Optimization of vector operations in the LLVM compiler

Stephen Rogers

April, 2014

B.A. (Mod.) Computer Science

Final Year Project

Supervisor: Dr. David Gregg

School of Computer Science and Statistics

Trinity College Dublin, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

______________________________  April 23, 2014
Stephen Rogers
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

______________________________  April 23, 2014
Stephen Rogers
Acknowledgements

First of all, I would like to say a big thank you to my supervisor, Dr. David Gregg. His help and feedback was an invaluable resource throughout this project.

I would also like to thank my family, friends and classmates for their support. In particular, I would like to thank my good friend Ian Hunter for his help in keeping me sane and on track throughout my final year.
Abstract

Naive code generation of large vector operations in LLVM results in an unnecessarily large generated assembly code size, and inefficient generated code. The aim of this project is to alleviate these problems by converting large vector operations into compact loops which use vector operations of the target architecture’s native vector width. The aim being to get the benefits of the small code size of a loop but also the execution speed benefits of using vector operations.

This report examines the design and implementation of this optimization, and gives an evaluation of its performance. The optimization is shown to provide an improvement to the generated code size and also to the execution time of compiled programs. Specifically in the test cases used, the optimization has been shown to provide up to a 17% reduction in execution time and also a huge reduction in code size (several thousand lines of code in some cases).
# Contents

1 Introduction ................................................. 3

2 Code representation in LLVM ................................ 4

3 Problem: Vector instruction selection in LLVM .............. 8

4 Solution design ................................................. 13
   4.1 My algorithm ............................................. 14
   4.2 List generation procedure ................................. 16
   4.3 Movement safety procedure ............................... 20
      4.3.1 The result of an instruction is used outside of its list ... 20
      4.3.2 An instruction is not a load, store or binary operand instruction ... 21
      4.3.3 An instruction is inside the body of a loop and is loop variant ... 23
      4.3.4 An instruction in the list is a store instruction with a non-zero constant source value ... 23
   4.4 Insert loop procedure ..................................... 24

5 Solution implementation ...................................... 30
   5.1 Optimization passes in LLVM ............................. 30
   5.2 Data structures ........................................... 30
   5.3 Algorithm implementation ................................. 31

6 Evaluation ...................................................... 33
   6.1 Safety tests ............................................... 33
   6.2 Benchmark tests .......................................... 34
      6.2.1 Generated code size .................................. 34
      6.2.2 Execution time ....................................... 39

7 Future Work ................................................... 43
   7.1 Transform more kinds of instructions ..................... 43
   7.2 Improve storing of constant vectors ..................... 44
   7.3 Alternative method for ordering instructions ............ 44

8 Conclusion ..................................................... 45
List of Figures

2.1 An example of a Control Flow Graph ........................................... 5
2.2 An example of a program converted into SSA form ......................... 6
2.3 The example in figure 2.2 represented as LLVM-IR ......................... 7

3.1 An example of a vector instruction ............................................. 8
3.2 A demonstration of how the number of generated instructions increases with vector width ......................................................... 11

4.1 Sample input and output of the optimization pass ............................ 13
4.2 CFG containing a phi-node which cannot be transformed ................. 22
4.3 Insert loop procedure example, input CFG .................................. 24
4.4 Insert loop procedure example, after splitting of the input CFG ........ 25
4.5 Insert loop procedure example, initializing the loop iterator and inserting two new basic blocks ................................................. 25
4.6 Insert loop procedure example, inserting the loop iterator check condition and branch instruction ...................................................... 26
4.7 Insert loop procedure example, after insertion of the converted vector instructions ................................................................. 28
4.8 Insert loop procedure example, final state of the CFG .................... 29

6.1 Graph of generated x86_64 instructions for a single zero initialization of a vector of varying sizes ......................................................... 35
6.2 Graph of generated x86_64 instructions for two loads, one operation and one store, with varying vector sizes ................................. 37
6.3 Graph of generated x86_64 instructions showing the difference a single operation makes ................................................................. 38
6.4 Graph of execution time in clock cycles for the single operation test .... 40
6.5 Graph of the percentage reduction in execution time for the single operation program with the optimization enabled ............................ 41
6.6 Graph of execution time in clock cycles for the two operation test ....... 42
6.7 Graph of the percentage reduction in execution time for the two operation program with the optimization enabled ............................ 42
Chapter 1

Introduction

LLVM is a modular infrastructure, made up of various compiler and toolchain technologies. For this project, I am interested primarily in the LLVM-core, which provides source and target independent code optimization. The core also provides assembly code generation for various target architectures.

A problem exists in the LLVM code generator, which results in naive assembly code generation for vector operations. Specifically, this is a problem since it results in needlessly large code sizes (plus large executable binary files) and also inefficient use of runtime resources (time and main memory usage).

Since this is a very specific problem in LLVM, there is not much work related to it. As a result, the algorithm described in this report is original, with some inspiration drawn from unrelated algorithms and designs in the world of compilers [9]. The algorithm described is entirely my own work, but should be viewed within the context of the LLVM project as a whole.

My algorithm provides code size, memory usage and execution time optimization for vector operations, which take a specific form. There are certain cases in which it is not safe to perform the optimization. These cases have been accounted for in a set of safety tests which need to be passed before the optimization is actually applied. These tests are examined in section 4.3.

In the remainder of this report, I provide a description of LLVM and its internal representation, a more in-depth description of the problem in LLVM, and information on the design and implementation of my own solution to this problem. An evaluation of my solution is also provided in chapter 6, followed by a short chapter on possible future work and how my implementation could be improved.
Chapter 2

Code representation in LLVM

Internally, LLVM uses its own high-level target independent language to represent code throughout the entire compilation process [1]. This language is commonly referred to as LLVM Intermediate Representation or LLVM-IR for short. It is important to note that there is a difference between IR that LLVM will successfully parse and “well-formed” IR. An example of IR that is not well-formed but still syntactically correct is shown below (taken directly from [1]).

%\text{x} = \text{add i32} 1, %\text{x}

This example is not well-formed since the variable \%x does not dominate all of its uses. In this case, this means that the value of \%x is used before it has been defined. This is a problem since LLVM-IR is in SSA form [11], which is examined later in this chapter. This line of code will be parsed by LLVM successfully, but is not well-formed.

LLVM provides a verification pass, which determines whether or not the input LLVM-IR is well-formed or not. Any LLVM-IR that is not well-formed on output from the LLVM optimizer implies bugs in transformation passes (like the one described later in this report). For the purposes of this project, I will ensure that all LLVM-IR generated by any code I write is well-formed. The IR verification pass will be used to do this.

