BTP : Project Report

# SuperOptimization based LLVM to x86 Compiler

Aseem Garg
2008CS10165

Ujjaval Kumar
2008CS10196

*Supervisor:*
Prof. Sorav Bansal

April 30, 2012


Indian Institute of Technology, Delhi

**Certificate**

This is to certify that the project report titled **SuperOptimization based LLVM to x86 Compiler** being submitted by Aseem Garg, entry number - 2008CS10165 and Ujjaval Kumar, entry number - 2008CS10196 to the Indian Institute of Technology, Delhi, is a result of bonafide work carried out by them under my supervision. The matter in this report has not been submitted to any other University or Institute for the award of any other degree or diploma.

Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016

**Acknowledgement**

We would like to thank our supervisor Prof. Sorav Bansal for his guidance during the course of the project.

Aseem Garg
2008CS10165

Ujjaval Kumar
2008CS10196

# Contents

# 1 Introduction

LLVM is a compiler infrastructure that can be used for various kinds of optimizations on higher level languages to generate a generic intermediate form (LLVM assembly). LLVM has become popular as a preferred intermediate form for many open source and commercial compilers (used by Apple).
In this project we have implemented peephole superoptimization based algorithm for LLVM backend for x86. Peephole superoptimizer based translation has been used for binary translation [3]. We have extended the technique for compilation of intermediate form. In this algorithm, the compiler is provided with a set of translation rules that specify a translation from a sequence of LLVM instructions to equivalent sequence of x86 instructions. Since multiple translations might be possible at any code point, we construct a dynamic programming problem which is subsequently solved to get an optimized translation.
Traditional compiler algorithms require hand-coding the translations from source language to destination assembly. This presents with following problems:

1. Modern architectures have complex instruction sets and provide for large number of optimization possibilities. Targeting all such optimizations can be a significant engineering challenge.

2. For different architecture large parts of the compiler need to be re-written. Hence targeting new architectures can be very time consuming.

Our design tries to address these problems. The first problem is addressed by having an automatic rule generator. The second problem is automatically addressed as application of the rules is not specific to any architecture. However some work needs to be done when generating the assembly as different architectures have different instructions for operations like mov.
The compiler has two parts:

1. An automatic rule generator that takes in as the input, a specification of the destination architecture and a set of training programs and outputs a set of rules that can be used by the translator.

2. A compiler that applies these rules to LLVM programs and generates the desired assembly.

The infrastructure has been designed to target x86 architecture. However it can be easily modified for targeting other architectures. At this point in time, we are targeting a small subset of the full LLVM. This small subset can be used to define a variety of experimental programs that can be compiled to see how does this method compare to the tradition compiler for LLVM. We were able to achieve good results even with the few rules used in this project.
In the second section we discuss the translation rules in detail. In the next section we give an overview of the algorithm. In the fourth section we discuss the scope of the current prototypes. In the next section we discuss the various challenges faced and their solutions or heuristics that have been used to address them. In the sixth section we discuss some of shortcomings of this technique. Next section describes the performance of some small programs compiled using this compiler. Next we discuss the rule generator. Finally we give the related work and conclusion and future work.

## 2 Translation Rules

A translation rule is a pair of equivalent instruction sequences in the source and destination instruction set. A sequence of X86 instructions is equivalent to another sequence of LLVM instructions if the value of variables mapped appropriately in their input and output have same values before and after their execution on arbitrary inputs.

Table 1: Example Rule

| LLVM instruction Sequence | In Map | Out Map | X86 instruction sequence |
|---|---|---|---|
| r0 = icmp slt i32 r1, i32 c0<br>br i1 r0, label l1, label l2 | r1:R0 | r1:R0 | cmpl c0, R0<br>jl l1<br>jmp l2 |

In the above example if r0 is mapped to a register R1 of x86 in the beginning, running the two sequences would place equal values in the variables specified in the out map(r0 mapped to x86 register R1). LLVM registers might be mapped to memory locations that are specified by M1, M2 and so on and constants are represented as generic values like c0, c1 etc. At this point of time rules can not be specified for specific constants. Hence we can not use translations like inc R0 for LLVM instruction r0=add i32 r0, i32 1.
Furthermore since LLVM has infinite register set and translation for various x86 registers would essentially produce same translation, the rules use canonicalized names. LLVM registers are named as r0, r1 etc, constants as c0, c1 and so on. x86 registers are named as R0, R1 and so on, memory locations as M0, M1 and so on, and constants as c0, c1 and so on.

