COMPACTING LOADS AND STORES FOR CODE SIZE REDUCTION

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Isaac Asay

February 2014

COMMITTEE MEMBERSHIP

TITLE:	Compacting Loads and Stores for Code Size Reduction
AUTHOR:	Isaac Asay
DATE SUBMITTED:	February 2014
COMMITTEE CHAIR:	John Oliver, PhD Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Chris Lupo, PhD Assistant Professor of Computer Science
COMMITTEE MEMBER:	Lynne Slivovsky, PhD Professor of Electrical Engineering

Abstract

Compacting Loads and Stores for Code Size Reduction

Isaac Asay

It is important for compilers to generate executable code that is as small as possible, particularly when generating code for embedded systems. One method of reducing code size is to use instruction set architectures (ISAs) that support combining multiple operations into single operations. The ARM ISA allows for combining multiple memory operations to contiguous memory addresses into a single operation. The LLVM compiler contains a specific memory optimization to perform this combining of memory operations, called ARMLoadStoreOpt. This optimization, however, relies on another optimization (ARMPreAllocLoadStoreOpt) to move eligible memory operations into proximity in order to perform properly. This mover optimization occurs before register allocation, while ARMLoadStoreOpt occurs after register allocation. This thesis implements a similar mover optimization (called *MagnetPass*) after register allocation is performed, and compares this implementation with the existing optimization. While in most cases the two optimizations provide comparable results, our implementation in its current state requires some improvements before it will be a viable alternative to the existing optimization. Specifically, the algorithm will need to be modified to reduce computational complexity, and our implementation will need to take care not to interfere with other LLVM optimizations.

Keywords: arm, compiler, loads, memory, optimization, stores

Acknowledgments

I am indebted to Dr. Chris Lupo for his role as advisor over the years. His experience and feedback have been invaluable in the development of this thesis, and his good humor and positivity have pulled me out of many project quagmires. Thanks also to Dr. John Oliver and Dr. Lynne Slivovsky for their roles on my thesis committee, and for the great experiences I had learning from them in the excellent classes they teach. Good teachers shaped my education more than anything, and these three in particular have been responsible for instilling in me a love of computer architecture, design, and engineering.

Over the years, I have had many teachers at Cal Poly who have made learning fun and have opened up entirely new avenues of thought in my life. Besides those already mentioned, I wish to particularly thank Dr. Phillip Nico, Dr. Chris Buckalew, and Dr. Franz Kurfess for introducing me to completely new areas of Computer Science, and keeping the spark of fascination alive as I studied with them. Though my focus now is on engineering, my first academic love has always been mathematics, and I want to particularly express my appreciation for Dr. Todd Grundmeier, who teaches mind-bending geometry with aplumb, and Dr. Dylan Retsek, who made my head hurt in a good way while teaching me how to prove things.

I cannot thank the teachers in my life without recognizing the overwhelming contributions made to my academic career by my parents. Both have encouraged me in my studies for as long as I can remember, and both have consistently modeled a love of learning and a commitment to lifelong learning. I wish to especially

thank my mother, Cheryl, for her tenacity in teaching me and my siblings as she homeschooled us through high school. I would not be where I am today without her support, perserverence, and positive attitude toward education. She is a model to me of what it looks like to sacrifice for the good of your children.

Finally, this thesis would never have been finished without the patient encouragement of my wonderful wife Emily. When the research was long, she fed me. When there was no light at the end of the tunnel, she was my friend and companion. When I felt like quitting, she wouldn't let me. Her gentle warrior spirit has kept me going these years to finish what I started. Thank you.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Embedded Computing Challenges

While general purpose microprocessors have historically focused on maximizing instruction throughput, the field of embedded computing includes the additional constraints of minimizing power dissipation and die size while still achieving acceptible performance for its desired applications. An embedded processor may often use power from a limited battery, and must thus conserve power used both by the processor and by RAM.

Many compiler techniques used for more general purpose computing can be adapted to reduce power dissipation. For example, reducing the numbers of loads and stores (which are instructions with high latency) can decrease execution time, which is important for general purpose machines in increasing throughput. This same technique can also be used by embedded processors to reduce power dissi-

pated. Thus, some optimizations are platform agnostic in that they create desirable effects for both general purpose computers and embedded computers.

Embedded computers are often designed for mass production. When many millions of units will be sold, there is more motivation for a company to increase up front development cost to reduce the manufacturing cost of the final product. Even a small reduction in manufacturing costs can lead to substantial savings over the life of a successful product. For most SoC (System on Chip) applications, the largest component present on the die is the memory [20]. All instructions required for these embedded systems to perform their functions must be present in some form of ROM on the SoC. It is very desirable to reduce the physical size of the code stored in the ROM, as by using a smaller code size for the same application, smaller ROM may be used, reducing the overall cost of the embedded system without reducing its feature set. Various techniques have been used in the history of computing for data and code compression; a good survey of compression techniques used for code size reduction was performed in [10].

This thesis will examine one such technique specific to ARM processors to reduce code size without impacting code functionality. In the rest of this chapter, we will examine some of the background necessary to perform this technique, which reduces individual memory instructions by combining them into block memory instructions, yielding reduced code size without affecting the functionality of the program. This is one such method among many that should be considered by companies buiding embedded systems to reduce their manufacturing costs by reducing ROM sizes.

1.2 ISAs

The most basic technique for reducing code size is to reduce the number of instructions used for a given program. The vocabulary of instructions used by a microprocessor to execute programs is called the Instruction Set Architecture (ISA) [29]. A processor's hardware is designed to accept a certain range of instructions encoded in machine code using a particular format. Machine code is simply a certain number of sequential bits which when taken together cause the processor to perform a particular instruction. The various types of instructions are called operations, and each operation may use one or more operands. An operand is supplemental information provided to the operation, which may include registers or constant values. Each operation has its own opcode, which is the bit pattern with which the operation is encoded in binary. Opcodes are combined with operands in certain well-defined formats to form a complete operation. Most processors use a fixed-width ISA, meaning that each operation takes up the same amount of memory in the instruction cache. At present, most embedded processors use 32-bit wide instructions, though smaller instruction widths such as 16-bits are also in use for certain applications.

During execution, a processor will fetch an instruction, break it into its constituent parts (opcode, operands, etc) and then execute it. Generally, the operands for an instruction are registers, but operands may also include constant values, such as in the case of using an offset from a particular base register. Instructions may provide various functions, including arithmetic operations, memory access operations, and conditional logic operations. All high-level language functionality must

be reduced by the compiler down to a series of ISA instructions. While some functionality may be relatively simple to emulate (such as a=b+c), others may require chains of low-level instructions to replicate the functionality of the high-level language, such as function calls.

For the more common fixed-width ISAs, the design of the ISA is severely restricted by the fact that every instruction must be contained within a fixed number of bits. Thus, there is a tradeoff between the number of opcodes, the number of registers used as operands, and the register space available to use for an operation. Some processors use a large number of registers, but can only support a small number of operation types. Others are restricted to using only two operands per instruction, which limits instruction flexibility, but provides more available bits for use as opcodes or to index registers. However, for a given number of instructions, a fixed-width ISA will always use the same number of bits regardless of what the operations executed by the processor actually do.

1.3 Significance of Memory Operations

Microprocessors store values in a number of locations. A processor will typically use a number of registers to store values that the processor is currently working with, but register banks are typically small. At some point the processor will need to *store* a register value elsewhere so it can use the register for another value. Processors will store the value in main memory, often called *RAM*, which can hold anywhere from several thousand to several billion memory values (often called *words*),

depending on the design and application. Onced stored, the processor can *load* the values back from main memory into a register and work with the value.

Because registers are part of the processor itself, they can typically be accessed in a single clock cycle. By contrast, reading from or writing to main memory can take hundreds of clock cycles, during which the processor typically remains idle, generating no-ops (instructions that produce no useful output, which are used to stall a processor core) while it waits for the memory to access the value and return it via the memory bus. Many processor designs utilize *caches* to reduce this delay by storing recently accessed (or predicted potential) values in a small region of memory located very near the processor. Such caches can return values within ones or tens of clock cycles, significantly reducing the penalty for memory accesses. Memory access times are unpredictable, but are nearly always longer than a register access. Hence, the more instructions a compiler generates that use registers rather than memory accesses, the faster the code will typically execute. Many compilers use a number of techniques to reduce memory accesses, allowing the generated code to run faster and use less power [9].

1.4 Target Platform: ARM Cortex-A8 Microprocessor

This thesis will be focused on optimizing code size for a specific ISA and microprocessor: the ARM Cortex-A8.

The ARM Cortex-A8 is a RISC-based microprocessor designed to implement

the ARM ISA and the Thumb-2 ISA [4]. It contains 16 registers, and contains a two level cache system with configurable sizing. The Cortex-A8 uses dynamic branch prediction to guess at which code path will be taken based on previous execution paths. It also contains a memory management unit along with instruction and data translation look-aside buffers to cache virtual address translations and thus reduce memory access latency. The Cortex-A8 has been used in several popular products, including mobile phones such as the Motorola Droid [7] and Palm Pre [1]. It is a popular choice for embedded systems and mobile devices.

In this thesis, we will use a Cortex-A8 processor mounted on a BeagleBoard development board [3]. The processor runs at 720 MHz and is contained in a Texas Instruments OMAP3530 chip. The board contains 2 Gb of SDRAM, as well as several expansion connectors, such as USB 2.0, S-Video, and stereo audio ports.

1.5 The Role of Compilers

Few programs are directly developed in assembly code using a specific target ISA's instructions. In practice, programs are written in a high-level language and later converted into assembly code. A compiler is a tool used to translate a high-level language expressed in plain text into a low-level assembly language representation that is specific to a target microprocessor. Compilers are usually used in conjunction with assemblers, which take the generated assembly code and translate it into the machine code that is actually executed by the microprocessor [9].

Compilers use several trasformative phases which are executed when compiling

a program. These phases are divided into two sections: the front end, which performs analysis of the program, and the back end, which uses this analysis to perform optimization and synthesis of the program to transform it into a target language. The front end takes as input a character encoded program text, performs lexical, syntactic, and semantic analysis of the text, and finally outputs an intermediate representation, which is usually a separate internal language defined by the compiler, to the back end of the compiler. This back end performs machine-independent optimizations to the intermediate representation, generates target-dependent assembly code, performs machine-dependent optimizations upon this assembly language representation, and finally outputs it to the assembler for final compaction.

This thesis will analyze a specific optimization pass within the machine-dependent optimization phase of the compiler, and thus we will not concern ourselves with the front end of the compiler at all. We will assume that the intermediate representation language has already been translated to the target-dependent assembly language, and will focus entirely on optimizing the memory instructions encoded in assembly code.

1.6 Ordering Optimizations

As mentioned above, a compiler goes through several phases as it transforms a high level language to an ISA representation. These phases each contain optimizations, which are responsible for reducing instruction count (and thus code size) without affecting program correctness. Generally these optimizations are ordered

based on the level of information needed by an optimization. An example of ordering optimizations can be found in the code generator, which is part of the back end of the compiler. Within the code generator, some optimizations are constrained to occur before or after the register allocation process. Register allocation is the process by which some large set of theoretic registers used by the internal compiler representation is mapped to the fixed, and generally smaller, set of registers present in the actual hardware. Some optimizations may be performed either before or after register allocation. For such an optimization, research must be done to decide where it is most optimal to schedule the optimization based on its performance when scheduled before or after register allocation. This thesis will examine the relative performance of an optimization, and its impact on code size, when implemented entirely after register allocation, compared with an existing implementation requiring phases to run both before and after register allocation.

1.7 Target Compiler: LLVM

The LLVM project is a compiler framework that provides high-level information to compiler transformations [25]. LLVM originally was a university project at the University of Illinois, but was later released with a BSD-style open source license. The framework is implemented in C++, with documentation available on the project website [5].

LLVM uses Static Single Assignment (SSA) form as its internal low-level program representation. In SSA form, there are an infinite number of write-once reg-

isters that may be read multiple times. Thus, a register in SSA form can be used in multiple calculations, but can only be defined once. SSA is used in the internal representation of many compilers due to the ease of applying certain types of optimizations.

1.8 Register Allocation During Compilation

During the compilation process, variables are initially not tied to any particular registers. During the front end of the compilation, the source code is analyzed and converted into an intermediate representation (IR). This representation is then analyzed for machine-independent optimizations such as the elimination of dead (unreachable) code. Eventually, the compiler must perform instruction lowering, which is a process by which the IR is converted to a target-specific representation, such as an assembly language. During this process, which is part of the code generation phase, register allocation must occur. Register allocation is the process of replacing variable names with actual registers, and determining when to store a register to main memory to allow the register to be reused for another variable, a process called *register spilling*.

1.9 Memory Optimization

As the instructions are lowered to the target representation, machine-dependent optimizations can be performed on the new low-level representation. Such oper-

ations involve target-specific techniques that can reduce the program's code size without impacting code correctness. Many of these optimizations involve memory, and the arrangement of memory operations. These optimizations can occur before or after register allocation, depending on whether the number and locations of the physical registers affect the optimization.

The ARM ISA, which we will be using as the output for our code generator, is a load/store architecture [6], meaning all values must be loaded into registers in order to be operated on. This is in contrast to a CISC architecture like x86, which is a register memory architecture and thus allows for operations to affect values directly in memory, not just values contained within registers. The ARM ISA contains a simple store operation abbreviated `STR`. This `STR` operation saves the contents of a particular register to a specified location in main memory. The `STR` operation involves only a single register and a single memory location; multiple stores require the use of multiple `STR` instructions. Later revisions of the ARM ISA included a multiple store operation abbreviated `STM`. This `STM` instruction stores several registers to contiguous memory locations in fewer clock cycles [2, 8] and in smaller code size than the associated number of individual `STR` operations. Thus, a target-specific optimization performed by many compilers is to attempt to find contiguous `STR` operations which reference contiguous locations in main memory, and convert the `STR` instructions into a single `STM` instruction. This reduces code size and increases instruction throughput by reducing the number of processor cycles required to perform the same memory accesses. There also exists analogous instructions for loading values from main memory, which allows memory accesses to be optimized regardless of the direction of data transfer.

In this thesis, we will be examining these multiple memory instructions, and attempt to improve the compiler's conversion of single memory instructions to multiple memory instructions.

1.10 Memory Optimization in LLVM

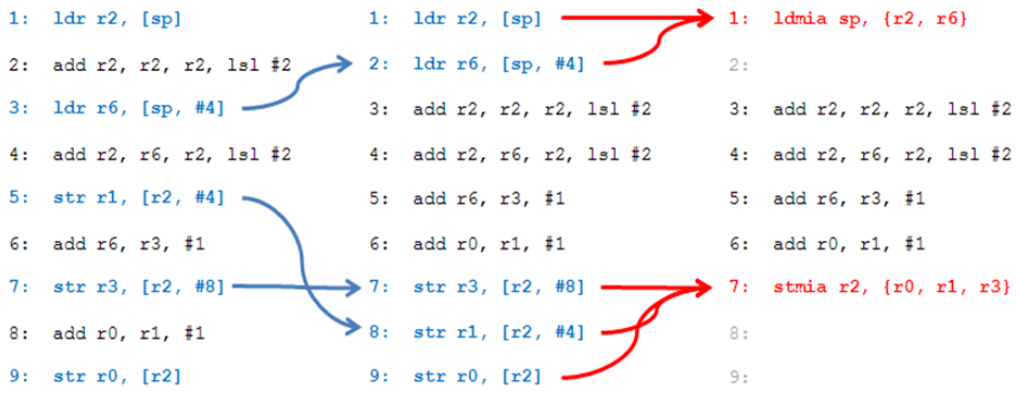
The LLVM compiler executes multiple optimization passes to reduce code size and increase performance. It uses passes on both the intermediate representation (IR) code and the machine-dependent assembly code created during the code generation phase of compilation. We will focus the attention of this thesis on one particular machine-dependent optimization performed by LLVM: the Load/Store optimizer.

The LLVM Load/Store optimizer is contained within a class called `ARMLoadStoreOpt` within the file `ARMLoadStoreOptimizer.cpp`. This file contains several classes which work together in a two part process to perform a single optimization. These classes operate on basic blocks, which are sequences of code that do not contain jumps or branches, and thus have only a single entry point and a single exit point.

The two parts of the Load/Store optimizer are shown in Figure 1.1. The first part of this optimization occurs before register allocation occurs. At this point, memory instructions within a basic block that share a base register are identified and moved to be contiguous, if this can be safely accomplished.

The second part of this optimization occurs after register allocation is per-

Figure 1.1: Optimization example



formed. The central member function within the `ARMLoadStoreOpt` class is the `LoadStoreMultiOpt` function, which iterates through basic blocks searching for contiguous memory operations (i.e., loads or stores). The member function attempts to combine these contiguous operations into a single multiple-memory operation (i.e., a `LDM` or a `STM`) by using base register offsets to correctly order the registers to be loaded or stored.

This thesis will compare and contrast this existing implementation with a newly developed, post-register allocation pass which will replace the analysis performed by the pre-register allocation phase of this optimization. Rather than replacing or modifying the LoadStoreMultOpt function, we will design a pass that will run on the assembly code directly before the LoadStoreMultOpt pass executes. The purpose of this pass will be to assist the LoadStoreMultOpt pass by moving memory operations into advantageous positions within individual basic blocks without violating any dependencies. The memory optimizations will be moved so that they are contiguous if possible, and will thus be optimized into multiple-memory instruc-

tions in the subsequent LoadStoreMultOpt pass.

1.11 Previous Work

Compiler optimization is a prolific area for computer science research, and many such research efforts choose to focus, directly or indirectly, on reducing energy expended during runtime. In the case of direct focus, a particular research effort may explicitly give its results in terms of power dissipation. However, an optimization may indirectly reduce power by reducing the dynamic instruction count, which is most often the metric given by researchers focusing on performance increases. In fact, code generators which focus on reducing power and code generators which focus on reducing execution cycles produce very similar assembly code [35].

Another way an optimization may save power is by reducing overall code size, enabling a smaller ROM chip to be used for a particular embedded application. While the block memory instructions we are examining in this thesis are an example of reducing code size, [26] examines compressing instruction memory through the use of a lookup table leading to substantial savings in program memory requirements. In [19], heuristics are introduced that improve code compression ratios using partial matching. The compression lookup tables can themselves be compressed, further increasing overall code density, as seen in [11]. Apart from compressing instructions, [17] introduces a link time optimization for the ARM platform which reduces the amount of library object code included in embedded contexts at link

time.

As mentioned above, an optimization may focus on reducing dynamic instruction count, which often can reduce power losses by reducing overall execution time. An example of two related techniques using post-register allocation optimizations were those proposed in [16] and later extended in [27]. The selection of which optimizations to use is non-trivial, as optimizations may interfere with each other and reduce overall performance. Using profiling in conjunction with performance counters to determine which combinations of optimizations maximize performance for specific workloads was examined in [13].

An example of a research project providing an overview of some low energy compilation techniques is [35]. In this survey of techniques, the authors examine reordering instructions to minimize switching power by reordering instructions to minimize bit flips between neighboring instructions. This work was applied to VLIW mobile devices in [34] and later extended in [14] which formulated heuristics to reduce switching on the instruction bus. The authors of [35] also consider optimizations which reduce memory instructions by improving global register allocation, a topic we will examine more closely below.

Using dynamic power dissipation from bit switching is one way to model power costs, but [36] constructs an instruction level model of overall power loss based on the number of cycles and the overall current required by each CPU instruction. This allows for a much more exact estimate of a program's overall power dissipation based on the sum of the individual power requirements for each instruction. This work is the basis for examining the utility of many other optimization techniques,

such as [18].

Because of the substantial costs associated with memory accesses (primarily loads and stores), much research has been performed in reducing the number of memory accesses through elimination of redundant code. Some have focused on static code size reduction using more efficient register allocators or other techniques. Two well studied register allocation techniques include graph coloring in [12, 15, 23] and the linear scan technique in [31, 37, 38].

This thesis is concerned with compiler techniques to speed up general purpose register loads and stores by using block memory instructions to move multiple values using one operation. There are other architectural techniques which have been developed to reduce latency when performing the same operation across multiple values. The most famous of these is likely Intel's MMX extension to its x86 architecture [30]. These extensions, which were introduced in the Pentium II, allowed for multiple packed integer values to be moved between memory and a special bank of MMX register, which allowed vector operations to be performed on multiple values at once. MMX is primarily used for media playback or encoding, and was later extended by Intel's SSE extensions [32] to work with larger packed values and floating point operations. These operations are known as *streaming* operations and are a type of *SIMD* (single instruction, multiple data) operation. The ARM architecture also allows for SIMD using *VFP* (Vector Floating Point) instructions [4]. These instructions were not true vector operations, as they operated in sequence rather than in parallel, and they were later supplanted by ARM's NEON media coprocessor. Like Intel's SSE, NEON performs vector operations in parallel across many values (known as *scalars*) simultaneously [4]. Unlike Intel's x86, due to ARM's load/s-

tore architecture it must load all such values into separate NEON registers before performing operations on the values. While vector operations are an interesting sub-field of computer architecture, this thesis focuses on improving block memory instructions involving general purpose registers, and we will thus not consider the NEON instructions further. Research in [21, 24] examines applying general purpose workloads to streaming SIMD instructions to achieve better parallelism in multi-core environments. Performing SIMD using interleaved data, rather than packed data, is shown to show promising speedups in [28]. While most use of vector operations involves the use of assembly code or special libraries, [33] examines improving compiler support for the use of multimedia vector operations in general purpose workloads, and what the current challenges in this field are.

1.12 Contribution of This Thesis

This thesis contains a method of optimizing memory operations performed by LLVM by heuristically selecting non-contiguous related memory operations and moving them to contiguous locations. This pass will be performed entirely after register allocation has occurred, in contrast to the current, similar optimization performed by LLVM which requires a mix of both pre- and post-register allocation information and actions. This new pass, which we call *MagnetPass*, will work in conjunction with the existing, post-register allocation LoadStoreMultOpt pass and the effectiveness of this pass will be compared against the existing memory optimization pass, which we will examine briefly in the next chapter. This new pass will be target-specific to ARM processors which use ISAs supporting multiple memory

operations. This pass, in a way similar to the existing memory operation optimization, will reduce code size, increase instruction throughput, and reduce power simultaneously.

1.13 Overview of Thesis

The organization of the rest of this thesis is as follows. We will begin by examining the theory necessary to ensure program correctness in Chapter 2. Next, in Chapter 3, we will delve into the algorithmic constructs necessary to support the theory, and determine a method which will be used to perform the preliminary analysis. We will also examine the heuristic used to determine where instructions should be moved to. In Chapter 4, we will explain the implementation details and specify our class structures. Chapter 5 will contain a summary of our results for particular benchmarks, and a comparison with the existing optimization. Our work will be summarized in Chapter 6, followed by Appendix A which will give further information about the benchmarks selected for use in this thesis. Appendix B will contain the full dataset from our test results. Finally, Appendix C will contain the new source code integrated into the LLVM compiler.

Chapter 2

Theory

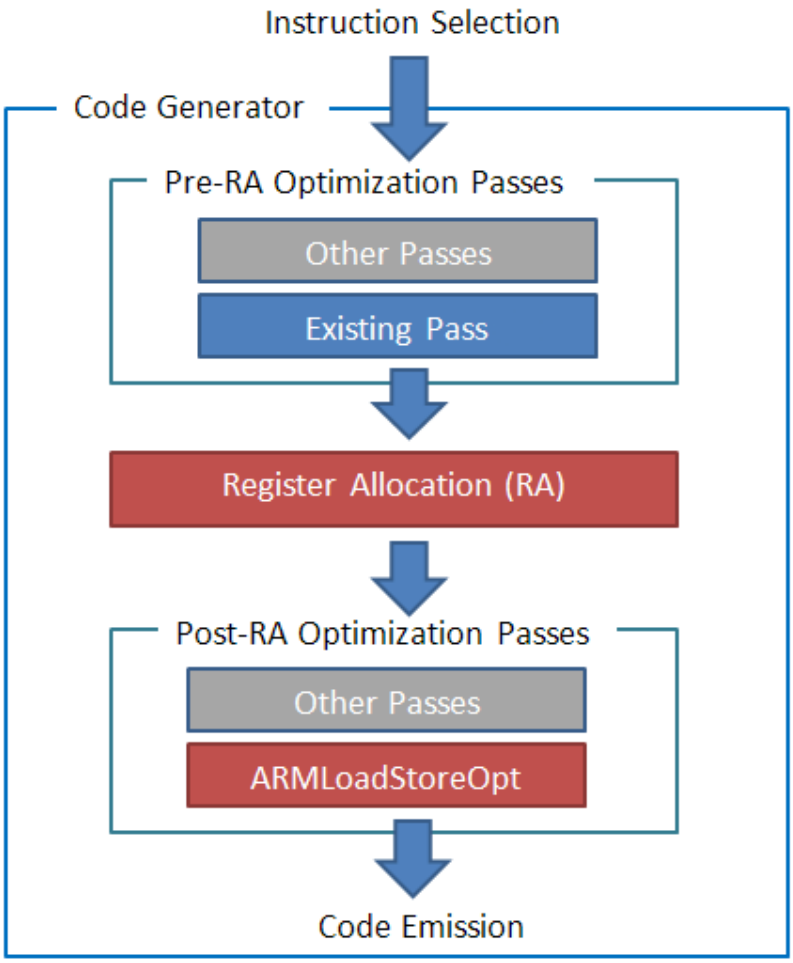
This chapter provides an overview of what is required to implement a post-register allocation version of the LLVM memory instruction optimization `ARM-LoadStoreOpt` (our MagnetPass). First, we will examine memory instructions, and the reasons to cluster them. We will then briefly discuss the differences between loads and stores prior to our examination of how the individual operations can be safely moved in machine-level code. Finally, we will look at where the instructions should be moved to once we determine their range of movements.

2.1 The Importance of Clustering

As mentioned in Section 1.10, the LLVM compiler has a post-register allocation pass that combines contiguous compatible memory instructions into a single multiple memory instruction. This pass is preceded by a pre-register allocation pass

which examines the memory operations and attempts to move them together. Figure 2.1 illustrates this sequence of optimization passes. It also shows that there are many optimizations within the code generator, some of which occur before register allocation and some of which occur afterwards.