Code represented as LLVM-IR is made up of one or more basic blocks. In this context, a basic block is a section of code with only one entry point and only one exit point. This means that it is impossible to branch out of or branch into the middle of a basic block. Typically, multiple basic blocks make up a single program that is being compiled. Such a program can be represented graphically using a Control Flow Graph (CFG) [10]. The basic blocks of a program are the nodes of that program’s CFG. The edges of the CFG denote how execution of the program can flow between the blocks.
Figure 2.1: An example of a Control Flow Graph

In the example shown in figure 2.1, there are 4 basic blocks labelled 1, 2, 3 and 4 respectively. This CFG shows that at the end of basic block 1, execution of the program can either flow to basic block 2 or 3. This is denoted by the edges joining the blocks. Which basic block is entered is usually determined by some conditional branch instruction (i.e. go to block 2 if a certain condition is met, to block 3 otherwise). The CFG also shows that once block 2 has finished executing, the program will move to the start of block 4. The same can be said for block 3.

LLVM-IR is in Static Single Assignment (SSA) form. Any code that is in SSA form has two distinct properties [2]:

1. Every variable in the code is assigned to by exactly one static instruction.

2. There exists $\phi$-functions that “choose” the value of a variable based on which control flow edge is incoming to a basic block.

In figure 2.2 there is an example of a simple program that has been converted to SSA form. In order to ensure the first property outlined above is kept, each variable needs to be renamed in the program. For example, in BB1 each assignment to the variable a is assigned a unique variant of its original name (a1, a2, etc). Similarly, the assignments to the variable b in BB2 and BB3 have been renamed.

However, this renaming presents a problem in BB4. The value of b is used in the assignment to c. With the renaming of the variables in BB2 and BB3 we need a way to choose between the two, depending on which execution path was taken through the CFG.
This is where the $\phi$-functions outlined in property 2 come in. In the new BB4, we insert the $\phi$-function which assigns to a new variable $b_3$. This function simply states that if the program is branching from BB2, then $b_3$ is assigned the value of $b_1$. Similarly, if the program is branching from BB3, then $b_3$ is assigned the value of $b_2$. In LLVM, $\phi$-functions are referred to as phi-nodes. This is the name that I will use to refer to these functions throughout the remainder of this report.
Figure 2.3: The example in Figure 2.2 represented as LLVM-IR.

Figure 2.3 shows the example in Figure 2.2 represented as LLVM-IR. Note that the variables used are named slightly differently than in Figure 2.2. When naming variables, LLVM uses the notation %number starting at 1 (i.e. %1, %2, %3, etc). Variables are named in this way in the order that they are encountered in the program. The same naming scheme is used for both variables and labels. You will notice that the last variable assigned to in the first basic block is %5. The next thing encountered in the program is the label of the basic block on the left most branch from this block. As a result, this block receives the label 6. This naming scheme ensures that there is no overlap between the names of labels and the names of variables.
Chapter 3

Problem: Vector instruction selection in LLVM

Within the context of this project, vectors are a form Single Instruction Multiple Data (SIMD) computing. This means that vector instructions provide a way of performing a single operation on multiple input data streams.

In figure 3.1, there is a simple example of a vector add instruction. The contents of the first vector are added to the contents of the second vector in the way shown to form a new vector.

In LLVM-IR, variables with a vector type can be created with any scalar type (int,
float, etc) and with any width greater than one. For example, the following instruction allocates memory space for a vector of four 32-bit integer values (like the vectors shown in figure 3.1).

\[
\%\text{test} = \text{alloca} \langle 4 \times \text{i32} \rangle, \text{align 16}
\]

As mentioned earlier, these vectors can be of any width that is greater than 1. As vectors become larger and larger, we can see a problem develop with the code generation for the associated instructions. The following example of LLVM-IR demonstrates this problem.

\[
\begin{align*}
\%2 &= \text{load} \langle 128 \times \text{i32} \rangle \%a, \text{align 512} \\
\%3 &= \text{load} \langle 128 \times \text{i32} \rangle \%b, \text{align 512} \\
\%4 &= \text{add} \langle 128 \times \text{i32} \rangle \%2, \%3 \\
\text{store} \langle 128 \times \text{i32} \rangle \%4, \langle 128 \times \text{i32} \rangle \%c, \text{align 512}
\end{align*}
\]

This is very straight-forward code. The program loads two vectors of 128 32-bit integers, called \%a and \%b respectively, adds them together, and finally stores the result (\%4) into \%c. The problem arises when we try to generate assembly code for our target machine. In this instance, we will try to generate code for an x86_64 target with a native vector width of 128-bits.

The code generator sees the pattern in the above code and tries to generate assembly for it, starting off by loading the two operands of the add instruction into registers from memory. So we get this assembly:

\[
\begin{align*}
\text{movaps} &\ 1536(\%\text{rsp}), \%\text{xmm1} \\
\text{movaps} &\ 1552(\%\text{rsp}), \%\text{xmm2} \\
\text{movaps} &\ 1568(\%\text{rsp}), \%\text{xmm3} \\
\text{movaps} &\ 1584(\%\text{rsp}), \%\text{xmm4} \\
\text{movaps} &\ 2048(\%\text{rsp}), \%\text{xmm5} \\
\text{movaps} &\ 2064(\%\text{rsp}), \%\text{xmm6} \\
\text{movaps} &\ 2080(\%\text{rsp}), \%\text{xmm7} \\
\text{movaps} &\ 2096(\%\text{rsp}), \%\text{xmm8} \\
\text{movaps} &\ 1600(\%\text{rsp}), \%\text{xmm9} \\
\text{movaps} &\ 2112(\%\text{rsp}), \%\text{xmm10} \\
\text{movaps} &\ 1616(\%\text{rsp}), \%\text{xmm11} \\
\text{movaps} &\ 2128(\%\text{rsp}), \%\text{xmm12} \\
\text{movaps} &\ 1632(\%\text{rsp}), \%\text{xmm13}
\end{align*}
\]
In this section of code, we can see that the code generator is attempting to load the two operands in their entirety into registers. Since the target machine vector width is 128-bits, it is trying to load each operand four integers at a time. The generator gets as far as loading 64 (16 * 4) of the 256 values (the 2 \(128 \times 32\) vectors) that are required from memory. So, at this point the code generator has no choice but to spill the values it has already loaded from memory onto the stack (in a completely new location).

Once it has spilled that register to memory, it attempts to load the next part of the operands from memory. Once again it has run out of registers and needs to spill one to memory. It does this so it can load the next part of the operands from memory. This cycle of spilling and loading continues until it has reached the end of both operands. At this point the code generator has reached the end of the two load instructions and can begin adding the values together.

