Developing a formally verified algorithm for static register allocation

David J. Barker
Jesus College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: David.Barker@cl.cam.ac.uk

May 16, 2014

Declaration

I David J. Barker of Jesus College, being a candidate for the Part III in Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: TBC

Signed:

Date:

Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page. TODO.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is the introduction where you should introduce your work. In general the thing to aim for here is to describe a little bit of the context for your work — why did you do it (motivation), what was the hoped-for outcome (aims) — as well as trying to give a brief overview of what you actually did.

It's often useful to bring forward some "highlights" into this chapter (e.g. some particularly compelling results, or a particularly interesting finding).

It's also traditional to give an outline of the rest of the document, although without care this can appear formulaic and tedious. Your call.

Chapter 2

Background

2.1 The problem of register allocation

When compiling code, we generally assume the existence of an infinite number of locations in which to store values. Each variable used by the program is treated as its own location, and temporaries and results are freely allocated to conceptual registers. However, in reality all processors have only finite numbers of physical registers, some of which are reserved for specific purposes, and so the compiler ultimately has to map variables in the code to registers in such a way that ensures the code's behaviour is the same.

There are many issues involved when doing this. The main problem is that code will often use far more 'virtual' registers than actually exist on the target processor. The simplest solution to this is to store any values which won't fit in memory, but this considerably worsens performance. The solution is to find variables which are never in use at the same time and place them in the same register, allowing registers to be re-used when the values originally assigned to them are not in use.

This optimisation is enabled by liveness analysis. Essentially, a variable is live where changing its value will affect the program's future behaviour [ref?]. Because this is difficult to decide with absolute precision, we generally compute a safe over-approximation of liveness: a variable is live if an instruction reachable from here reads its value, and its value is not written to in a preceding instruction. Liveness is computed by repeatedly iterating over instructions in the code from the end backwards, updating each instruction's set of live variables according the following liveness equation:

$$live(n) = (live(n+1) \setminus write(n)) \cup read(n)$$

(Where live(n) is the set of variables live at instruction n, live(n+1) is those live at the next instruction and read(n) and write(n) are the variables read and written by the instruction.)

This is repeated until the live variable sets stop changing. With this information, we can deduce which variables are live at the same time, and thus which variables cannot occupy the same register because doing so would result in one overwriting the other whilst the other was about to be used.

When no allocation can satisfy these clashes between variables, it becomes necessary to spill variables to memory. This means reading or writing said variables takes considerably longer, and so an effective register allocation algorithm should minimise spills wherever possible.

Another thing to consider is that some target architectures have constraints on what registers can be used for. For instance, a particular operator may require its operands to be stored in particular registers, or may place the output in a designated 'result' register. This means move instructions will be required to satisfy these requirements, slowing the overall program down. Good register allocators allow for this by biasing allocation so that these move instructions can be eliminated wherever possible.

2.2 Isomorphism to the graph-colouring problem

The most common approach to register allocation is to reduce the problem to one of graph colouring, where we have a graph which we wish to colour with a maximum of K colours such that no two adjacent vertices share the same colour (TODO ref). Given the knowledge of which variables are live at the same time, we construct a clash graph in which the vertices represent registers and an edge between two vertices exists where the corresponding registers are simultaneously live. Treating a colour as representing a physical register on the target machine, we then attempt to 'colour' the graph with target registers such that vertices which are linked by a clash edge are always given different colours.

A successful colouring will naturally give a mapping from virtual to physical registers enforcing that no registers which are simultaneously live are placed in the same physical register. Formally verifying that the colouring resulting from this process will never affect the behaviour of the code being compiled was the core objective of this project.

The K-colourability problem is known to be NP-complete wherever K is greater than two, and so finding a truly optimal solution is rarely feasible in practice. Instead, register allocation algorithms are designed primarily to find a solution which is correct, and from there use heuristics to find a solution which is as optimal as can be achieved in reasonable time. There are several approaches to this, and many of them were verified in detail in this project. As discussed later, the definitions and proofs in the project are designed to be largely independent of the algorithm and so extending it to verify some of the algorithms not covered would be relatively simple.

2.3 Basic approaches to colouring

The simplest way of colouring a graph is to take each vertex in turn and assign it a new colour until all vertices have been coloured. This approach is guaranteed to generate a valid colouring (as verified in Section (TODO section)), but is also very inefficient as registers are never re-used for variables which could reside in the same register. In a system with few physical registers this will result in registers being spilled to memory, and as the order colours are assigned is arbitrary this could result in frequently-used registers being stored in memory causing constant loads and stores and poor performance.

An improvement is to take each vertex in turn and assign it the lowest register which isn't already in use by a vertex with which it clashes. This is fairly simple to implement, and significantly improves register utilisation as it re-uses old registers whenever the opportunity presents itself (TODO ref?).

However, lowest-first allocation doesn't always find opportunities to re-use old registers. A further refinement is to check whether we can avoid using a new register by swapping colours of already-allocated vertices. This often improves register utilisation further, as sub-optimal decisions made early on in the process can be changed when colouring new vertices later.