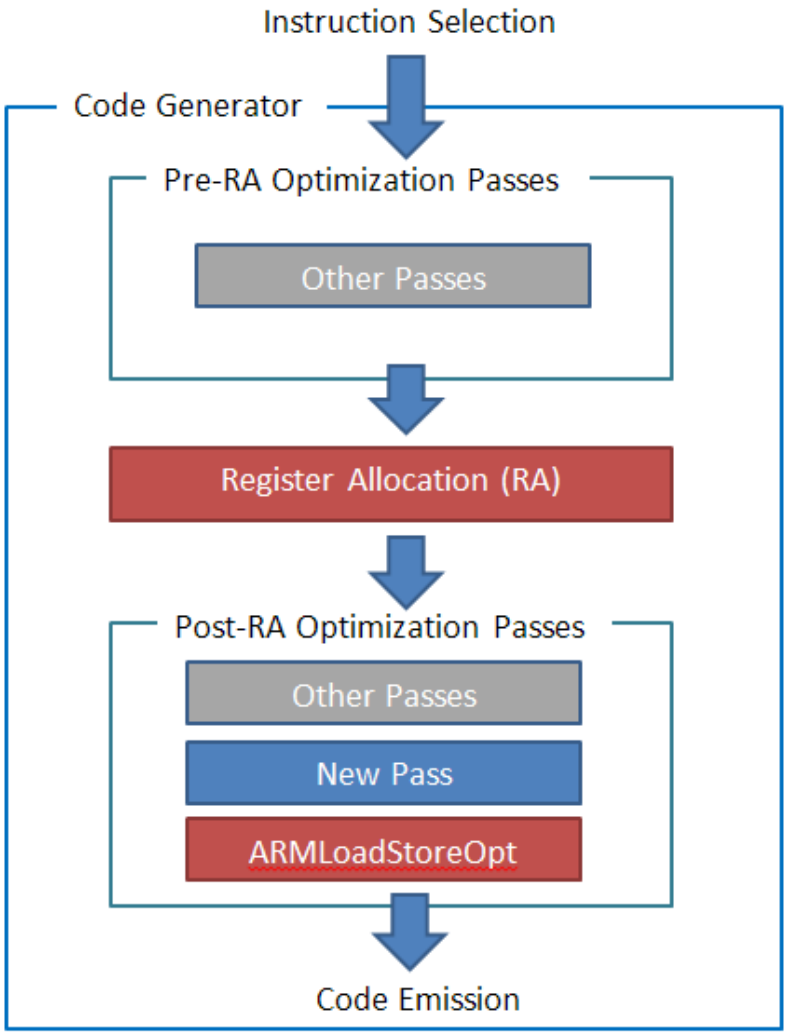
Figure 2.1: Existing pass architecture



This thesis presents a post-register allocation pass called MagnetPass that runs just before `ARMLoadStoreOpt` which will move similar memory instructions into clusters (contiguous groups of instructions) without the need for a separate pre-

register allocation pass. These clusters are then optimized into multiple memory instructions by the subsequent `ARMLoadStoreOpt` pass. This new sequence is shown in Figure 2.2.

Figure 2.2: New pass architecture



We wish to compare the efficiency of implementing this clustering phase after register allocation against the existing pre-register allocation implementation.

While running both optimizations while compiling code is possible, it is redundant as both passes should yield similar clusters of memory instructions.

2.2 Ranges of Movement

Even after instruction scheduling has occurred, it is still possible to reorder program instructions so long as we maintain program correctness. This is done by analyzing instruction dependencies on surrounding operations, and ensuring that we do not violate any of these dependencies. In particular, we are interested in moving memory instructions, and defining the dependencies governing their movements. We will refer to the space within which we can safely reorder the memory instructions as the instructions' range of movement. Because loads and stores are usually the definition and final use (respectively) of a variable, moving these instructions affects the live ranges of these variables. We must therefore ensure that our ranges include information that prevents any unsafe movement of memory instructions, particularly as they relate to the definitions and final uses of variables. We will define this criteria more precisely in the upcoming sections.

Note that because our optimization will occur after register allocation, we do not need to worry about later optimizations adding spill code due to the rescheduling performed by our optimization.

2.3 Looking at Loads and Stores

While loads and stores are both classified as memory operations, and are thus closely related, they will be examined individually to allow for differences in their dependency constraints. An example of such a constraint is that before a value can be safely loaded, it must have been stored to the proper memory location. A store has no such constraint, rather we must ensure that before a value is stored, the location the value will be stored must not have any loads pending using its previous value.

2.3.1 Range of Movement for Loads

A load operation pulls a value from memory into a register. While there are many ways to address the memory location, we will first look at the register receiving the loaded value, which we shall denote `Rx`. `Rx` may have been used in the past for some other value; if this is the case, by the time we encounter the load into `Rx`, this previous value may have been stored to memory. Alternatively, `Rx` may have been used for some temporary variable which was not finally stored, or this may be the first time `Rx` is used in this basic block. In either case, we must be sure we do not move the instruction before this last use of `Rx` or we will violate a write-after-read (*WAR*) dependency. In addition to these considerations, we must ensure that the base register used to index the load is unchanged by our instruction moving. Thus, if we wish to move the load operation, we cannot move the instruction before the point where the base register for the load's memory address is defined, or we

will violate a read-after-write (*RAW*) data dependency. Finally, we must also take care that we do not move the load instruction before a point where the value we are loading is changed via another store to the address we are loading from.

We therefore see that in order to prevent overwriting of an important value contained within `Rx`, an instruction to load `Rx` has an upper (earliest) range of movement bounded by the nearest of the following:

1. The beginning of the basic block
2. The last use of the previous value contained in Rx, including stores
3. The last write to the base register used by the load
4. The last write to the memory address used by the load

Note that because we will be examining instructions at the basic block level, we will consider range of movement to be bounded by the limits of the basic block, if no other limits present themselves earlier. We perform this optimization at the basic block level to match how the existing LLVM optimization, which is a *local* (bounded by the basic block) optimization, iterates through the instructions. Matching the optimization boundaries, as we do here, will allow for a better comparison between the two implementations.

Alternatively, we may wish to delay the load instruction. The requirements are quite similar, though rather than noting the last use of the value contained in `Rx`, we must take care to not move the load later than the first use of the newly loaded `Rx` register (a RAW dependency). Thus, we may move a single load instruction down, but not past the nearest of the following:

1. The end of the basic block

2. The first use of the new value to be loaded into Rx

3. The first write to the base register used by the load

4. The first write to the memory address used by the load

We may move the single load anywhere within the range bordered by these two boundaries.

A simple example we shall examine for load operations is the basic block fragment in Listing 2.1.

Listing 2.1: Basic Load Example

```
1 LDR r0, [sp, #4]
2 ADD r0, r0, r5
3 LDR r1, [sp, #8]
```

We wish to combine the `LDR` instructions as efficiently as possible. The instructions will not be combined unless we move them to be contiguous, as they are currently separated by non-memory instructions. We determine using our above criteria that the `LDR r0` instruction cannot be moved to be later than the first use of `r0` (the subsequent `ADD` instruction). Thus, we cannot move the `LDR r0` instruction at all. However, we note that the `LDR r1` instruction has a range of movement bounded by the top of the basic block. To see this, note that `r1` is not used previously in this basic block, no write is performed to the stack pointer, and no stores are performed on memory location `sp + 8`. We can therefore move the `LDR r1`

instruction up prior to the `ADD` instruction and make the two loads contiguous, allowing them to be optimized by the subsequent `LoadStoreMultOpt` pass.

2.3.2 Range of Movement for Stores

A store operation is the inverse of a load operation. Rather than pulling a value from memory into a register, a store copies a value from a register to main memory. As before, we will assume that the register in question is denoted `Rx`. When moving stores upwards, we must ensure that we do not move the store to be prior to the last write into `Rx`; moving past this point would cause us to store an register value that has not been fully computed (a RAW dependency). In addition, we cannot move the store before the point where a load instruction loads a value from the memory location used by the store, lest we erase the previous value in the memory location that the load meant to read. We must also ensure that the base register used to index the store is unchanged by our instruction moving, thus we cannot move the instruction above the point where the base register is modified (another RAW dependency). Finally, there is an additional subtle dependency in the form of other stores. If a store using one base+offset moves prior to a store using another base+offset which happens to be an alias of the first base+offset, we could violate a write-after-write (WAW) dependency. Thus, we must make sure we do not reorder stores to the same memory address.

We see that in order to ensure correctness, an instruction to store `Rx` has an upper range of movement bounded by the nearest of the following:

1. The beginning of the basic block

2. The last modification of the value contained in `Rx`, including loads

3. The last write to the base register used by the store

4. The last load from the memory address used by the store

5. The store store to the memory address used by the store

Alternatively, we may wish to delay the store instruction. The requirements are quite similar, though rather than noting the last modification of the value contained in `Rx`, we must take care not move the store later than the first modification of the value to store within the `Rx` register (a RAW dependency). Thus, we may move a single store instruction down, but not past the nearest of the following:

1. The end of the basic block

2. The first modification of the value in `Rx`

3. The first write to the base register used by the store

4. The first load from the memory address used by the store

5. The first store to the memory address used by the store

We may move the single store anywhere within the range bounded by these two sets of options.

As before, we will look at a simple example for store operations contained within the basic block fragment in Listing 2.2.

Listing 2.2: Basic Store Example

```
1 STR r1, [sp, #4]
2 ADD r0, r0, r5
3 STR r0, [sp, #8]
```

We wish to combine the `STR` instructions as efficiently as possible. The instructions must therefore move to be contiguous, if possible. We determine using our above criteria that the `STR r0` instruction cannot be moved up prior to the last modification of `r0` (the prior `ADD` instruction). Thus, we cannot move the `STR r0` instruction at all. However, we note that the `STR r1` instruction has a range of movement bounded by the end of the basic block; as `r1` is not modified later in this basic block, no write is performed to the stack pointer, and no loads are performed on memory location `sp + 4`. We can therefore move the `STR r1` instruction down below the `ADD` instruction and make the two stores contiguous, allowing them to be optimized by the subsequent `LoadStoreMultOpt` pass.

2.4 Selection of Cluster Points

Once we have determined the ranges of motion for each of the memory instructions in a basic block, we must decide where to move the individual instructions in order to maximize the number of operations combined into a multiple memory operation. We do this by partitioning the basic block into sets of operations that have memory operations using the same base pointer. This ensures that we do not attempt to cluster operations using different base pointers, as these operations would not be subsequently optimized using the `LoadStoreMultOpt` pass.

We will look at an example of how this selection process works. Consider the following set of ASM instructions (Listing 2.3):

Listing 2.3: Example of clustering selection

```
1  ldr r3, [r2, #-32]
2  ldr r12, [r2, #-12]
3  eor r3, r12, r3
4  ldr r12, [r2, #-56]
5  mov r1, sp
6  ldr r2, [r2, #-64]
7  eor r3, r3, r12
8  eor r2, r3, r2
9  ldr r1, [sp, #324]
10 ldr r0, [sp, #332]
11 bic r1, r1, r0
12 str r1, [r0]
13 ldr r2, [sp, #328]
14 ldr r3, [sp, #344]
15 mov r1, sp
16 cmp r0, #0
```

Because we consider the affects that non-memory instructions have on the ranges of movement in an earlier step, we have no need to examine the non-memory instructions here, so we will ignore these instructions (Listing 2.4):

Listing 2.4: Example after only focusing on memory instructions

```
1  ldr r3, [r2, #-32]
2  ldr r12, [r2, #-12]
3  --------------------
4  ldr r12, [r2, #-56]
5  --------------------
6  ldr r2, [r2, #-64]
7  --------------------
8  --------------------
9  ldr r1, [sp, #324]
10 ldr r0, [sp, #332]
11 --------------------
12 str r1, [r0]
13 ldr r2, [sp, #328]
14 ldr r3, [sp, #344]
15 --------------------
16 --------------------
```

Because loads can only combine with other loads (and similarly, stores with other stores), we will only consider clustering one type of memory operation at a time. We first consider loads (Listing 2.5):

Listing 2.5: Example after only focusing on loads

```
1  ldr r3, [r2, #-32]
2  ldr r12, [r2, #-12]
3  ----------
4  ldr r12, [r2, #-56]
5  ----------
6  ldr r2, [r2, #-64]
7  ----------
8  ----------
9  ldr r1, [sp, #324]
10 ldr r0, [sp, #332]
11 ----------
12 ----------
13 ldr r2, [sp, #328]
14 ldr r3, [sp, #344]
15 ----------
16 ----------
```

We see that we have two base pointers here: `r2` and `sp`. We arbitrarily select `r2` to examine first (Listing 2.6); our algorithm, detailed in Chapter 3, will simply select the first base register encountered in the basic block to examine first.

Listing 2.6: Example after only focusing on `r2` as base register

```
1  ldr r3, [r2, #-32]
2  ldr r12, [r2, #-12]
3  ------------------
4  ldr r12, [r2, #-56]
5  ------------------
6  ldr r2, [r2, #-64]
7  ------------------
8  ------------------
9  ------------------
10 ------------------
11 ------------------
12 ------------------
13 ------------------
14 ------------------
15 ------------------
16 ------------------
```

Finally, we see only the particular set of memory ops with a common base register that we wish to optimize by moving as close together as possible. For each set of instructions with the same base pointer, we will select as a cluster point the location of the greatest overlap of ranges of motion. At this point in the algorithm, we will have already calculated the ranges of motion, so we do not now need to go back and examine each instruction to see if it is safe to move. Note that not all the instructions in this grouping will necessarilly be able to move to the cluster point. We are only selecting the most profitable cluster point, where the most of these instructions may safely move. Any instruction which cannot safely move will

be left where it is, and will be analyzed again during a future iteration.

Once we have selected a cluster point, we will then move the memory operations to this point. We will go into more detail about this process, and the selection of the cluster point, in Chapter 3. Note also that the process of partitioning stores to allow for cluster point selection is identical to the process we have just illustrated for loads.

2.5 Summary

Loads and stores each have rules to determine an instruction's range of movement. Once these ranges have been determined, cluster points are selected using the maximum overlap of these ranges as a heuristic, and the instructions are finally moved to these cluster points, allowing them to be combined into multiple memory instructions during the later LoadStoreMultOpt pass.

Chapter 3

Algorithm

We will now present the algorithm to determine the range of movement for memory ops, as well as the method by which these instructions are moved to more advantageous positions.

In order to determine the maximum and minimum bounds for each range of movement for each instruction contained within a basic block, we must use two passes: one to determine the latest instruction a memory operation can be inserted after and one to determine the earliest instruction it can be inserted after. Once these dependencies are identified, we can determine a cluster point by using this information. We will then move the instructions to this cluster point and repeat until we have processed all memory instructions within each basic block.

This chapter will explain the data structures necessary to maintain dependency information for memory operations, and will also explain what operations are necessary during each function operating on each of the basic blocks.

3.1 Level of Optimization

The current `LoadStoreMultiOpt` optimization performed by LLVM is run when the compiler is passed an optimization flag of `-O1` or greater, and operates at the basic block level. As our optimization is a helper pass designed to run before this pass, we will incorporate our algorithm at the basic block level under similar circumstances.

3.2 Determining Range of Movement

Before we start moving memory instructions, we must analyze their dependency information to ensure that we do not compromise program correctness. To do this, we must first examine the range of movement for each memory instruction.

3.2.1 Tracking Memory Instructions

Our first task in determining range of movement for the memory instructions is to create data structures to contain the pertinent dependency data. We define a `MemOpRecord` class outlined in pseudocode in Listing 3.1.

Listing 3.1: The `MemOpRecord` Class

```
1 class MemOpRecord
2   MachineInstr *OpLocation
3   MachineInstr *LowerBound, *UpperBound
```

Each `MemOpRecord` object holds a pointer to the machine instruction that this record represents. It also holds a pointer to the determined maximum and minimum machine instruction the `OpLocation` instruction can be safely inserted after. A `MemOpRecord`'s `OpLocation` is set once during the initialization of the optimization class, and is not modified during the remainder of the code. In contrast, the two bounds pointers initially are set to `NULL`, and are set when the respective upper and lower bounds are discovered during the discovery passes. These bounds will be updated after every collection of memory ops is clustered, to account for any changes in dependency information that occur due to the movement of these instructions.

To keep track of the memory instructions, we create a vector of `MemOpRecords` and call it `AllMemOps`. This vector is initially empty, and will be filled during the initialization of the main optimization function. The `MemOpRecords` contained within this vector will be reused during the lifetime of the function, with each `MemOpRecord`'s range of motion being updated as instructions are moved.

3.2.2 Register Dependency Information

Now that we have a way of tracking the memory operations, we need some way of tracking the dependency information, and eventually ending the ranges when the dependencies are violated. To do this, we make a class called `RegisterDependencies` (see Listing 3.2).

Listing 3.2: The `RegisterDependencies` Class

```
1 class RegisterDependencies
2   vector<int> use_dep
3   vector<int> mod_dep
```

A memory instruction may contain two types of register dependencies: a use dependency and a modify dependency. A use dependency is violated when an instruction uses a particular register as part of its execution. An example would be a load instruction, which cannot be delayed past the point where a subsequent instruction requires the use of the value loaded. Similarly, a modify dependency is violated when an instruction modifies a register during its execution. An example of a modify dependency would be a store instruction, which cannot be delayed past the point of the stored register's first modification. Were it to be thus delayed, it would be entering the beginning of the live range for the next value contained within this register, and would thus be mingling two live ranges within a single register, which is an impossibility.

Accurate tracking of both types of dependencies is necessary to ensure program correctness. We wish to track the memory operations dependent on each individual register, so we create a fixed sized vector of `RegisterDependencies` to hold this dependency information called `RegDeps`. We will maintain a vector size of 16, one for each register used by the Cortex-A8 processor, and will thus be able to track which memory operations stored in `AllMemOps` are dependent on each particular register contained in the `RegDeps` vector. Each `RegisterDependencies` object will correspond to a particular register, and each `use_dep` or `mod_dep` will be an integer that indexes into the `AllMemOps` vector, allowing

each register to maintain information about what set of `AllMemOps` are dependent upon this register.

As an example, consider the two assembly instructions in Listing 3.3.

Listing 3.3: ASM Example 1

```
1 LDR r0, [r2, #8]
2 ADD r0, r0, r1
```

As we move through the instructions, we first encounter the `LDR` instruction. We thus insert a `MemOpRecord` into the `AllMemOps` vector, and create 3 entries in the `RegDeps` vector. Specifically, we insert `MemOpRecord` pointers into `RegDeps[r0].use_dep`, `RegDeps[r0].mod_dep`, and `RegDeps[r2].mod_dep`. Now, if any following instructions use `r0` or modify `r0` or `r2`, we will follow the pointers to the proper `MemOpRecord` contained in the `AllMemOps` vector, and end that `MemOpRecord`'s `LowerBound`. To see this in action, consider the next instruction. We see that the `ADD` instruction uses the `r0` register, and thus we must set the previous `LDR` instruction's `LowerBound` to point at the last instruction (itself), and then remove all the dependencies that point to the `MemOpRecord` from the `RegDeps` vector. In effect, the load in this example cannot move down at all, and thus its `LowerBound` pointer points to itself, signifying that the lowest instruction that the `LDR` instruction can be inserted below is itself.

3.2.3 Memory Dependency Information

In addition to register dependencies, a memory instruction may also have dependencies with respect to the memory location it operates on. We choose to handle memory dependencies in a fashion similar to register dependencies; we will create a vector of ints called `MemDeps`, which will contain indicies to the `AllMemOps` vector of any memory instructions encountered during the `findLowerBounds()` or `findUpperBounds()` functions. As these functions move through the basic block, they will check each instruction to see if it is a load or a store. If this current instruction is a load, we will conservatively end the range of movement for any stores we have already encountered which have not already been ended. We do this to avoid the load potentially retrieving a value from the same location in memory as the store uses. Similarly, if the instruction is a store we will end the range of movement of any previously detected loads or stores which have not yet been ended. This dependency checking is overly conservative, and is employed to avoid the need to track pointer aliasing.

As an example, consider the set of instructions in Listing 3.4.

Listing 3.4: ASM Example 2

```
1 LDR r0, [r2, #8]
2 STR r1, [r3, #8]
```

During the `findLowerBounds()` algorithm, we first encounter the `LDR` instruction. We add it to the `MemDeps` vector, and proceed. The next instruction is a `STR` instruction, which has no register dependencies that affect the `LDR` instruction. However, we do not know if the base pointer `r3` in this case is an alias of the `LDR`

`r2` base pointer, and thus this store is immediately overwriting the same memory location as the `LDR` instruction is using to populate `r0`. In light of this ambiguity, the algorithm will conservatively end the `LDR` range of movement here. Though the `r2` and `r3` registers might have different values, we have no way of being certain, and we choose to consider the pointer aliasing problem in this regard as future work for this algorithm.

3.2.4 `runOptimization()`

To run our optimization, the `MagnetPass` class must be instantiated, and the `runOptimization()` method must be executed. This is the sole public method of this class, and it is responsible for all initialization, execution, and tear down of our optimization. The pseudocode outline for this method is in Listing 3.5.

Listing 3.5: `MagnetPass.runOptimization()`

```
1  initialize()
2
3  while len(AllMemOps) > 0:
4    reset all LowerBounds and UpperBounds in AllMemOps to NULL
5
6    run findLowerBounds() to regenerate the LowerBounds
7    run findUpperBounds() to regenerate the UpperBounds
8
9    ClusterPoint bestInfo = getBestRangeOverlap(BB)
10
11   if all instructions in bestInfo are not contiguous
12     gatherAtBestRangeOverlap(BB, bestInfo)
13
14 cleanUp()
```

The `runOptimization()` method will first perform some basic initialization, including creating and initializing the `RegDeps` vector and populating `AllMemOps` using the basic block instructions. Next, it will begin a while loop using the exit criteria of `AllMemOps` emptying. As this loop progresses, `AllMemOps` will grow smaller as memory instructions are processed and moved, until finally all memory ops have been analyzed and the loop will terminate.

Inside this loop, every `MemOpRecord` contained within the `AllMemOps` vector will have its `LowerBound` and `UpperBound` pointers reset to `NULL`. This allows the subsequent `findLowerBounds()` and `findUpperBounds()` methods to set the bounds based on the current ordering of instructions within the basic block. Once all the bounds are determined, an instance of the `ClusterPoint`

class (which will be detailed in a later Section) called `bestInfo` is created. This variable contains information regarding which instructions must move and where they must move to. This information is first analyzed to ensure that the instructions to be moved are not already contiguous; if they are there is no need to reorder them as they should already get optimized by the later `LoadStoreMultiOpt` optimization without modification. If the instructions are not all contiguous, we will then use the `gatherAtBestRangeOverlap()` method along with the `bestInfo` variable to rearrange the instructions within the basic block.

After all the `AllMemOps` have been processed, we will call a `cleanUp()` method to empty any vectors still containing dependency information.

3.2.5 `findLowerBounds()`

The first pass will be used to determine the maximum amount of delay that can be added to each memory operation contained in `AllMemOps`. Because when we consider a code listing, a delay in an instruction's execution is seen as moving an instruction downward, we use the terms "down" and "lower bound" to refer to to delaying an instruction's execution with respect to other instructions in the code listing. Similarly, we will use the terms "up" and "upper bound" to refer to scheduling an instruction earlier in a code listing with respect to the other instructions.

Reviewing Chapter 2, we see that when moving through the basic block in order, we can move instructions downwards until one of the following cases occur. For loads:

1. The end of the basic block
2. The first use of the new value to be loaded into the destination register
3. The first write to the base register used by the load
4. The first store to the memory address used by the load, or to an ambiguous address that might be the load's memory address

And for stores:

1. The end of the basic block
2. The first modification of the value in the source register
3. The first write to the base register used by the store
4. The first load from the memory address used by the store, or to an ambiguous address that might be the store's memory address

We must ensure that this first pass properly ends the ranges when one of these conditions are met. The algorithm we will use to calculate the `LowerBound` values is shown in Listing 3.6.

Listing 3.6: `MagnetPass.findLowerBounds()`

```
1  clearAllDeps()
2
3  for instr in BB
4
5    endLowerBoundUsingRegsUsed(instr)
6    endLowerBoundUsingRegsModified(instr)
7    endLowerBoundUsingMem(instr)
8
9    if isMemoryOp(instr)
10     if instr has already been removed from AllMemOps, skip to next
           instr
11       add instruction to MemDeps
12     for all regs used, add a mod_dep to RegDeps pointing to instr
13
14     if isLoad(instr)
15       for all regs modified, add a use_dep to RegDeps pointing to
             instr
```

Because this function will be called many times when compiling a program, the first step is to clear any existing register dependencies in preparation for adding new ones. Once this is accomplished, we iterate through each instruction within the basic block and check to see if the instruction violates any existing dependencies, and thus ends a `LowerBound`.

The `endLowerBoundUsingRegsUsed()` function examines the `use_dep` vector for every register used by the instruction passed to it, and if pointers are found to `MemOpRecord`s, the `MemOpRecord`s are updated with new `Lower-`

Bound pointers, if the pointers are currently unset. This new pointer is the instruction which was passed into the function, and represents the last instruction that the MemOpRecord's instruction can be inserted after.

The endLowerBoundUsingRegsModified() function performs the exact same function as the endLowerBoundUsingRegsUsed() function, but it examines the mod_dep vectors rather than the use_dep vectors. The function thus examines the instruction's modified registers rather than used registers.

We have now checked for explicit register dependencies, but we also need to take care that we do not modify data in memory which could be addressed by a memory instruction. To do this, our algorithm includes a endLowerBoundUsingMem() function, which must check for the case that the current instruction is one of two types of memory operations. We first check to see if the instruction is a load. If we have such an instruction, we must ensure that the memory location from which we are loading does not have a pending store for that same location. If this case were to occur, we would be loading a value from memory which has not been stored yet. The first part of this function thus takes in a memory location and iterates through the MemDeps vector, examining each entry to ensure that it is not a store that could potentially use the same memory location as the instruction which was passed into the function. If it locates such a store, it sets the store's LowerBound to point to the instruction. Note that in this implementation, we ignore the pointer aliasing problem by being conservative; rather than keeping track of memory locations pointed to by registers and offsets, we assume that any store following a load could potentially write to that load's memory address, and thus conservatively end the LowerBound at that subsequent instruction. While this is a real limitation of

the above method, pointer aliasing is a difficult problem which is only related in passing to the rest of the algorithm. Future work may be done on this algorithm to include a method for better dealing with pointer aliasing.

After examining all loads, we must similarly ensure that if the instruction is a store instruction, that no active load instructions exist in the `AllMemOps` vector which may use the same memory address. If we neglect this, we could potentially overwrite a value needed by a previous load. As above, we prevent this inconsistancy from occuring by examining all `MemDeps` and ending the `LowerBound` entries of any subsequent loads. There is also an additional consideration with stores, as there is a possibility of a store with one base+offset moving past a second store with a different base+offset that nonetheless happens to point to the same memory location (the two base+offsets are aliases of each other). If this occurs, we could violate a write-after-write (WAW) dependency, so we conservatively end the `LowerBound` entries of any subsequent stores as well when examining stores.

Thus, at the conclusion of the first pass, all `LowerBound` pointers have been set for each memory instruction within the basic block. Next, we must use a second pass to evaluate the `UpperBound` pointers.