Now the problem is that the operands that are required were spilled onto the stack just after they were loaded. As a result, the code generator needs to reload the values as they are needed, before they can be used. So the program ends up reloading all the values that
it has just stored into memory.

This means that when this code is running, the machine has to spill registers 49 times (64 - 15) and reload those values as needed. This equates to 98 potentially unnecessary memory writes and reads. Also, with this simple example if we were to increase the width of the vectors involved, the number of spills and reloads would also increase.

The number of instructions generated is also far greater than it needs to be. For large vectors like this, it could potentially be better (in terms of code size) to perform the operation as part of a loop, rather than generating vast sections of code.

Figure 3.2: A demonstration of how the number of generated instructions increases with vector width

In figure 3.2, the generated code size of three different programs is plotted. The blue line represents a program which simply initialises every element in a single vector to zero. The red line represents a program like the example outlined above, with two loads of vectors from memory, a single operation on the values loaded, and then a store back to a third memory location. The final line represents a program similar to the red line, except it performs two operations on the loaded values, rather than just one.

Clearly in all three programs, as the width of the vectors increases so too does the number of instructions generated. As well as this, we can see that as the number of vector
operations in the program increases, the number of generated instructions increases dra-
matically as well. For example, the largest vector width in this data set is 8192 (integers).
The LLVM-IR for the one operation program with this width is:

%2 = load <8192 x i32>* %a, align 16
%3 = load <8192 x i32>* %b, align 16
%4 = mul <8192 x i32> %2, %3
store <8192 x i32> %4, <1024 x i32>* %c, align 16

and for the two operation program:

%2 = load <8192 x i32>* %a, align 16
%3 = load <8192 x i32>* %b, align 16
%4 = mul <8192 x i32> %2, %3
%5 = add <8192 x i32> %4, %2
store <8192 x i32> %5, <1024 x i32>* %c, align 16

With this vector width, the number of generated instructions for the one operation
program is 10,212, and for the two operation program this value is 14,291. This is a massive
difference in code size, especially when one considers that the only difference between the
first program and the second shown above is just one add instruction.

This kind of naive code generation can be very undesirable when compiling software
for any system in which resources like main system memory and storage space are scarce
(e.g. embedded systems). In these cases, the generation of a compact loop which does not
dramatically increase the size of the generated code is much more desirable.
Chapter 4

Solution design

In order to alleviate the problems outlined in chapter 3, I have designed an algorithm to transform large vector instructions into compact loops. Figure 4.1 gives a sample input and expected output of the transformation in LLVM-IR.

![Figure 4.1: Sample input and output of the optimization pass](image)

The basic block on the left of figure 4.1 shows a simple example of the kind of input we
can expect into this algorithm. Specifically, we want to focus on the four vector instructions near the end of the block. These four instructions are two load instructions, a vector multiply and finally a store instruction.

The algorithm described in this chapter is capable of taking these four instructions and converting them into a loop, like the one shown on the right of figure 4.1. This algorithm was written with LLVM specifically in mind. At points, it relies on specific constructs available in LLVM.

### 4.1 My algorithm

The following is a pseudo-code representation of my algorithm for the transformation:

```plaintext
for each function F in the program {
    change_made = true
    while change_made {
        change_made = false
        instruction_lists = generate_lists(F)
        move_lists = []
        for each list l in instruction_lists {
            if movement_safe(l)
                append l to move_lists
        }
        if move_lists is not empty {
            for each list l in move_lists {
                change_made = insert_loop(F, l)
                if change_made
                    break
            }
        }
    }
}
```

From line 1 of the algorithm, we can see that it operates on a function by function basis. This means that if there are any vector instructions in a function whose result is used outside the scope of that function, the transformation simply is not possible. This kind of transformation is well beyond the scope of this project.

The main body of the algorithm is executed inside a while loop. This loop exists so that the transformation will be performed as many times as necessary before the code reaches
a stable point (i.e. there are no more vector instructions on which the transformation is possible). Doing this ensures that all vector instructions that can be transformed have been transformed.

The first thing that needs to be done is generate a set of lists of vector instructions (line 5). These lists are formed based on the inter-dependencies between instructions in the current function. It is important to group instructions in this way since it is unsafe to change the type of the result of an instruction if all uses of that result are not also transformed as well. Doing so without transforming all uses of the result could bring about changing the semantics of the program being compiled or cause the compiler to raise a type error and exit. The procedure used to generate these lists is examined in section 4.2.

Once the lists of instructions have been formed, we need to check if it is safe to move all the instructions in each list to a single point in the program (lines 7-11). This safety check is performed on a list by list basis. All lists on which it is possible to move to a single point in the program are appended to move lists. We only attempt to perform the transformation on the lists of instructions in move lists. The procedure used to check the safety of the transformation is examined in section 4.3.

If there are any lists of instructions in this function which it is safe to transform, then we attempt to perform the transformation (lines 13-19). It is important to note here that we only perform the transformation on a single list. If the transformation has been successfully performed on a list then we must break out of the loop on line 17 and move back around to line 4 to begin a whole new iteration. This is the case since performing the transformation changes the structure of the CFG. Changing the structure in this way could mean that a previously safe transformation is no longer so. As such, it is much safer to generate a new set of lists and check the safety of each one again. A description of the procedure used to perform the transformation can be found in section 4.4.
4.2 List generation procedure

The procedure listed on the next page is used to generate lists of vector instructions that are found in a function $F$. This procedure is used as part of the main algorithm described in section 4.1. The output of this procedure is a set of lists of instructions. Each list of instructions generated is a potential candidate for the transformation described in this report. The transformation may be performed on a list if it meets the safety criteria described in section 4.3.

Every function $F$ that is passed to this procedure is made up of one or more basic blocks. These basic blocks should be iterated over (line 3) in a top-down, breadth-first manner [11]. This ensures that any use of a variable is seen by the procedure strictly after its definition. This will be important when it comes to performing the transformation at a later stage. It is possible that there will be edges in the CFG that flow back up to the top of the graph. In these instances, there can exist a use of a variable that is only defined later in the program (i.e. at a point that has not been parsed yet). It is OK for these uses to appear before their definitions (for this optimization) since such uses are guaranteed to be phi-nodes, which cannot be transformed. This is examined in greater detail in section 4.3.

