Max Flow Spill Code Placement Algorithm Implemented in GCC 4.4.3

A Senior Project

Presented to

the Faculty of the Computer Engineering Program

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Stephen Robert Beard

June, 2010

Table of Contents

Acknowledgments

I would first and foremost like to thank Dr. Chris Lupo for being an amazing professor, adviser, mentor, and sparking for my interest in computer architecture and code generation. I do not believe I would be headed where I am without your guidance and inspiration.

I would next like to thank Tina Reese for all the help she has given me, for keeping the Computer Engineering department running, and for doing it all with a smile. I would also like to thank Dr. Smith and Dr. Harris for their work as Program Director (even though everyone knows that Tina does all the work), for being great professors, and for allowing me to help out in your labs. I learned a great deal in your classes, but I probably learned more serving as a lab assistant; so thank you both for that opportunity.

I would also like to thank Dr. John Oliver for being an excellent professor and for being so supportive of me during my time here; Dr. Aaron Keen for teaching a great compilers course that was extremely useful in my work on this project; Dr. John Clements for being another wonderful professor and for the advice; Dr. Wayne Pilkington for being one of the best professor's I've had and keeping lectures so interesting; Dr. Joe Grimes for allowing me to TA for his Computer Architecture class; and all of the rest of my professors at Cal Poly.

My time at Cal Poly has been better than I could have hoped for, and I will always cherish the memories I have made here.

I. Introduction

The placement of spill code plays an important role in the register allocator of an optimizing compiler. Many computer architectures possess a register linkage convention that dictates which registers are preserved across function calls and which are not. This project addresses the problem of optimizing spill code that is associated with register linkage conventions.

This algorithm was created by Dr. Chris Lupo and is described in the paper *Beyond Register Allocation: a Novel Algorithm for Spill-Code Placement*. The algorithm was implemented for GCC 2.5.7 for a PA-RISC architecture [4]. The work in this project will involve porting the existing code to the newest version of GCC, v4.4.3, and modifying it for use with the x86 architecture.

To solve the problem of placing spill code associated with procedure call conventions, an algorithm based on the max-flow min-cut theorem will be used.

II. Background

This chapter will describe background technical information relevant to the project.

Control Flow Graph

The control flow graph, CFG, is an abstraction used to model the line of execution through a function. The code in a function is broken up into basic blocks, which are single-entry single-exit chunks of code. This means that all code in the block will be executed together; by definition there is no way to jump into or out of the middle of a basic block. The CFG itself is a directed

graph where each node represents a basic block and each edge represents a possible transfer of control between two blocks. [1]

Register Allocation

Registers are typically the fastest location in the memory hierarchy, and often the only location that many instructions can use directly. Thus, intelligent register allocation is paramount to good program performance. The register allocator in a compiler is tasked with the job of mapping the symbolic registers, used by the preceding phases of the compiler, to the real register set of the target architecture. At times, the number of *live values*, values that will be used later, in a program exceeds the number of available registers. This causes a *spill*, a temporary store to main memory, to reduce the *register pressure*, the number of values currently required to be stored in a register. [2] These spills are costly since they require access to main memory, which is much slower than access to registers. Thus, a good optimizing compiler will possess a register allocator that makes efficient use of registers and minimizes the number of spills required.

Procedure Call Convention

When procedures in a program are compiled separately, some convention must be followed so the compiler knows which registers are available inside the procedure. There are three possible choices for register availability: all registers, no registers, or some subset of registers. If all registers were available, the calling function would be required to save all live registers before the call. If no registers were available, the called function would be responsible for saving and restoring the value to all registers that it will use. In the general case, either of these options would result in a large number of memory accesses. For this reason, many architectures divide the registers into two sets, *callee-saved*, whose values are saved across procedure calls, and

caller-saved, whose values are not saved across calls. In this convention, callee-saved registers are only required to be saved and restored if the called function will make use of them, and the caller-saved registers are only required to be saved if they will be live across a function call. [3] This split allows the register allocator to make intelligent choices about register usage to minimize memory accesses.

Max-Flow Min-Cut Theorem

The max-flow min-cut theorem implies that for a given flow network, if there is some maximum flow found then there is also a cut of minimum weight. The *Ford-Fulkerson method* is a common method for solving the maximum flow problem. It is an iterative method that attempts to find an augmenting path on which additional flow can be pushed from source to sink during each iteration. If such a patch is found, the flow is updated along the path. The process is repeated until no augmenting path is found. The *Edmunds-Karp* algorithm is an implementation of this method that uses a breadth-first traversal of the flow network to find the augmenting paths. [4]

Max-flow Spill Code Placement Algorithm

The algorithm developed by Dr. Lupo to find optimal placement for spill code placement makes use of the *Edmunds-Karp* algorithm to locate locations of minimal cut. It depends upon the collection of dynamic run time data to construct the maximal flow networks and to locate the minimum cut locations. This is easily accomplished using profiling in GCC. The algorithm is given below, and is general to both caller and callee saved register types.