3.2.6 `findUpperBounds()`

The second pass will be used to determine the upper bound of movement allowable for each memory operation contained in the basic block. Again, we start by reviewing Chapter 2, and see that when moving through the basic block in reverse order, we can move instructions upwards until one of the following cases occur. For

loads:

1. The beginning of the basic block
2. The last use of the previous value in `Rx`
3. The last write to the base register used by the load
4. The last write to the memory address used by the load

And for stores:

1. The beginning of the basic block
2. The last modification of the value in `Rx`
3. The last write to the base register used by the store
4. The last load from the memory address used by the store

We must ensure that the second pass properly ends the upper bounds when one of these conditions are met. It is interesting to note that these requirements exactly match those found in the first pass if the word *first* is replaced with the word *last*. This suggests that the second pass will be almost identical to the first pass.

The algorithm we will use for the second pass is in Listing 3.7.

Listing 3.7: `MagnetPass.findUpperBounds()`

```
1  clearAllDeps()
2
3  for instr in reversed(BB)
4
5    endUpperBoundUsingRegsModified(instr)
6    endUpperBoundUsingRegsUsed(instr)
7    endUpperBoundUsingMem(instr)
8
9    if isMemoryOp(instr)
10     if instr has already been removed from AllMemOps, skip to next
           instr
11       add instruction to MemDeps
12     for all regs used, add a mod_dep to RegDeps pointing to instr
13
14     if isLoad(instr)
15       for all regs modified, add a use_dep to RegDeps pointing to
             instr
```

This second pass closely matches the first pass; the main difference is that the `UpperBound` pointers will be modified rather than the `LowerBound` pointers. We iterate through the basic block in reversed order, so that an instruction with dependencies that exist earlier in the basic block will have its `UpperBound` pointer set correctly.

Once both passes run, the `AllMemOps` vector will contain information regarding the range of movement for each memory operation. We must then decide where to move these memory operations.

3.3 Selection of Cluster Points

Once we have all the ranges of movement for the memory operations stored in `AllMemOps`, we are ready to begin reordering the instructions.The instructions are analyzed using the `getBestRangeOverlap()` method, which analyzes the `MemOpRecords` contained in `AllMemOps` and returns an object called `bestInfo`. This object is of a new class called `ClusterPoint`, which is defined in Listing 3.8.

Listing 3.8: The `ClusterPoint` Class

```
1 class ClusterPoint
2   MachineInstr* insertAfter
3   vector<MachineInstr*> instructionsToGather
```

This class holds information to be passed to the `gatherAtBestRangeOverlap()` function. It contains a vector of `MachineInstr` pointers describing which assembly instructions should be moved, and a pointer to the instruction the instructionsToGather instructions should be inserted after. This object is generated by the `getBestRangeOverlap()` function, the algorithm for which is expressed in Listing 3.9.

Listing 3.9: `MagnetPass.getBestRangeOverlap()`

```
1 getBestRangeOverlap(BB)
2   lookAtLoads = do any loads remain in AllMemOps?
3   baseReg = first base reg of memory op of type specified by
        lookAtLoads
4
```

```
5   create empty ClusterPoint objects called currentCluster and
        bestCluster
6
7   add instrs with NULL UpperBounds and given baseReg into
        currentCluster.instructionsToGather
8   bestCluster = currentCluster
9
10  for instr in BB
11    currentCluster.insertAfter = instr
12    for memop in AllMemOps
13      if memop is proper type according to lookAtLoads
14        if this memop has the proper base register
15          if this memop's UpperBound = instr
16            add to currentCluster.instructionsToGather
17          if this memop's LowerBound = instr
18            remove this instr from currentCluster.
                  instructionsToGather
19
20    if len(currentCluster.instructionsToGather) > len(bestCluster.
          instructionsToGather)
21        bestCluster = currentCluster
22
23  remove all bestCluster.instructionsToGather from AllMemOps
24  remove bestCluster.insertAfter from bestCluster.
        instructionsToGather, if relevant
25  return bestCluster
```

The algorithm requires some explanation. Initially, we must determine if we will move the loads or the stores, as it makes no sense to cluster loads with stores

because the LoadStoreMultiOpt pass will not combine them. We arbitrarily choose to move all loads first, and then any stores that are present in the basic block. We perform this by setting a boolean variable (lookAtLoads) to true if the type of memory operation we wish to optimize is a load and to false if it is a store. We must then determine which base register we will use to select instructions for clustering, and we choose to use a standard greedy approach and thus select the first base register of the type specified in lookAtLoads to be the baseReg for the rest of this method call.

We then create two instances of the ClusterPoint class called currentCluster and bestCluster. We must have some way of keeping track of the current cluster point (currentCluster.insertAfter) along with the instructions that can move there (currentCluster.instructionsToGather). We also wish to track the current *best* cluster point, which we define as the cluster point around which we could gather the most contiguous instructions. We have the bestCluster object to hold this information. The bestCluster will be constantly compared to the currentCluster object, and will always be set to the best cluster point we have yet found in the basic block.

We must initially add any MemOpRecords contained in AllMemOps that have NULL UpperBounds to currentCluster.instructionsToGather, assuming they have the base register given by baseReg. These are instructions whose range of movement allow them to move to the very top of the basic block.

Next, we iterate through the basic block, and initially set the currentCluster.insertAfter pointer to the current instruction. We must then determine

which instructions currently in `currentCluster.instructionsToGather` have a `LowerBound` that points to the current instruction. These instructions must be removed from `currentCluster.instructionsToGather`, as the current cluster point is outside of the range of movement for these instructions. Conversely, any instructions whose `UpperBound` points to the current instruction should be added to the `currentCluster.instructionsToGather` vector, as the current cluster point has just entered their range of movement.

Only instructions of the type specified in `lookAtLoads` will be added to `currentCluster.instructionsToGather`, and this is reflected in the first if statement. Additionally, only `MemOpRecords` which have the base register specified by `baseReg` should be considered, which is reflected in the second if statement. Once a `MemOpRecord` has passed these two tests, it will be added or removed from `currentCluster.instructionsToGather` according to the criteria above.

Finally, after every iteration the `bestCluster` object is compared with the `currentCluster` object, and whichever object holds a larger number of `instructionsToGather` entries becomes the new `bestCluster` object. Thus, the `bestCluster` object always contains the most profitable cluster point as well as every instruction that can be moved to that cluster point.

After iterating through the basic block, `bestCluster` will contain all the instructions that should be moved. Because we will move them in the next step, we remove them from `AllMemOps` to prevent them from being considered again. In addition, if `bestCluster.instructionsToGather` contains `bestClus-`

ter.insertAfter, we will remove this instruction from bestCluster.instructionsToGather to prevent the cluster point from being moved in the later gatherAtBestRangeOverlap() method. Finally, we return the bestCluster object to the runOptimization() public method.

Once the location and MemOpRecords of the most profitable instructions to be moved is determined, this information is passed to the gatherAtBestRangeOverlap() function.

3.4 Instruction Reordering

We now know which instructions to move and where to move them to. We pass that information to the gatherAtBestRangeOverlap() method, seen in Listing 3.10.

Listing 3.10: `MagnetPass.gatherAtBestRangeOverlap()`

```
1  gatherAtBestRangeOverlap(BB, bestCluster)
2    split bestCluster.instructionsToGather into moveUp and moveDown
         vectors
3
4    for instr in BB from first moveDown to bestCluster.insertAfter
5      remove any kill flags on instr regs
6    add these removed kill flags to last bestCluster.
         instructionsToGather using the regs
7
8    for instr in reversed(BB) from first moveUp to bestCluster.
         insertAfter
9      give any moveUp kill flags to last instr using that reg
10
11   insert all bestCluster.instructionsToGather after bestInfo.
         insertAfter in BB
```

This function also requires some explanation. The `bestCluster.instructionsToGather` memory operations are divided into two vectors: one for instructions that exist before `bestCluster.insertAfter` (stored in the moveDown vector) and another for instructions that exist after (stored in the `moveUp` vector). This is done because different procedures are required to prevent disrupting *kill flags* (defined below) for each group of instructions.

LLVM appends metadata to the instructions in a basic block, including when the final use of a register value occurs and the register is available for scavenging (able to be used for a new variable). This particular data is known internally as the *kill flag*. We must prevent a memory instruction from moving past an instruction

that kills one of the memory instruction's registers. If we do not avoid this scenario, LLVM may scavenge the register before the last use of its variable, effectively destroying program correctness.

To combat this, all the memory instructions in `bestCluster.instructionsToGather` that must move down scan each instruction they will move past and remove every kill flag that refers to registers that they themselves use. Once this is completed, any registers that used to have kill flags will be *subsumed* by the memory instructions using those registers, and the kill flags will thus migrate down to the lowest instruction using the value's register.

On the other hand, any memory instruction that must move up may move past the use of one of its registers, and if the memory instruction itself contains registers with kill flags, correctness will again be compromised. To prevent this, any memory instructions that must move up scan instructions they move past and *abdicate* the responsibility of killing the register value to the first instruction which uses the register which the memory op kills. Thus program correctness is preserved in both cases.

Once the kill flags have been migrated, the memory instructions are removed from the basic block and inserted back beneath the `bestCluster.insertAfter` instruction and the method ends. The `runOptimization()` method will loop again until no more `MemOpRecords` exist in `AllMemOps`, and thus all memory instructions have been analyzed and moved to more probably advantageous locations.

Chapter 4

Implementation

Now that we have an understanding of the algorithm and data structures we will use to perform this optimization, we must now look at how to implement these items in the LLVM compiler. We will look at how the existing post-register allocation optimizations are structured in LLVM, and illustrate where we will insert this new optimization. We will also examine some changes that became necessary to properly implement the algorithm in the existing codebase.

4.1 Organization of LLVM Code

The LLVM project is available on the Internet as a Subversion repository, allowing anyone to check out the source code and development history of the project. Once the repository has been checked out, it can be built using the Make tool and tested with a test suite similarly available from the LLVM website. The code, which

is written in the C++ language (along with some C modules and domain specific language compenents) may be modified and rebuilt. When testing our optimization, we built the standard version 2.8svn, moved the resulting binaries and libraries, then added our optimization and rebuilt the project. This allowed us to have both an unaltered version and a modified version of the binary, leading to easy comparison between the generated testcases.

The source tree contains many files and directories, only a few of which are relevant to this project. A very abbreviated diagram of the directory structure is shown in Listing 4.1.

Listing 4.1: Relevant portion of the LLVM source tree

```
1  USER/src/llvm
2   |-autoconf
3   |-bindings
4   |-cmake
5   |-Debug+Asserts
6   |---bin
7   |---lib
8   |-docs
9   |-examples
10  |-include
11  |---llvm
12  |-----ADT
13  |-----Analysis
14  |-----Assembly
15  |-----Bitcode
16  |-----CodeGen
```

```
17  |-----CompilerDriver
18  |-----Config
19  |-----ExecutionEngine
20  |-----MC
21  |-----Support
22  |-----System
23  |-----Target
24  |-----Transforms
25  |---llvm-c
26  |-----Transforms
27  |-lib
28  |---Analysis
29  |---Archive
30  |---AsmParser
31  |---Bitcode
32  |---CodeGen
33  |---CompilerDriver
34  |---ExecutionEngine
35  |---Linker
36  |---MC
37  |---Support
38  |---System
39  |---Target
40  |-----Alpha
41  |-----ARM
42  |-------AsmParser
43  |-------AsmPrinter
44  |-------Disassembler
45  |-------TargetInfo
```

```
|-----Blackfin
|-----CBackend
|-----CellSPU
|-----CppBackend
|-----MBlaze
|-----Mips
|-----MSIL
|-----MSP430
|-----PIC16
|-----PowerPC
|-----Sparc
|-----SystemZ
|-----X86
|-----XCore
|---Transforms
|---VMCore
|-projects
|-runtime
|-test
|-tools
|-unittests
|-utils
|-website
```

There are a few important directories to note here. First, we have the `Debug+Asserts` directory, which contains the binaries and libraries built by Make which are actually executed when running the compiler. We also have the `includes` directory, which contains the header files imported for various classes and

function collections. Finally, we have the `lib` directory, which contains the implementations for many of these header files. The `lib` directory contains many files that are used during any compilation process, but it also contains the `Target` directory, which holds any optimizations and definitions that are specific to a single architecture. In particular, we have shown the subdirectories of the ARM directory. It is in this directory that the `ARMLoadStoreOptimizer.cpp` file resides, and this is the file containing the existing two-phase memory operations optimization. We will add our code into this file as well, as explained below.

4.2 Current Post-Register Allocation Memory Optimization

The file `ARMLoadStoreOptimizer.cpp` contains the class and method definitions for the `MachineFunctionPasses` used to optimize memory instructions. An overview of this file is in listing 4.2.

Listing 4.2: ARMLoadStoreOptimizer.cpp class hierarchy

```
1  Function createARMLoadStoreOptimizationPass
2    // Creates and returns instances of below two classes
3
4  Class ARMPreAllocLoadStoreOpt
5    // To be replaced by our new optimization
6
7  Class ARMLoadStoreOpt
8    runOnMachineFunction
9      LoadStoreMultipleOpti
10       MergeLDR_STR
11         MergeOpsUpdate
12           MergeOps
13       MergeBaseUpdateLSMultiple
14       MergeBaseUpdateLoadStore
15     MergeReturnIntoLDM
```

There are two main classes in this source file: the `ARMPreAllocLoadStoreOpt` class is used prior to register allocation to rearrange memory ops, while the `ARMLoadStoreOpt` class is used to actually perform the memory op merges. We will be attempting to replace the `ARMPreAllocLoadStoreOpt` functionality with our own post-register allocation MagnetPass version. We will continue to use the `ARMLoadStoreOpt` class to actually combine contiguous memory operations. Also included in this file is the `createARMLoadStoreOptimizationPass` function, which is part of the global `llvm` namespace. This function simply creates an object of one of these two classes (depending on whether an argument is passed into it) and returns a `FunctionPass` pointer to its caller pointing

to this new object. The caller is then free to use the object's methods to perform optimizations.

When the optimization is performed, LLVM passes a `MachineFunction` reference (i.e., a function of machine-dependent assembly instructions) to the `runOnMachineFunction()` method within the `ARMLoadStoreOpt` class. This method breaks the `MachineFunction` into basic blocks (called `MachineBasicBlocks`), initializes some member variables for the subsequent methods based on the `TargetMachine` object derived from the `MachineFunction` object, and passes the basic blocks to the `LoadStoreMultipleOpti()` and the `MergeReturnIntoLDM()` methods, which we will examine below. Like most of the methods within this class, `runOnMachineFunction()` operates on object references, and will simply return a boolean indicating whether or not the method was successful in performing its task.

The `MergeReturnIntoLDM()` method is a simple function that checks a basic block for a closing unconditional branch to the link register (`LR`). If such a branch exists, and the instruction immediately preceding the branch is a multiple load instruction, the method will *roll up* the branch into the multiple load instruction by loading the `LR` register contents into the program counter (`PC`), effectively causing a branch to occur without using an explicit branch instruction. This is one of several simple optimizations that LLVM performs to save code size within this source file.

The `LoadStoreMultipleOpti()` method is the main hub of activity for this optimization. It iterates through a basic block and calls several helper meth-

ods to merge all series of contiguous memory instructions which do not overwrite their base registers and use varying but not necessarily ordered base register offsets. Each series of such instructions is merged into a single multiple memory instruction using the `MergeLDR_STR()` method, which in turn calls `MergeOpsUpdate()` which tracks the merges and finally performs the merge using the `MergeOps()` method. Along the way, the `MergeBaseUpdateLoadStore()` method will combine a trailing increment of a base register with the preceding memory instruction and the `MergeBaseUpdateLSMultiple()` method will do the same for a multiple memory instruction. These smaller optimizations complement the main optimization of compacting multiple memory instructions into a single multiple memory instruction.

4.3 New Optimization Pass Class

The full source code is located in Appendix C. The classes explained here are in an anonymous namespace in the `ARMLoadStoreOptimizer.cpp` file. As mentioned in Chapter 2, we require classes for certain data structures, namely `MemOpRecord`, `RegisterDependencies`, and `ClusterPoint`. Each of these classes are pure data structures, that is they do not have any methods associated with them, including constructors. We also define a class `BBPrinter` with methods to make it easier to debug problems with the optimization by printing out the instructions within the basic blocks. This class and its associated methods do not modify the basic blocks in any way, and the code is included in the Appendix for completeness, but will not be futher examined here. Also in this namespace we

include a number of simple functions to work with registers and make boolean conditional statements in the code more readable. Some of these simple functions have been copied from other locations in the LLVM source code, as they give insight into ARM specific instruction nuances.

The most important class defined as part of our optimization is the `MagnetPass` class, which contains many methods to implement the algorithm explained earlier into C++ code. Apart from a constructor and destructor that exist to initialize and clean up logging framework, the class has only a single public method called `runOptimization`. This method is the main entry point to the optimization, and when called it iterates through the algorithm described earlier in Chapter 2. This `runOptimization` method is called once for each basic block, and operates only on the given basic block which is passed by reference to the method, allowing it to operate directly on the machine instructions within the basic block. Each function described in Chapter 2 has a corresponding private method in the `MagnetPass` class, which modifies the class object's internal state stored in private object properties. This state primarily includes the `AllMemOps` array, the `RegDeps` and `MemDeps` arrays. As we are implementing this algorithm in C++, array datatypes are stored in C++ standard library vectors. The process of looping through these vectors is accomplished using the C++ idiom of object iterators. For completeness, the source listing also contains our logging framework and logging messages, which also serve as documentation in addition to the code comments.

4.4 Using the New Class During Compilation

The `ARMLoadStoreOpt` class is the post-register allocation pass which combines sequential memory operations if they have the same base address. It has a public method `runOnMachineFunction` which takes in a function containing machine instructions and performs the instruction compaction on each basic block contained within. This `ARMLoadStoreOpt` class is instantiated by the `createARMLoadStoreOptimizationPass` function, which returns the newly created object to be used to run on machine instruction functions by a pass manager contained in another source file.

The `MagnetPass` class is instantiated in the constructor of the `ARMLoadStoreOpt` class. We then simply run its `runOptimization` method on each basic block just prior to running the instruction merging process normally executed by the `runOnMachineFunction` method. Thus, our optimization rearranges the order of the memory operations into advantageous locations, and the subsequent merging occurs more efficiently. Because this pass depends on the `ARMLoadStoreOpt` class to merge instructions, we chose to couple these two passes relatively closely rather than add additional scaffolding to make our `MagnetPass` class more independent.

4.5 Prototype

During the development of the algorithm, we created a prototype of the new optimization pass in Python. We developed the basic harness necessary to test the new

pass; as an example we created a `MachineInstr` class having the attributes and methods we expected to have in LLVM. The algorithm in Chapter 2 was developed as we created this prototype, and the source code for the prototype is included in Appendix C.

The Python prototype was written as a proof of concept, and contains many Python idioms that do not directly translate into C++. Because the algorithm was designed to work with C++ concepts, such as pointers, that are less prominent in Python, the prototype contains several features included to make the C++ design work when implemented in Python. An example of this are the "handles" attached to several of the classes, which are used in place of more traditional C++ pointers. Using these handles, we can assign machine instructions to a unique identifier that would usually come for free in C++ if we used the address of the memory object.

In addition, when we began implementing the pass in C++, we discovered that some of the methods we had assumed would be available did not exist in LLVM, and we had to make several changes in our final C++ implementation of our algorithm. These changes are explained below. We include the Python prototype because the code is easier to understand than our final C++ code, as a starting point to understand our algorithm.

4.6 Changes to Our Algorithm

4.6.1 Mapping Registers

While a program running on a Cortex-A8 processor only has access to 16 registers (`r0` - `r15`), there are in fact 112 registers the compiler must keep track of. The `RegDeps` vector of Dependencies only needs to keep track of these 16 registers, and because the registers are internally stored in LLVM as enums, we need a way of mapping these register enums to integers we can use to index the `RegDeps` vector. LLVM uses a function called `getRegisterNumbering()` for exactly this purpose. Unfortunately, it can only accept certain numbered registers as valid inputs, and if a passed register does not have a corresponding number, the function will cause LLVM to crash. An example of such a register which has no corresponding number would be the Current Program Status Register (CPSR), which is used to control conditional instruction execution (such as IT blocks). Because we have no control over what registers might be stored in the operands of the machine instructions, we chose to create a similar function called `getRegNum()` which performs precisely the same mapping, but rather than crashing when passed a bad register our function simply returns `-1`. The return value is tested by the function caller, and registers that are not in the range `r0` - `r15` are ignored. Positive return values are used to index into the `RegDeps` vector.

4.6.2 Deleting `RegDeps` entries

Our prototype would endeavor to remove all entries stored in the `RegDeps` vectors once their associated `MemOpRecord`s had their bounds set. This kept the vectors holding only the unused dependencies. This reduced the time needed to update dependencies (the only values in the `RegDeps` vectors were values that needed updating), but the process of searching for and removing dependencies pointing to `MemOpRecord`s which had already had their bounds updated was time consuming. For our C++ implementation, we substituted a simple if statement during the dependency updating process which checked before overwriting the bounds that the current bound was `NULL`. If it was not, it had already been set, and the `MemOpRecord` would be skipped. This if statement removed the need to remove `RegDeps` entries once their `MemOpRecord` ranges had been updated. Because the basic blocks on average are short, the time required to skip entries in the `RegDeps` vectors was deemed to be acceptable, and the vector entries were not removed once their `MemOpRecord`s were updated.

4.6.3 Dependencies at Basic Block Ends

The algorithm implemented in the prototype ran two passes to determine ranges of movement. The first pass would run `endLowerBoundUsingRegs()` functions as it iterated through the basic block to set the bounds properly when iterating past dependencies. Once all instructions were iterated through, the `endLowerBoundUsingRegs()` functions were again run to set the remaining dependencies to `NULL` values. In our C++ implementation, we initialized all dependencies

with NULL values, and any dependencies that were not overwritten in the iteration through the basic block would remain NULL. Because these are the same instructions that can be moved to the ends of the basic blocks, we would want them to have NULL bounds anyway. In this way, by initializing them to NULL, we can avoid this final step of running the endLowerBoundUsingRegs() functions after the end of the loop. Similarly, the second pass no longer need the trailing endUpperBoundUsingRegs() functions after executing its reversed order iteration through the basic block.

4.6.4 Algorithm Rewrite

In the early stages of this project, we employed a simple two pass algorithm to generate dependencies, then moved all memory operations within a basic block iteratively without regenerating the dependency information. This was done to minimize the number of passes through the basic block required to perform the optimization. Unfortunately, several months after the initial design we discovered a flaw in this algorithm. We will illustrate using Listing 4.3.

Listing 4.3: Block Before Moves

```
1  ADD r0, r6, #18
2  LDR r1, sp, #4  <-+ UpperBound
3  STR r1, r0, #4 ---|
4  LDR r0, sp, #12 <-+ LowerBound
```

This block is taken from the the qsort benchmark. We see that the two pass range of movement generation has been performed and that the STR instruction cannot move up beyond the LDR instruction pointed to by UpperBound (there is

a write dependency on the `r1` register). This `LDR` instruction itself has a range of movement (not shown) with an upper bound of the top of this code block. If this `LDR` instruction is moved by our optimization, the result can be seen in Listing 4.4.

Listing 4.4: Block After Moves

```
1 LDR r1, sp, #4  <-+ UpperBound
2 ADD r0, r6, #18   |
3 STR r1, r0, #4 ---|
4 LDR r0, sp, #12 <-+ LowerBound
```

As shown, the `LDR` instruction has moved to the top of this block of code. This is perfectly legal, however note that because the `UpperBound` is a pointer to the `LDR` instruction, the `UpperBound` moves with the `LDR` instruction. As far as the optimization is concerned, the `STR` can move up above the intermediate `ADD` instruction. However, in reality if the `STR` instruction was to move above the `ADD` instruction it would be violating a dependency on `r0` (which is defined by the `ADD` instruction) and would lead to a violation of code correctness. Clearly, whenever a series of instructions is moved, any range of movements could be outdated. Rather than attempt to track these dependencies on the fly (i.e., attempt to repair them during a move), we elected to rewrite the algorithm so that after every clustering of instructions all register dependencies are cleared and regenerated. Unfortunately this leads to a significant increase in passes through the basic blocks, but this is necessary to prevent the loss of program correctness.

The corrected algorithm was presented in Chapter 3, and differs from our original design. Thus, the code listed for the Python prototype uses this same early version of our design and will not correctly handle this special case. Because the

prototype was designed to aid our refinement of the algorithm and not to be used, we have elected to not update it to match our final algorithm.

4.7 Additional Considerations

Because the pre-register allocation pass we are comparing against is designed to integrate with the `ARMLoadStoreOpt` pass, there is no way to disable the pre-register allocation pass using LLVM command line options without disabling the merging pass as well. We must therefore comment out the pre-register allocation pass in the code to disable it, and thus fairly compare our new optimization against the old one. Because we maintain separately built binaries for versions of LLVM with and without this optimization, comparisons between optimizations were simple.

Chapter 5

Results

Having implemented our MagnetPass post-register allocation algorithm in code, we must now examine how it compares to the existing optimization in LLVM. To do this, we will compile a set of source files and compare the outputs generated by different levels of optimizations.

5.1 Benchmarks Used

When testing compiler performance, it is important to select a suite of source files to serve as a representative sampling of the sorts of programs the compiler would be expected to build. These programs are used to verify that the compiler maintains program correctness and give the compiler (or, in this case, compiler optimization) raw material to use to display the efficacy of a particular compilation technique. There are many benchmark suites in existance that are used to demon-

strate compiler or system performance; an example would be the popular SPEC CPU benchmark suites, which is used to demonstrate CPU-intensive workloads.

Because our optimization focuses on ARM processors, we choose to select a series of benchmarks that targets embedded system workloads; namely the freely available MiBench suite [22]. MiBench is a set of 35 applications that are representative of typical programs executed in an embedded context. These applications are split into a number of different domains: Automotive/Industrial Control, Consumer Devices, Office Automation, Network, Security, and Telecommunications. We choose to select a sampling of these 35 benchmarks to gather data from and use to compare optimizations. We will select the following 4 benchmarks:

5.1.1 dijkstra

The `dijkstra` benchmark is part of the Network domain, and constructs a large graph using an adjacency matrix representation. It then calculates the shortest path between every pair of nodes by using Dijkstra's algorithm in $O(n^2)$ time. The source file for this benchmark is 174 lines of C code long, and it generates an ARM ASM file that is 628 lines in length.

5.1.2 qsort

Part of the Automotive/Industrial Control domain, the `qsort` benchmark implements the well known quick sort algorithm to sort large arrays of strings into ascending order. This is a very common requirement for many different applica-

tions. This program has a source file that is only 55 lines of C code, and generates 289 lines of ARM ASM when compiled.