This procedure iterates over every instruction $i$ in the function $F$ on a basic block by basic block basis (lines 3-4). Every single vector instruction that exists in $F$ must be added to an instruction list. First of all, we check if the result of the instruction $i$ is a vector (line 6). If it is, then we must add $i$ to an instruction list. If the result of $i$ is used by an instruction in any of the pre-existing instruction lists, then we simply append $i$ to that list (lines 8-12). Otherwise, we create a new instruction list and append $i$ to it (lines 14-19).

If the result of $i$ is not a vector, then we must check each of its operands to see if any of them are vectors (line 22). If there is an operand of $i$ that is a vector, we search the pre-existing instruction lists for the definition of that vector operand. If the defining instruction for that operand is found in a list, then we append $i$ to that list (lines 24-28). Otherwise, we create a new instruction list and append $i$ to it (lines 30-34).
generate_lists(Function F) {
    instruction_lists = []
    for each basic block BB in F {
        for each instruction i in BB {
            added = false
            if the result of i is a vector {
                for each list l in instruction_lists {
                    if the result of i is used in l {
                        add i to l
                        added = true
                        break
                    }
                }
                if not added {
                    create a new instruction list new_l
                    append i to new_l
                    append new_l to instruction_lists
                    added = true
                }
            }
            if not added {
                if any operand of i is a vector {
                    for each list l in instruction_lists {
                        if any operand of i is defined in l {
                            add i to l
                            added = true
                            break
                        }
                    }
                    if not added {
                        create a new instruction list new_l
                        append i to new_l
                        append new_l to instruction_lists
                    }
                }
            }
        }
    }
    return consolidate_lists(instruction_lists)
}
This approach to generating lists of instructions does not always produce complete lists of instructions. By this, I mean that it is possible that there are two (or more) lists that have inter-dependencies. As a result of this, we need to do a final pass over all of the instruction lists and merge any lists that have inter-dependencies (line 39). This function simply states that if any instruction in a list $l$ is dependent on any instruction in a list $m$, and $l$ is not $m$ then we must merge $l$ and $m$ “appropriately”. Here, “appropriately” means that the lists must be merged in a way such that the original order of the instructions in the program is maintained in the new list. A pseudo-code representation of this function is shown below.

```
consolidate_lists(instruction_lists lists) {
    for each list $l$ in lists {
        for each list $m$ in lists {
            if $l$ is not $m$ {
                for each instruction $li$ in $l$ {
                    for each instruction $mi$ in $m$ {
                        if $li$ is dependent on $mi$ or $mi$ is dependent on $li$ {
                            join $l$ and $m$ appropriately
                        }
                    }
                }
            }
        }
    }
}
```

It is important that these lists are merged such that the instructions in both lists are ordered correctly within the new list. This means that it is not sufficient to simply append one list onto another.

For example, take the following snippet of LLVM-IR:

```
%2 = load <128 x i32>* %a, align 512
%3 = load <128 x i32>* %b, align 512
%4 = add <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c, align 512
```

When the procedure outlined above is run on this snippet of code, it will produce two lists of instructions. The first list will contain instructions %2 (the first load instruction), %4 (the add instruction) and the final store instruction. The second list will contain just %3 (the second load instruction). This happens since the procedure will sort the instructions into lists in a top-down approach. It will create a new list for %2 (since no others exist)
and then create a new list for %3, since there are no inter-dependencies between %2 and %3.

However, as you can see both %2 and %3 are used by the add instruction %4. Because of this, we need to merge the two lists such that the original four instructions are in the order they appear in the original program. If we were to simply append the second list to the first, we would get the following code in the one list:

```
%3 = load <128 x i32>* %b, align 512
%4 = add <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c, align 512
%2 = load <128 x i32>* %a, align 512
```

Of course, this is incorrect since it is impossible to use the result of the instruction %2 before it has been evaluated. So, we need to maintain some kind of ordering information for instructions while generating lists. One way of maintaining this ordering is examined in Chapter 5.
4.3 Movement safety procedure

```java
1  movement_safe(instruction_list l) {
2      for each instruction i in l {
3          if the result of i is used outside of l {
4              return false
5          }
6          if i is not a load, store or binary operand instruction {
7              return false
8          }
9          if i is inside a loop and is loop variant {
10             return false
11          }
12          if i is a store instruction {
13             if the source value of i is a constant and is not all zeroes {
14                 return false
15             }
16          }
17      }
18      return true
19  }
```

Once lists of instructions have been generated as described in section 4.2, we must check the lists generated to see if it is safe to perform the transformation on them. In order for the transformation to be deemed safe, a number of criteria must be met. In broad terms, the following are the conditions under which the transformation is deemed to be unsafe:

1. The result of an instruction in the list is used outside of the list (line 3)
2. An instruction in the list is not a load, store or binary operand instruction (line 6)
3. An instruction in the list is inside the body of a loop and is variant to that loop (line 9)
4. An instruction in the list is a store instruction with a non-zero constant source value (lines 12-13)

4.3.1 The result of an instruction is used outside of its list

When we perform the transformation on a list of instructions, the original instructions are removed from the program and replaced with instructions of a new vector type (the target
architecture’s native vector width). Once this change in type has been performed, any remaining uses of the removed instructions will no longer be able to be executed. This is the case since those instructions expect an operand of a very specific type. If we pass it an operand of the converted type, either a type error will be raised by the compiler or it will simply crash.

Clearly, performing the transformation in this instance will also change the semantics of the program being compiled (since we are only transforming some but not all uses of a variable). This is an incorrect thing for a compiler to do, and as such we should not attempt to perform the transformation in this case.

4.3.2 An instruction is not a load, store or binary operand instruction

This condition comes directly out of LLVM’s class hierarchy for representing Instructions [3]. There is only a small subset of the types of instructions that LLVM can represent that we can safely transform in the way described. Specifically, we can only transform three types of instructions, which are LoadInst, StoreInst and BinaryOperator. It is possible that other kinds of instructions could be accounted for by the algorithm, but these would require a different kind of transformation to be performed. Some of these possibilities are examined in Chapter 7.

This condition protects against a number of things. First of all, it prevents the incorrect handling of specific kinds of instructions. For example, it is not possible to transform Intrinsic instructions with this algorithm since they can have very specific semantics, defined by the target architecture. Similarly, it would be unwise to attempt to transform instructions like casts and memory allocations since these instructions do not necessarily translate directly into target assembly code and have very specific semantics.

The second (and possibly most important) thing this condition protects against is moving instructions onto execution paths they did not previously exist on. Take the section of a CFG shown in figure 4.2 (on the next page) as an example.