MAX-FLOW SPILL CODE PLACEMENT ALGORITHM
1. Construct load network G_L
2. Use Edmunds-Karp algorithm to find maximal flow f_L of G_L
3. Construct store network G_S
4. Use Edmunds-Karp algorithm to find maximal flow f_S of G_S
5. **If** $|f_L| + |f_S| <$ Max Allowed Cost
6. Insert loads on minimum cost edge cut set of G_L
7. Insert stores on minimum cost edge cut set of G_S
[4]

III. Description

This chapter will cover the technical details of the project.

Preliminary Work

To begin the project, Dr. Lupo provided me with both the *Beyond Register Allocation: a Novel Algorithm for Spill-Code Placement* paper and his implementation for GCC 2.5.7. A good amount of time was spent in reading and understanding the paper, and looking through the previous implementation to understand it.

Dr. Lupo expressed concern about the memory management in his code, and suggested that I investigate the way that GCC handles its internal memory management. The previous implementation relied heavily upon 'malloc,' which is both slow and prone to memory leaks. Using the memory management page of the GCC guide [5], I discovered that there are different methods of handling data objects based upon their size and longevity. I noted that there was extensive use of registers sets, and then a smaller amount of fixed size data objects in the previous implementation. Based on this, I decided to use GCCs register set handlers for the register sets, and allocation pools for the other objects.

Register sets were a natural choice, as that is the way that GCC handles its use of register sets. The allocation pools are the preferred choice for fixed size, fixed lifetime objects for three main reasons. The first is that they are much faster than multiple calls to 'malloc.' When created, the allocation pool will grab large chunk of memory, large enough to hold some specified amount of objects. Then, as you add items to the pool, you simply use this extra space, rather than having to make the slow system call to 'malloc' once again. If the allocation pool runs out of space, it will automatically increase its size. The second reason is that you can free individual elements of the pool, and they return to the pools free list. These freed items can then be reused, rather than completely releasing the memory and then having to allocate new memory for a new object. The final reason is that GCC currently has better statistic tracking and memory checking with pools. [5]

In the provided code, there was extensive manipulation of the CFG. A good deal of time was spent in understanding the way that GCC represents the CFG, and the provided methods for accessing and traversing it. The CFG is composed of basic blocks. Each block contains the information needed to traverse the CFG (block id, predecessors, successors, instructions, etc). There are also macros provided to easily traverse the graph; the most frequently used is FOR_ALL_BB(bb), which will visit all blocks in the graph.

Adding to GCC

After the preliminary understanding of both the previous code and the relevant internals of GCC was obtained, I began to port the existing implementation to the new version of GCC. This involved first setting up the internal structure of GCC to accept the new code, which was a multistep process.

First, I added a new pass to the compiler. Each step in compilation for GCC is implemented as a pass. Each of these passes is executed in a specified order to correctly compile the file. Since Dr. Lupo's algorithm is meant to be done as an optional optimization, and done before register allocation, I added it as such. Since this should be an optional pass, a flag was needed to tell GCC to use the optimization. I created the flag ' -fcaller-max-flow' which seemed appropriately named.

Next, the make file template for GCC need to be altered to compile my added code. This involves altering the 'Makefile.in' file included in the 'gcc' subdirectory of the main GCC source code. When adding to this file, much care is needed to ensure that all dependences are accounted for, and that the make rule is in the correct location. After this was completed, GCC was ready to correctly compile and use my additions.

Code Refactoring

Once GCC would recognize my code, I began the process of refactoring Dr. Lupo's previous implementation to use GCC's memory management and CFG.

Analysis

To construct the flow graph needed for the algorithm, three sets of information are needed: the reaching definitions, upwards exposed uses, and reaching calls. The reaching definitions are used to track which definitions reach a given use of a register. The upwards exposed uses are used to track which uses of a register are exposed to a given definition. These are both standard dataflow calculations; however, there is a slight change required for use in this algorithm. That is that calls kill definitions and uses for the reaching definitions and upwards exposed uses

respectively. The reaching calls analysis is used to track which call reaches a given use of a register.