5.1.3 sha

The SHA (Secure Hash Algorithm) benchmark is part of the Security domain, and is used to generate message digests 160 bits in size for an arbitrary input. The SHA family of hash algorithms is commonly used to generate digital signatures and store password information without revealing a plaintext password. The MiBench suite uses the original SHA algoritm, now known as SHA-0, which was later revised into SHA-1. Later generations of this algorithm (SHA-2, etc) are still in common use today and are published by the National Institute of Standards and Technology (NIST). The source file for this program is 210 lines of C code, which is compiled to 600 lines of ARM ASM. There is also a driver file which applies the dataset to the algorithm in the main file; it is 31 lines of C code and 107 lines of ARM ASM in length.

5.1.4 stringsearch

The Office domain contains the `stringsearch` benchmark, which simply moves through text looking for a particular string without regard to case. This program is composed of 4 source files containing over 3000 lines of code, though most of those are the input dataset within the driver file. Nearly 5000 lines of ARM ASM is generated by these files, though again most of this is constant data.

Each of these benchmarks comes with its own dataset, and its own set of expected output. For most of these benchmarks, the data is stored separately, and is not reflected in the numbers given above (the notable exception being `stringsearch`). For more details about the MiBench suite, see Appendix A.

5.2 Method of Testing

To generate our comparisons, we modified each benchmark's Makefile to use LLVM to compile the program. We generated ARM ASM using a two step compilation procedure. First, we used llvm-gcc to convert the high level language files into LLVM bytecode with the -emit-llvm and -c flags. This bytecode file was then used as input for the llc program in the LLVM library to convert the LLVM bytecode to human readable assembly language. The bytecode was also compiled down to an executable and tested on a BeagleBoard ARM development board running an ARM Cortex-A8 CPU, which implements the ARMv7 ISA. The output was compared against the benchmark references to ensure that program correctness was maintained.

We compiled the benchmarks using a variety of scenarios to illustrate the effectiveness of our post-register allocation optimization, and to compare it against the existing pre-register allocation version, a version with no pre- or post-register allocation optimizations, and a version with no optimizations at all (-O0). This allows us to see how effective the optimization is compared to several different parallel data points.

The version with no optimization is used as a baseline, and due to the large number of other optimizations it disables, we expect that it will always compare unfavorably with any -O3 compiler build. It is denoted in the Tables below as "O0 none".

The version without either the pre- or post-register allocation optimization includes all the normal optimizations found when using -O3, but the `ARMLoadStoreOpt` class which combines single memory ops into multiple memory ops does not have any passes that run before it to move memory ops into advantageous positions. This provides a more realistic baseline, as any useful optimization should increase the rate of single ops turning into multiple ops, whether the optimization runs before or after the conversion process. In the Tables below, we mark this set of results as "O3 nopreorpost".

The final two Table columns include one or the other of the optimizations which run before the combining process, and are denoted as "O3 pre_ra" and "O3 MagnetPass" respectively. We wish to especially compare these two columns, as it will allow us to make a direct comparison between the pre- and post-register allocation implementations of the optimization.

To ease testing, we maintained three versions of the LLVM compiler built with various combinations of optimizations enabled or disabled. After generating the assembly files, we used a Python script to extract a list of all the assembly instructions and categorize them to give us an overview of the effectiveness of the optimizations. We then examined the assembly files by hand where the aggregate data indicated an interesting modification to the generated assembly files.

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	160	123	123	123
LOAD_MULTIPLE	0	3	3	3
STORE_SINGLE	92	53	53	53
STORE_MULTIPLE	0	2	2	2
OTHER	149	138	138	138
Total	401	319	319	319

Table 5.1: Summary of instructions in dijkstra_large

5.3 Comparative Results

5.3.1 dijkstra

In the case of the `dijkstra` benchmark, neither the pre- or the post-register optimization had any effect on the efficiency of the code. As a matter of fact, the O3 version without the pre optimization is exactly the same as the version with the pre optimization. We will, however, take this as an opportunity to show the differing heuristic used by the MagnetPass optimization. Listing 5.1 displays the modifications to the ASM performed by the post-RA optimization.

Listing 5.1: Dijkstra ASM Movements

```
1     bgt .LBB5_2
2    @ BB#1:                                 @ %bb
3     ldr r4, .LCPI5_0
4  +-
5  |  ldr r0, .LCPI5_1
6  |  mov r2, #27
7  |  mov r1, #1
8  +>  ldr r3, [r4]
```

```
 9 |     bl  fwrite
10 | +-
11 | |   ldr r0, .LCPI5_2
12 | |   mov r2, #40
13 | |   mov r1, #1
14 | +>  ldr r3, [r4]
15 |     bl  fwrite
16 |    .LBB5_2:                                 @ %bb1
17 |     ldr r0, [sp, #20]
18 | ... snipped ...
19 |     orr r0, r1, r0, lsl #16
20 |     ldr r1, .LCPI5_3
21 |     bl  fopen
22 | +-
23 | |   str r4, [sp, #12]
24 | +>  str r0, [sp]
25 |     b   .LBB5_7
26 |    .LBB5_3:                                 @ %bb2
27 |                                             @   in Loop: Header=
   |                                                 BB5_7 Depth=1
28 | ... snipped ...
29 |     b   .LBB5_5
30 |    .LBB5_4:                                 @ %bb3
31 |                                             @   in Loop: Header=
   |                                                 BB5_5 Depth=2
32 | +-
33 | |   add r2, sp, #4
34 | |   mov r1, r5
35 | +>  ldr r0, [sp]
```

```
36      bl  __isoc99_fscanf
37      ldr r0, [sp, #12]
38      ldmib   sp, {r1, r2}
39  ... snipped ...
```

In Listing 5.1, we see that the memory ops at lines 4, 10, 22, and 32 are moved downward in the code as far as possible since no suitable cluster point is found. This is performed to reduce the live range of the value held in the registers. However, the density of memory operations is not sufficient to allow for clustering, and Table 5.1 illustrates that there are no additional multiple memory operations.

5.3.2 qsort

In `qsort`, again we see that the pre optimization has little effect on the ASM code, and in fact merely moves a single store as shown in Listing 5.2.

Listing 5.2: Qsort Pre ASM Movements

```
1       ldr r2, [sp, #12]
2       add r2, r2, r2, lsl #2
3       add r2, r6, r2, lsl #2
4   +>  str r0, [r2, #12]
5   |   str r1, [r2, #16]
6   +-
7       ldr r1, [sp, #12]
8       add r0, r1, #1
9       str r0, [sp, #12]
10  ... snipped ...
```

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	81	50	50	51
LOAD_MULTIPLE	0	4	4	4
STORE_SINGLE	56	21	21	21
STORE_MULTIPLE	0	3	3	3
OTHER	96	107	107	109
Total	233	185	185	188

Table 5.2: Summary of instructions in qsort_large

We see that the order of the two stores on lines 4 and 5 has simply been switched. Because the later combining optimization does not care about instruction order, this movement is completely ineffective in assisting the combining optimization. Our code has a condition that if a set of instructions are contiguous, it is unnecessary to reorder them if no other memory ops are present apart from the contiguous block. This avoids this unnecessary movement of instructions.

Though there are two memory instructions, the later combining optimization does not combine them, and thus the movement has no effect. This is due to the fact that the `MergeOps` method, which is responsible for the final merging of the contiguous instructions, will not merge only two instructions unless the starting offset is equal to 0. If the starting offset is greater than 0, LLVM would need to add an instruction to create the proper offset in a register, then use that register for the multiple memory instruction base register. Thus, we would be adding an instruction to remove an instruction, and `MergeOps` simply skips this unprofitable exercise.

Though the MagnetPass optimization moves other memory operations, it does not enable any more multiple memory operations to form, due to the sparseness of memory ops in these code regions (see Listing 5.3).

Listing 5.3: Qsort Post ASM Movements

```
1     str r1, [sp, #12]
2     ldr r0, [lr, #4004]
3     cmp r0, #1
4   * ble .LBB1_11
5    @ BB#1:                                @ %bb1
6     add r6, sp, #73, 18 @ 1196032
7     ldr r0, [r6, #4000]
8  ... snipped ...
9     str r0, [sp, #12]
10   .LBB1_3:                               @ %bb3
11                                          @ =>This Inner Loop
                                               Header: Depth=1
12 +-
13 |   add r2, sp, #8
14 |   mov r1, r4
15 +>  ldr r0, [sp, #20]
16    bl  __isoc99_fscanf
17    cmp r0, #1
18    bne .LBB1_7
19 ... snipped ...
20   @ BB#4:                                @ %bb4
21                                          @   in Loop: Header=
                                               BB1_3 Depth=1
22 +-
23 |   add r2, sp, #4
24 |   mov r1, r4
25 +>  ldr r0, [sp, #20]
26    bl  __isoc99_fscanf
```

```
27      cmp r0, #1
28      bne .LBB1_7
29  ... snipped ...
30     @ BB#5:                                  @ %bb5
31                                              @   in Loop: Header=
                                                   BB1_3 Depth=1
32  +-
33  |   mov r2, sp
34  |   mov r1, r4
35  +>  ldr r0, [sp, #20]
36      bl  __isoc99_fscanf
37      cmp r0, #1
38      bne .LBB1_7
39  ... snipped ...
40      add r4, sp, #24
41      ldr r5, .LCPI1_6
42      bl  printf
43  +-
44  |   mov r2, #20
45  |   ldr r3, .LCPI1_5
46  |   mov r0, r4
47  +>  ldr r1, [sp, #12]
48      bl  qsort
49      mov r0, #0
50      b   .LBB1_9
51  ... snipped ...
52      ldr r0, [sp, #12]
53      ldr r1, [sp, #16]
54      cmp r1, r0
```

```
55    + blt .LBB1_8
56    +@ BB#10:                                   @ %bb12
57    + mov r0, #0
58    * add lr, sp, #73, 18 @ 1196032
59    * str r0, [lr, #3992]
60    + str r0, [lr, #3996]
61    * add r1, sp, #73, 18 @ 1196032
62    + sub sp, r11, #24
63    + ldr lr, [sp, #28]
64    * ldr r0, [r1, #3996]
65    * ldmia   sp, {r4, r5, r6, r7, r8, r9, r11}
66    + add sp, sp, #32
67    + bx  lr
68    *.LBB1_11:                                  @ %bb
69      ldr r0, .LCPI1_0
70      ldr r3, [r0]
71      mov r1, #1
72  ... snipped ...
73      bl  fwrite
74      mvn r0, #0
75      bl  exit
76    *@ BB#12:
77      .align  2
78     .LCPI1_0:
79      .long   stderr
80  ... snipped ...
```

Besides the previously mentioned loads moving (lines 15, 25, 35, and 47), we must also note a couple of things here. First, by moving the loads down, and past

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	192	178	178	178
LOAD_MULTIPLE	0	2	2	2
STORE_SINGLE	120	91	91	91
STORE_MULTIPLE	0	1	1	1
OTHER	190	174	174	175
Total	502	446	446	447

Table 5.3: Summary of instructions in sha

some of the arithmetic ops below them, to just before the branch instructions, we may be reducing performance, as the arithmetic instructions can act as a buffer of instructions to be performed while the loads are finishing.

Secondly, we see that the post-RA MagnetPass optimization has to potential to actually increase code size when optimizing conditional instructions, as seen by the large block of contiguous instructions on lines 55 - 68 being modified and instructions being added. This occurs because by breaking apart the conditional instructions into a new basic block, we increase the code size, which is the opposite of what we wish to do here. We have noted this problem under the Future Work Section at the end of this thesis, but because we are choosing to focus on increasing code density through combining memory instructions, and because conditional blocks are more rarely seen in practice, we are choosing to ignore this issue to keep our algorithm (and C++ code) simpler.

5.3.3 sha

The `sha` program has both a core module (sha.c) and a driver module (sha_driver.c), which we will examine separately.

The sha.c file is similar to the `qsort` benchmark in that the pre RA optimization merely moves a single store, but does not assist in the creation of any new multiple memory operations. We have omitted the code listing due to its similarity to the `qsort` benchmark, and it not adding anything to our analysis.

Listing 5.4: Sha ASM Movements

```
1     ldr r3, [r2, #-32]
2     ldr r12, [r2, #-12]
3     eor r3, r12, r3
4   + ldr r12, [r2, #-56]
5     ldr r2, [r2, #-64]
6   * eor r3, r3, r12
7     eor r2, r3, r2
8     str r2, [r1, r0, lsl #2]
9     ldr r1, [sp, #344]
10 ... snipped ...
11    b   .LBB0_8
12   .LBB0_7:                                @ %bb6
13                                           @   in Loop: Header=
                                                BB0_8 Depth=1
14 +>  ldr r1, [sp, #324]
15 |   ldr r0, [sp, #332]
16 +-
17    ldr r2, [sp, #328]
```

```
18      ldr r3, [sp, #344]
19      bic r1, r1, r0
20  ... snipped ...
21      b   .LBB0_11
22     .LBB0_10:                                @ %bb10
23                                              @   in Loop: Header=
                                                    BB0_11 Depth=1
24  +-
25  |   mov r1, sp
26  +>  ldr r0, [sp, #344]
27  +>  ldr r2, [sp, #332]
28  |   ldr r3, [sp, #328]
29  +-
30      ldr r0, [r1, r0, lsl #2]
31      eor r1, r2, r3
32      ldr r2, [sp, #324]
33  ... snipped ...
34      b   .LBB0_17
35     .LBB0_16:                                @ %bb16
36                                              @   in Loop: Header=
                                                    BB0_17 Depth=1
37  +-
38  |   mov r1, sp
39  +>  ldr r0, [sp, #344]
40  +>  ldr r2, [sp, #332]
41  |   ldr r3, [sp, #328]
42  +-
43      ldr r0, [r1, r0, lsl #2]
44      eor r1, r2, r3
```

```
45      ldr r2, [sp, #324]
46  ... snipped ...
47      .type   sha_init,%function
48     sha_init:                                @ @sha_init
49     @ BB#0:                                  @ %entry
50    + sub sp, sp, #4
51      ldr r1, .LCPI2_0
52      str r1, [r0]
53    + str r0, [sp]
54      ldr r0, .LCPI2_1
55      ldr r1, [sp]
56      str r0, [r1, #4]
57  ... snipped ...
58     .LBB5_2:                                 @ %bb2
59                                              @ =>This Inner Loop
                                                    Header: Depth=1
60      add r4, sp, #2, 20  @ 8192
61  +-
62  |   mov r0, sp
63  |   mov r1, #1
64  |   mov r2, #2, 20  @ 8192
65  +>  ldr r3, [r4, #4]
66      bl  fread
67      str r0, [r4]
68      cmp r0, #0
69  ... snipped ...
```

In Listing 5.4, the clustering algorithm of the MagnetPass optimization is clearly seen, as in several places groups of loads are brought together to form clusters of

sufficient size to be combined by the later optimization. Unfortunately, the clusters are not combined due to them not all using contiguous memory locations. For example, the load instructions at lines 39-41 use memory addresses `sp + 328`, `332`, and `344` which are not all 4 bytes apart (`336` and `340` are missing), and thus the three instructions cannot be combined by the later optimization. While `sp + 328` and `332` could be combined, as mentioned above `MergeOps` does not combine them because they do not have an initial base register offset of 0.

There are some other peculiarities to note with this code listing. Near the top, at line 4, we see that a `ldr` appears to be added, but checking the table of instruction counts, we see that this is not the case. In fact, the `ldr` instruction used to appear above the first `eor` instruction, but was moved below it. In the process, the `ldr` register was changed from its original register of `r4` to `r12`, and thus was treated as a *new* instruction that replaced the *old* ldr instruction. This also explains why the second `eor` instruction below has been changed; rather than using the original `r4` register it is using the `r12` register, and the text diff program we are using reports this as a change, and marks it with an asterisk.

More interesting is the second to last change (lines 50 and 53) where it appears that two instructions have been added. In fact, the top `sub` instruction previously was a `str` instruction which was moved down (to the second *addition* point). The `str` instruction previously was an instruction that simultaneously stored a value and updated the `sp` pointer by subtracting 4 from it. Because the `str` instruction moved below a use of the `str` value, the instruction was *split* into an initial subtraction from the `sp` pointer and then a later store. This is where the new `sub` instruction comes from, and the reason why there is an extra "other" instruction

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	18	13	13	13
LOAD_MULTIPLE	0	1	1	1
STORE_SINGLE	17	8	8	8
STORE_MULTIPLE	0	1	1	1
OTHER	29	30	30	30
Total	64	53	53	53

Table 5.4: Summary of instructions in sha_driver

listed for the MagnetPass optimization in the Summary Table. Because this move was not useful enough to cause a new multiple memory instruction to be created, it would be better for the original `str` instruction to simply not move at all, and thus the program would not incur the code size penalty of adding an unnecessary new instruction. Unfortunately, our algorithm does not allow us to make this determination (indeed, the original pre RA optimization also does not know whether a change will cause a combination; both pre and post use heuristics).

The sha_driver.c file creates a very small ASM file, and in fact there is no difference between the pre RA optimization and no optimization at all. Our post-RA MagnetPass optimization moves a single ldr instruction down, but does not lead to any new combinations. We have omitted both code listings due to neither of them adding to this analysis.

5.3.4 stringsearch

While we have seen clustering in previous benchmarks, until this point we have not seen any actual combining of memory ops into a multiple memory op. The

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	88	67	65	65
LOAD_MULTIPLE	0	2	3	3
STORE_SINGLE	35	24	24	24
STORE_MULTIPLE	0	2	2	2
OTHER	78	63	63	63
Total	201	158	157	157

Table 5.5: Summary of instructions in bmhasrch

bmhasrch.c file requires one of these two optimizations in order to combine one section of code into such an op. The section of code is shown in Listing 5.5.

Listing 5.5: bmhasrch ASM Movements

```
1  -------- None --------
2  .LBB0_8:                                @ %bb11
3                                          @   in Loop: Header=BB0_9
                                               Depth=1
4    ldr  r0, [r4]
5    ldr  r2, [sp, #4]
6    ldr  r3, [r4, #4]
7    ldrb  r2, [r0, r2]
8    add  r0, r3, r0
9  -------- Pre RA --------
10 .LBB0_8:                                @ %bb11
11                                         @   in Loop: Header=BB0_9
                                               Depth=1
12   ldmia  r4, {r0, r2}
13   ldr  r3, [sp, #4]
14   ldrb  r3, [r0, r3]
15   add  r0, r2, r0
16 -------- MagnetPass --------
17 .LBB0_8:                                @ %bb11
18                                         @   in Loop: Header=BB0_9
                                               Depth=1
19   ldr  r2, [sp, #4]
20   ldmia  r4, {r0, r3}
21   ldrb  r2, [r0, r2]
22   add  r0, r3, r0
```

As seen here, the original O3 compilation yields two ldr instructions separated by another `ldr` instruction with a different base. The pre RA optimization moves

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	103	69	67	67
LOAD_MULTIPLE	0	4	5	5
STORE_SINGLE	51	32	32	32
STORE_MULTIPLE	0	2	2	2
OTHER	96	83	83	83
Total	250	190	189	189

Table 5.6: Summary of instructions in bmhisrch

the lower `ldr` up, while our MagnetPass optimization moves the upper `ldr` down, and in both cases they are combined by the later optimization. Because the pre RA optimization occurs before register allocation, the registers used for the resulting `ldmia` instruction are different, and subsequent code reflects the different registers chosen for these values. Because the post-RA optimization occurs after the registers have been chosen, the register numbers are unchanged.

This listing also illustrates that two contiguous instructions will be combined if the first base register offset is 0, as seen in line 4. While previously we have seen two memory instructions being skipped over, here there is no need to add an instruction to incorporate the starting offset into a base register, and it is thus profitable to combine the two instructions into a single multiple load instruction.

Our optimization also moves some other `ldr` instructions down in the ASM file, however no further combinations are performed, so we omit them here.

The bmhisrch.c file is very similar to the bmhasrch.c file above, including a single instance where both the pre- and post-RA optimizations enabled a combining of two `ldr` instructions into a single `ldmia` instruction. There were also some instances of single `ldr` instructions moving down, as usual, but no additional com-

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	74	53	51	51
LOAD_MULTIPLE	0	2	3	3
STORE_SINGLE	36	26	26	26
STORE_MULTIPLE	0	1	1	1
OTHER	71	60	60	60
Total	181	142	141	141

Table 5.7: Summary of instructions in bmhsrch

Instruction Type	O0 none	O3 nopreorpost	O3 pre_ra	O3 MagnetPass
LOAD_SINGLE	78	57	57	58
LOAD_MULTIPLE	0	3	3	3
STORE_SINGLE	45	26	26	26
STORE_MULTIPLE	0	2	2	2
OTHER	93	78	78	80
Total	216	166	166	169

Table 5.8: Summary of instructions in pbmsrch_large

binations were performed.

The bmhsrch.c file is also very similar to the bmhasrch.c file above, including a single instance where both the pre- and post-RA optimizations enabled a combining of two `ldr` instructions into a single `ldmia` instruction. There were also some instances of single `ldr` instructions moving down, as usual, but no additional combinations were performed.

In the pbmsrch_large.c file, there are no modifications made by the pre RA optimization, so the two ASM files are identical. This file contains a rather substantial section of conditional instructions, and as the MagnetPass optimization does not handle this well, we actually add an entire basic block and some extra instructions in place of the logic flow that used to pass through a set of conditional instructions.

This is unfortunate, and in instances like these with conditional sections of code, our algorithm would be advised to simply skip over these sections rather than make the modifications. Due to the length and complexity of the conditional section, and given that this is not an area this optimization is focusing its attention on, we have opted not to include an ASM code listing for this file.

5.4 Effect on Compilation Time

In addition to checking correctness and that our algorithm is working as intended, we also wish to compare the time it takes to execute our new pass with the time expended by the old pass. In Table 5.9, we show the comparative analysis. For each file, we used LLVM's `--time-passes` parameter to measure the total compilation time used by llvm in building the assembly (*.s) files. We built each file five times and measured and compared the user+system time, rather than the wall-clock time. We choose to measure total compile time, rather than time the individual pass. We do this because the benchmark files we are using are relatively small, and the compile time for each is very short. The `--time-passes` option reports with a minimum granularity on the order of a few hundreds of microseconds. Because our total compilation times are on the order of tens of milliseconds, the shorter pass-specific times are less accurate for calculating a percent difference.

We see in Table 5.9 that our new pass takes on average 35% more time to compile than the existing two phase optimization. Much of this is likely due to our algorithm requiring a pair of passes through the basic block to regenerate our range

Filename	Pre_RA	MagnetPass	% diff
dijkstra	0.0960	0.1368	42.50
qsort	0.0648	0.0888	37.04
sha_driver	0.0208	0.0224	7.69
sha	0.1184	0.1936	63.51
pbmsrch_large	0.0536	0.0712	32.84
bmhisrch	0.0616	0.0808	31.17
bmhsrch	0.0432	0.0584	35.19
bmhasrch	0.0520	0.0656	26.15
Average	0.0638	0.0897	34.51

Table 5.9: Compilation time (in seconds) comparison

of movement, and the fact that we must perform this pair of passes before every set of memory ops that must be clustered together. Thus, the more sets of memory ops with differing base registers there are in a basic block, the more passes must occur. This relationship is linear with the number of clusters, and is affected not only by the absolute number of memory ops in a basic block, but also the number of different base registers these ops use. By contrast, the current pre-register optimization only iterates forward through the basic block, and is thus less affected by these same two properties.

5.5 Analysis

These four benchmark examples show that in most cases the MagnetPass post-register allocation optimization we have developed provides similar improvements to code size as the existing optimization that runs before register allocation. In the `dijkstra` and `qsort` benchmarks, we see that the optimization reduces the

live ranges of register values by moving loads down if no clustering is possible. In the `sha` benchmark, we see clustering occur in probabilisticly advantageous ways, moving distant memory ops into contiguous blocks, which increases the probability that the later combining optimization will combine them. Unfortunately, due to the lower density of single memory instructions in these two benchmarks, we do not see any clustering performed that allows the ARMLoadStoreOpt optimization to generate block memory instructions. However, in the `stringsearch` benchmark, we see this clustering actually result in a combination of two memory ops, reducing code size in a way similar to the pre register allocation optimization, but with the added benefit of moving loads down and thus (slightly) reducing the live range of the variable and reducing register pressure.

We must also note some of the disadvantages to this approach. Many of these are simply due to the relative maturities of the two code bases, and our reluctance to over-complicate the algorithm by accounting for certain edge cases. In both `qsort` and `stringsearch`, we see that the post-register allocation optimization does not currently gracefully handle conditional code blocks, and can thus increase code size if a memory op within a conditional block is moved. Also, in the `sha` benchmark, we see an instance where our optimization's unawareness of certain micro-optimizations causes code expansion by moving a memory op away from an arithmetic op that affects the memory op's base register. A more robust version of our optimization would have to account for these issues in its heuristic algorithm.

More broadly, we must also compare the two optimizations by which point in compilation provides more information to assist in moving memory ops for the purposes of combining them. Because the LLVM bytecode model is an infinite register

machine (due to this intermediate representation using SSA), any target machine with a fixed set of registers (such as ARM) will introduce the possibility of aliasing or spilling a value, particularly in situations with increased register pressure. Thus, it is generally more advantageous to perform this sort of optimization prior to register allocation, as discrete values are more easily identified and tracked when these two issues are not a concern. In situations with higher register pressure, we expect the pre-register allocation algorithm to outperform our post-register allocation optimization for this reason. Some of these advantages are reduced somewhate by the set of functions and properties LLVM makes available for this purpose, such as the *kill flag* marking the last use of a register for a particular value. Further examination would be necessary to examine the relative efficiencies of these two domains, and their respective functions.

Our analysis thus indicates that while it is possible to create a very serviceable post-register allocation version of the existing optimization (as we have made the beginnings of here), there exist a number of advantages to implementing this optimization before register allocation has occured. In addition, our heuristic of blindly moving all memory ops to reduce their live ranges may lead to more disadvantages due to missed later optimizations than advantages due to reduced register pressure.

Chapter 6

Summary

In this thesis, we examined a pre-register allocation memory optimization in LLVM and attempted to recreate it as a post-register allocation optimization (our MagnetPass) and compare the results. We examined what must be present for such an optimization to work, and developed an algorithm to generate a data structure containing safe ranges to move each memory op. We then implemented this using C++ in the LLVM compiler and tested it against several benchmarks from the MiBench benchmark suite to prove its operation. Finally, we compared our optimization against the existing pre-register allocation optimization both using the raw data from the benchmark compilation and using reasoning about which would likely have access to better data more quickly. We found that our algorithm is on average 35% slower than the existing implementation, but provides comparable results outside of a few edge cases.

