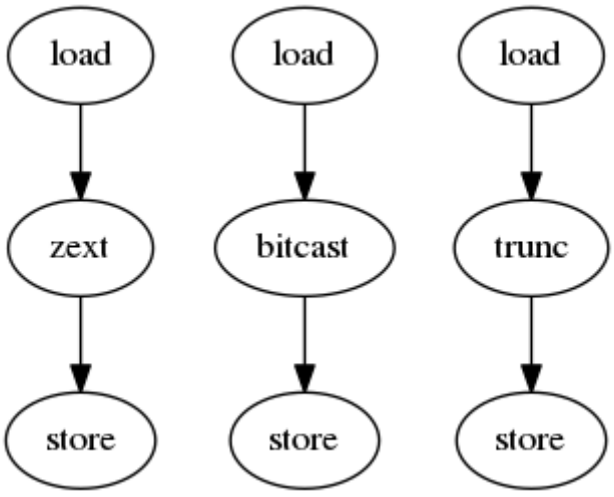

Figure 3

Figure 3 shows "simple" load-load-op-store subgraphs which also frequently turned up among the most common subgraphs. The fact that subgraphs so simple would exist in real code was surprising; it would be interesting to see whether these subgraphs remain after optimization.

Figure 4 shows two "extended" load-load-op-store subgraphs. It is common to see a conversion

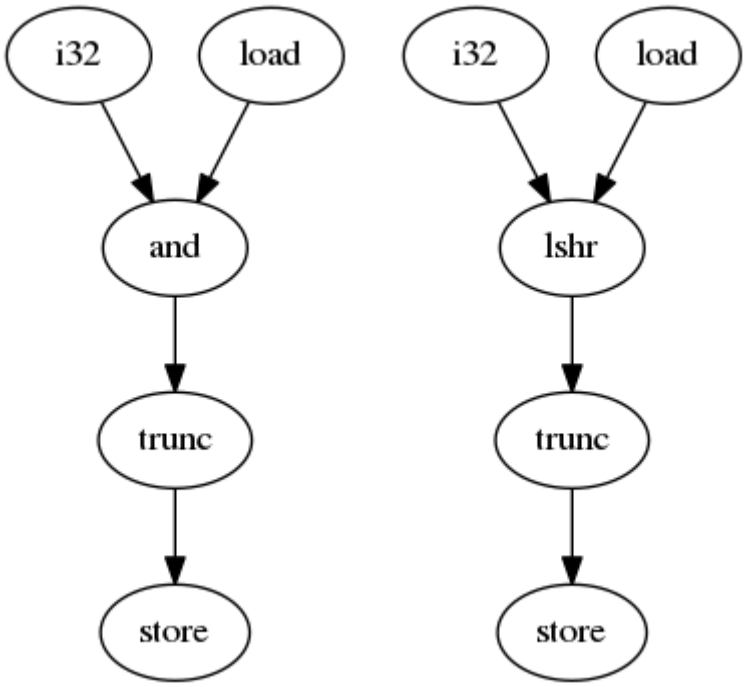

Figure 4

operation (such as truncation) applied to the output of the operation before being stored.

Figures 5 and 6 show increasingly complex subgraphs found in the code. There are two important things to note. First, these subgraphs are among the most frequently appearing subgraphs in their respective programs. Second, these subgraphs are entirely offloadable: their rear frontiers are composed entirely of loads, and their frontiers are composed entirely of stores.

Figure 7 presents an incredibly complex (but completely offloadable) subgraph. Interestingly, this subgraph occurs more frequently in its program than any of the subgraphs in figures 5 and 6.

In the remaining figures, we pull out all the stops. These subgraphs are no longer chosen from the list of most frequently occurring subgraphs; rather, these are the largest completely offloadable subgraphs found across all applications and benchmarks.

