CS201 Course Project

Promoting Name Spaces to Register While Propagating Constants

|  |  |
| --- | --- |
| Abhay Singh | Manjusnath Shivashankarappa |
| [asing073@ucr.edu](mailto:asing073@ucr.edu) | [mshiv001@ucr.edu](mailto:mshiv001@ucr.edu) |
| 862041345 | 862041345 |

Abstract

This project has been completed towards partial completion of the compiler design coursework: CS201. This project report aims to present our observations while studying and designing passes. For the project we chose to combine constant propagation functionality with memory to register pass. In order to create the new pass we studied the open source llvm passes: mem2reg.cpp and ConstantProp.cpp. The report presents a comprehensive study on the working principles of promotion of name spaces (variables) to register. We also present a number of examples in order to demonstrate the workings of the pass mem2reg. The report will also take the examples and run them on the designed pass and compare the results.

Introduction

As per [1] the LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project. Here for the coursework we are working with clang compiler. Clang is an open source multi-pass compiler. Each pass is an optimization on the original source code. Each pass communicates to another pass via an intermediate representation. One of the most important passes in clang are simplifycfg, mem2reg and constant propagation. We try to combine the optimizations done by constant propagation with that of mem2reg. Mem2reg promotes the eligible variables to registers and constantprop replaces any variable that may have a constant value assigned to it with the constant and further propagate the changes.

The report is divided into two major sections. The first one discussing in depth about the mem2reg pass and the second section talks about the new pass modification designed.

Section 1: Studying mem2reg

1.1 Introduction to mem2reg

This section focuses on the mem2reg pass in Clang llvm. The load and store operations are slow and thus must be minimized, mem2reg thus, analyses the intermediate representation and tries to minimize the aforementioned. On a high level, it checks for each name space if it is unambiguous. If it is found to be unambiguous, replace the alloca and all its uses with equivalent code based on a register. In the coming subsections we will discuss the basic principles and how the pass was run to analyse the differences it makes.

1.2 Basic Principles: mem2reg

The basic idea of the pass is to minimize the load and store operations that happen from the memory. In order to achieve this, the pass assumes an infinite set of registers and assigns all the unambiguous name spaces a register. These are called the virtual registers. This is done because the register allocation will strictly depend on the architecture of the machine, the type of registers it has and the number of registers. In the second step of the pass it gets information from the system about the aforementioned parameters and depending on the frequency of being used, a name space now termed as alloca is assigned a register. Please note, there is a subtle difference in allocation and assignment. The first stage allocates a register the second assigns it.

Mem2reg constructs a mapping from virtual registers in the intermediate representation into a combination of physical registers and memory locations and rewrites code to reflect that mapping. Register allocation needs to create a correct program for the targeted machine.

1.2 Algorithms: mem2reg

In order to simplify the algorithms used for the register allocation, we assume a basic block is the entire program. However the discussed algorithms can be extended into a SSA form for the complete transformation across the basic blocks. The second assumption taken here is that the registers are all general purpose registers. The algorithms can be easily extended to handle multiple register classes.

The next two segments will be exploring various methods to perform register allocation.

1.2.1 Top-Down Register Allocation

The principle behind this is that the heavily used values should occupy the registers. It finds the number of times each virtual register appears in a block. Then finally it allocates the virtual registers to physical registers in decreasing order of frequency.

Algorithm:

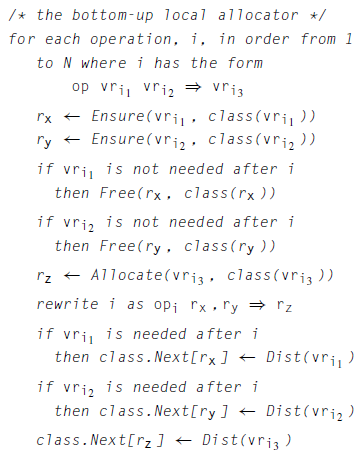
1. Compute a priority for each virtual register
2. Sort virtual registers in decreasing order.
3. Assign the virtual registers to physical registers in order of priority.
4. Rewrite the code to remove corresponding loads/stores.

The drawback for this algorithm is that is dedicates a whole register for an alloca for the entity of the program. In other words, if a virtual register is being used heavily for the first quarter of the program and is not used for the rest of the code, it can be replaced by another alloca, in the later three quarters of the code.

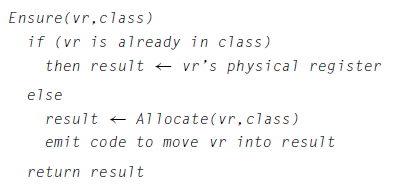
1.2.1 Bottom -up Register Allocation

The basic idea is to allocate the registers on demand basis. It iterates over the blocks to observe usage patterns and allocates registers to avoid most spills (load/stores). The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand. There are, however, some subtleties. By considering and in order, the allocator avoids using two physical registers for an operation with a repeated operand, such as add . Similarly, trying to free and before allocating avoids spilling a register to hold an operation’s result when the operation actually frees a register. Harder parts of the algorithm are in the sub-routines Ensure, Allocate, and Free.

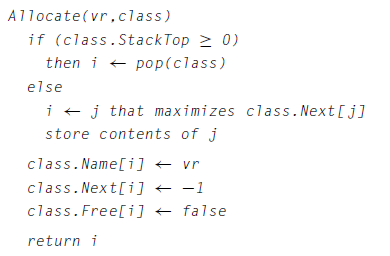
The following pseudo-code from [2] illustrates the algorithm.



Code 1: Bottom-up driver program



Code 2: Subroutine Ensure



code 3: Allocate subroutine. Assigning the alloca

The source code for mem2reg pass can be found at link [Mem2Reg\_8cpp\_source.html](http://llvm.org/doxygen/Mem2Reg_8cpp_source.html) .