Our optimization has several clear deficiencies, many of which are noted in the

below Section regarding Future Work. However, the optimization does correctly identify memory ops that may be safely moved into contiguous blocks and successfully moves them whilst updating the basic block instructions if necessary. With additional effort, most of the disadvantages noted in Chapter 5 could be overcome and the optimization would likely produce code sizes of similar quality to the existing pre-register allocation optimization. However, while the code generated would be of similar quality, because the information required for this algorithm is more easily accessed before transforming the LLVM intermediate representation in SSA form to the target ARM ISA, the pre allocation optimization would almost certainly be more efficient in terms of time taken to execute. Because of this, and because of the existing maturity of the current implementation for this optimization, it is unlikely that development on our optimization will continue or be submitted to the LLVM project for consideration or improvement.

6.1 Future Work

As mentioned in Chapter 3, our current implementation does not handle pointer aliasing for base pointers. Instead, it conservatively ends all stores when a load is detected, and vice versa. It would be desirable to improve the flexibility of this optimization pass by implementing handling of pointer aliasing.

Currently, this implementation doesn't attempt to track changes to base pointers; if the base pointer changes, we do not attempt to change any of the subsequent offsets using that base pointer to widen the range of movements. Implementing

this could allow for wider range of movements, and thus more efficent collecting of related memory operations.

Our optimization also does not work with conditional instructions, and in fact increases the code size by removing the conditional instructions and inserting a basic block. This does not really affect the main comparison made by this thesis, but if this were to be turned into a real compiler optimization to be submitted to the LLVM project, support for conditional instructions would have to be added.

Bibliography

[1] The palm pre vs. apple's iphone 3g: Preliminary results and 3gs discussion. http://www.anandtech.com/show/3547, June 2009.

[2] Arm architecture reference manual armv7-a and armv7-r edition version 4.0. http://arm.com/, June 2010.

[3] Beagleboard system reference manual rev. c4. http://beagleboard.org/, Nov. 2010.

[4] Cortex-a8 technical reference manual rev. r3p2. http://arm.com/, July 2010.

[5] The llvm compiler infrastructure. http://llvm.org/, Oct. 2010.

[6] Realview compilation tools assembler guide version 4.0. http://arm.com/, Dec. 2010.

[7] Droid by motorola. http://www.android.com/devices/detail/droid-by-motorola, Dec. 2013.

[8] What is the fastest way to copy memory on a cortex a8? http:

//infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html, Feb. 2014.

[9] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. 2007.

[10] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys (CSUR)*, 35(3):223267, 2003.

[11] T. Bonny and J. Henkel. Efficient code density through look-up table compression. In *Proceedings of the conference on Design, automation and test in Europe*, pages 809–814. EDA Consortium, 2007.

[12] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[13] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 185–197. IEEE, 2007.

[14] N. Chabini and M. Wolf. Reordering the assembly instructions in basic blocks to reduce switching activities on the instruction bus. *Computers & Digital Techniques, IET*, 5(5):386–392, 2011.

[15] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins,

and P. W. Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[16] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of Conference on Programming Language Design and Implementation*, 1988.

[17] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of arm binaries. In *ACM SIGPLAN Notices*, volume 39, pages 211–220. ACM, 2004.

[18] S. Devadas and S. Malik. A survey of optimization techniques targeting low power vlsi circuits. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 242–247. ACM, 1995.

[19] M. Drinic, D. Kirovski, and H. Vo. Code optimization for code compression. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 315–324. IEEE, 2003.

[20] T. Edler von Koch, I. Böhm, and B. Franke. Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16-and 32-bit instructions. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 180–189. ACM, 2010.

[21] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming general-purpose multicore processors using streams. *ACM SIGPLAN notices*, 43(3):297–307, 2008.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[23] S. Hack and G. Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[24] A. Kudriavtsev and P. Kogge. Generation of permutations for simd processors. In *ACM SIGPLAN Notices*, volume 40, pages 147–156. ACM, 2005.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *In Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[26] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 194–203. IEEE, 1997.

[27] C. Lupo and K. Wilken. Post register allocation spill code optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[28] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *ACM SIGPLAN Notices*, volume 41, pages 132–143. ACM, 2006.

[29] D. Patterson and J. Hennessy. *Computer organization and design*. Morgan Kaufmann, third edition, 2005.

[30] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, 1996.

[31] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[32] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *Micro, IEEE*, 20(4):47–57, 2000.

[33] G. Ren, P. Wu, and D. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 89b–89b. IEEE, 2005.

[34] D. Shin and J. Kim. An operation rearrangement technique for low-power vliw instruction fetch. In *Proceedings of the Workshop on Complexity-Effective Design*, 2000.

[35] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 38–39. IEEE, 1994.

[36] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[37] O. Traub, G. Holloway, and M. D. Smith. *Quality and speed in linear-scan register allocation*, volume 33. ACM, 1998.

[38] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179. ACM, 2010.

Appendix A

MiBench Benchmark Information

A.1 Introduction

The benchmark used for this thesis was MiBench version 1.0 [22] which originated from the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor. All benchmarks were accessed from http://www.eecs.umich.edu/mibench/index.html on 10/2/2010. Output was verified with sample output downloaded from the same site.

All benchmarks were compiled using the -mthumb switch using LLVM frontend CLANG on an ARM Cortex-A8 microprocessor running on a BeagleBoard. Full results are listed below of the results of attempted compilation using CLANG running on Ubuntu 10.04.1 LTS.

A.2 Compilation

automotive			
Name	**In Results**	**Compiled**	**Notes**
basicmath	no	yes	Outputs differed due to FP rounding
bitcount	no	yes	None
qsort	yes	yes	None
susan	no	yes	None
consumer			
jpeg	no	yes	None
lame	no	no	termcap.h: No such file or directory
mad	no	no	unrecognized command line option "-fforce-mem"
tiff2bw	no	no	Test for tiff-v3.5.4 only; no compilation
tiff2rgba	no	no	Test for tiff-v3.5.4 only; no compilation
tiff-data	no	no	Data for tiff-v3.5.4 only; no compilation
tiffdither	no	no	Test for tiff-v3.5.4 only; no compilation
tiffmedian	no	no	Test for tiff-v3.5.4 only; no compilation
tiff-v3.5.4	no	no	tif_luv.c: undefined reference to 'log'
typeset	no	yes	None
network			
dijkstra	yes	yes	None
patricia	no	yes	Outputs differed, but that may be normal
office			
ghostscript	no	no	macro "dprintf" passed 3 arguments, but takes just 1
ispell	no	no	correct.c: conflicting types for 'getline'
rsynth	no	no	Could not configure. Need to specify system type.
sphinx	no	no	blk_cdcn_norm.c: invalid storage class for function 'block_actual_cdcn_norm'
stringsearch	yes	yes	None
security			
blowfish	no	no	Seems to compile correctly, but segfaults when run.
pgp	no	no	make reported "nothing to be done for 'all'." during first compile attempt.
rijndael	no	no	aesxam.c: aggregate value used where an integer was expected
sha	yes	yes	None
telecomm			
adpcm	no	yes	None
CRC32	no	yes	None
FFT	no	yes	Outputs differed, but that may be normal.
gsm	no	yes	None

Table A.1: Compilation results of benchmarks

Appendix B

Full Test Results

B.1 Introduction

We include here the full dataset generated via compilation during the test of our optimization. This raw data is broken down by basic block, and includes the counts of the different categories of ARM instruction relevant to this analysis. The categories we include are single load instructions (ex: ldr), multiple load instructions (ex: ldmia), single stores (ex: str), multiple stores (ex: stmia), and all other instructions. Each Table analyzes a different source file used to compare the optimizations, and looks at the resultant assembly code when compiled without any memory optimizations (denoted "O0 none"), without moving instructions before combining into multiple memory instructions ("O3 nopreorpost"), using the default pre-register allocation optimization ("O3 pre_ra"), and using our new post-register allocation optimization ("O3 MagPs"). Total counts are also given to make com-

parison easier across files.

B.2 Data

Each Table corresponds to a single source file (ex: dijkstra_large.c); some benchmarks use multiple source files, such as the `sha` benchmark which uses sha.c and sha_driver.c.

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	2 / 10 / 10 / 10	0 / 0 / 0 / 0	5 / 3 / 3 / 3	0 / 0 / 0 / 0	7 / 11 / 11 / 11	14 / 24 / 24 / 24
BB 2 / BB 2 / BB 2 / BB 2	3 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	5 / 1 / 1 / 1
BB 3 / BB 3 / BB 3 / BB 3	4 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 6 / 6 / 6	0 / 0 / 0 / 0	2 / 5 / 5 / 5	8 / 14 / 14 / 14
BB 4 / BB 4 / BB 4 / BB 4	1 / 8 / 8 / 8	0 / 0 / 0 / 0	0 / 4 / 4 / 4	0 / 0 / 0 / 0	3 / 3 / 3 / 3	4 / 15 / 15 / 15
BB 5 / BB 5 / BB 5 / BB 5	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	1 / 3 / 3 / 3
BB 6 / BB 6 / BB 6 / BB 6	6 / 2 / 2 / 2	0 / 0 / 0 / 0	12 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 0 / 0 / 0	24 / 3 / 3 / 3
BB 7 / BB 7 / BB 7 / BB 7	5 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 2 / 2 / 2	11 / 4 / 4 / 4
BB 8 / BB 8 / BB 8 / BB 8	8 / 2 / 2 / 2	0 / 0 / 0 / 0	4 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 0 / 0 / 0	15 / 3 / 3 / 3
BB 9 / BB 9 / BB 9 / BB 9	2 / 3 / 3 / 3	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 3 / 3 / 3	4 / 7 / 7 / 7
BB 10 / BB 10 / BB 10 / BB 10	0 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 5 / 5 / 5	1 / 8 / 8 / 8
BB 11 / BB 11 / BB 11 / BB 11	2 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 12 / BB 12 / BB 12 / BB 12	2 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 5 / 5 / 5	0 / 0 / 0 / 0	2 / 3 / 3 / 3	4 / 11 / 11 / 11
BB 13 / BB 13 / BB 13 / BB 13	2 / 14 / 14 / 14	0 / 0 / 0 / 0	1 / 5 / 5 / 5	0 / 0 / 0 / 0	0 / 2 / 2 / 2	3 / 21 / 21 / 21
BB 14 / BB 14 / BB 14 / BB 14	2 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	4 / 3 / 3 / 3
BB 15 / BB 15 / BB 15 / BB 15	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	4 / 1 / 1 / 1
BB 16 / BB 16 / BB 16 / BB 16	0 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 3 / 3 / 3	1 / 8 / 8 / 8
BB 17 / BB 17 / BB 17 / BB 17	3 / 0 / 0 / 0	0 / 0 / 0 / 0	8 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 1 / 1 / 1	15 / 1 / 1 / 1
BB 18 / BB 18 / BB 18 / BB 18	15 / 2 / 2 / 2	0 / 0 / 0 / 0	6 / 3 / 3 / 3	0 / 1 / 1 / 1	2 / 5 / 5 / 5	23 / 11 / 11 / 11
BB 19 / BB 19 / BB 19 / BB 19	1 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 3 / 3 / 3	0 / 0 / 0 / 0	3 / 2 / 2 / 2	4 / 8 / 8 / 8
BB 20 / BB 20 / BB 20 / BB 20	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	1 / 3 / 3 / 3
BB 21 / BB 21 / BB 21 / BB 21	2 / 0 / 0 / 0	0 / 1 / 1 / 1	2 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	5 / 3 / 3 / 3
BB 22 / BB 22 / BB 22 / BB 22	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 3 / 3 / 3
BB 23 / BB 23 / BB 23 / BB 23	0 / 8 / 8 / 8	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 7 / 7 / 7	1 / 17 / 17 / 17
BB 24 / BB 24 / BB 24 / BB 24	1 / 1 / 1 / 1	0 / 0 / 0 / 0	7 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 5 / 5 / 5	12 / 6 / 6 / 6
BB 25 / BB 25 / BB 25 / BB 25	5 / 3 / 3 / 3	0 / 0 / 0 / 0	3 / 1 / 1 / 1	0 / 0 / 0 / 0	4 / 3 / 3 / 3	12 / 7 / 7 / 7
BB 26 / BB 26 / BB 26 / BB 26	2 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	4 / 4 / 4 / 4
BB 27 / BB 27 / BB 27 / BB 27	2 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 3 / 3 / 3	4 / 7 / 7 / 7
BB 28 / BB 28 / BB 28 / BB 28	1 / 9 / 9 / 9	0 / 0 / 0 / 0	1 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 4 / 4 / 4	4 / 15 / 15 / 15
BB 29 / BB 29 / BB 29 / BB 29	4 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	6 / 1 / 1 / 1	12 / 2 / 2 / 2

BB 30 / BB 30 / BB 30 / BB 30	4 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	8 / 4 / 4 / 4
BB 31 / BB 31 / BB 31 / BB 31	7 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	6 / 3 / 3 / 3	14 / 3 / 3 / 3
BB 32 / BB 32 / BB 32 / BB 32	4 / 5 / 5 / 5	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 6 / 6 / 6	8 / 11 / 11 / 11
BB 33 / BB 33 / BB 33 / BB 33	8 / 1 / 1 / 1	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	11 / 3 / 3 / 3
BB 34 / BB 34 / BB 34 / BB 34	15 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 1 / 1 / 1	22 / 1 / 1 / 1
BB 35 / BB 35 / BB 35 / BB 35	2 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 2 / 2 / 2	0 / 1 / 1 / 1	1 / 3 / 3 / 3	4 / 7 / 7 / 7
BB 36 / BB 36 / BB 36 / BB 36	2 / 5 / 5 / 5	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 6 / 6 / 6	4 / 11 / 11 / 11
BB 37 / BB 37 / BB 37 / BB 37	0 / 8 / 8 / 8	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 7 / 7 / 7	3 / 17 / 17 / 17
BB 38 / BB 38 / BB 38 / BB 38	7 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 1 / 1 / 1	16 / 2 / 2 / 2
BB 39 / BB 39 / BB 39 / BB 39	3 / 3 / 3 / 3	0 / 1 / 1 / 1	0 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 5 / 5 / 5	6 / 11 / 11 / 11
BB 40 / BB 40 / BB 40 / BB 40	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	1 / 3 / 3 / 3
BB 41 / BB 41 / BB 41 / BB 41	1 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 1 / 1 / 1	0 / 0 / 0 / 0	4 / 1 / 1 / 1	10 / 3 / 3 / 3
BB 42 / BB 42 / BB 42 / BB 42	7 / 1 / 1 / 1	0 / 0 / 0 / 0	4 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 2 / 2 / 2	16 / 3 / 3 / 3
BB 43 / BB 43 / BB 43 / BB 43	6 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 4 / 4 / 4	14 / 5 / 5 / 5
BB 44 / BB 44 / BB 44 / BB 44	0 / 5 / 5 / 5	0 / 0 / 0 / 0	1 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 8 / 8 / 8	3 / 15 / 15 / 15
BB 45 / BB 45 / BB 45 / BB 45	7 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 2 / 2 / 2	15 / 4 / 4 / 4
BB 46 / BB 46 / BB 46 / BB 46	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 47 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	3 / - / - / -
BB 48 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 49 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	3 / - / - / -	5 / - / - / -
BB 50 / - / - / -	6 / - / - / -	0 / - / - / -	6 / - / - / -	0 / - / - / -	9 / - / - / -	21 / - / - / -
BB 51 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 52 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	2 / - / - / -
Total / Total / Total / Total	160 / 123 / 123 / 123	0 / 3 / 3 / 3	92 / 53 / 53 / 53	0 / 2 / 2 / 2	149 / 138 / 138 / 138	401 / 319 / 319 / 319

Table B.1: Categories of instructions in dijkstra_large by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	11 / 8 / 8 / 8	0 / 0 / 0 / 0	13 / 4 / 4 / 4	0 / 2 / 2 / 2	8 / 6 / 6 / 6	32 / 20 / 20 / 20
BB 2 / BB 2 / BB 2 / BB 2	4 / 3 / 3 / 3	0 / 1 / 1 / 1	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 5 / 5 / 5	7 / 10 / 10 / 10
BB 3 / BB 3 / BB 3 / BB 3	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 4 / BB 4 / BB 4 / BB 4	0 / 2 / 2 / 2	0 / 1 / 1 / 1	1 / 3 / 3 / 3	0 / 0 / 0 / 0	1 / 2 / 2 / 2	2 / 8 / 8 / 8
BB 5 / BB 5 / BB 5 / BB 5	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 3 / 3 / 3	0 / 1 / 1 / 1	1 / 9 / 9 / 9	3 / 14 / 14 / 14
BB 6 / BB 6 / BB 6 / BB 6	0 / 7 / 7 / 7	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 8 / 8 / 8	2 / 16 / 16 / 16
BB 7 / BB 7 / BB 7 / BB 7	1 / 11 / 11 / 11	0 / 0 / 0 / 0	2 / 6 / 6 / 6	0 / 0 / 0 / 0	0 / 36 / 36 / 36	3 / 53 / 53 / 53
BB 8 / BB 8 / BB 8 / BB 8	3 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 5 / 5 / 5	7 / 6 / 6 / 6
BB 9 / BB 9 / BB 9 / BB 9	1 / 1 / 1 / 1	0 / 0 / 0 / 0	8 / 0 / 0 / 0	0 / 0 / 0 / 0	10 / 5 / 5 / 5	19 / 6 / 6 / 6
BB 10 / BB 10 / BB 10 / BB 10	3 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 5 / 5 / 5	9 / 6 / 6 / 6
BB 11 / BB 11 / BB 11 / BB 11	6 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	6 / 2 / 2 / 2	13 / 3 / 3 / 3
BB 12 / BB 12 / BB 12 / BB 12	28 / 5 / 5 / 5	0 / 0 / 0 / 0	21 / 0 / 0 / 0	0 / 0 / 0 / 0	20 / 7 / 7 / 7	69 / 12 / 12 / 12
BB 13 / BB 13 / BB 13 / BB 13	2 / 3 / 3 / 3	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 5 / 5 / 5	6 / 9 / 9 / 9
BB 14 / BB 14 / BB 14 / BB 14	2 / 3 / 3 / 2	0 / 1 / 1 / 0	0 / 3 / 3 / 1	0 / 0 / 0 / 0	4 / 6 / 6 / 2	6 / 13 / 13 / 5
BB 15 / BB 15 / BB 15 / BB 15	2 / 3 / 3 / 2	0 / 0 / 0 / 1	0 / 0 / 0 / 2	0 / 0 / 0 / 0	4 / 5 / 5 / 6	6 / 8 / 8 / 11
BB 16 / - / - / BB 16	1 / - / - / 3	0 / - / - / 0	0 / - / - / 0	0 / - / - / 0	4 / - / - / 5	5 / - / - / 8
BB 17 / - / - / -	4 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	6 / - / - / -	12 / - / - / -
BB 18 / - / - / -	6 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	5 / - / - / -	13 / - / - / -
BB 19 / - / - / -	2 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	4 / - / - / -
BB 20 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	2 / - / - / -	4 / - / - / -
BB 21 / - / - / -	4 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	4 / - / - / -	8 / - / - / -
Total / Total / Total / Total	81 / 50 / 50 / 51	0 / 4 / 4 / 4	56 / 21 / 21 / 21	0 / 3 / 3 / 3	96 / 107 / 107 / 109	233 / 185 / 185 / 188

Table B.2: Categories of instructions in qsort_large by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 2 / 2 / 2	0 / 0 / 0 / 0	4 / 3 / 3 / 3	8 / 5 / 5 / 5
BB 2 / BB 2 / BB 2 / BB 2	4 / 4 / 4 / 4	0 / 0 / 0 / 0	2 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	9 / 8 / 8 / 8
BB 3 / BB 3 / BB 3 / BB 3	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 4 / BB 4 / BB 4 / BB 4	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 5 / BB 5 / BB 5 / BB 5	6 / 6 / 6 / 6	0 / 0 / 0 / 0	2 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 6 / 6 / 6	14 / 13 / 13 / 13
BB 6 / BB 6 / BB 6 / BB 6	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 7 / BB 7 / BB 7 / BB 7	10 / 10 / 10 / 10	0 / 0 / 0 / 0	6 / 5 / 5 / 5	0 / 0 / 0 / 0	2 / 2 / 2 / 2	18 / 17 / 17 / 17
BB 8 / BB 8 / BB 8 / BB 8	14 / 14 / 14 / 14	0 / 0 / 0 / 0	7 / 6 / 6 / 6	0 / 0 / 0 / 0	10 / 10 / 10 / 10	31 / 30 / 30 / 30
BB 9 / BB 9 / BB 9 / BB 9	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 10 / BB 10 / BB 10 / BB 10	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 11 / BB 11 / BB 11 / BB 11	14 / 14 / 14 / 14	0 / 0 / 0 / 0	7 / 6 / 6 / 6	0 / 0 / 0 / 0	9 / 9 / 9 / 9	30 / 29 / 29 / 29
BB 12 / BB 12 / BB 12 / BB 12	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 13 / BB 13 / BB 13 / BB 13	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 14 / BB 14 / BB 14 / BB 14	14 / 14 / 14 / 14	0 / 0 / 0 / 0	7 / 6 / 6 / 6	0 / 0 / 0 / 0	11 / 11 / 11 / 11	32 / 31 / 31 / 31
BB 15 / BB 15 / BB 15 / BB 15	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 16 / BB 16 / BB 16 / BB 16	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 17 / BB 17 / BB 17 / BB 17	14 / 14 / 14 / 14	0 / 0 / 0 / 0	7 / 6 / 6 / 6	0 / 0 / 0 / 0	9 / 9 / 9 / 9	30 / 29 / 29 / 29
BB 18 / BB 18 / BB 18 / BB 18	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 4 / 4 / 4
BB 19 / BB 19 / BB 19 / BB 19	15 / 16 / 16 / 16	0 / 0 / 0 / 0	5 / 5 / 5 / 5	0 / 0 / 0 / 0	5 / 7 / 7 / 7	25 / 28 / 28 / 28
BB 20 / BB 20 / BB 20 / BB 20	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	4 / 1 / 1 / 1
BB 21 / BB 21 / BB 21 / BB 21	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 4 / 4 / 4	0 / 0 / 0 / 0	1 / 4 / 4 / 4	1 / 9 / 9 / 9
BB 22 / BB 22 / BB 22 / BB 22	1 / 17 / 17 / 17	0 / 0 / 0 / 0	7 / 9 / 9 / 9	0 / 0 / 0 / 0	4 / 2 / 2 / 2	12 / 28 / 28 / 28
BB 23 / BB 23 / BB 23 / BB 23	17 / 2 / 2 / 2	0 / 0 / 0 / 0	10 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 5 / 5 / 5	29 / 8 / 8 / 8
BB 24 / BB 24 / BB 24 / BB 24	2 / 11 / 11 / 11	0 / 0 / 0 / 0	0 / 8 / 8 / 8	0 / 0 / 0 / 0	2 / 3 / 3 / 4	4 / 22 / 22 / 23
BB 25 / BB 25 / BB 25 / BB 25	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 26 / BB 26 / BB 26 / BB 26	11 / 10 / 10 / 10	0 / 0 / 0 / 0	9 / 7 / 7 / 7	0 / 0 / 0 / 0	2 / 7 / 7 / 7	22 / 24 / 24 / 24
BB 27 / BB 27 / BB 27 / BB 27	0 / 6 / 6 / 6	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 9 / 9 / 9	2 / 17 / 17 / 17
BB 28 / BB 28 / BB 28 / BB 28	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	1 / 3 / 3 / 3
BB 29 / BB 29 / BB 29 / BB 29	2 / 4 / 4 / 4	0 / 0 / 0 / 0	7 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 5 / 5 / 5	14 / 9 / 9 / 9

BB 30 / BB 30 / BB 30 / BB 30	2 / 6 / 6 / 6	0 / 0 / 0 / 0	1 / 7 / 7 / 7	0 / 0 / 0 / 0	1 / 8 / 8 / 8	4 / 21 / 21 / 21
BB 31 / BB 31 / BB 31 / BB 31	6 / 5 / 5 / 5	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 11 / 11 / 11	11 / 16 / 16 / 16
BB 32 / BB 32 / BB 32 / BB 32	7 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 0 / 0 / 0	0 / 0 / 0 / 0	8 / 2 / 2 / 2	18 / 4 / 4 / 4
BB 33 / BB 33 / BB 33 / BB 33	1 / 7 / 7 / 7	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 10 / 10 / 10	3 / 19 / 19 / 19
BB 34 / BB 34 / BB 34 / BB 34	3 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 1 / 1 / 1	2 / 5 / 5 / 5	5 / 9 / 9 / 9
BB 35 / BB 35 / BB 35 / BB 35	1 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 3 / 3 / 3	5 / 5 / 5 / 5
BB 36 / BB 36 / BB 36 / BB 36	6 / 1 / 1 / 1	0 / 0 / 0 / 0	8 / 1 / 1 / 1	0 / 0 / 0 / 0	9 / 7 / 7 / 7	23 / 9 / 9 / 9
BB 37 / BB 37 / BB 37 / BB 37	7 / 1 / 1 / 1	0 / 1 / 1 / 1	2 / 0 / 0 / 0	0 / 0 / 0 / 0	14 / 5 / 5 / 5	23 / 7 / 7 / 7
BB 38 / BB 38 / BB 38 / BB 38	2 / 3 / 3 / 3	0 / 1 / 1 / 1	0 / 4 / 4 / 4	0 / 0 / 0 / 0	5 / 4 / 4 / 4	7 / 12 / 12 / 12
BB 39 / - / - / -	6 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	4 / - / - / -	12 / - / - / -
BB 40 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	4 / - / - / -	5 / - / - / -
BB 41 / - / - / -	3 / - / - / -	0 / - / - / -	8 / - / - / -	0 / - / - / -	6 / - / - / -	17 / - / - / -
BB 42 / - / - / -	2 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	3 / - / - / -	5 / - / - / -
BB 43 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	7 / - / - / -	9 / - / - / -
BB 44 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 45 / - / - / -	2 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	5 / - / - / -	7 / - / - / -
BB 46 / - / - / -	9 / - / - / -	0 / - / - / -	9 / - / - / -	0 / - / - / -	4 / - / - / -	22 / - / - / -
BB 47 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	3 / - / - / -	4 / - / - / -
Total / Total / Total / Total	192 / 178 / 178 / 178	0 / 2 / 2 / 2	120 / 91 / 91 / 91	0 / 1 / 1 / 1	190 / 174 / 174 / 175	502 / 446 / 446 / 447