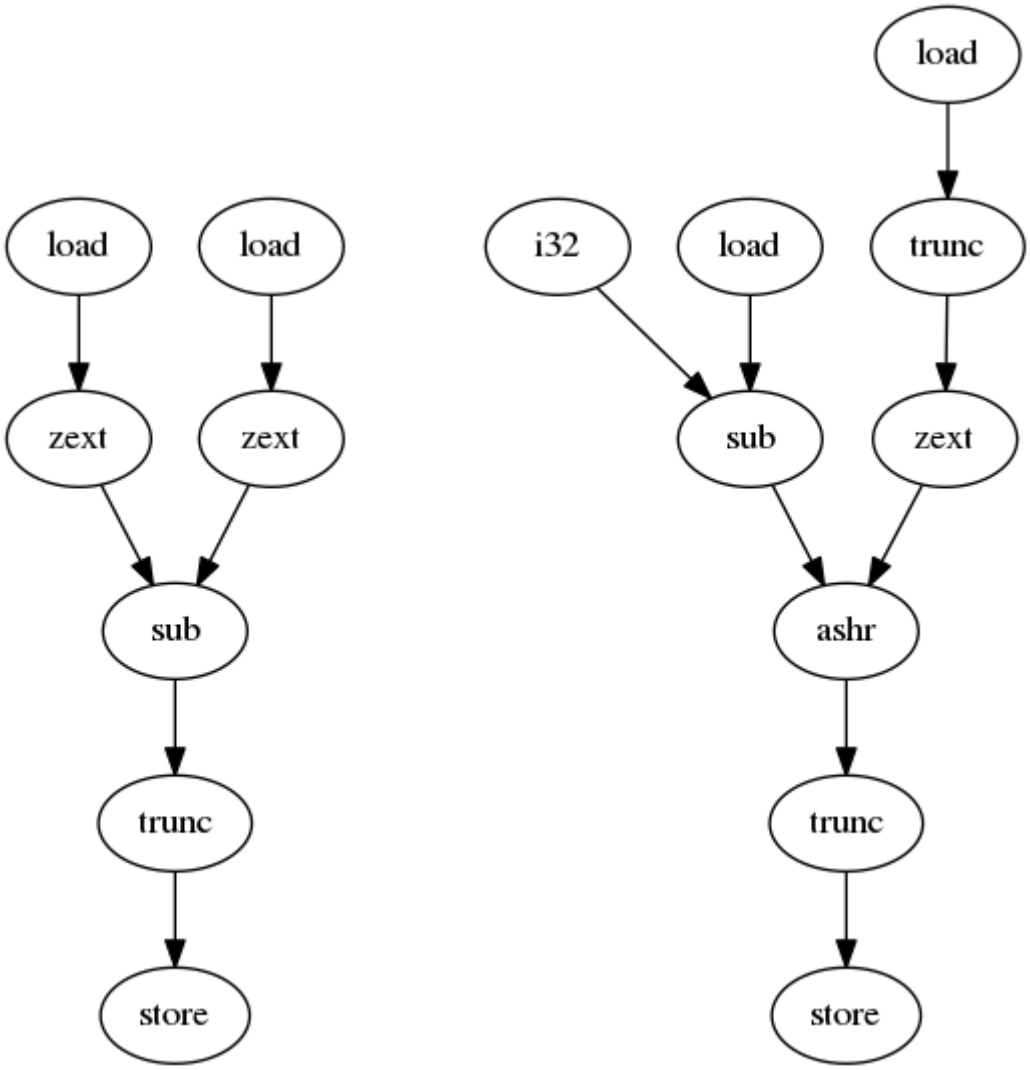
load
load
zext
zext
sub
trunc
store
load
i32
load
trunc
sub
zext
ashr
trunc
store

