CS4071 SSA Optimiser

Miles McGuire Tom Mason Stephen Rogers
Lonan Hugh Lardner

January 2, 2014

Contents

Chapter 1

Introduction

The aim of this assignment is to design and implement an optimiser for ARM7 assembly. The problem has been split up into three parts, one of which will be described in this report. Specifically, we will examine the conversion of an intermediate representation (IR) into SSA (Static Single Assignment) form, various optimisations performed on the IR in SSA form, and conversion back out of this form.

In the Design chapter, we will begin by describing the IR that is passed into our application to be optimised. Then we will examine how conversion into SSA form is achieved. In the following section, we will describe the various optimisations that are performed on the IR in SSA form. Finally, we will look at how conversion out of SSA is completed.

In the Implementation chapter, we will give a brief overview of how the application is implemented, focusing on code structure and any technologies used.

Chapter 2

Design

2.1 Intermediate Representation

The intermediate representation was designed specifically to use JSON for passing between different parts of the overall application. JSON was chosen because of its simplicity and the wide range of libraries available for working with it.

Any program that is being passed into our optimiser is represented by a single object. This object will generally contain two members. The first is a comments section, which can be ignored by the optimiser. It exists primarily for human reading only, to place context on the program represented. The second is an array of objects representing the basic blocks.

Each of these basic block objects contain three members. The first is the name of the basic block. The second is an array of objects, each representing an instruction contained by the basic block. The third and final member is an array which contains the name of all blocks that succeed this one. Each object representing an instruction must contain an "op" member. They may also contain members for a destination register and up to two source registers.

2.2 Conversion into SSA

Before conversion into SSA form can begin, we need to generate certain pieces of information about the program being converted. First, we must generate the Control Flow Graph (CFG) for the program. This is done simply in two steps. We iterate over the program, adding each basic block as a node in the graph. Then we iterate over the program again, adding an edge from a node x to a node y where y is in the "next block" set for node x.

From the CFG we can now calculate the Dominance Frontier (DF) of each of the nodes in the graph (i.e. each basic block). The DF of each node is needed to determine where to insert ϕ-functions later in the process. The DF of each node x is the set w where x dominates some immediate predecessor of w but does not strictly dominate w. The set of all DFs can be calculated programmatically by iterating over the CFG.

Finally, we calculate the set of "definition sites" for each variable in the program. This set, for a variable contains all basic blocks in which there is an assignment to that variable.

Once we have all of this information, we can begin inserting ϕ-functions into the program. This is done using the following algorithm.

```
for each variable x in the program
    worklist = definition sites of x

    while worklist is not empty
        remove node n from the worklist

        for each node y in DF(n)
            if x does not have a phi-function in y
                insert phi-function for x at the top of block y

                if y is not in the definition sites for x
                    add y to worklist
```

Once we have successfully inserted all ϕ-functions, we can begin renaming variables to complete the conversion into SSA.

2.3 Optimisations

2.3.1 Dead-code elimination

Simple dead-code elimination is achieved using the algorithm examined in class on the "SSA Optimization Algorithms" handout.

```
worklist = all variables in the SSA program
while worklist is not empty
    remove variable v from worklist
    if v's list of uses is empty
        let s be v's statement of definition
        if s has no side effects (other than assignment to v)
            delete s from the program
            for each variable x used by s
                delete s from the list of uses for x
                worklist = worklist U {x}
```

This algorithm simply removes any assignment statements to variables which are not used at any other point in the program. Whenever a statement is removed, the list of uses of any variables used by that statement must be updated. When this is done those variables are added back onto the worklist to be checked again. This continues until the worklist is empty.

2.3.2 Simple constant propogation

The algorithm used for simple constant propogation is also taken from the "SSA Optimization Algorithms" handout. The algorithm has been modified slightly to include copy propogation and constant folding into the one process.

```
worklist = all statements in SSA program
while worklist is not empty
```

```
remove statement s from worklist
if s is a phi-function with all the same arguments
    replace s with a copy operation
if s is a foldable operation
    if both source operands are constants
        fold constant statement s
if s is a copy of some constant c to destination v
    delete s from the program
    for each statement t that uses v
        substitute c for v in t
        worklist = worklist U {t}
```

This algorithm performs three distinct transformations on the original SSA program.

- Eliminates phi functions where all phi-function operands are equal, replacing such functions with a copy operation.
- Constant folds operations on two constant variables, by performing the specified operation and replacing the statement with a copy operation with the result, eg. "ADD R0, #1, #5" becomes "MOV R0, #6".
- Copy propagation taking single argument ϕ-functions or copy assignments of the form x ¡- ϕ(y) or x ¡- y, deleting them and replacing all uses of 'x' by 'y'.

2.3.3 Aggressive dead code elimination

This optimisation aggressively finds and eliminates dead code using the algorithm described in the "SSA Optimization Algorithms" handout. The algorithm performs the following 9 steps.

1. Marks all statements to be deleted.
2. Unmarks statements that perform "live operations":
    - I/O
    - Memory writes (STR)
    - Branch & Exchange (BX) and Branch & Link (BL)
    - Software Interrupts (SWI)
    - Status register updates (CMP), operations with the 'S' flag.
3. Unmarks statements defining variables used in live statements.
4. Unmarks conditional branches that directly control execution of live statements.
5. Removes blocks that will not be reachable from the START node after deletion.
6. Deletes all marked statements.

7. Removes variables from phi functions whose definitions have been eliminated.

8. Removes blocks which contain no statements.

9. Iteratively finds a least fixed point at which no further statements are removed.

A least fixed point solution is used despite not being mentioned in the description of the algorithm as testing showed subsequent attempts calls to the function may eliminate further code. This arises from the three phases of unmarking, particularly "unmark_live_variable_definitions", interacting with one another and finding new results to be deleted after the removal of certain blocks/edges.

2.3.4 Conditional constant propogation

Once again, the algorithm used for conditional constant propogation is described in the "SSA Optimization Algorithms" handout. At the beginning, this algorithm assumes that a basic block cannot be executed until there is evidence that shows it can be. Similarly, the algorithm assumes that a variable is never assigned to, unless there is evidence that shows it is.

In the algorithm, the set V tracks the "evidence" associated with each variable in the program. For any variable v, V[v] is true if there is evidence that v will have at least two values at various points in the program, or an unpredictable value (e.g. read from a file). V[v] is false otherwise. Also, the set D tracks whether or not basic blocks should be deleted or not. At the start, D[b] is true for all blocks b in the program. If it is found that a block is reachable, then D[b] is set to false.

The algorithm works by iterating over a worklist of basic blocks. This worklist initially contains only the starting block, since this is the only block at this point that we can say for certain will be executed. Blocks are popped off the worklist one at a time. Whenever a block with a single successor s is encountered, D[s] is set to false. In the event that there are two successors to a block (called B1 and B2), if both source operands to the conditional jump are found to be constant, then the result of the jump can be evaluated at compile time. Depending on the result, either D[B1] is set to false or D[B2] is set to false.

The algorithm performs all of the steps outlined in the handout iteratively on each statement of each basic block on the worklist. Once the worklist is empty, the algorithm deletes any basic blocks b where D[b] is still set to true.

2.4 Conversion out of SSA

Chapter 3

Implementation

Chapter 4

Conclusion