Table B.3: Categories of instructions in sha by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	1 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 2 / 2 / 2	0 / 1 / 1 / 1	6 / 5 / 5 / 5	13 / 9 / 9 / 9
BB 2 / BB 2 / BB 2 / BB 2	4 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 1 / 1 / 1	12 / 3 / 3 / 3
BB 3 / BB 3 / BB 3 / BB 3	0 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 6 / 6 / 6	1 / 9 / 9 / 9
BB 4 / BB 4 / BB 4 / BB 4	3 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 2 / 2 / 2	0 / 0 / 0 / 0	4 / 5 / 5 / 5	9 / 9 / 9 / 9
BB 5 / BB 5 / BB 5 / BB 5	3 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 3 / 3 / 3	6 / 5 / 5 / 5
BB 6 / BB 6 / BB 6 / BB 6	3 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	4 / 6 / 6 / 6	9 / 8 / 8 / 8
BB 7 / BB 7 / BB 7 / BB 7	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	5 / 4 / 4 / 4
BB 8 / BB 8 / BB 8 / BB 8	0 / 1 / 1 / 1	0 / 1 / 1 / 1	2 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 2 / 2 / 2	3 / 6 / 6 / 6
BB 9 / - / - / -	3 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	3 / - / - / -	6 / - / - / -
Total / Total / Total / Total	18 / 13 / 13 / 13	0 / 1 / 1 / 1	17 / 8 / 8 / 8	0 / 1 / 1 / 1	29 / 30 / 30 / 30	64 / 53 / 53 / 53

Table B.4: Categories of instructions in sha_driver by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	5 / 4 / 4 / 4	0 / 0 / 0 / 0	8 / 4 / 4 / 4	0 / 1 / 1 / 1	5 / 6 / 6 / 6	18 / 15 / 15 / 15
BB 2 / BB 2 / BB 2 / BB 2	5 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 1 / 1 / 1	9 / 5 / 5 / 5
BB 3 / BB 3 / BB 3 / BB 3	8 / 6 / 6 / 6	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	10 / 8 / 8 / 8
BB 4 / BB 4 / BB 4 / BB 4	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	3 / 1 / 1 / 1
BB 5 / BB 5 / BB 5 / BB 5	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 3 / 3 / 3	3 / 5 / 5 / 5
BB 6 / BB 6 / BB 6 / BB 6	1 / 8 / 8 / 8	0 / 0 / 0 / 0	0 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 6 / 6 / 6	3 / 17 / 17 / 17
BB 7 / BB 7 / BB 7 / BB 7	5 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	8 / 3 / 3 / 3
BB 8 / BB 8 / BB 8 / BB 8	3 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	6 / 4 / 4 / 4
BB 9 / BB 9 / BB 9 / BB 9	2 / 10 / 8 / 8	0 / 0 / 1 / 1	1 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 5 / 5 / 5	5 / 16 / 15 / 15
BB 10 / BB 10 / BB 10 / BB 10	1 / 2 / 2 / 2	0 / 1 / 1 / 1	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 4 / 4 / 4	3 / 8 / 8 / 8
BB 11 / BB 11 / BB 11 / BB 11	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 12 / BB 12 / BB 12 / BB 12	3 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 3 / 3 / 3	0 / 1 / 1 / 1	2 / 5 / 5 / 5	7 / 11 / 11 / 11
BB 13 / BB 13 / BB 13 / BB 13	10 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	13 / 8 / 8 / 8
BB 14 / BB 14 / BB 14 / BB 14	4 / 4 / 4 / 4	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 3 / 3 / 3	7 / 8 / 8 / 8
BB 15 / BB 15 / BB 15 / BB 15	1 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 3 / 3 / 3	3 / 5 / 5 / 5
BB 16 / BB 16 / BB 16 / BB 16	3 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 3 / 3 / 3	0 / 0 / 0 / 0	3 / 5 / 5 / 5	6 / 12 / 12 / 12
BB 17 / BB 17 / BB 17 / BB 17	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	4 / 4 / 4 / 4
BB 18 / BB 18 / BB 18 / BB 18	0 / 7 / 7 / 7	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	1 / 9 / 9 / 9
BB 19 / BB 19 / BB 19 / BB 19	2 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 2 / 2 / 2	12 / 3 / 3 / 3
BB 20 / BB 20 / BB 20 / BB 20	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 2 / 2 / 2
BB 21 / BB 21 / BB 21 / BB 21	2 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 3 / 3 / 3	4 / 6 / 6 / 6
BB 22 / BB 22 / BB 22 / BB 22	5 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	9 / 1 / 1 / 1
BB 23 / BB 23 / BB 23 / BB 23	2 / 2 / 2 / 2	0 / 1 / 1 / 1	0 / 2 / 2 / 2	0 / 0 / 0 / 0	5 / 1 / 1 / 1	7 / 6 / 6 / 6
BB 24 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 25 / - / - / -	5 / - / - / -	0 / - / - / -	3 / - / - / -	0 / - / - / -	5 / - / - / -	13 / - / - / -
BB 26 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	3 / - / - / -	5 / - / - / -
BB 27 / - / - / -	9 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	11 / - / - / -
BB 28 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 29 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	3 / - / - / -

BB 30 / - / - / -	3 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	3 / - / - / -	7 / - / - / -
BB 31 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 32 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	1 / - / - / -
BB 33 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -
BB 34 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
Total / Total / Total / Total	88 / 67 / 65 / 65	0 / 2 / 3 / 3	35 / 24 / 24 / 24	0 / 2 / 2 / 2	78 / 63 / 63 / 63	201 / 158 / 157 / 157

Table B.5: Categories of instructions in bmhasrch by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	6 / 2 / 2 / 2	0 / 0 / 0 / 0	8 / 3 / 3 / 3	0 / 1 / 1 / 1	6 / 6 / 6 / 6	20 / 12 / 12 / 12
BB 2 / BB 2 / BB 2 / BB 2	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 3 / 3 / 3	2 / 4 / 4 / 4
BB 3 / BB 3 / BB 3 / BB 3	1 / 4 / 4 / 4	0 / 1 / 1 / 1	2 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	6 / 8 / 8 / 8
BB 4 / BB 4 / BB 4 / BB 4	8 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	13 / 5 / 5 / 5
BB 5 / BB 5 / BB 5 / BB 5	3 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	5 / 4 / 4 / 4
BB 6 / BB 6 / BB 6 / BB 6	0 / 3 / 3 / 3	0 / 0 / 0 / 0	1 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 6 / 6 / 6
BB 7 / BB 7 / BB 7 / BB 7	5 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	8 / 3 / 3 / 3
BB 8 / BB 8 / BB 8 / BB 8	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 9 / BB 9 / BB 9 / BB 9	0 / 10 / 8 / 8	0 / 0 / 1 / 1	1 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 6 / 6 / 6	3 / 18 / 17 / 17
BB 10 / BB 10 / BB 10 / BB 10	15 / 2 / 2 / 2	0 / 0 / 0 / 0	5 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 3 / 3 / 3	26 / 6 / 6 / 6
BB 11 / BB 11 / BB 11 / BB 11	3 / 3 / 3 / 3	0 / 1 / 1 / 1	0 / 4 / 4 / 4	0 / 0 / 0 / 0	3 / 6 / 6 / 6	6 / 14 / 14 / 14
BB 12 / BB 12 / BB 12 / BB 12	12 / 7 / 7 / 7	0 / 0 / 0 / 0	8 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 4 / 4 / 4	26 / 12 / 12 / 12
BB 13 / BB 13 / BB 13 / BB 13	5 / 2 / 2 / 2	0 / 1 / 1 / 1	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 4 / 4 / 4	7 / 8 / 8 / 8
BB 14 / BB 14 / BB 14 / BB 14	4 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	7 / 2 / 2 / 2
BB 15 / BB 15 / BB 15 / BB 15	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 16 / BB 16 / BB 16 / BB 16	3 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 3 / 3 / 3	0 / 1 / 1 / 1	3 / 5 / 5 / 5	6 / 11 / 11 / 11
BB 17 / BB 17 / BB 17 / BB 17	1 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	4 / 3 / 3 / 3
BB 18 / BB 18 / BB 18 / BB 18	0 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 3 / 3 / 3	1 / 7 / 7 / 7
BB 19 / BB 19 / BB 19 / BB 19	2 / 4 / 4 / 4	0 / 0 / 0 / 0	6 / 1 / 1 / 1	0 / 0 / 0 / 0	6 / 3 / 3 / 3	14 / 8 / 8 / 8
BB 20 / BB 20 / BB 20 / BB 20	0 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 3 / 3 / 3	3 / 5 / 5 / 5
BB 21 / BB 21 / BB 21 / BB 21	2 / 4 / 4 / 4	0 / 0 / 0 / 0	1 / 3 / 3 / 3	0 / 0 / 0 / 0	1 / 5 / 5 / 5	4 / 12 / 12 / 12
BB 22 / BB 22 / BB 22 / BB 22	5 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	9 / 4 / 4 / 4
BB 23 / BB 23 / BB 23 / BB 23	2 / 6 / 6 / 6	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 3 / 3 / 3	7 / 9 / 9 / 9
BB 24 / BB 24 / BB 24 / BB 24	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 3 / 3 / 3
BB 25 / BB 25 / BB 25 / BB 25	5 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 1 / 1 / 1	13 / 3 / 3 / 3
BB 26 / BB 26 / BB 26 / BB 26	1 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	5 / 6 / 6 / 6
BB 27 / BB 27 / BB 27 / BB 27	7 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 1 / 1 / 1	10 / 2 / 2 / 2
BB 28 / BB 28 / BB 28 / BB 28	1 / 2 / 2 / 2	0 / 1 / 1 / 1	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 5 / 5 / 5
BB 29 / BB 29 / BB 29 / BB 29	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 1 / 1 / 1	3 / 1 / 1 / 1

BB 30 / BB 30 / BB 30 / BB 30	3 / 3 / 3 / 3	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 2 / 2 / 2	7 / 6 / 6 / 6
BB 31 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 32 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	1 / - / - / -
BB 33 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -
BB 34 / - / - / -	2 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	3 / - / - / -	5 / - / - / -
BB 35 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	1 / - / - / -
BB 36 / - / - / -	2 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	5 / - / - / -
BB 37 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
Total / Total / Total / Total	103 / 69 / 67 / 67	0 / 4 / 5 / 5	51 / 32 / 32 / 32	0 / 2 / 2 / 2	96 / 83 / 83 / 83	250 / 190 / 189 / 189

Table B.6: Categories of instructions in bmhisrch by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	5 / 3 / 3 / 3	0 / 0 / 0 / 0	8 / 4 / 4 / 4	0 / 1 / 1 / 1	5 / 4 / 4 / 4	18 / 12 / 12 / 12
BB 2 / BB 2 / BB 2 / BB 2	5 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 1 / 1 / 1	8 / 6 / 6 / 6
BB 3 / BB 3 / BB 3 / BB 3	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 3 / 3 / 3
BB 4 / BB 4 / BB 4 / BB 4	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 5 / BB 5 / BB 5 / BB 5	8 / 5 / 3 / 3	0 / 0 / 1 / 1	2 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	13 / 9 / 8 / 8
BB 6 / BB 6 / BB 6 / BB 6	3 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 2 / 2 / 2	5 / 5 / 5 / 5
BB 7 / BB 7 / BB 7 / BB 7	8 / 2 / 2 / 2	0 / 1 / 1 / 1	4 / 3 / 3 / 3	0 / 0 / 0 / 0	5 / 5 / 5 / 5	17 / 11 / 11 / 11
BB 8 / BB 8 / BB 8 / BB 8	5 / 7 / 7 / 7	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 4 / 4 / 4	7 / 12 / 12 / 12
BB 9 / BB 9 / BB 9 / BB 9	4 / 2 / 2 / 2	0 / 1 / 1 / 1	1 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 4 / 4 / 4	7 / 8 / 8 / 8
BB 10 / BB 10 / BB 10 / BB 10	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 1 / 1 / 1	3 / 1 / 1 / 1
BB 11 / BB 11 / BB 11 / BB 11	3 / 2 / 2 / 2	0 / 0 / 0 / 0	0 / 4 / 4 / 4	0 / 0 / 0 / 0	3 / 5 / 5 / 5	6 / 11 / 11 / 11
BB 12 / BB 12 / BB 12 / BB 12	1 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	4 / 7 / 7 / 7
BB 13 / BB 13 / BB 13 / BB 13	0 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 3 / 3 / 3	1 / 8 / 8 / 8
BB 14 / BB 14 / BB 14 / BB 14	2 / 2 / 2 / 2	0 / 0 / 0 / 0	5 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 3 / 3 / 3	12 / 5 / 5 / 5
BB 15 / BB 15 / BB 15 / BB 15	0 / 4 / 4 / 4	0 / 0 / 0 / 0	1 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 5 / 5 / 5	3 / 12 / 12 / 12
BB 16 / BB 16 / BB 16 / BB 16	2 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 2 / 2 / 2	4 / 4 / 4 / 4
BB 17 / BB 17 / BB 17 / BB 17	5 / 5 / 5 / 5	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 2 / 2 / 2	9 / 7 / 7 / 7
BB 18 / BB 18 / BB 18 / BB 18	2 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 2 / 2 / 2	7 / 3 / 3 / 3
BB 19 / BB 19 / BB 19 / BB 19	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 2 / 2 / 2
BB 20 / BB 20 / BB 20 / BB 20	5 / 2 / 2 / 2	0 / 0 / 0 / 0	3 / 1 / 1 / 1	0 / 0 / 0 / 0	5 / 3 / 3 / 3	13 / 6 / 6 / 6
BB 21 / BB 21 / BB 21 / BB 21	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1	5 / 1 / 1 / 1
BB 22 / BB 22 / BB 22 / BB 22	6 / 3 / 3 / 3	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 2 / 2 / 2	8 / 7 / 7 / 7
BB 23 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 24 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	3 / - / - / -
BB 25 / - / - / -	3 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	3 / - / - / -	7 / - / - / -
BB 26 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -
BB 27 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	1 / - / - / -	1 / - / - / -
BB 28 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -
BB 29 / - / - / -	1 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	3 / - / - / -

Total / Total / Total / Total	74 / 53 / 51 / 51	0 / 2 / 3 / 3	36 / 26 / 26 / 26	0 / 1 / 1 / 1	71 / 60 / 60 / 60	181 / 142 / 141 / 141

Table B.7: Categories of instructions in bmhsrch by basic block

O0 none/O3 nopreorpost/O3 pre_ra/O3 MagPs	LOAD_SINGLE	LOAD_MULTIPLE	STORE_SINGLE	STORE_MULTIPLE	OTHER	TOTAL
BB 1 / BB 1 / BB 1 / BB 1	3 / 2 / 2 / 2	0 / 0 / 0 / 0	7 / 4 / 4 / 4	0 / 0 / 0 / 0	5 / 4 / 4 / 4	15 / 10 / 10 / 10
BB 2 / BB 2 / BB 2 / BB 2	5 / 3 / 3 / 3	0 / 0 / 0 / 0	2 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 1 / 1 / 1	8 / 6 / 6 / 6
BB 3 / BB 3 / BB 3 / BB 3	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 3 / 3 / 3
BB 4 / BB 4 / BB 4 / BB 4	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 2 / 2 / 2	3 / 2 / 2 / 2
BB 5 / BB 5 / BB 5 / BB 5	7 / 3 / 3 / 3	0 / 1 / 1 / 1	2 / 1 / 1 / 1	0 / 0 / 0 / 0	3 / 3 / 3 / 3	12 / 8 / 8 / 8
BB 6 / BB 6 / BB 6 / BB 6	3 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 4 / 4 / 4	5 / 10 / 10 / 10
BB 7 / BB 7 / BB 7 / BB 7	2 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 1 / 1 / 1
BB 8 / BB 8 / BB 8 / BB 8	1 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 3 / 3 / 3	0 / 1 / 1 / 1	3 / 4 / 4 / 4	4 / 12 / 12 / 12
BB 9 / BB 9 / BB 9 / BB 9	0 / 2 / 2 / 2	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	1 / 4 / 4 / 4
BB 10 / BB 10 / BB 10 / BB 10	4 / 2 / 2 / 2	0 / 0 / 0 / 0	6 / 0 / 0 / 0	0 / 0 / 0 / 0	5 / 2 / 2 / 2	15 / 4 / 4 / 4
BB 11 / BB 11 / BB 11 / BB 11	0 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 2 / 2 / 2	1 / 7 / 7 / 7
BB 12 / BB 12 / BB 12 / BB 12	2 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 2 / 2 / 2	4 / 3 / 3 / 3
BB 13 / BB 13 / BB 13 / BB 13	2 / 4 / 4 / 4	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 6 / 6 / 6	4 / 11 / 11 / 11
BB 14 / BB 14 / BB 14 / BB 14	5 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 1 / 1 / 1	8 / 3 / 3 / 3
BB 15 / BB 15 / BB 15 / BB 15	1 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 1 / 1 / 1	0 / 0 / 0 / 0	2 / 1 / 1 / 1	3 / 3 / 3 / 3
BB 16 / BB 16 / BB 16 / BB 16	7 / 2 / 2 / 2	0 / 0 / 0 / 0	2 / 0 / 0 / 0	0 / 0 / 0 / 0	6 / 2 / 2 / 2	15 / 4 / 4 / 4
BB 17 / BB 17 / BB 17 / BB 17	1 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	3 / 2 / 2 / 2
BB 18 / BB 18 / BB 18 / BB 18	1 / 2 / 2 / 2	0 / 1 / 1 / 1	1 / 1 / 1 / 1	0 / 0 / 0 / 0	1 / 1 / 1 / 1	3 / 5 / 5 / 5
BB 19 / BB 19 / BB 19 / BB 19	2 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	2 / 1 / 1 / 1	4 / 1 / 1 / 1
BB 20 / BB 20 / BB 20 / BB 20	0 / 6 / 6 / 6	0 / 0 / 0 / 0	1 / 0 / 0 / 0	0 / 1 / 1 / 1	1 / 15 / 15 / 15	2 / 22 / 22 / 22
BB 21 / BB 21 / BB 21 / BB 21	1 / 4 / 4 / 10	0 / 0 / 0 / 0	1 / 0 / 0 / 3	0 / 0 / 0 / 0	0 / 5 / 5 / 10	2 / 9 / 9 / 23
BB 22 / BB 22 / BB 22 / BB 22	2 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	3 / 3 / 3 / 3	5 / 4 / 4 / 4
BB 23 / BB 23 / BB 23 / BB 23	0 / 1 / 1 / 1	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	1 / 3 / 3 / 3	1 / 4 / 4 / 4
BB 24 / BB 24 / BB 24 / BB 24	5 / 9 / 9 / 2	0 / 1 / 1 / 0	6 / 6 / 6 / 1	0 / 0 / 0 / 0	14 / 12 / 12 / 2	25 / 28 / 28 / 5
BB 25 / - / - / BB 25	7 / - / - / 2	0 / - / - / 1	3 / - / - / 2	0 / - / - / 0	8 / - / - / 7	18 / - / - / 12
BB 26 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	4 / - / - / -
BB 27 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	1 / - / - / -	3 / - / - / -
BB 28 / - / - / -	6 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	6 / - / - / -	13 / - / - / -
BB 29 / - / - / -	2 / - / - / -	0 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	5 / - / - / -

BB 30 / - / - / -	1 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	3 / - / - / -	6 / - / - / -
BB 31 / - / - / -	2 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	4 / - / - / -	6 / - / - / -
BB 32 / - / - / -	0 / - / - / -	0 / - / - / -	2 / - / - / -	0 / - / - / -	2 / - / - / -	4 / - / - / -
BB 33 / - / - / -	3 / - / - / -	0 / - / - / -	0 / - / - / -	0 / - / - / -	5 / - / - / -	8 / - / - / -
Total / Total / Total / Total	78 / 57 / 57 / 58	0 / 3 / 3 / 3	45 / 26 / 26 / 26	0 / 2 / 2 / 2	93 / 78 / 78 / 80	216 / 166 / 166 / 169

Table B.8: Categories of instructions in pbmsrch_large by basic block

Appendix C

Source Code

C.1 Introduction

We include all source code used in the post-register allocation optimization we developed below. Because our optimization was added as a helper to an existing optimization (`ARMLoadStoreOpt`), we elected to simply add our source code into the same file (`ARMLoadStoreOptimizer.cpp`) in a separate namespace. All of our final code was implemented in C++ in this file.

C.2 `ARMLoadStoreOptimizer.cpp`

The implementation challenges have been discussed in the implementation Chapter, however we wanted to include the full and rather lengthy source code for our optimization. We will be including both the optimization code, and also the modi-

fications we made to implement the optimization in the existing post-register allocation optimization in the `ARMLoadStoreOpt` pass.

The optimization code in Listing C.1 also includes code for logging what the pass is doing for debug purposes. Some of this logging code may be commented out, but was left in the code to aid debug of future issues.