## 3 Translation algorithm

LLVM functions consist of basic blocks which consist of a sequence of instruction. Each basic block has exactly one entry point (top). The last instruction specifies a branch (can be conditional) in other basic blocks of the same function or a return instruction. Hence the basic blocks form a CFG structure.

### 3.1 Register Map Selection within a Basic Block

At each code point we specify maps from the live LLVM registers to x86 locations. These locations can be register or memory locations. For each location there can be many such maps mapping different variables to memory locations or registers. The map doesn't specify the naming of x86 register. Hence for a code point with live variables being x1, x2, x3 the possible maps can be [x1-R, x2-R, x3-R], [x1-R, x2-R, x3-M], [x1-R, x2-M, x3-R], [x1-R, x2-M, x3-M], [x1-M, x2-R, x3-R], [x1-M, x2-R, x3-M], [x1-M, x2-M, x3-R], [x1-M, x2-M, x3-M] constrained by the total number of x86 registers.
Suppose we have a certain map M1 at a code point i, and the rule set has a rule R for translating k instructions starting at point i. The rule might require certain movements of variables from registers to memory and vice versa. This rule generates a final map say M2 at code point i+k. The cost of creating this M2 is given by the cost of starting map (M1 in this case) plus the cost of moving variables from memory to registers and vice versa

plus the cost of the rule.
Now there might be multiple maps at code points i, i+1, .., i+k-1 that might with corresponding rule generate M2 at code point i+k. Hence if we have defined all the maps till code points i-1, we can define all possible maps at code point i with cost being given as
$cost(M2) = Min_{over\ all\ possible\ predecessor\ maps}[cost(M) + cost(movements) + cost(rule)]$
Hence if the maps are defined for starting of a basic block, we can use the above dynamic programming model to specify the maps are subsequent code points. For the entry block, we can assume all variables are mapped to memory locations. For any other block, first we generate all possible R/M maps. These maps are then assigned costs based on the final maps in the predecessor block. For a block having multiple predecessors, we use weighted average of end maps in each predecessor block that gives the minimum cost of reaching the map in this block. This is described in detail in next section.
At this point we are not optimizing across basic blocks.
Working in this fashion, we can find the possible maps at end location and their corresponding costs. Then starting from min cost map at end point, we backtrack to find the min-cost translation.

### 3.2 Target specific code generator

Once the min cost maps are chosen, we specify the mapping to exact register and memory locations (functions: coalace_regisers and coalace_memory). This requires specifying of register names. Finally we output the code by patching the register and memory names in the rules. The rule generator needs to know instructions for moving variables between registers and memory locations. Hence this has to be specific to an architecture.

## 4 Scope of the current prototype

Since the aim of this project was to understand the viability of such a design, the tool has many simplifications that need to be dealt with for using the compiler for larger programs.

1. Presently we are targeting only a small subset of LLVM. In particular we are dealing with only 32 bit integer types and pointers. Other basic types have not been dealt with. Furthermore not all instructions for integer types are translated. Only the instructions occurring in our test program have been translated. At this time derived types have not been dealt with. We can translate programs using malloc or where the array is passed as a pointer from functions compiled outside our translator.
2. As the program sizes grow, there is a need to constrain the number of maps that are saved at each code point. A simple heuristic can be to keep a specific number of min cost maps. However since our training programs are small, this functionality is not implemented.
3. Even with constraining the number of maps, there can be large memory usage. Hence we need to write the maps to file in extreme cases.
4. The translator doesn't deal with multithreaded programs.
5. Conditionals are saved in memory or register locations and not in flag register. A 1 bit conditional takes a 32 bit location with the LSB specifying the value. However

a rule might map an intermediate variable to flag register and use it.