In this example, there are two execution paths through this particular part of the CFG. Execution can either flow through the basic block labelled 4 and into the block labelled 12, or it can flow through the block labelled 8 and then into 12. This kind of structure presents a problem for my transformation. As you can see, the result of the vector instructions %7 and %11 are used in the phi-node %13. The result of %13 is subsequently used in the proceeding store instruction.
With the list generation procedure described in section 4.2, the following list of instructions would be generated:

\[
\begin{align*}
%5 &= \text{load } <128 \times i32>* %a, \text{ align } 512 \\
%6 &= \text{load } <128 \times i32>* %b, \text{ align } 512 \\
%7 &= \text{mul } <128 \times i32> %5, %6 \\
%9 &= \text{load } <128 \times i32>* %a, \text{ align } 512 \\
%10 &= \text{load } <128 \times i32>* %b, \text{ align } 512 \\
%11 &= \text{add } <128 \times i32> %9, %10 \\
%13 &= \text{phi } <128 \times i32> [ %7, %4 ], [ %11, %8 ] \\
\text{store } <128 \times i32> %13, <128 \times i32>* %b, \text{ align } 512
\end{align*}
\]

The algorithm would then attempt to transform these instructions into a loop at a single point in the program. Specifically, this point is the start of the basic block labelled 12. Movement of all these instructions to this specific point completely changes the semantics.
of the program being compiled and will result in unexpected behaviour for whoever is running this code. Specifically, the two sets of instructions in the blocks labelled 4 and 8 respectively will be moved onto execution paths in the program that they did not previously exist on.

The safety condition described in this section prevents the compiler from attempting to perform the loop transformation on this list of instructions. Since a phi-node is not a store, load or binary operand instruction in LLVM, this list will be rejected from the transformation and the code will be left as-is.

4.3.3 An instruction is inside the body of a loop and is loop variant

It can occur that a vector instruction is used as part of a loop, and that the output of the instruction is dependent on some loop iterator. In these instances it is impossible to move the instruction out of the loop, or to transform its type (at least not within the parameters defined in this algorithm). As a consequence, any lists of instructions that contain a loop variant vector instruction such as this are rejected by this check condition and the transformation is not attempted on that list.

4.3.4 An instruction in the list is a store instruction with a non-zero constant source value

Sometimes the value operand of a store instruction can be a constant (for example, in the case of initialization). In such cases, it is only safe to perform the transformation if the constant vector involved has the same value in every field of the vector. If there are different values in each field of the vector, then it is unsafe to perform the transformation. The transformation is not applicable in these cases since it is not possible to turn these store instructions into a loop since a different constant would be required on each iteration.

Due to restraints in LLVM, I was only able to find a way to check if all the values in the vector constant are zero or not. As a result, this condition will only pass for a store instruction with a constant source value if that constant is an all zero vector. It is possible that there is a way to check for other values in such constants within LLVM, but I was unable to find any with the time constraints of the project.
4.4 Insert loop procedure

The simplest way to describe how this part of the algorithm works is to go through an example step by step. For this, we will use the example shown in figure 4.1.

Figure 4.3 shows the starting point of the CFG of our example. The instruction list passed to this procedure (after having passed all safety checks) is the following:

```assembly
%1 = alloca i32, align 4
%a = alloca <128 x i32>, align 64
%b = alloca <128 x i32>, align 64
%c = alloca <128 x i32>, align 64
store i32 0, i32* %1
%2 = load <128 x i32>* %a, align 64
%3 = load <128 x i32>* %b, align 64
%4 = mul <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c, align 64
ret i32 0
```

Figure 4.3: Insert loop procedure example, input CFG

Figure 4.3 shows the starting point of the CFG of our example. The instruction list passed to this procedure (after having passed all safety checks) is the following:

```assembly
%2 = load <128 x i32>* %a, align 64
%3 = load <128 x i32>* %b, align 64
%4 = mul <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c, align 64
ret i32 0
```

Before attempting to perform the transformation, we must first check if these instructions have a vector width greater than the target architecture’s native vector width. For this example, let us assume we are compiling this code for a machine with a 128-bit vector unit. This means that the vector width to check against in this example is \(\langle 4 \times i32 \rangle\). 128 is greater than 4 so we may continue with the transformation. In the event that the instructions in the list had a vector width less than or equal to 4, we would have simply returned false, exiting from the procedure. At that point, the main algorithm would simply take the next available instruction list and pass it into this procedure.

The first step in performing this operation is to split the basic block that is our input in half, at the point where we are going to insert the loop. In this example, that point is the point in the program of the last instruction in the input list. While we do this, we will also remove all instructions in the list from their respective points in the CFG. As a result, we end up with a CFG that looks like that shown in figure 4.4.
Next, we must initialize our loop iterator at the end of the first (original) basic block. This is done by inserting an instruction to allocate space on the stack for a 32-bit integer value, and another instruction to store zero to that location. After this, we also create two completely new basic blocks and insert them into the CFG. The first of these blocks will be used for the loop iterator check condition (at the start of each loop iteration), and the other as the main body of our loop. Once this is done, the CFG should look like that shown in figure 4.5.

The next step in the process is to add in the loop iterator check condition into our
first new basic block. First we must insert a load instruction to load the loop iterator from memory. We can then insert a compare instruction, to perform an integer “equals to” comparison. In this example, we want to check when the iterator reaches 32 (128/4). Finally, we need to insert a conditional branch instruction which uses the result of the new comparison. If the comparison returns true, then we want to branch out of the loop and move onto the start of the second half of the original block. Otherwise, we need to branch to the loop body and perform the next iteration. Once this is done, the CFG should look like that shown in figure 4.6.

![Figure 4.6: Insert loop procedure example, inserting the loop iterator check condition and branch instruction](image)

After this, we are ready to insert new vector instructions with the converted types. For this process, we need to maintain a map of old instructions to new instructions. Whenever we encounter a vector operand in an instruction we can simply use the old instruction that generated that operand to index into the map and fetch the new instruction for that operand. Whenever we generate a new instruction which generates some result, we add it to the map with an index of its corresponding old instruction. Since we are guaranteed that all instructions in the list will have all their operands defined at an earlier point in
the list, this map should work for all cases.

For load instructions, we need to generate instructions to cast the original load source to the new type, and then index the pointer using the loop iterator. This means that each original load instruction has three corresponding new instructions (at the IR level). These three instructions are a pointer cast instruction, a “get element pointer” instruction and finally, the load instruction itself.

For store instructions, we need to find the new source value of the store using the map described above. This is only the case if the source value of the original store instruction is a variable. In the case of a constant source value, we need to convert the type of the value to the new vector type. This is only possible if the constant vector has the value 0 in all fields (as described in section 4.3.4).