Figure 5

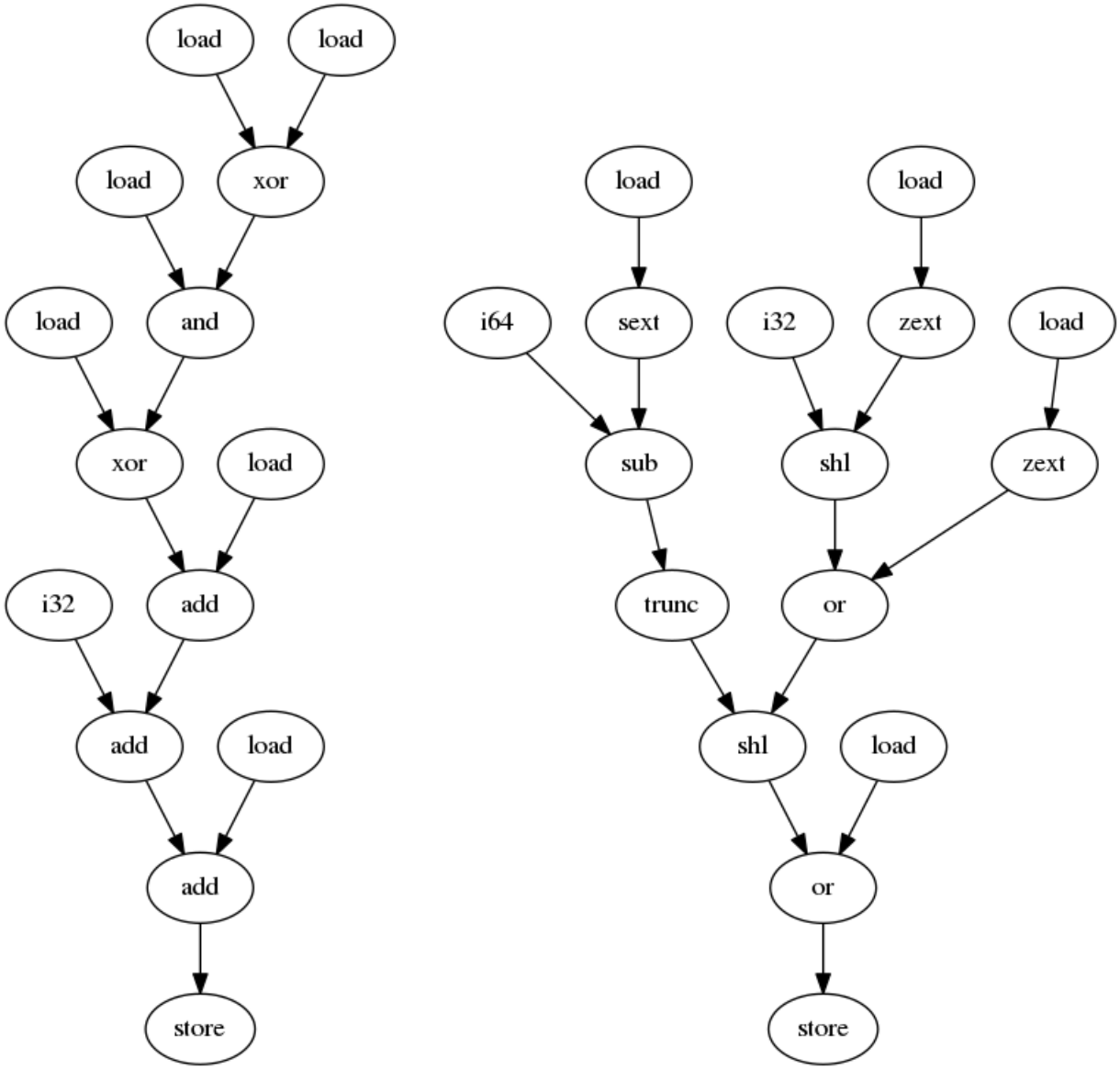
load
load
load
xor
load
and
xor
load
i32
add
add
load
add
store
load
load
i64
sext
i32
zext
load
sub
shl
zext
trunc
or
shl
load
or
store