The algorithm resulting from applying all these refinements will work fairly well, but in practice the quality of the colouring is often also heavily dependent on the order in which we colour vertices. Decisions made at the beginning naturally constrain decisions made later, even if we allow the algorithm to swap existing colours. We therefore need to use a heuristic ordering step at the beginning to improve the order in which vertices are presented to the main colouring algorithm. Some such heuristics are described in the next section.

2.4 Heuristics

Heuristics are used in graph-colouring algorithms to select a vertex ordering which gives a more optimal solution. There are several heuristics which are used in practice, some of which were formally verified during the course of this project.

One of the simplest heuristics is to order the vertices by their degree. By presenting the vertices to the colouring algorithm in descending order of degree, vertices which are most likely to clash with others are coloured first so that those left over towards the end will be easier to colour without assigning new colours. There are also a few variations on this: saturation degree ordering, for instance, picks the next vertex to colour based on which vertex is currently adjacent to the most different colours. Incidence degree colouring is another variant which sorts based on the number of adjacent vertices which have already been coloured (TODO ref).

A more complex heuristic, which is useful when attempting to colour with at most K colours, is to pick at each stage the vertex of highest degree less than K (this constraint on its degree ensures that it can definitely be coloured if only K colours are available). This vertex is then removed from the graph and placed on top of a vertex stack. The process repeats, each time removing a vertex which is guaranteed to be colourable, until no vertices of degree less than K remain. Those which have not yet been removed are spilled to memory, and the rest of the vertices are popped off the stack one by one and passed to the colouring algorithm. The heuristic essentially priorities registers with few conflicts, spilling those with many conflicts to memory, and colours those of lowest degree first.

This is not an exhaustive list of the heuristics used in practice, but these two types are very common and so verification focussed primarily on these. As mentioned, most of the verification conditions were designed to be as general as possible such that other more exotic heuristics could also be verified without much additional effort.

Chapter 3

Related Work

TODO

Chapter 4

Live variable analysis and clash graph generation

Introduction

In order to achieve a complete end-to-end verification, it was decided that the verified allocator should take in some representation of three-address code (typical of this stage in the compilation process), and output three-address code with registers allocated. The overall proof would therefore show that the behaviour of the resulting code was exactly the same as that of the source in all situations. For this reason, a simplified representation of three-address code and an evaluation function were developed and an algorithm to generate clash graphs was verified. This chapter explores the representation used, the way 'colouring' registers was represented and the development and verification of the clash graph generator, along with the way a 'correct' register allocation was defined. The proofs showing that these definitions are correct and linking them to the projects overall goal will follow in Chapter (TODO: bringing everything together).

4.1 Three-address code representation

Here we explore the simplified representation of three-address code, how evaluation was defined and how colourings were modelled and applied.

4.1.1 Registers and instructions

In real systems, register naming conventions tend to vary between processors, and there are often multiple groups of registers available. For the purposes of verification, this was simplified by using natural numbers to refer to registers, as in practice these numbers could trivially be mapped to real register names. Furthermore, there will generally be a great many different operations available on a given machine, but as the focus is on register allocation the only thing that matters here is the registers read and those assigned. Compilers commonly use three-address code as an intermediate representation, so a given instruction essentially consists of two source registers and one destination register. This led to the very simplified instruction representation given below:

```
inst = Inst of num ⇒ num ⇒ num
```

For the main verification task, this was entirely satisfactory as it captures all the information needed. Later work extended this representation slightly: for instance, the addition of preferences meant it was necessary to include move instructions so that these could be used to compute preference graphs. Extensions to the three-address representation are discussed in Chapter (TODO: extensions chapter).

Code blocks were then represented as simple lists of these instructions, to simplify verification. The extension to more complex code featuring basic blocks is not too great a leap, and is discussed in Section (TODO: extensions-complex code). The main difference in using this representation is that live variable analysis is somewhat simpler.

4.1.2 Code evaluation

As mentioned, it was not necessary for verification to have a wide range of instructions. The register allocation can be proven correct simply by showing that evaluating each instruction with an arbitrary binary operator and storing the result in the destination register has the same effect as it did before allocation. The precise meaning of ‘has the same effect’ is explained in Section (TODO bringing together - proving stuff about code evaluation), where it is proven to hold for the register allocator built during this project. The evaluation function used is given below:

```
eval f s [] = s
eval f s (Inst w r1 r2::code) =
eval f ((w =+ f (s r1) (s r2)) s) code
```

Here ‘f’ is an arbitrary binary operator and ‘s’ is the store, a total function representing the values of all registers. Each instruction is evaluated by looking up the two source registers in the store and applying the binary operation to them, then updating the store function to map the destination register to the resulting value. The base case, where we have reached the end of the code block, simply returns the store as it is.

Chapter 5

Colouring algorithms

TODO

Chapter 6

Bringing everything together

TODO

Chapter 7

Extension features

TODO

Chapter 8

Summary and conclusions

TODO