6. We do not allow large rules that might incorporate multiple blocks for across block optimization.

7. The generated assembly is not optimized by assembly specific constructs like placement of blocks etc. Furthermore jmp instructions that might be avoided (due to fall through) are not removed by the assembly generator at this time.

8. At this time there is no way to specify if certain rule requires certain variables to be mapped into specific registers. This might be the case with instructions like mul which requires one of the source operands and the destination use eax register. Similarly call instructions under the present caller saved - callee saved registers convention requires that specifically eax, ecx and edx be caller saved. Hence such instructions can't be used.

These simplifications have allowed for simplification of code. All variables can be memory or register mapped and can be assumed to be 32 bit.

# 5 Other Challenges

Various challenges had to be addressed to include certain intuitive optimizations and other problems.

## 5.1 Infinite Instruction Set

Unlike an assembly corresponding to most architectures, LLVM assembly includes instructions that may not have a fixed form. An example is the call instruction. This instruction can take variable number of arguments. For all such instructions we need to generate a translation. Even by using a large number of training programs we can never exhaustively cover the whole space of call instructions. Similar problems would arise in instructions like getelementpointer, load, store etc. In fact for most LLVM instructions, we would run into such problem. To overcome this problem, we have defined a function single_inst that can generate a translation for a single instruction. The algorithm is then tweaked such that incase we can't find any translation to reach a given code point, this function is called over the instruction at code point i-1. The single_inst function, as is clearly evident, needs to be architecture specific. At this point the function is not complete and caters only to instructions that occur in our test cases and have no translation rules specified. Furthermore the function only caters to forms used in our translation programs.

## 5.2 Phi instruction

Phi instructions can have no corresponding translations in machine assembly. Hence before beginning the DP analysis the phi instructions are removed from beginning of each block and moved into preceding blocks. This destroys the SSA structure of LLVM. However since we are optimizing only within a basic block at this point of time, we can still go on without considering that a variable might be defined at multiple locations. A better way to deal with phi instructions would be to rather put small blocks of code to move values from the end locations of the previous blocks to the locations as required by this block.

This will also save on extra movements needed in the previous method. However due to lack of time in the end, this approach could not be tested.

### 5.3 Caller Saved and Callee Saved Registers

LLVM having an infinite register set, has no concept of caller saved and callee saved registers. However, in x86 (using linux) eax, ecx, edx are caller saved while ebx, ebp, esi, edi are callee saved. To deal with this we assume that at starting of the program 4 registers are busy while 3 are free. Freeing the callee saved registers is done at the beginning of the translation. This is done so as to ensure that freeing doesn't happen inside a loop where it would cost a lot more. Since the 4 busy registers would give equivalent translations independent of which is used, we essentially run the DP with 3, 4, 5, 6 and 7 registers independently and report the translation with minimum cost. For using more than 3 registers the initial map at entry block take a cost of (total_regs-CALLERSAVED) x MEMCOST specifying the freeing of regs in beginning and saving them back at end.

### 5.4 Named Registers

Instructions like call require that specifically eax, ecx and edx be free rather than any three registers. Similarly many string operations require esi and edi specifically rather than any two registers. To incorporate the cost of moving between registers, we would need to use named register maps in our DP analysis. However this would greatly increase the size of our maps. Furthermore, when dealing with loops, there might arise a case where register mapping is reversed. For instance, let a:eax, b:ebx, c:ecx. Now suppose a becomes dead in between hence eax is freed which is then mapped to a variable d. Now ecx is freed and later when a is defined it is mapped to ecx. Finally d becomes dead and c is mapped to eax. Hence we have starting map of a:eax, b:ebx, c:ecx and final map of a:ecx, b:ebx, c:eax. However in our DP analysis both there are represented by a:R, b:R, c:R. Hence there is a zero cost assumed for translating at end before re-entering the loop. However, to constrain the size of no of maps, we are ignoring this cost. The rulegen, however, never generates a rule where a variable mapped to R1 in in-map gets mapped to some other register at end.