Figure 6

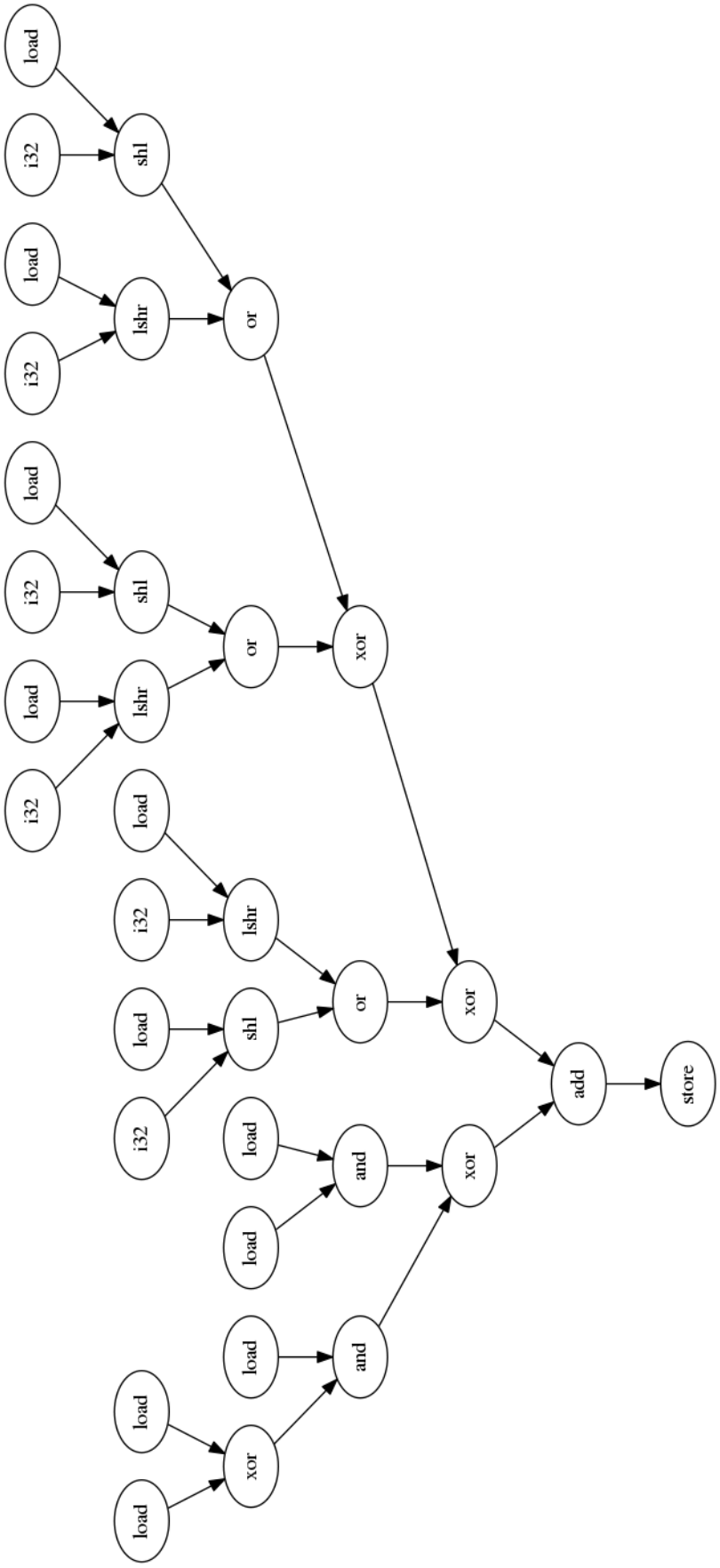
load
i32
shl
load
i32
lshr
or
load
i32
shl
load
i32
lshr
or
xor
load
i32
lshr
load
i32
shl
or
xor
load
load
and
load
load
xor
add
store