GCC contains structure to perform the standard reaching definition analysis, but not the reaching calls or upwards exposed uses. I spent some time trying to understand the way the reaching definition code for GCC works. In general, data flow problems work very similarly. GCC takes advantage of this fact by having a generalized framework for data flow analyses. Each analysis is represented as a structure with data and functions to perform the tasks specific to its particular needs. The time required to fully understand this structure, re-implement it for three new analyses, and integrate them into the max flow proved to be too extensive. So I instead made the necessary modifications to the existing data flow analysis code to allow it to run in GCC v4.4.3.

Graph Creation

Once the analysis code was updated, I began to work on the graph creation code. This essentially involves forming a mapping between the instruction locations and control flow to the flow graphs. The majority of the work here was in changing the flow graphs to be created using GCC v4.4.3's CFG structures.

Graph Solving and Instruction Insertion

The prior work required the vast majority of the time spent on the project. The final steps in the process, graph solving and instruction insertion, have been left as a future project.

The graph solving should be straightforward to update. It simply involves updating the existing code to make use of GCC v4.4.3's profiling data, which is easily accessed as the count member of a basic block.

Instruction insertion will be slightly more involved. To insert a spill instruction you must be able to relate the spilled pseudo to a memory location. This mapping must be created at the store location, and recalled at the load location. The file 'caller-save.c' contains the functions which are responsible for saving and restoring call-clobbered registers across calls. The basic functionality contained in this file would need to be ported to the max flow code.

IV. Evaluation

There are several key points in this project that may be evaluated. The most important metric is proper compilation. This can be tested in two ways. The first is by using GCC's internal compilation test tools. These tools are designed to extensively test new builds of GCC against known inputs. The second is by using the SPEC benchmark suite. SPEC will compile its benchmarks using a compiler that is known to be correct and the experimental compiler. It will then compare the output of the two sets of benchmarks to ensure that the programs compiled with the experimental compiler produce the same output as the correct compiler.

After flow graph creation, the first part of the max-flow optimization can be tested. A program with a known flow graph can be compiled with debugging output enabled to display the compilers created flow graph. These two can then be compared for correctness.

Once the optimization is shown to correctly compile the code, and the flow graphs are being created correctly, the optimization can be checked for effect. That is, the dynamic execution counts of loads and stores and be compared between the base and experimental compiler. If successful, the max-flow algorithm should improve the spill code placement, which will reduce the number of dynamic loads and stores. The run times of the programs compiled with and

without the optimization can also be compared. My prior work in creating analysis tools for dynamic load and store counts in GCC can be used along with the SPEC suite to compare the dynamic instruction counts, and the SPEC suite can be used to compare run times.

V. Conclusions

Due to time constraints, the final implementation was forced to omit a few features which would increase the code quality. One feature is the method used for the dataflow analysis. To bring this optimization up to the quality required for submission to GCC would require modifying and using the internal GCC dataflow analysis. This includes creating the structure for the reaching calls and modified upwards exposed uses, and either adding new structure for the modified reaching definitions or adding a flag to alternate between the standard and modified versions. The graph solving and instruction placement functions also need to be finished as previously mentioned.

This project has first and foremost reinforced the idea of how difficult it is to develop additional features to a preexisting program with poor documentation. GCC is a massive program consisting of over four million lines of code. While only a fraction of this was relevant to my additions, it is still a tremendously difficult task to discover the relevant portions and understand the way that they interact with the rest of the program. There was often little documentation to assist in this effort.

It has also served to highlight some of the strengths and weaknesses of free open source projects. The great strength is that anyone, even an undergrad student, can develop new additions to the program. The great weakness is that cohesiveness and documentation suffer. Even though

there are preferred practices and requirements for submission, there are many contributors and each person's style of coding is slightly different. This means the program reads like a book written by multiple authors; each section differs slightly in style, grammar and feel. Once you grow comfortable with one section of the program and move onto another, it is like starting from the beginning again. Also, since people are usually developing for the project on their free time, and no one really likes documenting their code, the documentation tends to be lacking. While certain key sections have ample documentation, others that are more obscure have almost none. Using some sort of automated documentation generation system like Doxygen within GCC would help to reduce the time it takes to become comfortable with the inner workings of the program.

Between this project and my project for my compilers course, I have gained a great deal more respect for object oriented languages. My progress and understanding were greatly diminished by the amount of global state that is passed around in GCC. Often times I needed on particular bit of information, but did not know how to access it. Then it would turn out that this information is simply accessed with some globally defined macro or variable. While the lack of good documentation exacerbated this problem, it is still a significant issue. I feel that GCC would greatly benefit from being ported to an object oriented language; C++ being the natural choice since the vast majority of GCC is written in C.