### 5.5 Multiple Parents

Suppose in the CFG a block has multiple predecessor blocks. In that case the starting maps should accommodate the cost of all the parents. We do this by taking an average over all parents. Furthermore, a block might have a parent that is part of a loop. Consider the following CFG for example:
In this, when finding the costs for maps in 2, we would somehow want to give more weight to edge $3 \rightarrow 2$ than $1 \rightarrow 2$. To implement this we first identify all the loops (described in next sub-section). For loops, we give a weight of 10 to the corresponding edges. Also, if 2 waits on 3 and 3 waits on 2, we can never complete the translation. Hence in the first run, we leave out the blocks for which analysis is yet to be done. Then a second run is done and for the blocks yet to be updated, costs from last run are used.

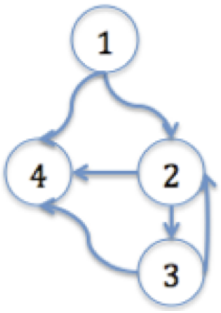


Figure 1: CFG

### 5.6 loops

To identify loops, we use the algorithm described in [1]. Essentially first for each block, it's dominators are defined. A block d dominates n if every path from entry block to n goes through d. Then if there is an edge from n to d then this is a loop. The blocks in this loop are given by ([1])

```
procedure insert (m):
if m is not in loop then begin
loop := loop U {m}
push m onto stack
end

/* main program */

stack := empty
loop := {d}
insert(n)
while stack is not empty do begin
pop m, the first element off the stack
for each predecessor p of m do insert(p)
end
```

This will not identify all the loops in the graph. However if LLVM code is formed from a higher level language that uses constructs like while and for and doesn't use constructs like goto, all the loops will be detected. Although nested loops can also be detected from this, we haven't given them any preferential treatment. We have also kept a weight metric with each block. This is an experimental heuristic. Each cost incurred in a block is multiplied by the weight heuristic. The intuition behind this was that we would like to incur extra cost if such an operation occurs in a loop. The metric would be removed if we don't see any improvement when running the tests.

### 5.7 Choice of map in parent node

It might be possible that while backtracking from end node, we are required to choose conflicting maps by two successors. Consider the CFG given in 1. Suppose we choose map M1 as end map of 4. Hence we choose M2 as starting map of 4 and M2 as end map of 3. Backtracking we choose M3 as starting map of 3. Hence for 2 we have a conflict of

choosing M2 or M3. In such a scenario we choose the map as indicated by the block with higher edge cost (3 in this case).

# 6 Shortcomings of Technique

Peephole superoptimizers based compilation requires a large memory usage. Since a large number of maps are created at each step, there is significant memory usage expected. Furthermore, the compilation time for large programs can be significant. This would also make the technique almost inapplicable for dynamic translation without significant downsizing.
There is no concept of scheduling in the algorithm. Although some amount of scheduling is done by application of rule on multiple instructions, generally rules would not span many LLVM instructions and hence many scheduling based optimizations are lost.
There is no way overcome the problems discussed in section 5.4 without named maps. Hence in some cases the program would take large cost. Other heuristics as discussed in last section might also yield inefficient translation but had to be chosen for lack of better options.

# 7 Results

To test the compiler designed we chose to check the performance of assemblies created for four functions against performance for assemblies created by the LLC tool that is shipped with the LLVM tool chain. Although we wanted to test recursive programs too, however at this time it was not possible due to un-availability of call instruction.
We chose four programs. The tests were performed on the Palasi server at the department of computer science and engineering. All the tests were single threaded. The rules used for analysis were hand written. We believe we can get better results with a larger rule set.
First was a simple tail-recursive implementation of fibonacci numbers. This was mainly needed to test the correctness of code and correctness of generated assembly.
Next was an implementation of the bubblesort algorithm. We were able to observe the following performance on input arrays of various lengths.