Figure 7

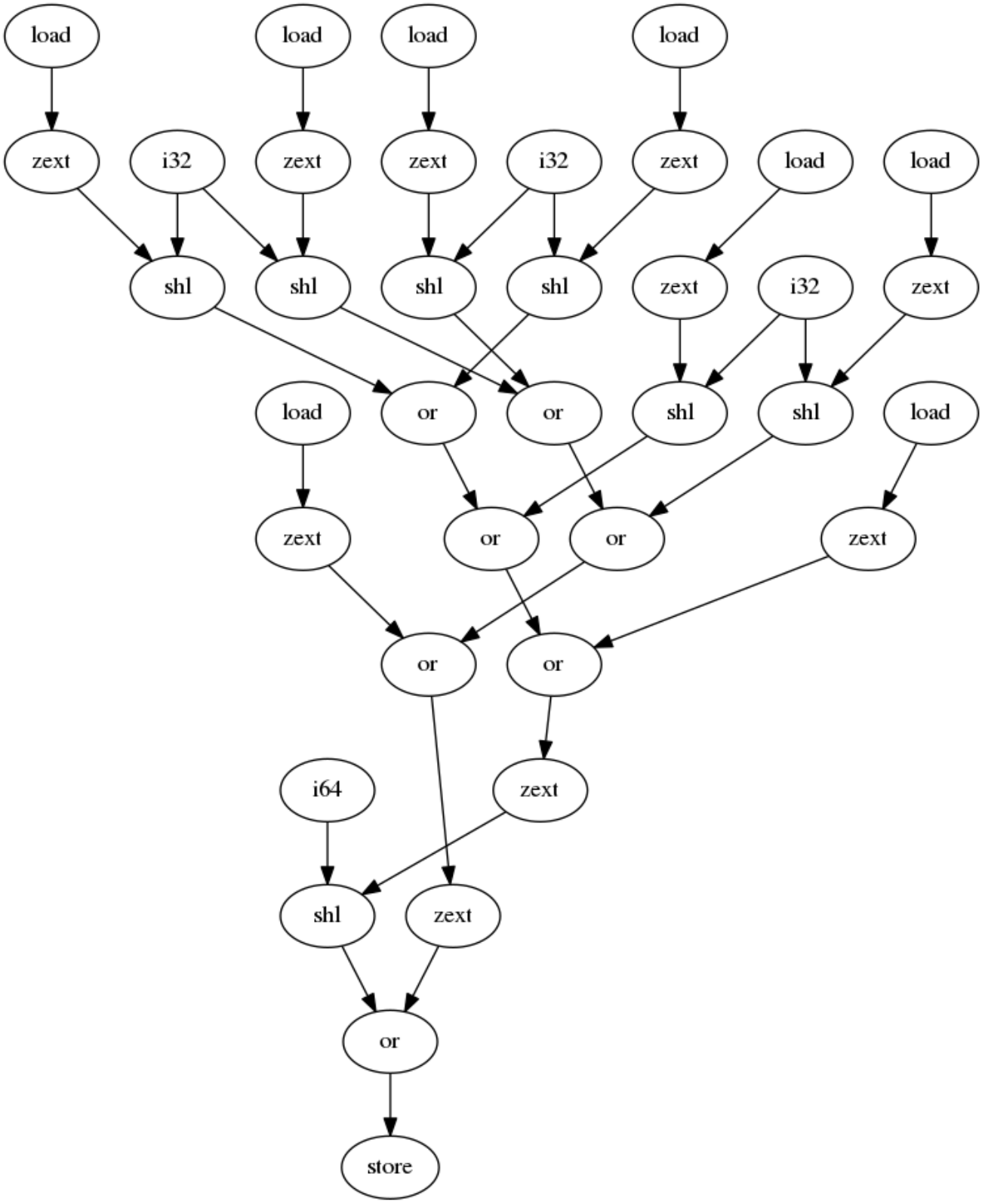
load
load
load
load
zext
i32
zext
zext
i32
zext
load
load
shl
shl
shl
shl
zext
i32
zext
load
or
or
shl
shl
load
zext
or
or
zext
or
or
i64
zext
shl
zext
or
store