Listing C.1: Optimization Code

```
1  namespace {
2    using namespace std;
3
4    class MemOpRecord {
5    public:
6      MachineInstr* OpLocation;
7      MachineInstr* LowerBound;
8      MachineInstr* UpperBound;
9
10     MemOpRecord() : OpLocation(NULL), LowerBound(NULL), UpperBound
           (NULL) {}
11   };
12
13   class RegisterDependencies {
14   public:
15     vector<int> use_dep;
16     vector<int> mod_dep;
17   };
18
19   struct ClusterPoint {
20     MachineInstr* insertAfter;
```

```
21      vector<MachineInstr*> instructionsToGather;
22
23      ClusterPoint() : insertAfter(NULL) {}
24    };
25
26    class MagnetPass {
27      vector<MemOpRecord> AllMemOps;
28      vector<RegisterDependencies> RegDeps;
29      vector<int> MemDeps;
30      unsigned numRegs;
31
32      void clearAllDeps();
33      int getAllMemOpsIndexOf(MachineInstr* MI);
34      vector<int> getRegsUsed(MachineInstr* MI);
35      vector<int> getRegsModified(MachineInstr* MI);
36      void endLowerBoundUsingRegsUsed(MachineInstr* MI);
37      void endLowerBoundUsingRegsModified(MachineInstr* MI);
38      void endUpperBoundUsingRegsUsed(MachineInstr* MI);
39      void endUpperBoundUsingRegsModified(MachineInstr* MI);
40      void endLowerBoundUsingMem(MachineInstr* MI);
41      void endUpperBoundUsingMem(MachineInstr* MI);
42      ClusterPoint getBestRangeOverlap(MachineBasicBlock &MBB);
43      bool allMIsContiguous(MachineBasicBlock &MBB, const
            TargetMachine &TM, ClusterPoint bestCluster);
44      void gatherAtBestRangeOverlap(MachineBasicBlock &MBB, const
            TargetMachine &TM, ClusterPoint bestCluster);
45      void printBB(MachineBasicBlock &MBB, const TargetMachine &TM,
            string s);
46      void findLowerBounds(MachineBasicBlock &MBB, const
```

```
        TargetMachine &TM);
47      void findUpperBounds(MachineBasicBlock &MBB, const
        TargetMachine &TM);
48      void initialize(MachineBasicBlock &MBB, const TargetMachine &
        TM);
49      void cleanUp();
50    public:
51      raw_os_ostream* errLog;
52      MagnetPass() {
53        //errLog = new raw_os_ostream(cerr);  // Log progress to
            stderr
54        errLog = NULL;  // Disable logging
55      }
56
57      ~MagnetPass() {
58        if (errLog)
59          delete errLog;
60      }
61
62      void runOptimization(MachineBasicBlock &MBB, const
        TargetMachine &TM);
63    };
64
65    class BBPrinter {
66    public:
67      raw_os_ostream* errLog;
68      BBPrinter(bool before) {
69        if (before) {
70          errLog = new raw_os_ostream(cout);
```

```
71        } else {
72          errLog = new raw_os_ostream(cerr);
73        }
74      }
75
76      void printBB(MachineBasicBlock &MBB, const TargetMachine &TM);
77      void printInstr(MachineBasicBlock &MBB, const TargetMachine &
            TM, int index);
78
79      ~BBPrinter() {
80        delete errLog;
81      }
82    };
83
84    void BBPrinter::printBB(MachineBasicBlock &MBB, const
        TargetMachine &TM) {
85      MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end();
86      for (; MBBI != E; ++MBBI) {
87        MachineInstr *MI = MBBI;
88        *errLog << "          " << MI << ": ";
89        MI->print(*errLog, &TM);
90      }
91      *errLog << "\n";
92      errLog->flush();
93    }
94
95    void BBPrinter::printInstr(MachineBasicBlock &MBB, const
        TargetMachine &TM, int index) {
96      MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end();
```

```
97      int i = 0;
98      for (; MBBI != E; ++MBBI) {
99        if (i == index) {
100         MachineInstr *MI = MBBI;
101         *errLog << "          " << MI << ": ";
102         MI->print(*errLog, &TM);
103       }
104       ++i;
105     }
106     if (i >= index) {
107       // *errLog << "\n";
108       errLog->flush();
109     }
110   }
111
112
113   unsigned getBaseReg(MachineInstr* MI) {
114     // Pulled from FixInvalidRegPairOp()
115     const MachineOperand &BaseOp = MI->getOperand(2);
116     return BaseOp.getReg();
117   }
118
119   int getRegNum(unsigned RegEnum) {
120     using namespace ARM;
121     switch (RegEnum) {
122       case R0: return 0;
123       case R1: return 1;
124       case R2: return 2;
125       case R3: return 3;
```

```
126         case R4: return 4;
127         case R5: return 5;
128         case R6: return 6;
129         case R7: return 7;
130         case R8: return 8;
131         case R9: return 9;
132         case R10: return 10;
133         case R11: return 11;
134         case R12: return 12;
135         case SP: return 13;
136         case LR: return 14;
137         case PC: return 15;
138         default: return -1;
139       }
140     }
141
142     // Borrowing later definitions by LLVM
143     static bool isT2i32Load2(unsigned Opc) {
144       return Opc == ARM::t2LDRi12 || Opc == ARM::t2LDRi8;
145     }
146     static bool isi32Load2(unsigned Opc) {
147       return Opc == ARM::LDR || isT2i32Load2(Opc);
148     }
149
150     bool isLoad(MachineInstr* MI) {
151       int Opcode = MI->getOpcode();
152       return isi32Load2(Opcode) || Opcode == ARM::VLDRS || Opcode ==
              ARM::VLDRD;
153     }
```

```
154
155     static bool isT2i32Store2(unsigned Opc) {
156       return Opc == ARM::t2STRi12 || Opc == ARM::t2STRi8;
157     }
158     static bool isi32Store2(unsigned Opc) {
159       return Opc == ARM::STR || isT2i32Store2(Opc);
160     }
161
162     bool isStore(MachineInstr* MI) {
163       int Opcode = MI->getOpcode();
164       return isi32Store2(Opcode) || Opcode == ARM::VSTRS || Opcode
              == ARM::VSTRD;
165     }
166
167 }
168
169 // Defined later by LLVM
170 // Only detects memory operations that this optimization CAN ACT
      UPON
171 static bool isMemoryOp(const MachineInstr *MI);
172
173 int MagnetPass::getAllMemOpsIndexOf(MachineInstr* MI) {
174   for (int i = 0; i < (int)AllMemOps.size(); ++i) {
175     if (AllMemOps[i].OpLocation == MI)
176       return i;
177   }
178   return -1;
179 }
180
```

```
181 vector<int> MagnetPass::getRegsModified(MachineInstr* MI) {
182   vector<int> regs;
183   const TargetInstrDesc &TID = MI->getDesc();
184   // If this is a branch, ALL registers are "modified" (
          LowerBounds and UpperBounds must end)
185   if (TID.isBranch() || TID.isTerminator()) {
186     for (unsigned i = 0; i < 16; ++i)
187       regs.push_back(i);
188   } else {
189     for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
190       MachineOperand &MO = MI->getOperand(i);
191       if (MO.isReg() && MO.isDef()) {
192         // Hack because getRegisterNumbering crashes
193         const int rnum = getRegNum(MO.getReg());
194         // If register is one of the 16 addressable registers
195         if (rnum >= 0) {
196           if (find(regs.begin(), regs.end(), rnum) == regs.end())
                {
197             regs.push_back(rnum);
198           }
199         }
200       }
201     }
202   }
203   return regs;
204 }
205
206 vector<int> MagnetPass::getRegsUsed(MachineInstr* MI) {
207   vector<int> regs;
```

```
208   const TargetInstrDesc &TID = MI->getDesc();
209   // If this is a branch, ALL registers are "used" (LowerBounds
          and UpperBounds must end)
210   if (TID.isBranch() || TID.isTerminator()) {
211     for (unsigned i = 0; i < 16; ++i)
212       regs.push_back(i);
213   } else {
214     for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
215       MachineOperand &MO = MI->getOperand(i);
216       if (MO.isReg()) {
217         // Hack because getRegisterNumbering crashes
218         const int rnum = getRegNum(MO.getReg());
219         // If register is one of the 16 addressable registers
220         if (rnum >= 0) {
221           if (find(regs.begin(), regs.end(), rnum) == regs.end())
                  {
222             regs.push_back(rnum);
223           }
224         }
225       }
226     }
227   }
228   return regs;
229 }
230
231 void MagnetPass::endLowerBoundUsingRegsUsed(MachineInstr* MI) {
232   vector<int> regs = getRegsUsed(MI);
233   for (vector<int>::iterator it = regs.begin(); it != regs.end();
          ++it) {
```

```
234       /*
235       if (errLog) {
236         *errLog << "    Register used: " << *it;
237         *errLog << ", with " << RegDeps[*it].use_dep.size() << "
              dependencies\n";
238         errLog->flush();
239       }
240       */
241       // Set all NULL LowerBoundes with this reg to point to this
            instr
242       vector<int>::iterator it2 = RegDeps[*it].use_dep.begin(),
243         E = RegDeps[*it].use_dep.end();
244       for (; it2 != E; ++it2) {
245         if (AllMemOps[*it2].LowerBound == NULL) {
246           AllMemOps[*it2].LowerBound = MI;
247           /*
248           if (errLog) {
249             *errLog << "      Setting a LowerBound\n";
250             errLog->flush();
251           }
252           */
253         }
254       }
255     }
256   }
257
258   void MagnetPass::endLowerBoundUsingRegsModified(MachineInstr* MI)
        {
259     vector<int> regs = getRegsModified(MI);
```

```
260     for (vector<int>::iterator reg_modified = regs.begin();
            reg_modified != regs.end(); ++reg_modified) {
261       const int rnum = *reg_modified;
262       /*
263       if (errLog) {
264         *errLog << "    Register defined: " << rnum;
265         *errLog << ", with " << RegDeps[rnum].mod_dep.size() << "
              dependencies\n";
266         errLog->flush();
267       }
268       */
269       vector<int>::iterator dependent_reg = RegDeps[rnum].mod_dep.
            begin(),
270         E = RegDeps[rnum].mod_dep.end();
271       for (; dependent_reg != E; ++dependent_reg) {
272         if (AllMemOps[*dependent_reg].LowerBound == NULL) {
273           AllMemOps[*dependent_reg].LowerBound = MI;
274           /*
275           if (errLog) {
276             *errLog << "      Setting " << AllMemOps[*dependent_reg
                  ].OpLocation << " LowerBound to " << MI << "\n";
277             errLog->flush();
278           }
279           */
280         }
281       }
282     }
283 }
284
```

```
285 void MagnetPass::endUpperBoundUsingRegsUsed(MachineInstr* MI) {
286   vector<int> regs = getRegsUsed(MI);
287   for (vector<int>::iterator reg_used = regs.begin(); reg_used !=
          regs.end(); ++reg_used) {
288     const int rnum = *reg_used;
289     /*
290     if (errLog) {
291       *errLog << "    Register used: " << rnum;
292       *errLog << ", with " << RegDeps[rnum].use_dep.size() << "
            dependencies\n";
293       errLog->flush();
294     }
295     */
296     // Set all NULL LowerBoundes with this reg to point to this
          instr
297     vector<int>::iterator dependent_reg = RegDeps[rnum].use_dep.
          begin(),
298       E = RegDeps[rnum].use_dep.end();
299     for (; dependent_reg != E; ++dependent_reg) {
300       if (AllMemOps[*dependent_reg].UpperBound == NULL) {
301         AllMemOps[*dependent_reg].UpperBound = MI;
302         /*
303         if (errLog) {
304           *errLog << "      Setting a UpperBound\n";
305           errLog->flush();
306         }
307         */
308       }
309     }
```

```
310     }
311   }
312
313   void MagnetPass::endUpperBoundUsingRegsModified(MachineInstr* MI)
          {
314     vector<int> regs = getRegsModified(MI);
315     for (vector<int>::iterator it = regs.begin(); it != regs.end();
            ++it) {
316       const int rnum = *it;
317       /*
318       if (errLog) {
319         *errLog << "    Register defined: " << rnum;
320         *errLog << ", with " << RegDeps[rnum].mod_dep.size() << "
              dependencies\n";
321         errLog->flush();
322       }
323       */
324       vector<int>::iterator it2 = RegDeps[rnum].mod_dep.begin(),
325         E = RegDeps[rnum].mod_dep.end();
326       for (; it2 != E; ++it2) {
327         if (AllMemOps[*it2].UpperBound == NULL) {
328           AllMemOps[*it2].UpperBound = MI;
329           /*
330           if (errLog) {
331             *errLog << "      Setting a UpperBound\n";
332             errLog->flush();
333           }
334           */
335         }
```

```
336       }
337     }
338   }
339
340   void MagnetPass::endLowerBoundUsingMem(MachineInstr* MI) {
341     // Set all NULL LowerBounds with this reg to point to this instr
342     vector<int>::iterator it2 = MemDeps.begin(), E = MemDeps.end();
343     if (isLoad(MI)) {
344       for (; it2 != E; ++it2) {
345         if (AllMemOps[*it2].LowerBound == NULL && isStore(AllMemOps
              [*it2].OpLocation)) {
346           AllMemOps[*it2].LowerBound = MI;
347           if (errLog) {
348             *errLog << "      Setting a LowerBound using MEM at " <<
                  AllMemOps[*it2].OpLocation << " to " << MI << "\n";
349             errLog->flush();
350           }
351         }
352       }
353     } else if (isStore(MI)) {
354       for (; it2 != E; ++it2) {
355               // Conservative approach to the aliasing problem means
                    we cannot move stores beyond stores, lest two
                    base+offsets point to the same memory location
356         if (AllMemOps[*it2].LowerBound == NULL && (isLoad(AllMemOps
              [*it2].OpLocation) || isStore(AllMemOps[*it2].OpLocation
              ))) {
357           AllMemOps[*it2].LowerBound = MI;
358           if (errLog) {
```

```
359             *errLog << "      Setting a LowerBound using MEM at " <<
                    AllMemOps[*it2].OpLocation << " to " << MI << "\n";
360             errLog->flush();
361           }
362         }
363       }
364     }
365   }
366
367   void MagnetPass::endUpperBoundUsingMem(MachineInstr* MI) {
368     // Set all NULL UpperBounds with this reg to point to this instr
369     vector<int>::iterator it2 = MemDeps.begin(),
370       E = MemDeps.end();
371     if (isLoad(MI)) {
372       for (; it2 != E; ++it2) {
373         if (AllMemOps[*it2].UpperBound == NULL && isStore(AllMemOps
                [*it2].OpLocation)) {
374           AllMemOps[*it2].UpperBound = MI;
375           if (errLog) {
376             *errLog << "      Setting a UpperBound using MEM at " <<
                    AllMemOps[*it2].OpLocation << " to " << MI << "\n";
377             errLog->flush();
378           }
379         }
380       }
381     } else if (isStore(MI)) {
382       for (; it2 != E; ++it2) {
383             // Conservative approach to the aliasing problem means
                  we cannot move stores beyond stores, lest two
```

```
                base+offsets point to the same memory location
384       if (AllMemOps[*it2].UpperBound == NULL && (isLoad(AllMemOps
            [*it2].OpLocation) || isStore(AllMemOps[*it2].OpLocation
            ))) {
385         AllMemOps[*it2].UpperBound = MI;
386         if (errLog) {
387           *errLog << "      Setting a UpperBound using MEM at " <<
                AllMemOps[*it2].OpLocation << " to " << MI << "\n";
388           errLog->flush();
389         }
390       }
391     }
392   }
393 }
394
395 void MagnetPass::printBB(MachineBasicBlock &MBB, const
      TargetMachine &TM, string s) {
396   *errLog << s << "\n";
397   MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end();
398   for (; MBBI != E; ++MBBI) {
399     MachineInstr *MI = MBBI;
400     *errLog << "  " << MI << ": ";
401     MI->print(*errLog, &TM);
402   }
403   errLog->flush();
404 }
405
406 void MagnetPass::clearAllDeps() {
407   for (unsigned i = 0; i < RegDeps.size(); ++i) {
```

```
408       RegDeps[i].use_dep.clear();
409       RegDeps[i].mod_dep.clear();
410     }
411     MemDeps.clear();
412   }
413
414   void MagnetPass::findLowerBounds(MachineBasicBlock &MBB, const
          TargetMachine &TM) {
415     if (errLog) {
416       *errLog << "findLowerBounds: Starting Analysis\n";
417       errLog->flush();
418     }
419     clearAllDeps();
420     // Iterate though basic block
421     MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end();
422     for (; MBBI != E; ++MBBI) {
423       MachineInstr *MI = MBBI;
424
425       if (errLog) {
426         *errLog << "   " << MI << ": ";
427         MI->print(*errLog, &TM);
428         errLog->flush();
429       }
430
431       endLowerBoundUsingRegsModified(MI);
432       endLowerBoundUsingRegsUsed(MI);
433       endLowerBoundUsingMem(MI);
434
435       // FIXME: isMemoryOp() apparently doesn't catch STR r2, [r1,
```

```
        r0 #16386]
436     if (isMemoryOp(MI)) {
437       /*
438       if (errLog) {
439         for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i
              ) {
440           MachineOperand &MO = MI->getOperand(i);
441           *errLog << "    " << MO << " " << MO.getType() << "\n";
442           // *errLog << "    " << MO.getReg() << " " << MO.getType
                 () << "\n";
443         }
444         errLog->flush();
445       }
446       */
447       int uMOR = getAllMemOpsIndexOf(MI);
448
449       // If this MemOp has already been removed from AllMemOps,
             skip
450       if (uMOR < 0)
451         continue;
452
453       MemDeps.push_back(uMOR);
454
455       vector<int> regsUsed = getRegsUsed(MI);
456       for (vector<int>::iterator it = regsUsed.begin(); it !=
             regsUsed.end(); ++it) {
457         RegDeps[*it].mod_dep.push_back(uMOR);
458         /*
459         if (errLog) {
```

```
460             *errLog << "      Adding mod_dep using reg " << *it << "
                    to this MI (" << AllMemOps[RegDeps[*it].mod_dep.
                   size() - 1].OpLocation << ")\n";
461             errLog->flush();
462           }
463           */
464         }
465
466         // Only loads create use dependencies
467         if (isLoad(MI)) {
468           vector<int> regsModified = getRegsModified(MI);
469           for (vector<int>::iterator it = regsModified.begin(); it
                != regsModified.end(); ++it)
470             RegDeps[*it].use_dep.push_back(uMOR);
471         }
472       }
473     }
474     if (errLog) {
475       *errLog << "findLowerBounds: Finished\n";
476       *errLog << "findLowerBounds: Current AllMemOps (" << AllMemOps
           .size() << ")\n";
477       for (unsigned i = 0; i < AllMemOps.size(); ++i) {
478         MemOpRecord a = AllMemOps[i];
479         *errLog << "  " << a.OpLocation << " with UpperBound = " <<
              a.UpperBound << " and LowerBound = " << a.LowerBound <<
              "\n";
480         errLog->flush();
481       }
482       /*
```

```
483       *errLog << "findLowerBounds: Current RegDeps:\n";
484       for (unsigned i = 0; i < numRegs; ++i) {
485         if (RegDeps[i].use_dep.size() > 0) {
486           *errLog << "  use_dep[" << i << "]: ";
487           for (vector<int>::iterator it = RegDeps[i].use_dep.begin()
                  ; it != RegDeps[i].use_dep.end(); ++it) {
488             *errLog << AllMemOps[*it].OpLocation << ", ";
489           }
490           *errLog << "\n";
491           errLog->flush();
492         }
493         if (RegDeps[i].mod_dep.size() > 0) {
494           *errLog << "  mod_dep[" << i << "]: ";
495           for (vector<int>::iterator it = RegDeps[i].mod_dep.begin()
                  ; it != RegDeps[i].mod_dep.end(); ++it) {
496             *errLog << AllMemOps[*it].OpLocation << ", ";
497           }
498           *errLog << "\n";
499           errLog->flush();
500         }
501       }
502       */
503       errLog->flush();
504     }
505   }
506
507   void MagnetPass::findUpperBounds(MachineBasicBlock &MBB, const
        TargetMachine &TM) {
508     if (errLog) {
```

```
509       *errLog << "findUpperBounds: Starting Analysis\n";
510       errLog->flush();
511     }
512     clearAllDeps();
513     // Iterate through basic block in reverse order
514     MachineBasicBlock::iterator MBBRI = MBB.end(), E = MBB.begin();
515     for (; MBBRI != E; MBBRI--) {
516       MachineBasicBlock::iterator MBBRI2 = MBBRI;
517       --MBBRI2;
518       MachineInstr *MI = MBBRI2;
519
520       /*
521       if (errLog) {
522         *errLog << "  " << MI << ": ";
523         MI->print(*errLog, &TM);
524         errLog->flush();
525       }
526       */
527
528       endUpperBoundUsingRegsModified(MI);
529       endUpperBoundUsingRegsUsed(MI);
530       endUpperBoundUsingMem(MI);
531
532       if (isMemoryOp(MI)) {
533         // Locate MemOpRecord corresponding to this instr
534         int uMOR = getAllMemOpsIndexOf(MI);
535
536         // If this MemOp has already been removed from AllMemOps,
              skip
```

```
537         if (uMOR < 0)
538           continue;
539
540         MemDeps.push_back(uMOR);
541
542         vector<int> regsUsed = getRegsUsed(MI);
543         for (vector<int>::iterator it = regsUsed.begin(); it !=
                regsUsed.end(); ++it) {
544           RegDeps[*it].mod_dep.push_back(uMOR);
545         }
546
547         // Only loads create use dependencies
548         if (isLoad(MI)) {
549           vector<int> regsModified = getRegsModified(MI);
550           for (vector<int>::iterator it = regsModified.begin(); it
                != regsModified.end(); ++it)
551             RegDeps[*it].use_dep.push_back(uMOR);
552         }
553       }
554     }
555     if (errLog) {
556       *errLog << "findUpperBounds: Finished\n";
557       *errLog << "findUpperBounds: Current AllMemOps (" << AllMemOps
              .size() << ")\n";
558       for (unsigned i = 0; i < AllMemOps.size(); ++i) {
559         MemOpRecord a = AllMemOps[i];
560         *errLog << "  " << a.OpLocation << " with UpperBound = " <<
              a.UpperBound << " and LowerBound = " << a.LowerBound <<
              "\n";
```

```
561         errLog->flush();
562       }
563       /*
564       *errLog << "findUpperBounds: Current RegDeps:\n";
565       for (unsigned i = 0; i < numRegs; ++i) {
566         if (RegDeps[i].use_dep.size() > 0) {
567           *errLog << "  use_dep[" << i << "]: ";
568           for (vector<int>::iterator it = RegDeps[i].use_dep.begin()
                  ; it != RegDeps[i].use_dep.end(); ++it) {
569             *errLog << *it << ", ";
570           }
571           *errLog << "\n";
572           errLog->flush();
573         }
574         if (RegDeps[i].mod_dep.size() > 0) {
575           *errLog << "  mod_dep[" << i << "]: ";
576           for (vector<int>::iterator it = RegDeps[i].mod_dep.begin()
                  ; it != RegDeps[i].mod_dep.end(); ++it) {
577             *errLog << *it << ", ";
578           }
579           *errLog << "\n";
580           errLog->flush();
581         }
582       }
583       */
584       errLog->flush();
585     }
586   }
587
```

```
588 ClusterPoint MagnetPass::getBestRangeOverlap(MachineBasicBlock &
      MBB) {
589   if (errLog) {
590     *errLog << "getBestRangeOverlap: Starting Analysis (AllMemOps
          has " << AllMemOps.size() << " entries)\n";
591     errLog->flush();
592   }
593
594   // If any loads remain in AllMemOps, evaluate loads
595   bool lookAtLoads = false;
596   unsigned baseReg = 1337;
597   for (vector<MemOpRecord>::iterator it = AllMemOps.begin(), e =
        AllMemOps.end()
598       ; it != e; ++it) {
599     MachineInstr *MI = it->OpLocation;
600     if (isLoad(MI)) {
601       lookAtLoads = true;
602       baseReg = getBaseReg(MI);
603       break;
604     }
605   }
606     // If we are evaluating loads, we already found the baseReg in
          the search loop
607   if (!lookAtLoads)
608     baseReg = getBaseReg(AllMemOps[0].OpLocation);
609
610   if (errLog) {
611     *errLog << "  lookAtLoads set to " << lookAtLoads << " and
          baseReg set to " << baseReg << "\n";
```

```
612       errLog->flush();
613     }
614
615     // Adds all initial loads or stores that can move to top of
            basic block
616     ClusterPoint currentCluster, bestCluster;
617     for (vector<MemOpRecord>::iterator it = AllMemOps.begin(), e =
            AllMemOps.end()
618          ; it != e; ++it) {
619       MachineInstr *MI = it->OpLocation;
620       // If lookAtLoads, we only match loads, else we only match
              stores
621       if ((lookAtLoads && isLoad(MI)) || (!lookAtLoads && !isLoad(MI
              ))) {
622         // If the baseReg is what we want and UpperBound is NULL,
                add it to currentCluster
623         if ((getBaseReg(MI) == baseReg) && (it->UpperBound == NULL))
624           currentCluster.instructionsToGather.push_back(it->
                  OpLocation);
625       }
626     }
627     bestCluster = currentCluster;
628
629     // FIXME: Currently loads AND stores BOTH move as far DOWN as
            they can.
630     //        Make stores move UP!
631     // Look through BB for concurrent live Ranges of Movement at
            every BB instruction
632     for (MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end
```

```
      ()
633       ; MBBI != E; ++MBBI) {
634     MachineInstr *MI = MBBI;
635     currentCluster.insertAfter = MI;
636         // Check to see which AllMemOps with proper base reg have
              live Ranges of Movement at this instruction
637     for (vector<MemOpRecord>::iterator it = AllMemOps.begin(), e =
          AllMemOps.end()
638         ; it != e; ++it) {
639       MachineInstr *AMOMI = it->OpLocation;
640       // If lookAtLoads, we only match loads, else we only match
            stores
641       if ((lookAtLoads && isLoad(AMOMI)) || (!lookAtLoads && !
            isLoad(AMOMI))) {
642         // If the baseReg is what we want
643         if (getBaseReg(AMOMI) == baseReg) {
644                     // Add to set of live Ranges of Movement
645           if (it->UpperBound == MI)
646             currentCluster.instructionsToGather.push_back(it->
                  OpLocation);
647                     // Remove from set of live Ranges of Movement
648           if (it->LowerBound == MI) {
649             // FIXME: Is this broken like the contigous check was
                  broken?
650             vector<MachineInstr*>::iterator temp = remove(
                  currentCluster.instructionsToGather.begin(),
                  currentCluster.instructionsToGather.end(), MI);
651             currentCluster.instructionsToGather.erase(
                  currentCluster.instructionsToGather.begin(), temp)
```

```
                ;
652           }
653         }
654       }
655     }
656         // Keep track of where the largest cluster of live Ranges
                of Movement are
657     if (currentCluster.instructionsToGather.size() >= bestCluster.
          instructionsToGather.size())
658       bestCluster = currentCluster;
659   }
660
661   if (errLog) {
662     *errLog << "  bestCluster created with " << bestCluster.
          instructionsToGather.size() << " entries\n";
663     errLog->flush();
664   }
665
666   // Remove from AllMemOps all bestCluster.instructionsToGather,
        since we're going to move them
667   for (vector<MachineInstr*>::iterator it = bestCluster.
        instructionsToGather.begin(), e = bestCluster.
        instructionsToGather.end()
668        ; it != e; ++it) {
669     for (vector<MemOpRecord>::iterator toRemove = AllMemOps.begin
          ()
670          ; toRemove != AllMemOps.end(); ++toRemove) {
671       if (toRemove->OpLocation == *it) {
672         AllMemOps.erase(toRemove);
```

```
673           break;
674         }
675       }
676     }
677
678     // If bestCluster.insertAfter is in bestCluster.
          instructionsToGather, remove it since it's already been "
          moved"
679     for (vector<MachineInstr*>::iterator it = bestCluster.
          instructionsToGather.begin(), e = bestCluster.
          instructionsToGather.end()
680          ; it != e; ++it) {
681       if (*it == bestCluster.insertAfter) {
682         bestCluster.instructionsToGather.erase(it);
683         break;
684       }
685     }
686
687     if (errLog) {
688       *errLog << "getBestRangeOverlap: Finished Analysis (AllMemOps
            has " << AllMemOps.size() << " entries)\n";
689       errLog->flush();
690     }
691
692     return bestCluster;
693   }
694
695   bool MagnetPass::allMIsContiguous(MachineBasicBlock &MBB, const
        TargetMachine &TM, ClusterPoint bestCluster) {
```

```
696   if (errLog) {
697     *errLog << "allMIsContiguous: bestCluster.instructionsToGather
          Before Any Operations\n";
698     for (vector<MachineInstr*>::iterator it = bestCluster.
          instructionsToGather.begin(), e = bestCluster.
          instructionsToGather.end();
699          it != e; ++it) {
700       *errLog << "  " << *it << ": ";
701       (*it)->print(*errLog, &TM);
702     }
703     *errLog << "allMIsContiguous: bestCluster.insertAfter Before
          Any Operations\n";
704     *errLog << "  " << bestCluster.insertAfter << ": ";
705     bestCluster.insertAfter->print(*errLog, &TM);
706     *errLog << "allMIsContiguous: Starting contiguous check\n";
707     errLog->flush();
708   }
709
710   bestCluster.instructionsToGather.push_back(bestCluster.
       insertAfter);
711
712   // Check for contiguousness of all instructionsToGather
713   bool started = false;
714   for (MachineBasicBlock::iterator MBBI = MBB.begin()
715        ; MBBI != MBB.end(); ++MBBI) {
716     MachineInstr *MI = MBBI;
717     if (bestCluster.instructionsToGather.end() != find(bestCluster
          .instructionsToGather.begin(), bestCluster.
          instructionsToGather.end(), MI)) {
```

```
718         if (!started)
719           started = true;
720         if (errLog) {
721           *errLog << "  MI found, removing " << MI << " - there were
                  " << bestCluster.instructionsToGather.size();
722           errLog->flush();
723         }
724         vector<MachineInstr*>::iterator temp = remove(bestCluster.
              instructionsToGather.begin(), bestCluster.
              instructionsToGather.end(), MI);
725         bestCluster.instructionsToGather.erase(temp, bestCluster.
              instructionsToGather.end());
726         if (errLog) {
727           *errLog << " and are now " << bestCluster.
                 instructionsToGather.size() << " instructionsToGather"
                  << "\n";
728           errLog->flush();
729         }
730       } else {
731         if (started) {
732           if (errLog) {
733             *errLog << "  MI not found after started, " <<
                    bestCluster.instructionsToGather.size() << "
                    instructionsToGather are left\n";
734             errLog->flush();
735           }
736           if (bestCluster.instructionsToGather.size() == 0)
737             return true;
738           else
```

```
739             return false;
740         }
741         if (errLog) {
742           *errLog << "  MI not found, " << bestCluster.
                instructionsToGather.size() << " instructionsToGather
                are left\n";
743           errLog->flush();
744         }
745       }
746     }
747     if (started)
748       return true;
749
750     return false;
751   }
752
753   // Clustering is performed by removing all affected memory ops
      from the BB, removing all instructions after the
754   // insertion point, appending the memory ops after the insertion
      point, and finally appending the trailing
755   // instructions at the end.
756   void MagnetPass::gatherAtBestRangeOverlap(MachineBasicBlock &MBB,
      const TargetMachine &TM, ClusterPoint bestCluster) {
757     if (errLog) {
758       *errLog << "gatherAtBestRangeOverlap: bestCluster.
            instructionsToGather Before Any Operations\n";
759       for (vector<MachineInstr*>::iterator it = bestCluster.
            instructionsToGather.begin(), e = bestCluster.
            instructionsToGather.end();
```

```
760         it != e; ++it) {
761       *errLog << "  " << *it << ": ";
762       (*it)->print(*errLog, &TM);
763     }
764     printBB(MBB, TM, "gatherAtBestRangeOverlap: MBB Before Any
          Operations");
765     *errLog << "gatherAtBestRangeOverlap: Beginning <kill>
          analysis for " << bestCluster.instructionsToGather.size()
          << " bestCluster MIs\n";
766     errLog->flush();
767   }
768
769   // Split instructionsToGather into MIs that must move up and MIs
        that must move down
770   vector<MachineInstr*> moveDown, moveUp;
771   bool passedInsertAfter = false;
772   for (MachineBasicBlock::iterator MBBI = MBB.begin()
773       ; MBBI != MBB.end(); ++MBBI) {
774     MachineInstr *MI = MBBI;
775     // If MI is in instructionsToGather, insert it into the proper
          vector
776     if (bestCluster.instructionsToGather.end() != find(bestCluster
          .instructionsToGather.begin(), bestCluster.
          instructionsToGather.end(), MI)) {
777       if (passedInsertAfter)
778         moveUp.push_back(MI);
779       else
780         moveDown.push_back(MI);
781     }
```

```
782       if (MI == bestCluster.insertAfter)
783         passedInsertAfter = true;
784     }
785
786       // Memory ops moving below another instruction which has a
              register kill must SUBSUME that kill.
787       // Memory ops with a register kill that move above another
              instruction using that register must
788       // ABDICATE that kill to the other instruction.
789     enum reg_stats { NONE, TOSUBSUME, SUBSUMED, TOABDICATE,
          ABDICATED };
790
791     // FIXME: Since numRegs isn't const, use workaround for custom
          number of regs
792     reg_stats regStatus[16];
793     for (unsigned i = 0; i < 16; ++i)
794       regStatus[i] = NONE;
795
796     // START ANALYZING moveDown MIs for kills they may need to
          SUBSUME
797       // Any register used by any MI in moveDown could be:
798       //   1. Possibly killed in an instruction it moves after (NONE
              -> TOSUBSUME), in which case would need to SUBSUME
799       //   2. Killed in an instruction it moves after, in which case
              it must SUBSUME (NONE/TOSUBSUME -> SUBSUME)
800
801     // Iterate through BB even if no MIs need to be moved down, to
          set removeStartingHere
802     MachineBasicBlock::iterator removeStartingHere;
```

```
803   for (MachineBasicBlock::iterator MBBI = MBB.begin()
804       ; MBBI != MBB.end(); ++MBBI) {
805     MachineInstr *MI = MBBI;
806
807     bool moveDownContainsMI = false;
808     // If MI is in moveDown, prepare to SUBSUME later kills to MI'
          s regs
809     if (moveDown.end() != find(moveDown.begin(), moveDown.end(),
          MI))
810       moveDownContainsMI = true;
811
812     // Examine all regs used by MI
813     for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
814       MachineOperand &MO = MI->getOperand(i);
815       if (MO.isReg()) {
816         const int rnum = getRegNum(MO.getReg());
817         // If reg is one of the 16 addressable regs
818         if (rnum >= 0) {
819           // Prepare to SUBSUME all reg kills this MI uses
820           if (moveDownContainsMI) {
821             if (MO.isKill()) {
822               regStatus[rnum] = SUBSUMED;
823               MO.setIsKill(false);
824             } else {
825               if (regStatus[rnum] != SUBSUMED)
826                 regStatus[rnum] = TOSUBSUME;
827             }
828           // Update whether moveDown MIs must SUBSUME this reg
                kill
```

```
829             } else {
830               if (MO.isKill() && (regStatus[rnum] != NONE)) {
831                 regStatus[rnum] = SUBSUMED;
832                 MO.setIsKill(false);
833               }
834             }
835           }
836         }
837       }
838
839       // Once we hit insertAfter, set removeStartingHere and leave
              MBB loop
840       if (MI == bestCluster.insertAfter) {
841         removeStartingHere = MBBI;
842         ++removeStartingHere;
843         break;
844       }
845     } // End of MBBI iteration
846
847     // Remove moveDown MIs from MBB
848     for (vector<MachineInstr*>::iterator it = moveDown.begin(), e =
          moveDown.end();
849          it != e; ++it)
850       MBB.remove(*it);
851
852     // Update moveDown kill flags based on regStatus values
853     for (unsigned i = 0; i < 16; ++i) {
854       if (regStatus[i] == SUBSUMED) {
855         bool stopUpdating = false;
```

```
856         // Find LAST moveDown MI that uses i reg and add a kill flag
857         for (vector<MachineInstr*>::reverse_iterator rit = moveDown.
                rbegin(), e = moveDown.rend();
858              rit != e; ++rit) {
859           //MachineInstr *MI = rit;
860           // Examine all regs used by MI
861           for (unsigned i = 0, e = (*rit)->getNumOperands(); i != e;
                  ++i) {
862             MachineOperand &MO = (*rit)->getOperand(i);
863             if (MO.isReg()) {
864               // If reg is one of the 16 addressable regs and is i
865               if ((int)i == getRegNum(MO.getReg())) {
866                 (*rit)->addRegisterKilled(MO.getReg(), TM.
                        getRegisterInfo());
867                 stopUpdating = true;
868               }
869             }
870           }
871           // We only want to update last moveDown MI using i
872           if (stopUpdating)
873             break;
874         }
875       }
876     }
877
878     // DONE ANALYZING moveDown MIs, START ANALYZING moveUp MIs for
          register kills it may need to ABDICATE
879       // Any register killed by any MI in moveUp could move behind
            an instruction using that register,
```

```
880       // in which case it must ABDICATE the kill to that instruction
881
882     // Reset regStatus values
883     for (unsigned i = 0; i < 16; ++i)
884       regStatus[i] = NONE;
885
886     // If no MIs need to be moved UP, don't iterate through BB
            backwards
887     if (moveUp.size() > 0) {
888       for (MachineBasicBlock::reverse_iterator rMBBI = MBB.rbegin()
889           ; rMBBI != MBB.rend(); ++rMBBI) {
890         //MachineInstr *MI = rMBBI;
891
892         // Once we hit insertAfter, leave MBB loop
893         if (bestCluster.insertAfter == &(*rMBBI))
894           break;
895
896         bool moveUpContainsMI = false;
897         // If MI is in moveUp, prepare to ABDICATE later kills to MI
                's regs
898         if (moveUp.end() != find(moveUp.begin(), moveUp.end(), &(*
                rMBBI)))
899           moveUpContainsMI = true;
900
901         // Examine all regs used by MI
902         for (unsigned i = 0, e = (rMBBI)->getNumOperands(); i != e;
                ++i) {
903           MachineOperand &MO = (rMBBI)->getOperand(i);
904           if (MO.isReg()) {
```

```
905             const int rnum = getRegNum(MO.getReg());
906             // If reg is one of the 16 addressable regs
907             if (rnum >= 0) {
908               // Prepare to ABDICATE all regs this MI uses
909               if (moveUpContainsMI) {
910                 if (MO.isKill()) {
911                   if (regStatus[rnum] != ABDICATED)
912                     regStatus[rnum] = TOABDICATE;
913                 }
914               // Update MI and mark regStatus ABDICATED if we are
                     waiting to abdicate
915               } else {
916                 if (regStatus[rnum] == TOABDICATE) {
917                   regStatus[rnum] = ABDICATED;
918                   (rMBBI)->addRegisterKilled(MO.getReg(), TM.
                        getRegisterInfo());
919                 }
920               }
921             }
922           }
923         }
924       } // End of MBB reverse iteration
925     }
926
927     // Remove moveUp MIs from MBB
928     for (vector<MachineInstr*>::iterator it = moveUp.begin(), e =
          moveUp.end();
929          it != e; ++it)
930       MBB.remove(*it);
```

```
931
932     // Remove moveUp kill flags based on regStatus values
933     for (unsigned i = 0; i < 16; ++i) {
934       if (regStatus[i] == ABDICATED) {
935         // Find all moveDown MI that uses i reg and remove kill flag
936         for (vector<MachineInstr*>::iterator it = moveUp.begin(), e
                = moveUp.end();
937              it != e; ++it) {
938           MachineInstr *MI = *it;
939           // Examine all regs used by MI
940           for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i
                ) {
941             MachineOperand &MO = MI->getOperand(i);
942             if (MO.isReg()) {
943               // If i reg is ABDICATED and moveUp MI kills reg
944               if ((int)i == getRegNum(MO.getReg())) {
945                 if (MO.isKill())
946                   MO.setIsKill(false);
947               }
948             }
949           }
950         }
951       }
952     }
953
954     // DONE ANALYZING moveUp MIs
955
956     if (errLog) {
957       *errLog << "gatherAtBestRangeOverlap: Finished <kill> analysis
```

```
        \n";
958       printBB(MBB, TM, "gatherAtBestRangeOverlap: MBB After MemOp
            Removal");
959     }
960
961     // Set temp to hold all MBB non-MemOps after insertAfter
962     vector<MachineInstr*> temp;
963     bool saveTrailing = false;
964     for (MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end
            ()
965          ; MBBI != E; ++MBBI) {
966       MachineInstr *MI = MBBI;
967       if (saveTrailing)
968         temp.push_back(MI);
969       if (MI == bestCluster.insertAfter)
970         saveTrailing = true;
971     }
972
973     // Remove non-MemOps after insertAfter from MBB
974     for (vector<MachineInstr*>::iterator it = temp.begin(); it !=
            temp.end(); ++it)
975       MBB.remove(*it);
976
977     // Append bestCluster.instructionsToGather MemOps after
            insertAfter
978     for (vector<MachineInstr*>::iterator it = bestCluster.
            instructionsToGather.begin(), e = bestCluster.
            instructionsToGather.end();
979          it != e; ++it)
```

```
980       MBB.push_back(*it);
981
982     // Append all non-MemOps after insertAfter back onto MBB
983     for (vector<MachineInstr*>::iterator it = temp.begin(); it !=
            temp.end(); ++it)
984       MBB.push_back(*it);
985
986     if (errLog)
987       printBB(MBB, TM, "gatherAtBestRangeOverlap: MBB After
              Rearrangement");
988   }
989
990   void MagnetPass::initialize(MachineBasicBlock &MBB, const
        TargetMachine &TM) {
991     if (errLog) {
992       *errLog << "initialize: Building AllMemOps and RegDeps\n";
993       errLog->flush();
994     }
995
996     // Setup the RegDeps
997     // FIXME: Creates 112 RegisterDependencies instead of 16 - why?
998     //numRegs = TM.getRegisterInfo()->getNumRegs();
999     numRegs = 16;
1000    for (unsigned i = 0; i < numRegs; ++i) {
1001      RegisterDependencies d;
1002      RegDeps.push_back(d);
1003    }
1004
1005    // Setup AllMemOps
```

```
1006    for (MachineBasicBlock::iterator MBBI = MBB.begin(), E = MBB.end
            ()
1007         ; MBBI != E; ++MBBI) {
1008      MachineInstr *MI = MBBI;
1009      if (isMemoryOp(MI)) {
1010        MemOpRecord MOR;
1011        MOR.OpLocation = MI;
1012        AllMemOps.push_back(MOR);
1013      }
1014    }
1015  }
1016
1017  void MagnetPass::cleanUp() {
1018    if (errLog) {
1019      *errLog << "cleanUp: Clearing AllMemOps and RegDeps\n";
1020      errLog->flush();
1021    }
1022    AllMemOps.clear();
1023    clearAllDeps();
1024  }
1025
1026  void MagnetPass::runOptimization(MachineBasicBlock &MBB, const
        TargetMachine &TM) {
1027    initialize(MBB, TM);
1028
1029    while (AllMemOps.size() > 0) {
1030      // Nullify all LowerBound and UpperBound values in AllMemOps
1031      for (vector<MemOpRecord>::iterator it = AllMemOps.begin(), e =
              AllMemOps.end()
```

```
1032          ; it != e; ++it) {
1033        it->UpperBound = NULL;
1034        it->LowerBound = NULL;
1035      }
1036
1037      // Regenerate the ranges for the MemOps still remaining in
              AllMemOps
1038      findLowerBounds(MBB, TM);
1039      findUpperBounds(MBB, TM);
1040
1041      ClusterPoint bestCluster = getBestRangeOverlap(MBB);
1042      if (!allMIsContiguous(MBB, TM, bestCluster))
1043        gatherAtBestRangeOverlap(MBB, TM, bestCluster);
1044    }
1045
1046    cleanUp();
1047  }
```