Table 2: BubbleSort

| **Array Length** | **Time taken by our tool** | **Time taken by LLC** |
|---|---|---|
| 50000 | 6.64 | 4.24 |
| 60000 | 9.48 | 5.93 |
| 70000 | 12.19 | 8.05 |
| 80000 | 15.99 | 10.52 |
| 90000 | 20.56 | 13.30 |
| 100000 | 24.91 | 16.64 |

We also implemented the insertion sort algorithm. We were able to observe the following performance on input arrays of various lengths. Although there is considerable difference in performance of the assemblies generated by two tools, it should be noted that these algorithms require considerable number of jumps due to the conditionals involved. Since our assembly generator is a simple one, such programs are bound to perform badly due to

Table 3: InsertionSort

| Array Length | Time taken by our tool | Time taken by LLC |
|---|---|---|
| 50000 | 1.31 | 0.65 |
| 60000 | 1.90 | 0.92 |
| 70000 | 2.58 | 1.26 |
| 80000 | 3.38 | 1.65 |
| 90000 | 4.28 | 2.08 |
| 100000 | 5.27 | 2.60 |

no optimization in placement of blocks and no removal of extra jump statements.
Hence we decided to add another test that might have a more input independent structure. For this we used matrix multiplication like algorithm - we are calling it "matrix accumulation". However since we couldn't use the mul statement, the multiply operations were replaced by addition operations. Hence $C_{i,j}$, where $C = A \times B$, is given by $\sum_k A_{i,k} + B_{k,j}$. This was implemented as an $O(n^3)$ algorithms similar to simple matrix multiplication. We were able to observe the following performance for square matrices of various sizes.

Table 4: Matrix Accumulation

| Array Length | Time taken by our tool | Time taken by LLC |
|---|---|---|
| 1000 | 13.6 | 13.48 |
| 2000 | 134.63 | 128.36 |
| 3000 | 580.14 | 578.5 |

As it is evident, the performance of our tool is very close to the LLC tool. Due to nature of the program, the effect of not performing optimizations at assembly level are much less defined. Furthermore, the performance can improve with the use of larger rule set.

# 8 Rule Generator

In an arbitrary instruction set, there are hundreds of instructions, each having a different functionality and constraint. Thus, there exist too many such equivalent rules to be enumerated by hand. Moreover, not all LLVM instruction sequences appear as frequently. Having larger rule set would slow down the compiling process, though such a large set would produce more efficient translation. Hence, we collect a set of sequences occurring frequently in a set of training programs and find corresponding rules for them. The large space of such rules calls for the automatic rule generation.

Our Rule Generator is an automatic way of computing equivalent instructions sequence pairs. Goal is to generate as many rules for a specific LLVM instruction sequence as possible, keeping in mind certain constraints designed to facilitate good rule generation. For purpose of this project, we have selected only a small subset of LLVM and X86 instruction types, that occur in our test programs. For a generic compiler, we can extend the current implementation to include more instruction types as well as size of instruction sequences.

## 8.1 Components of Rule Generator

Here we describe the overall approach and steps taken in construction of this tool. Basically, there are three broad steps involved in the entire process -

1. Collection of LLVM instruction sequences for whom translation rules are needed.
2. Generating possible X86 sequences that can possibly be a part of a rule
3. Checking for equivalence between these sets of LLVM and X86 instruction sequences.

From now on, we shall call these steps as **Harvesting, Enumerating and Equivalence Checking** respectively. Following these steps essentially builds a table of rules, called *dictionary*, that is build only one time using a set of training programs, and used in subsequent compiling of new programs. Being a one time operation, we can justify its large running time and memory requirements as tolerable. The following section will describe in detail each of these components, and how we have build them in our project.

### 8.1.1 Harvester

This part of rule generator prepares input LLVM instructions for the later part of the tool. It takes a set of training LLVM programs, creates a table of all the instruction sequences (in generic format) and counts the frequency of their occurrence. Since each instruction has to be translated, all single instructions have to be taken. For instruction sequences of larges sizes, a threshold on frequency can be applied to extract them from rest. In our present implementation, we have put the critical sequences manually since it is more easy to identify them in small programs manually.