Figure 8

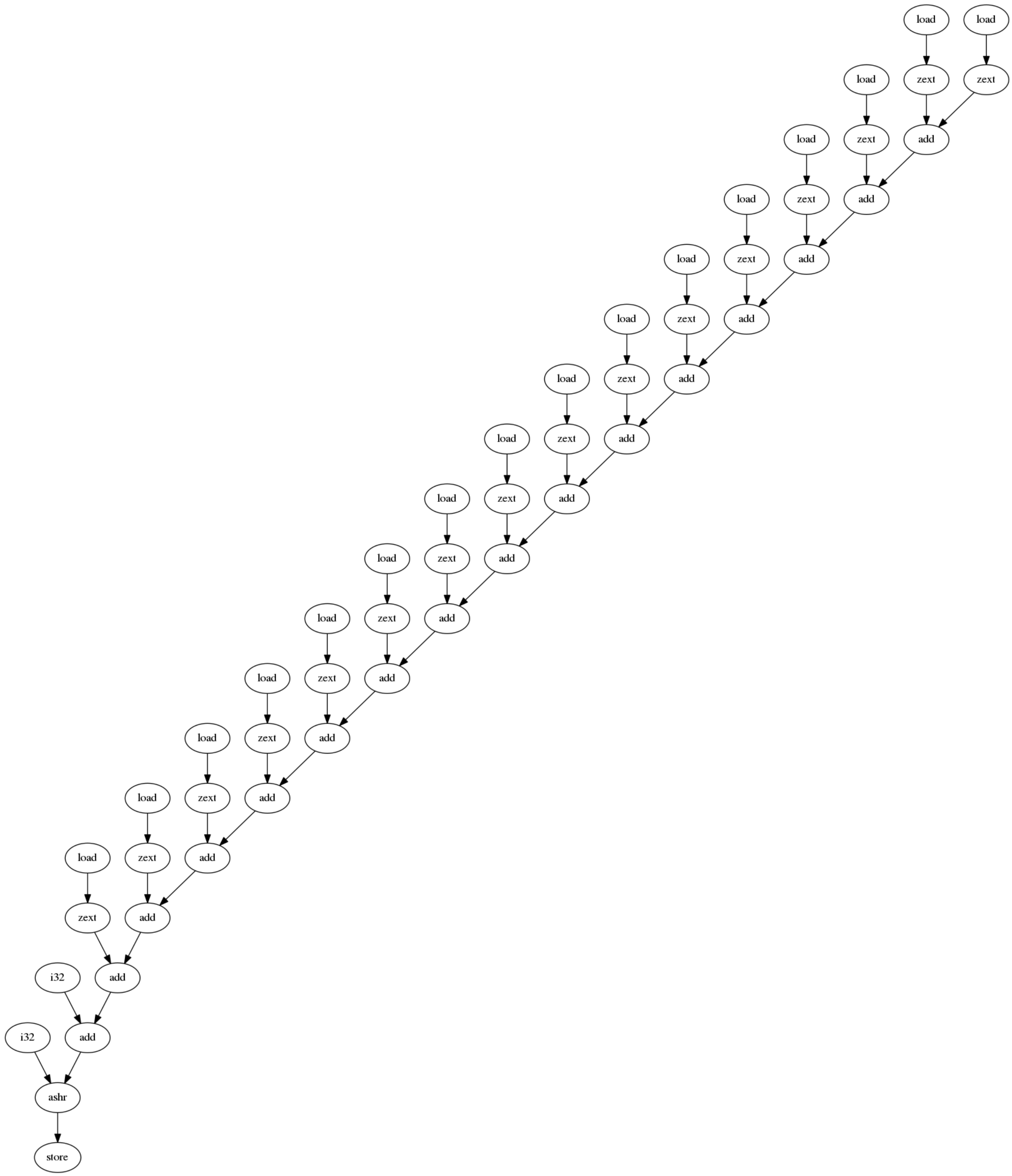
load
load
load
zext
zext
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
load
zext
add
i32
add
i32
add
ashr
store