Listing C.2: Enabling Optimization

```
1  /// ARMAllocLoadStoreOpt - Post- register allocation pass the
       combine
2  /// load / store instructions to form ldm / stm instructions.
3  namespace {
4    struct ARMLoadStoreOpt : public MachineFunctionPass {
5      static char ID;
6    MagnetPass MagPass;
7    BBPrinter PrintBefore, PrintAfter;
8      ARMLoadStoreOpt() : MachineFunctionPass(ID), MagPass(),
           PrintBefore(true), PrintAfter(false) {}
```

```
9
10  ... skip rest of class definition ...
11
12  bool ARMLoadStoreOpt::runOnMachineFunction(MachineFunction &Fn) {
13    const TargetMachine &TM = Fn.getTarget();
14    AFI = Fn.getInfo<ARMFunctionInfo>();
15    TII = TM.getInstrInfo();
16    TRI = TM.getRegisterInfo();
17    RS = new RegScavenger();
18    isThumb2 = AFI->isThumb2Function();
19
20    bool Modified = false;
21    for (MachineFunction::iterator MFI = Fn.begin(), E = Fn.end();
          MFI != E;
22          ++MFI) {
23      MachineBasicBlock &MBB = *MFI;
24
25    MagPass.runOptimization(MBB, TM);
26
27    //PrintBefore.printBB(MBB, TM);
28      Modified |= LoadStoreMultipleOpti(MBB, TM);
29      Modified |= MergeReturnIntoLDM(MBB);
30    //PrintAfter.printBB(MBB, TM);
31    }
32
33    delete RS;
34    return Modified;
35  }
```

C.3 Python Prototype

During development of our algorithm, we found it useful to create a prototype of our optimization to experiment with using the Python programming language. We did this due to Python's excellent expressiveness and flexibility, which enabled us to validate our ideas without needing full knowledge of how to develop LLVM optimizations in C++. We choose to include this prototype code in this Appendix not only for completeness, but also because by virtue of this same expressiveness it may be easier to understand the underlying algorithm here than in the equivalent C++ code used in the full implementation.

Listing C.3: prototype.py

```
1  #!/usr/bin/python
2
3  import instr_ops
4
5  ops = []
6  ops.append(instr_ops.MachineInstr('LDR r2 r3'))
7  ops.append(instr_ops.MachineInstr('ADD r1 r1 r1'))
8  ops.append(instr_ops.MachineInstr('LDR r0 r3'))
9  ops.append(instr_ops.MachineInstr('ADD r0 r0 r1'))
10 ops.append(instr_ops.MachineInstr('ADD r1 r0 r2'))
11 ops.append(instr_ops.MachineInstr('STR r1 r3'))
12 ops.append(instr_ops.MachineInstr('ADD r2 r1 r3'))
13
14 print "Preparing for first pass"
15 instr_ops.debugInfo()
16
```

```
17 for o in ops:
18
19   print "Analyzing op", o.text
20
21   instr_ops.endRangeMaxUsingRegsModified(o.getRegsModified(), o)
22   instr_ops.endRangeMaxUsingRegsUsed(o.getRegsUsed(), o)
23
24   if o.isLoad():
25     t = instr_ops.MemOpRecord(o)
26     for r in o.getRegsUsed():
27       print "Adding mod_dep", r
28       instr_ops.RegDeps[r].mod_dep.append(t.handle)
29     print "Adding use_dep", o.text.split()[1]
30     instr_ops.RegDeps[o.text.split()[1]].use_dep.append(t.handle)
31     instr_ops.AllMemOps.append(t)
32   if o.isStore():
33     t = instr_ops.MemOpRecord(o)
34     for r in o.getRegsUsed():
35       print "Adding mod_dep", r
36       instr_ops.RegDeps[r].mod_dep.append(t.handle)
37     instr_ops.AllMemOps.append(t)
38
39   instr_ops.debugInfo()
40
41 print "Running check before second pass"
42
43 instr_ops.endRangeMaxUsingRegsModified(instr_ops.RegDeps.keys(),
      None)
44 instr_ops.endRangeMaxUsingRegsUsed(instr_ops.RegDeps.keys(), None)
```

```
45
46  instr_ops.debugInfo()
47
48  for o in reversed(ops):
49
50    print "Analyzing op", o.text
51
52    instr_ops.endRangeMinUsingRegsModified(o.getRegsModified(), o)
53    instr_ops.endRangeMinUsingRegsUsed(o.getRegsUsed(), o)
54
55    if o.isLoad():
56      for a in instr_ops.AllMemOps:
57        if a.opHandle == o.handle:
58          t = a
59          break
60      for r in o.getRegsUsed():
61        print "Adding mod_dep", r
62        instr_ops.RegDeps[r].mod_dep.append(t.handle)
63      print "Adding use_dep", o.text.split()[1]
64      instr_ops.RegDeps[o.text.split()[1]].use_dep.append(t.handle)
65    if o.isStore():
66      for a in instr_ops.AllMemOps:
67        if a.opHandle == o.handle:
68          t = a
69          break
70      for r in o.getRegsUsed():
71        print "Adding mod_dep", r
72        instr_ops.RegDeps[r].mod_dep.append(t.handle)
73
```

```
74    instr_ops.debugInfo()
75
76  print "Running final check"
77
78  instr_ops.endRangeMinUsingRegsModified(instr_ops.RegDeps.keys(),
        None)
79  instr_ops.endRangeMinUsingRegsUsed(instr_ops.RegDeps.keys(), None)
80
81  instr_ops.debugInfo()
82
83  # ============================================================
84  # Dependency info done
85  # ============================================================
86
87  class CanMoveInfo:
88    """ Holds MachineInstr handle to insert before, and MachineInstr
          handles
89        that can be inserted there """
90    def __init__(self):
91      self.insertAfter = None
92      self.canMoveMIs = []
93
94  def memOpRecToMachineInstr(h):
95    for o in ops:
96      if o.handle == h.opHandle:
97        return o
98    raise RuntimeError
99
100 def isL(memOpRec):
```

```
101   return memOpRecToMachineInstr(memOpRec).isLoad()
102
103 def splitBySameBase(main, verbose=False):
104   if verbose: print "Starting splitBySameBase on", len(main), "
          memOpRecords"
105
106   ret = []
107   curBase = None
108   mainCopy = []
109   mainCopy.extend(main)
110   for a in mainCopy:
111     if curBase == None:
112       curBase = memOpRecToMachineInstr(a).getBaseReg()
113       if verbose: print " curBase now set to", curBase
114     if curBase == memOpRecToMachineInstr(a).getBaseReg():
115       if verbose: print " Appending"
116       ret.append(a)
117       main.remove(a)
118     else:
119       if verbose: print " Not appending"
120
121   if verbose: print "End of splitBySameBase, returning", len(ret),
          "memory ops"
122   return main, ret
123
124 def getMaxInfoUsingLoads(loadsWithSameBaseReg, verbose=False):
125
126   if verbose: print "Starting getMaxInfoUsingLoads with", len(
          loadsWithSameBaseReg)
```

```
127
128   curInfo = CanMoveInfo()
129   maxInfo = CanMoveInfo()
130
131   # Initially, see if we have RegDeps pointing to top of BB
132   for l in loadsWithSameBaseReg:
133     if l.rmin == None:
134       curInfo.canMoveMIs.append(memOpRecToMachineInstr(l))
135
136   # Set maxInfo to be whatever curInfo is after checking BB top
137   maxInfo.canMoveMIs = []
138   maxInfo.canMoveMIs.extend(curInfo.canMoveMIs)
139   maxInfo.insertAfter = None
140
141   if verbose: print " curInfo size:", len(curInfo.canMoveMIs)
142
143   # Scan through ops, updating curInfo as we go
144   for o in ops:
145     curInfo.insertAfter = o
146     # Check loads for beginning and end dependency markers
147     for l in loadsWithSameBaseReg:
148       if l.rmin == o.handle:
149         curInfo.canMoveMIs.append(memOpRecToMachineInstr(l))
150       if l.rmax == o.handle:
151         curInfo.canMoveMIs.remove(memOpRecToMachineInstr(l))
152     # If we have a new max, or if we can move down, update maxInfo
153     if len(curInfo.canMoveMIs) >= len(maxInfo.canMoveMIs):
154       maxInfo.canMoveMIs = []
155       maxInfo.canMoveMIs.extend(curInfo.canMoveMIs)
```

```
156         maxInfo.insertAfter = curInfo.insertAfter
157       if verbose: print " curInfo size:", len(curInfo.canMoveMIs)
158     if verbose: print "End of getMaxLoads with maxInfo having", len(
          maxInfo.canMoveMIs), "entries and points to", maxInfo.
          insertAfter.handle
159     return maxInfo
160
161   def moveOpsUsingMaxInfo(ops, loads, maxInfo, verbose=False):
162     retOps = []
163     opsCopy = []
164     opsCopy.extend(ops)
165     for o in opsCopy:
166       dontAppend = False
167       # FIXME should be l in maxInfo.canMoveMIs, not loads!
168       for l in loads:
169         if memOpRecToMachineInstr(l).handle == o.handle:
170           dontAppend = True
171           if verbose: print o.handle, "skipped"
172       if not dontAppend:
173         retOps.append(o)
174         if verbose: print o.handle, "appended"
175       if o.handle == maxInfo.insertAfter.handle:
176         break
177     # Inserts load MachineInstrs and removes them from loads list
178     loadsCopy = []
179     loadsCopy.extend(loads)
180     for l in loadsCopy:
181       if memOpRecToMachineInstr(l) in maxInfo.canMoveMIs:
182         retOps.append(memOpRecToMachineInstr(l))
```

```
183         loads.remove(l)
184         if verbose: print memOpRecToMachineInstr(l).handle, "
                inserted"
185       passed = False
186       for o in ops:
187         if passed:
188           retOps.append(o)
189           if verbose: print o.handle, "concatonated"
190         # FIXME might not work with other cases - should append after
              loads
191         if o.handle == maxInfo.insertAfter.handle:
192           passed = True
193       if verbose:
194         print "After moving some loads"
195         for o in retOps:
196           print " ", o.handle, o.text
197         print
198       return retOps, loads
199
200   print "Before load reordering"
201   for o in ops:
202     print o.handle, o.text
203   print
204
205   loads = [a for a in instr_ops.AllMemOps if isL(a)]
206
207   # Loop until we have no more loads
208   loads, loadsWithSameBaseReg = splitBySameBase(loads, verbose=False
        )
```

```
209 while len(loadsWithSameBaseReg) > 0:
210
211   # Loop until we have no more loads with base reg to move
212   while len(loadsWithSameBaseReg) > 0:
213     maxInfo = getMaxInfoUsingLoads(loadsWithSameBaseReg, verbose=
          False)
214     ops, loadsWithSameBaseReg = moveOpsUsingMaxInfo(ops,
          loadsWithSameBaseReg, maxInfo, verbose=False)
215
216   loads, loadsWithSameBaseReg = splitBySameBase(loads, verbose=
        False)
217
218 print "After load reordering"
219 for o in ops:
220   print o.handle, o.text
221 print
```

Listing C.4: instr_ops.py

```
1  #!/usr/bin/python
2
3  import logging
4
5  NUMREGS = 4
6
7
8  class MachineInstr:
9    """ Hold instr text, a handle, and identity methods """
10   nextHandle = 0
11   def __init__(self, instr):
```

```
12      self.text = instr
13      self.handle = MachineInstr.nextHandle
14      MachineInstr.nextHandle += 1
15      self.op = instr.split()[0]
16      self.regsUsed = list(set(instr.split()[1:]))
17      self.regsModified = []
18      if not self.isStore():
19        self.regsModified.append(instr.split()[1])
20
21    def isLoad(self):
22      return self.op == 'LDR'
23
24    def isStore(self):
25      return self.op == 'STR'
26
27    def isMemOp(self):
28      return self.isLoad() or self.isStore()
29
30    def getRegsUsed(self):
31      return self.regsUsed
32
33    def getRegsModified(self):
34      return self.regsModified
35
36    def getBaseReg(self):
37      return self.regsUsed[-1]
38
39
40  class Dependencies:
```

```
41    """ Holds MemOpRecord handles for reg dependencies """
42    def __init__(self):
43      self.use_dep = []
44      self.mod_dep = []
45
46
47  class MemOpRecord:
48    """ Holds an instr handle and a MemOpRecord handle, along with
          ranges """
49    nextHandle = 0
50    def __init__(self, o):
51      self.handle = MemOpRecord.nextHandle
52      MemOpRecord.nextHandle += 1
53      self.opHandle = o.handle
54      self.rmax = None
55      self.rmin = None
56
57    def logme(self):
58      logging.debug("MemOpRecord %s with op %s using MAX %s and MIN
            %s",
59           self.handle, self.opHandle, self.rmax, self.rmin)
60
61
62  RegDeps = {}
63  for i in range(0, NUMREGS):
64    RegDeps["r" + str(i)] = Dependencies()
65
66  AllMemOps = []
67
```

```
68
69  def debugInfo():
70    for a in AllMemOps:
71      a.logme()
72    for d in RegDeps:
73      logging.debug(d)
74      logging.debug("   U: %s", RegDeps[d].use_dep)
75      logging.debug("   M: %s", RegDeps[d].mod_dep)
76    logging.debug("")
77
78
79  def removeRegDepsUsingMemOp(d):
80    """ Move through all RegDeps and remove MemOpRecord with handle
        d """
81    for d2 in RegDeps:
82      RegDeps[d2].use_dep = [v for v in RegDeps[d2].use_dep if v !=
          d]
83      RegDeps[d2].mod_dep = [v for v in RegDeps[d2].mod_dep if v !=
          d]
84
85
86  def setRMaxUsingMemOpTo(d, finalInstr):
87    """ Set all MemOpRecords with handle d to have RangeMax
        finalInstr (handle) """
88    for a in AllMemOps:
89      if a.handle == d:
90        if finalInstr != None:
91          a.rmax = finalInstr.handle
92        else:
```

```
93          a.rmax = None
94        break
95
96
97  def setRMinUsingMemOpTo(d, finalInstr):
98    """ Set all MemOpRecords with handle d to have RangeMin
        finalInstr (handle) """
99    for a in AllMemOps:
100     if a.handle == d:
101       if finalInstr != None:
102         a.rmin = finalInstr.handle
103       else:
104         a.rmin = None
105       break
106
107
108 def endRangeMaxUsingRegsUsed(regs, finalInstr):
109   """ Each used reg has deps checked and RangeMax set """
110   for r in regs:
111     for d in RegDeps[r].use_dep:
112       logging.debug("Detected violated use_dep %s", d)
113       setRMaxUsingMemOpTo(d, finalInstr)
114       removeRegDepsUsingMemOp(d)
115
116
117 def endRangeMaxUsingRegsModified(regs, finalInstr):
118   """ Each modified reg has deps checked and RangeMax set """
119   for r in regs:
120     for d in RegDeps[r].mod_dep:
```

```
121         logging.debug("Detected violated mod_dep %s", d)
122         setRMaxUsingMemOpTo(d, finalInstr)
123         removeRegDepsUsingMemOp(d)
124
125
126 def endRangeMinUsingRegsUsed(regs, finalInstr):
127   """ Each used reg has deps checked and RangeMin set """
128   for r in regs:
129     for d in RegDeps[r].use_dep:
130       logging.debug("Detected violated use_dep %s", d)
131       setRMinUsingMemOpTo(d, finalInstr)
132       removeRegDepsUsingMemOp(d)
133
134
135 def endRangeMinUsingRegsModified(regs, finalInstr):
136   """ Each modified reg has deps checked and RangeMin set """
137   for r in regs:
138     for d in RegDeps[r].mod_dep:
139       logging.debug("Detected violated mod_dep %s", d)
140       setRMinUsingMemOpTo(d, finalInstr)
141       removeRegDepsUsingMemOp(d)
```