This kind of work is important as long as computer architectures use the caller/callee saved register scheme. Intelligent register allocation can greatly improve the program speed, and can also be used to reduce power consumption used for the program. Since poorly performed register allocation can add many additional loads and stores, that are simply overhead, they cause the program to use more power from their longer run time increased instruction count. Power issues

are becoming increasingly more important with the sharp rise in the mobile computing industry and increased concerns over global power usage by computing centers. Even architectures that don't use the caller/callee saved scheme, such as the register windows in SPARC, typically have some subset of registers that are preserved across calls and those that are not. These architectures could probably benefit from a modified version of this optimization.

I feel as though this work has been a great help to me in my future career. My graduate work will most likely involve compiler work where I will once again be thrust into a large, preexisting code base. This experience has helped me to become more comfortable with learning a new environment, as well as greatly expanded my knowledge of compilers.

I have two main pieces of advice for students working in this area. The first would be to take a compilers class beforehand. This will give you a solid understanding of the basic operation of a compiler. I spent a good deal of time reading and talking with Dr. Lupo to gain this fundamental knowledge, and still had areas that I didn't fully understand. After spending this time, the compilers class became quite simple, but I'm sure it would have been easier to do it the other way around. The second would be to invest a significant portion of time at the beginning of the project reading through the GCC manual. It contains some good information about the inner workings of the compiler that would give a solid basis of understanding to then proceed to gain more in depth knowledge on specific areas of interest.

Appendix A: References

[1] Cooper, Keith D. and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[3] Patterson, David A. and John L. Hennessy. *Computer Organization and Desgin: The Hardware/Software Interface*. Morgan Kaufmann, 2007.

[4] Christopher Lupo. *Precise Register-Allocation Spill-Code Costing and Placement*. PhD thesis, University of California, Davis, September 2008

[5] Gnu Compiler Collection Wiki, "Memory Management." Gnu Compiler Collection. http://gcc.gnu.org/wiki/Memory_management (accessed April 22nd, 2010)

Appendix B: Analysis of Senior Project Design

Summary of Functional Requirements

This project is an optimization to the Gnu Compiler Collection (GCC). The optimization is an implementation of Dr. Chris Lupo's max-flow spill-code placement algorithm that is designed to optimize the placement of spill-code associated with procedure linkage conventions. These linkage conventions define hardware registers as either callee or caller saved registers.

Primary Constraints

The primary difficulty in my project was learning how to develop for GCC. It is a huge program consisting of over 1.2 million lines of code. While I only interacted with a portion of this, it was still very difficult to become familiar enough with the program to add new features to it.

Economic

This is theoretical research in the area of code generation related to hardware registers on computer architectures, and as such the only components needed were servers of varying computer architectures that the school already owned.

Estimated time: 200 hours
Actual time: 300 hours

Manufacturing

The goal of this project is to be included in the Gnu Compiler Collection, a free open-source software package, which is distributed over the internet, so there are no commercial manufacturing concerns.

Environmental and Sustainability

This project increases the sustainability of computing. Spill code is added to programs when the number of live values at a given time exceeds the number of real registers available. This spill code is pure overhead to store and restore values from memory. By optimizing the placement of spill code we reduce the number of dynamic instructions, thereby reducing the run time and power consumption of the program.

The project could be enhanced by making better use of GCCs internal dataflow analysis which would most likely reduce compilation time, thereby reducing the power consumption to run GCC itself.

Ethical, Health and Safety, Social and Political

Since this project is used to compile other programs, there are some ethical issues involved. A malicious compiler programmer could have his compiler create some kind of backdoor in any program that was compiled with his compiler. Besides this obviously unethical behavior, a compiler writer must also perform many tests to ensure that his compiler produces correct code. If he ignores some aspect, the compiler could potentially create an incorrect program. This would be a problem for all programs, but could potentially be a serious issue if the compiler is used to compile some kind of sensitive application. These would include things like programs for medical devices, databases for public records, voting machine firmware, etc.

Development

I learned and made extensive use of the program 'cscope' during this project. This program is used to browse source code quickly and efficiently. It was incredibly useful during my time learning and developing for GCC. It allows quick access to any definition or use of any symbol. This allowed me to learn how to use the various functions in GCC much faster than if I had to scan through the entire source tree by 'grep'ing for key words.