Figure 9

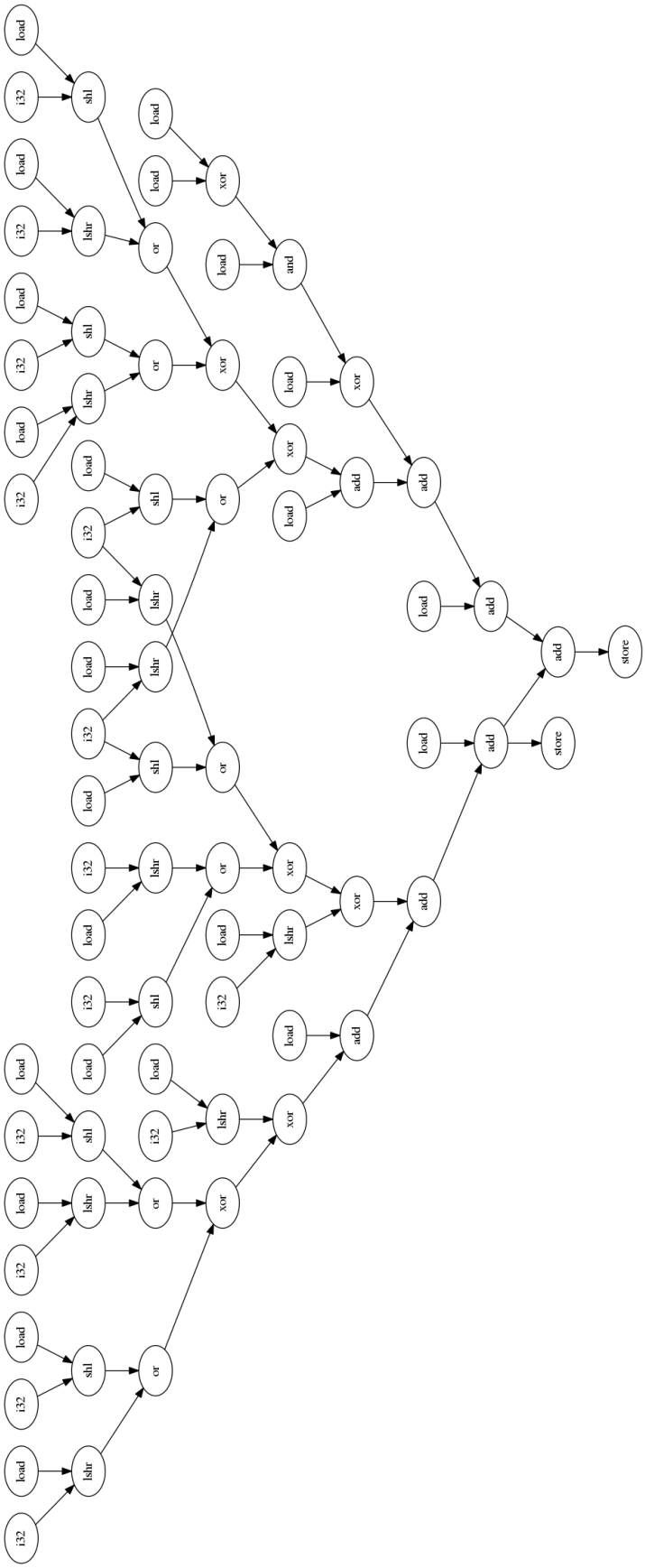

Figure 10