1.3 Experimentation: mem2reg

In order to investigate into the workings of the pass, we decide to let the pass transform the following experiment code samples, including:

1. A basic calculation program.
2. A basic program with a global variable.
3. A program with switch case.
4. A program with for loop.
5. A program with nested if else.
6. A program with pointer manipulation.

To check the intermediate representations and the differences the pass made to it, we used the following commands:

1. clang -S -emit-llvm eg1.cpp -o eg1.ll -Xclang -disable-llvm-optzns -O3
2. opt --time-passes -S eg1.ll -o eg1\_final.ll -mem2reg

The first would generate a readable .ll file for the intermediate representation and we have disabled any optimizations. The second command transforms the .ll file by using the mem2reg pass.

The system specifications where the above tests were conducted are as follows:

1. Clock speed: 2.7GHz
2. Processor: i7 7th generation
3. RAM: 16 GB
4. llvm version: 7.0

1.4 Results and Analysis: mem2reg

The above mentioned tests were run one at a time to yield the following results. Each section contains the experiment code snippet, List & Label Preview files(.ll format) before and after running the pass. Please note here only the part of .ll files that see a change have been included. To have a look at the complete code and List & Label Preview files go to the link : [www. MakeLaude.com](http://www.TerimaKaBhosadaMakelaude.com) Upload the files and paste link

1.4.1 basic calculation program(I)

1.4.2 basic program with a global variable(II)

1.4.3 Switch case program(III)

1.4.4 For Loop program(IV)

1.4.5 Nested if else program(V)

1.4.6 Pointer manipulation program(VI)

Analysis

In all the foregoing subsections we have a few strong observations:

* The pass has totally removed all the load and store operations wherever possible.
* Global variables are represented by @ and local variables by %.
* Although all the name spaces have been allotted a virtual register the name spaces assigned a register do not need load/store. Marking the difference between allocation and assignment.
* To add a couple of more observations.

1.4 mem2reg Conclusion

The mem2reg pass, aims to optimize a program’s runtime performance by reducing the amount of load or store operations. It is achieved by promoting the variables to registers. Experiments were conducted on the pass using seven test cases and observations were made. It was found that wherever possible, the pass tries to avoid the load and store operations. The pass could omit the load store operations by register allocation.

Section 2: Implementing a new pass

2.1 Motivation

Looking closely at the code test cases used, we find that most variables are essentially constants. So if we are able to propagate the constants, we should be able to reduce the number of virtual registers and eventually the number of physical registers occupied. Thus, to create our own pass we decided to combine the functionalities of mem2reg and constant Propagation.

Constant propagation and mem2reg do not have a dependency on each other and do not interfere with each other’s transformations by the virtue of their tasks. In other words, if constant propagation decides to omit a line of code because a variable was assigned a constant, this variable assignment omission will not affect the transformations being done by mem2reg. It can be argued that if the variable was actually promoted, the register will hold a constant value, however we’ll see that the mem2reg operation and constant propagation operations are done hand in hand and this situation does not arise.

2.2 Basic Principle and Algorithms

The constant propagation algorithm identifies variables that have been assigned a constant value and uses the constant value instead of the variable till the point in the code where the variable is assigned another value. The algorithm is composed of three major components it iteratively does initialization and CFG traversal in DFS manner till when no more changes are found. Upon no change found termination.

Now with llvm’s functionPass’s runOnFunction method, we can do iterative propagation of constants and Bottom-up register assignment operation we saw in the last section. We check if either traversal in the tree or bottom-up allocation algorithm recommends a variable promotion. If yes, make the recommended change and continue with the iteration. Once there are no recommended changes, termination occurs and the output is presented. Broadly speaking the algorithm can be seen as:

1. While(changes made)
2. cp = Check if CFG traversal creates a change (constant propagation)
3. mr = check if bottom-up allocation assigned some value to a register (mem2reg)
4. Changes made = mr || cp

The source code for our new pass can be found at link: Upload the files and paste link. The source code has also been attached as an appendix to this report.

2.3 Experimentation

In order to check if our designed pass works as expected, we do the tests designed on the similar pattern as the tests for the mem2reg pass.

To check the intermediate representations and the differences our pass made to it, we used the following commands:

1. Enter command 1
2. Enter command 2

System specifications used to conduct these tests were the same as the ones mentioned in section 1.2 above.

2.4 Results and Analysis

The same tests used for the mem2reg pass were run one at a time to yield the following results. Each section contains the experiment code snippet, List & Label Preview files(.ll format) before and after running the pass. Please note here only the part of .ll files that see a change have been included. To have a look at the complete code and List & Label Preview files go to the link: [www. MakeLaude.com](http://www.TerimaKaBhosadaMakelaude.com) Upload the files and paste link

2.4.1 Basic calculation program(I)

Paste snip from before and after, and of separate mem2reg and constantPropagation

2.4.2 Basic program with a global variable(II)

2.4.3 Switch case program(III)

2.4.4 For Loop program(IV)

2.4.5 Nested if else program(V)

2.4.6 Pointer manipulation program(VI)

Analysis

Write about the changes that happened now after combining the 2 functionalities. Compare time with mem2reg alone

|  |  |  |
| --- | --- | --- |
| Experiment Program | Time taken by mem2reg | Time taken by modified pass |
| (I) |  |  |
| (II) |  |  |
| (III) |  |  |
| (IV) |  |  |
| (V) |  |  |
| (VI) |  |  |

Table 1: Time taken mem2reg v/s modified pass

2.4 Conclusion

We must write a conclusion. It would depend on the analysis

APPENDIX: Source code for modified pass.