Compiler Analysis

CPE 431: Programming Languages II

Professor Lupo

March 19, 2019

Brian Hicks and Eitan Simler

Contents

1 Introduction

RTL (Register Transfer Level), is a form of abstraction of C which describes the behavior of registers throughout a program. By using RTL, a compiler can create optimizations independently of the machine architecture—along with machine-specific optimizations.

From the RTL a control flow graph is generated with basic blocks as nodes and edges between them. Next, registers are allocated for each instruction using a graph coloring algorithm. Optimizations like removing redundant code and instruction scheduling increase the performance of the code. Finally, an ARM assembly file is generated.

2 Architecture

The compiler takes an RTL file, generated by GCC, and generates optimized ARM assembly. The majority of the compiler was written in Java. Indeed, all optimizations and assembly creation were implemented in Java. However, the initial parsing was completed in Python due to its ease of use. A JSON file was used to transfer the parsed RTL information from the Python parser to the Java compiler. To run the entire program, a Makefile was created.

2.1 Parsing

The intermediate language RTL is parsed into a JSON file using the `RTLparser.py` and `RTLprocesser.py` scripts. We decided to work in Python for the parsing because it is significantly easier than Java's parsing. Next, we created RTL classes for each parsed instruction in the JSON object using the `JSONparser.java` file.

2.2 Control Flow Graph (CFG)

The Java file `CFG.java` is used to construct a digital and visual representation of the Control Flow Graph. The digital graph is shown in the DOT format. Each CFG consists of a basic block with edges between the block's successors and predecessors.

2.3 Translation

Each instruction in each basic block is translated from an RTL instruction to its corresponding ARMv7 instruction.

3 Optimizations

To improve the performance of the assembly, we implemented a few optimizations. These reduce the cycle count and the runtime of the outputted program. We added the removal of redundant moves as an additional optimization to further improve our assembly.

3.1 Register Allocation

When using register allocation, most loads and stores are removed from the RTL. This algorithm in `CFG.java` generates an interference graph of all instructions in a basic block. The instruction with the most interferences is removed and given a register that is not in use by one of the interfering instructions. This continues until there are no more instructions in the graph. If there are not enough registers, an instruction is spilled; each time the register is needed it is loaded and any modifications are stored on the stack. The graph coloring algorithm used is not optimal, but has highly accurate results.

3.2 Instruction Scheduling

The overall purpose of this algorithm is to reduce cycle count by increasing the distance between high-latency pairs of instructions. For example, an add instruction using a register defined in an load instruction previously has a high latency. According to the instruction scheduling algorithm discussed in lecture, each basic block's instructions was added to a directed acyclical graph (DAG) In `DAG.java`. Next, all instructions without predecessors were added to a ready priority queue; these instructions are ready to be scheduled and have no dependencies. The queue is sorted in descending order by longest path to a leaf, then most successors, and then random. The instruction with highest priority is added to a list and all of its successors without predecessors are added to the ready list. This algorithm is repeated until all instructions have been scheduled.

3.3 Removing Redundant Moves

Upon finding a move where the target and source registers are the same, we omitted the instruction from the program. These instructions were leftover from the RTL and can safely be deleted.

4 Performance Analysis