Similar to load instructions, we also need to generate pointer cast and “get element pointer” instructions for the destination operand of every store instructions. This is done in the same manner as described for load instructions.

Finally, for binary operand instructions (i.e. all other instructions) we need to generate a new instruction with the same operator as the original instruction. We do however need to fetch all vector source operands from the instructions map, as described previously. Once the new instruction has been inserted, we need to also add it to the instructions map, with an index of its corresponding old instruction. For our example, this process should result in the CFG shown in figure 4.7.

The remainder of this procedure is primarily general house-keeping on the CFG and the loop iterator. First of all, we need to increment the loop iterator at the end of the loop body. This is done simply by inserting an add instruction to the end of the basic block, and a store instruction to store the new value into the iterator’s position on the stack. Then we can insert an unconditional branch instruction back up to the loop iterator check block. We also need to insert an instruction into the end of the original basic block that branches to the loop iterator check block.

For our example, the final state of the CFG is shown in figure 4.8. At this point, the procedure returns true, exiting back to the original algorithm.
Figure 4.7: Insert loop procedure example, after insertion of the converted vector instructions

```
%1 = alloca i32, align 4
%a = alloca <128 x i32>, align 64
%b = alloca <128 x i32>, align 64
%c = alloca <128 x i32>, align 64
store i32 0, i32* %1
%2 = alloca i32
store i32 0, i32* %2

%4 = load i32* %2
%5 = icmp eq i32 %4, 32
br i1 %5, label %16, label %7

False

%7 = bitcast <128 x i32>* %a to <4 x i32>*
%8 = getelementptr <4 x i32>* %7, i32 %6
%9 = load <4 x i32>* %8
%10 = bitcast <128 x i32>* %b to <4 x i32>*
%11 = getelementptr <4 x i32>* %10, i32 %4
%12 = load <4 x i32>* %11
%13 = mul <4 x i32> %9, %12
%14 = bitcast <128 x i32>* %c to <4 x i32>*
%15 = getelementptr <4 x i32>* %14, i32 %4
store <4 x i32> %15, <4 x i32>* %12

True

<label>:16
ret i32 0
```
Figure 4.8: Insert loop procedure example, final state of the CFG
Chapter 5

Solution implementation

In this chapter, I examine particular aspects of my implementation of the algorithm described in chapter 4. This chapter can be used as an introduction to the implementation work that was performed and provide important information related to the C++ source code of the project.

5.1 Optimization passes in LLVM

LLVM provides a framework for implementing optimization passes in its optimizer [4]. Since the main algorithm described in chapter 4 performs the optimization on a function by function basis, I have implemented it using a FunctionPass in LLVM. A FunctionPass is used since each function in the program being compiled can be optimized independently of all others. The PassManager handles invoking my new optimization on each function of the program, one by one.

To implement this, I have simply declared a struct called VectorToLoop which inherits from FunctionPass, which is provided by LLVM. The main body of the algorithm is implemented in the member function runOnFunction, a virtual function in FunctionPass which has been overridden. This is the function that is invoked by PassManager on each function of the program being compiled.

5.2 Data structures

In general, LLVM makes extensive use of the C++ Standard Template Library (STL) [5]. Because of this, I have done the same in my optimization pass. In particular, the pass
primarily uses std::vector [6] and std::map [7] to represent the different data structures used by the algorithm.

Any other data structures used in this implementation are those provided internally by LLVM. These include the likes of BasicBlocks, Instructions, Values, Types, and Users. Each of these are described later in this chapter, as they arise in the implementation of the algorithm.

### 5.3 Algorithm implementation

In the main algorithm, lists of instructions are gathered together, the safety of the transformation is checked for each list, and finally the transformation is attempted on those lists which it is safe to do so. These lists of instructions are implemented as a std::map. The key type of the map is an unsigned integer, and the value type is a pointer to an Instruction. The reason these lists are represented by a map and not an array or a std::vector is that the unsigned integer key is used to provide a form of ordering to the instructions in lists. Each instruction that is encountered in the program receives a unique key value. Instructions are assigned a key value in the order they are parsed by the algorithm. These keys provide a way of ordering instructions between multiple lists (rather than the more focused ordering provided by the likes of an array). This ordering is particularly important when it comes to combining lists together, as described in section 4.2. There are other ways in which this ordering of instructions could be provided, one of which is examined in chapter 7.

In order to iterate over the basic blocks in each function and the instructions in each basic block, as described in section 4.1, LLVM provides simple iterators in each class. So, for example the Function::iterator can be used to iterate over each basic block in that function in a straight-forward for loop, like that shown below.

```c++
for (Function::iterator i = F.begin(), e = F.end(); i != e; ++i)
{
    // loop body using basic block i
}
```

Similar code can be written for iterating over the instructions in a basic block using the iterator BasicBlock::iterator.

The majority of the implementation work involves accessing different pieces of information related to instructions, their operands and value types. This involves working with
Instruction objects[3]. More specifically, this involves working with the various classes that inherit from the Instruction class and that it inherits from.

In order to access these various sub and super classes of Instruction, LLVM provides three functions: isa, cast and dyn_cast [8]. My implementation primarily makes use of dyn_cast. This function provides a way to easily cast an instantiation of a particular class into one of that class’s sub or super classes. It first checks if the cast is possible. If it is possible, then dyn_cast returns an object of the desired type, otherwise it returns NULL.

There are two classes in particular that I cast Instruction objects to in my implementation. The first of these classes is the Value class. Casting an Instruction to a Value gives the Value object that represents the result of that particular Instruction. With this Value object, it is then possible to check the type of the result of the Instruction, and also find all uses of that Value (amongst many other things).

The other class that I cast Instructions to is the User class. The User class provides a way of accessing the different operands of an Instruction. These operands are accessed as Value objects. This is particularly useful when performing the actual transformation itself since these Value objects can be cast to Instructions to quickly find their defining instructions in the program. Both the Value and User classes are super classes of the Instruction class.

In the next chapter, there is a description of how this implementation was tested. The results of these tests are also examined and evaluated in the chapter.
Chapter 6

Evaluation

In this chapter, I describe the approach I took to testing and benchmarking the transformation described in this report. I also present and describe the results of this testing.

6.1 Safety tests

For this type of project it is typical to use the existing LLVM test framework to ensure that the code transformations involved are safe and do not negatively affect the proper code generation of the compiler. I did run this optimization through the testing framework and it did not affect the pass rate of the near 10,000 tests in the framework. However, given the unique nature of the optimization this is almost to be expected. All of the vector operation tests in the LLVM test framework are already the correct size for the target machine. As a result, running this optimization through the framework only ensures that the optimization is not performed on vectors of the correct size. Of course, this is an important thing to check. However, it does not ensure that the optimization is performed correctly and without error for all cases covered by the algorithm. It also does not ensure that the optimization is not performed in cases in which it is unsafe to do so.