### 8.1.2 Enumerator

This part creates X86 instruction sequences that can be potentially equivalent to some of the harvested LLVM sequences. The major aspects of its construction are -

1. X86 instruction representation and sequence representation.
2. Operand pattern for operands appearing in a given X86 instruction sequence. For example: Following three differ only in pattern of operands

| | | | | |
|---|---|---|---|---|
| add R0, R1 | vs | add R0, R1 | vs | add R0, R1 |
| add R2, R3 | | add R1, R2 | | add R2, R0 |

3. Representation of In-map and Out-map, i.e. mapping specifying which LLVM variables are mapped to which X86 register/memory operands before and after execution. This will form the basis of equivalence.

We start by creating one class for each X86 instruction type. Each instructions has some common property, like opcode and argument list. So, we first create a base class containing these common attributes. Then we extend it for a specific type of instruction. Each of these initialized instructions specific values, like number of arguments, opcode, etc. Now, we create a vector of pointers, each pointing to one object of each of these classes. So, our representation of X86 instruction sequence is just a vector of indices, indexed into this global vector. Enumerator creates a list of list of indices taken $i$ at a time ($\forall i : 1 \leq i \leq n$, n: max. size of an X86 sequence). Then for each LLVM instruction sequence and each

X86 sequence, we generate Operands Pattern. This is done using a systematic pattern generation algorithm. Second problem is dealt by creating a list of *effective variables*- list of distinct register/memory/constant placeholder. The operand pattern of a X86 sequence is then specified by referencing operands of each x86 instruction (within the sequence) to the corresponding effective variable. Also, In-map and Out-map are represented by references to effective variables.

For e.g.: $\begin{matrix}\text{add R0, R1}\\ \text{add R1, R2}\end{matrix}$ has 3 effective variables (R0, R1, R2) and operands in individual instruction (object of that class) has references to these.

The X86 sequences are then generated as list of list of instruction types taken $i$ at a time ($\forall i : 1 \leq i \leq n$, n: max. size of an X86 sequence). Specifically, number of X86 instruction sequence types are $\sum_i m^i$, m: number of types of X86 instructions considered. And each X86 sequence type, having all operands distinct, can have more variants in Operand Pattern.

Enumerator creates these sequences for all possible combinations and we deal with Operand Patterns by a systematic process that avoids redundant patterns. It also deals with constraints where memory and constants can be put up. Specifically, an X86 instruction sequence having all register operands gives rise to its variants where some of these registers is replaced by memory or constants. In case of constants, we make sure that constant values aren't changed in that rule. This part of enumerator (creating memory and constant containing sequences) is carried out later after equivalence checking, on filtered sequences only, and so, till now, we only create register only sequences.

### 8.1.3 Equivalence Tester

Checking if two instruction sequences are equivalent is a critical as well as a tedious part of Rule generator. Ideally, two sequences are equivalent iff they produce same variable value (as specified in out-map) for every input value of input variables (as specified by in-map). However, it is practically impossible to test this for all possible values. So, our current approach is to do this by checking equivalence for a few random values of input variables. Since we are considering not only arithmetic, but also non-arithmetic X86 instructions, like Compare(cmp), Jump(jl, jeq, jmp), etc. we had to consider, not only variables appearing as operands in these instructions, but also other environment variables, like EFLAGS, Program Counter (PC) while checking for equivalence. So, in essence, this is a small simulator for an X86 machine, that simulates essential machine state through program variables. Equivalence testing then proceed as follows -

1. Randomize Machine state (effective variable, PC, EFLAGS) for both LLVM and X86 instruction sequence
2. Copy values of LLVM variables that are mapped to some X86 register as specified by In-Map
3. Execute Sequentially each of these sequences. Since variables/PC are references, sequential execution simulates actual processing.
4. Check for equal values of all variables as mapped by Out-map