I have written 60 tests to check for these properties. Specifically, these test cases were written to check that the optimization is performed correctly for:

- Vectors of varying widths
- Vectors with varying scalar types
- Programs with differently structured CFGs
These tests also ensure that the optimization is not performed in all the cases outlined in section 4.3.

All 60 tests that were written for this project execute, and behave correctly when compiled with and without my optimization enabled. More importantly, the programs behave in the exact same manner in both cases. While this kind of testing does not guarantee that the optimization will work correctly in 100% of cases, it does ensure that the optimization is performed safely and correctly in the set of tests provided. These tests were written in an attempt to comprehensively cover all known cases in which the optimization should and should not be performed. The tests used are available on the CD included with this report.

6.2 Benchmark tests

I have also written a benchmarking tool to test the actual performance of the optimization. The benchmarking tool measures two things:

- The relative generated code sizes of vector operations compiled with and without my optimization
- The relative execution times of vector operations compiled with and without my optimization

6.2.1 Generated code size

I created very small, and specific test cases in order to benchmark the generated code size. The programs used are not real-world examples, but were written to demonstrate the capabilities of the optimization in terms of generated code size.

The benchmarking tool I wrote was run on two different machines, with processors with two different native vector sizes. The two processors involved were an Intel Xeon E7-4820 with a 128 bit vector unit (SSE 4.1/4.2) and an Intel Core i7-4700MQ with a 256 bit vector unit (AVX 2.0).

Single vector initialization to zero

The first set of results was generated for a single zero initialization of a vector. The LLVM-IR used is:
%1 = alloca i32, align 4
%a = alloca <128 x i32>, align 32
store <128 x i32> zeroinitializer, <128 x i32>* %a, align 32
store i32 0, i32* %1
ret i32 0

The test was run with varying vector widths (128 x i32 shown) from 4 to 2048 (multiples of i32) for the 128 bit vector unit and vector widths between 8 and 2048 for the 256 bit vector unit. Figure 6.1 is a graph of a subset of this data (from 4 up to 256).

Figure 6.1: Graph of generated x86_64 instructions for a single zero initialization of a vector of varying sizes

In figure 6.1, we can see that for the smallest vector sizes the generated code size is the same with and without the optimization enabled. This is because the vectors involved are already at the native machine width and as a result the optimization simply ignores them, since there is nothing to do. Clearly, the optimization is not beneficial for code size for initializing small vectors. This is because vectors of these sizes do not suffer from the problems outlined in chapter 3 since there are enough registers to perform the operation in one go.

We can see that for the 128 bit target, the generated code size is the same with and without the optimization when the vector involved is 68 x i32. This value is the point at
which there are one too few registers to perform the operation. The target has 16 registers, each capable of taking 4 x i32. At a vector size of 68 x i32, we would need 17 registers to perform this operation without any spills (and reloads). Beyond this point, it becomes beneficial (in terms of code size) to enable the optimization. The upward trend seen in the code size without the optimization enabled is maintained for the full set of data up to 2048 x i32 vectors. In this test case, every additional 4 elements added to the vector translates to one extra line of code without the optimization enabled. Similar results can be seen for the 256 bit target, except instead of the increase occurring every 4 elements, it occurs every 8 (since 256 bit registers can store 8 x i32).

Two loads, one operation, one store

A test case was specifically generated to benchmark the capabilities of my optimization in this case. This test case was generated from the following C source using Clang (a C front end for LLVM).

```c
// declare our vector of 512 bytes => 128 * int
typedef int v128int __attribute__((vector_size (512)));

int main()
{
    v128int a, b, c;
    c = a * b;
    return 0;
}
```

The following LLVM-IR was generated from the C source for this benchmark:

```
%1 = alloca i32 , align 4
%a = alloca <128 x i32>, align 32
%b = alloca <128 x i32>, align 32
%c = alloca <128 x i32>, align 32
store i32 0 , i32* %1
%2 = load <128 x i32>* %a , align 32
%3 = load <128 x i32>* %b , align 32
%4 = mul <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c , align 32
```
Once again, this is not a real-world example, it is a test case generated specifically to benchmark the capabilities of my optimization. In this case, we are specifically interested in the code generated for the section of vector instructions which is made up of the two load instructions (%2 and %3), the single operation (%4) and the subsequent store. Figure 6.2 shows a subset of the results for the code generation of the above LLVM-IR.

![Graph of generated x86_64 instructions for two loads, one operation and one store, with varying vector sizes](image)

Figure 6.2: Graph of generated x86_64 instructions for two loads, one operation and one store, with varying vector sizes

In figure 6.2, we can see that just like the results for the single zero initialization, there is a clear trend in the number of generated instructions without the optimization enabled. Once the vector widths reach the size 68 x i32 (for the 128 bit target) and 120 x i32 (for the 256 bit target), there is a constant increase of 10 lines of generated assembly for every increase by the size of the machine’s native vector width (either 4 or 8 x i32). For example, on the 128 bit vector target increasing the vector width from 124 to 128 x i32 results in an additional 10 lines of generated x86_64 assembly code in this test case.

We can also see in figure 6.2, that the point at which it is beneficial to apply the optimization is much lower than in the case of the single zero initialization. This is because this test case requires the use of two registers for every 4 x i32 operation being performed, whereas the previous test case only required the one. Indeed, if we add an additional level
of complexity to this example (by adding an extra operation before the store instruction) we get the graph shown in figure 6.3.

![Graph of generated x86_64 instructions showing the difference a single operation makes](image)

Figure 6.3: Graph of generated x86_64 instructions showing the difference a single operation makes

Figure 6.3 shows the number of generated x86_64 assembly instructions for the 256 bit target, with a single additional vector multiplication operation included before the store instruction in the example on the previous page. With the one operation, the code size increased by 10 lines as described on the previous page. However, by adding just one more operation to the LLVM-IR, the code increases by 14 lines for every increase (8 x i32 increase in this case) in the vector size. However, the generated code size with the optimization only increases by 1 line for all vector sizes. This one line is simply the extra instruction that has been added to the code.

From this, it is clear that the optimized version of the code increases at a far lower rate than without the optimization, as the number of operations involved increases. This is a very desirable feature of the optimization, since real-world examples are likely to be a lot more complicated than the test cases used for benchmarking.
6.2.2 Execution time

The benchmarking tool that I have written uses the last two tests from the previous section (i.e. the one and two operation tests) in order to gather statistics on the effect my optimization has on the execution time of the generated assembly code.