5. If Check fails at any time, break. Otherwise, repeat above procedure for a number of times.

Since this is a probabilistic method of checking equivalence, there is a finite probability of getting false positives, i.e. two non-equivalent instruction sequences getting same results. Repetition amplifies probability of success and thus, relies on random number generation method deployed for testing. Also, instructions that deals in absolutes, like add, are more likely to get correct X86 corresponding instruction, where as instructions like icmp, that allow for significant change in input values while keeping output same, are more prone to false positives. In fact, our experiments show that for icmp, there are wrong equivalence for 1 out of 6 runs of the rule generator on an average, using standard rand() function of C++ stdlib.h library. Using better random number generators can reduce this frequency even more.
Also, while number of times the process is repeated affects the probability amplification, it is exponentially small increment with each repetition. This can be argued if one visualizes it as a problem belonging to RP complexity class. Our experiments show that for 5 repetitions, there are many false positives with icmp LLVM instruction in nearly every run. However, they got significantly reduced after increasing to 10 repetitions, and negligible (1 in 6) for 20 steps. So, these are the two major factors affecting correctness of our rule generator. While improving randomness adds to complexity, increasing repetitions increases run time by multiplicative factor and doesn't ensure great improvement with large repetitions too. This problem of false positive doesn't come with the *SAT solver*, which is basically a constraint satisfiability solver, constraint being in-map and out-map and doesn't involve testing with random values, but rather a simultaneous equation satisfiability problem. Since were able to generate rules satisfactorily from randomized equivalence testing, we didn't go for SAT solver and chose to remove rare false positives manually.

## 8.2 Rule Generator Model Overview and Extendibility

The entire rule generator model is such that it is applicable for any source - destination rule generation. Here be briefly summarize the entire model and how it can be extended. First of all, all instruction types, whether LLVM or X86, is represented by a class, which is in turn extended from a base class containing information common to all types. Each individual class may contain instruction specific constructs, for example, icmp class contains another opcode to specify operation mode - less than, equal, greater, etc. Also, all of their operands are not values, but pointers to values. This feature facilitates sequential execution of set of instructions, as output of any instruction becomes visible to all in that sequence while executing.
Then, all instructions have one object in a list of pointers to them. In this way, an instruction sequence is merely a set of indices, which helps in generating all combinations of instructions taken any number at a time. Secondly , Operand Patterns are represented by pointing operands to corresponding *effective variable*, so that value changed in one place appears everywhere in the pattern where that operand appears.
Thirdly, we generate correspondence in in-map and out-map with the *effective variables* of this pair. Here we enforce the constraint that an input variable of LLVM mapped to a register is either allowed to get destroyed or if survives in out-map, must remain mapped to same register. This prevents register swaps needed in case the LLVM sequence is a loop and helps in cost minimization during map generation phase of compiler.

Finally, for each LLVM and X86 instruction sequence pair, we know the in-map and out-map and thus are ready to execute both of them for equivalence testing. Each class has an execution function, declared virtual in base class, that implements its functionality. Executing an sequence is simple executing each instruction within sequentially, one followed by other. We execute only on random inputs for certain number of times repeated, and thus, our rules generated are correct with high probability. To extend this model to more instructions on any side, simple add more classes and input LLVM instructions. To extend this model to other architectures, simply change functionality in each class and common properties (architecture specific) in the base class.

# 9 Related Work

Techniques used in our current project are an extension previous work [3]. In this the technique was used for binary translation from powerPC to x86. Peephole superoptimizers have also been used in [2]. However this is the first work we know where the technique is used for translating from intermediate form to machine assembly.

# 10 Conclusion and Future Work

As is evident from the results obtained on the small test programs, the technique has a lot of potential for producing very efficient assemblies. Firstly the tool needs to be extended so that it can be tested on larger programs like spec integer benchmarks. For further extension, firstly the issues discussed in section 4 need to be addressed. We also need to do an analysis of memory usage and time taken by the compiler to make it useful for large programs. Finally most parts that are architecture specific are hard coded. We would like to provide an interface so that new architectures can be easily added.

## References


[1] Alfred V. Aho, Ravi Sethi, J. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education Asia, 2000.

[2] Bansal, S., and Aiken, A. Automatic generation of peephole superoptimizers. *SIGPLAN Not. 41*, 11 (Oct. 2006), 394–403.

[3] Bansal, S., and Aiken, A. Binary Translation Using Peephole Superoptimizers. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008)* (Dec. 2008).