Each test is run 10,000 times, the execution time of each run is recorded in an array. The array is sorted after the 10,000th run and the median value is extracted from the array. This helps to prevent the statistics from being affected by outside factors (e.g. the program being descheduled by the Operating System). A hardware counter is used to gather accurate execution time information. These tests were run on an Intel Core i7-4700MQ with a 256 bit vector unit (AVX 2.0). The x86_64 instruction set provides an instruction called RDTSC, which returns the number of clock cycles since the counter was last reset. This instruction is used in conjunction with the reset instruction to get the execution time of the tests in terms of clock cycles.

As mentioned above, the following LLVM-IR was used to benchmark the execution time of generated code:

```llvm
%1 = alloca i32, align 4
%a = alloca <128 x i32>, align 32
%b = alloca <128 x i32>, align 32
%c = alloca <128 x i32>, align 32
store i32 0, i32* %1
%2 = load <128 x i32>* %a, align 32
%3 = load <128 x i32>* %b, align 32
%4 = mul <128 x i32> %2, %3
store <128 x i32> %4, <128 x i32>* %c, align 32
ret i32 0
```

Figure 6.4 shows the results of the benchmark tests for the test above, with a single operation.

In figure 6.4, we can see that for small sized vectors the execution time statistics are very fuzzy. It is very difficult to get accurate execution time results (particularly in terms of clock cycles) for these tests since they take such a small amount of time to execute. This is because outside factors (such as OS descheduling) have a far greater impact on these lower execution times than with the larger vector operations. However, once the vector size hits around 512 elements, the statistics begin to stabilize to a point that we can analyse...
Figure 6.4: Graph of execution time in clock cycles for the single operation test

There is a clear difference in the execution times of the code generated with and without my optimization enabled. Indeed, if we examine the differences in the execution time with respect to the execution time without the optimization enabled, we can get the reduction in execution time as a percentage of the original. So, in figure 6.5 we can see the percentage reduction in execution time with my optimization enabled. For this graph, the data has been split up into sections of 64 items each. This means the vector sizes are grouped into sections of 512 elements each. Each point plotted on the graph is the average of each section of data.

As we can see, once the vector sizes reach 1024 x i32, there is at least a 5% reduction in the execution time of the generated code. Once the vector sizes reach 2048 x i32, there is at least a 10% reduction in the execution time. For the range tested, the reduction in execution time is at its highest for 8196 x i32, with a 15.67% reduction in execution time.
We can see very similar results when we increase the complexity of the program being tested slightly, by including a single extra vector instruction before the store instruction. In figure 6.6, we can see that the execution time of the program is on average faster when compiled using my optimization.

And again, if we examine the reduction in execution time as a percentage we get very similar results to before. In figure 6.7, you can see these reductions graphed in the same fashion as before. Here, the reduction in execution time hits its peak for the 6736-7240 range, at 17.07%.
Figure 6.6: Graph of execution time in clock cycles for the two operation test

Figure 6.7: Graph of the percentage reduction in execution time for the two operation program with the optimization enabled
Chapter 7

Future Work

In this chapter, I will examine some possible improvements that can be made to the work described in this report and also ways that the optimization pass can be expanded upon in the future.

7.1 Transform more kinds of instructions

As I mentioned in section 4.3.2, my algorithm could be modified to account for instructions that are not load, store or binary operand instructions. The trouble with doing this is that other instructions would have to be modified in a different manner to the loop transformation.

For example, take InsertElementInst and ExtractElementInst. The first of these instructions is used to insert a scalar value into some arbitrary position of a vector. The kind of LLVM-IR that is typically generated (by Clang, the C programming language front end for LLVM for example) with these kinds of instructions is as follows:

```c
%7 = load i32* %i, align 4
%8 = load <128 x i32>* %a, align 512
%9 = insertelement <128 x i32> %8, i32 %7, i32 123
store <128 x i32> %9, <128 x i32>* %a, align 512
```

This example has been extracted from one of the test cases described in chapter 6. This code snippet is taken from the body of a loop, with iterator %i. The vector %a is loaded from memory, the value 123 is inserted into position %7 (the value of %i) of the vector and then the entire vector is stored back into memory.
This code snippet suffers from the same problems described in this report, where the entire vector is loaded from memory, the value is inserted and the entire vector is stored back again. Of course, it would make more sense to simply load the section of the vector from memory, insert the element and then store it back. It might make even more sense to simply index the memory location directly and just store the single value 123 to the appropriate memory location.

The same kind of transformation could be performed for ExtractElementInst. Rather than loading the entire vector from memory in order to extract one element, we could modify the LLVM-IR to simply load that single value from memory.

7.2 Improve storing of constant vectors

As outlined in section 4.3.4, the optimization is not performed if an instruction in the instruction list is a store with a non-zero constant. The implementation of the algorithm could be expanded to include other constant operands to these store instructions. The main problem that would need to be solved with this is properly detecting constant vectors that either have the same value in all fields or contain some pattern throughout. The reason this does not need to be done for all zero constant vectors is because LLVM provides a specific type to identify such vectors.

7.3 Alternative method for ordering instructions

At the moment, instructions are ordered amongst instruction lists using unique integer values assigned to every instruction. This is described fully in section 5.3. This method of ordering relies heavily on the CFG being parsed in a particular fashion. Specifically, a top-down approach is assumed to be taken. If the basic blocks in the CFG are not parsed in this way, then this form of ordering will not work. A much more robust way of providing this ordering is using the use-def chains associated with instructions to order them. This would of course require much more processing of instruction information during the optimization process, but it would be far safer and less prone to failure.
Chapter 8

Conclusion

The aim of this project was to alleviate the problems associated with naive code generation for vector operations in LLVM. Specifically, the main focus was on the size of the generated assembly code, with improved execution speed and memory usage as a secondary goal.

The primary aims of the project have been achieved. With the optimization enabled, there is a significant reduction in the generated code size for many vector operations. In some cases, this reduction was as much as several thousand lines of assembly, which is a significant amount. There is also a reduction in the run time of programs which use vectors equal to or greater than a particular size, or for programs with complex enough vector operations. In the test cases used, this reduction in execution time ranges from 5% to 17%.

The work done in this report could be expanded upon to provide more complex analysis of vector operations and the capabilities of this optimization. There is a great range of future work that can be done to improve upon and supplement my optimization.

In conclusion, the goals of this project were met. The algorithm I have designed and implemented provides significant reductions in both generated code size and the run time of compiled programs.
References


46