Minimal Register Allocation

Kent Lee
Department of Computer Science
The University of Iowa
Iowa City, IA 52242
email: kent-lee@uiowa.edu

May 13, 1999

Abstract

Register allocation is a part of the code generation phase of a compiler where symbolic registers are assigned physical registers of a particular machine architecture. Classically, register allocation has been assumed to be equivalent to graph coloring, which is an NP-Complete problem. This paper shows register allocation is not necessarily as general as graph coloring and therefore can be solved in polynomial time. In addition, an algorithm is presented that solves the problem in $O(n^2)$ time.

1 Introduction

Aho, Sethi, and Ullman [Aho86] define a compiler as "... a program that reads a program written in one language - the source language - and translates it into an equivalent program in another language - the target language ...". A compiler typically accomplishes this task in phases which are usually described as scanning, parsing, and code generation. When the target language is the assembly or machine language of a particular machine, one part of the code generation phase is register allocation. Code generation can be simplified if the assumption can be made that the target machine has an unlimited number of registers. This is especially true for pools of similar registers such as general purpose registers or floating point registers. These unlimited sets of registers are referred to as *symbolic registers*, while the actual registers of the machine are referred to as *physical registers*.

A symbolic register is requested each time the code generator needs a physical register of the target machine. When a symbolic register is no longer needed by the computation being compiled, it is discarded and not used again. In other words, a symbolic register is *alive* from the time it is requested until it is discarded. Then, after code generation has completed, each symbolic register is assigned to a physical register so that no two symbolic registers that are alive at the same time share the same physical register. A more formal definition of the problem follows.

> Definition: A **register interference graph** is a simple undirected graph, $G = (V, E)$, where $V = \{s_1, s_2, ..., s_n\}$ is a set of symbolic registers and E is a set of edges where $(s_i, s_j) \in E$ iff s_i and s_j are *alive* at the same time.

Graph coloring is a problem from the area of Graph Theory that has similarities to register allocation. The graph coloring problem can be stated as: Given a simple undirected graph, $G = (V, E)$, find the minimum number of colors required so that each vertex of V is assigned a color and no two vertices in V share the same color if they are joined by an edge in E. The graph coloring problem is NP-Complete. The problem is described by Garey and Johnson [Garey79] and Karp [Karp72].

Register allocation and graph coloring are obviously closely related. In fact many texts on compiler construction claim they are equivalent. Aho, Sethi, and Ullman [Aho86] don't claim they are equivalent, but they do give heuristics that can be applied to a register interference graph to attempt to color it in a given number of colors. Pittman and Peters [Pittman92] claim that they are equivalent but also say "We have found an informed demand allocation scheme to give good results without the complexities of graph coloring...". Chaitin, et al. [Chaitin81] claim that given any simple undirected graph, G, they can generate a program whose corresponding register interference graph is isomorphic to G, and therefore register allocation is an NP-Complete problem.

These claims are misleading. In fact, an equally strong argument can be made against the equivalence of register allocation and graph coloring. Claims both for and against the equivalence of the two problems make an underlying assumption about the *register allocation methodology.* Section 2 contains a discussion of register allocation methodologies. Section 3 proves a significant result about one of the allocation methodologies, called *relaxed register allocation.* Section 4 presents an algorithm for relaxed register allocation. Section 5 concludes by pointing out areas of continuing research.

2 Allocation Methodologies

A register allocation methodology describes a decision policy that dictates when new symbolic registers should be created during code generation. At one extreme it is possible to create only one symbolic register for each physical register of the machine. While this methodology uses the minimum number of symbolic registers, it does not assist in register allocation.

A little less extreme methodology assigns one symbolic register to each *program variable.* Chaitin [Chaitin81] describes this register allocation methodology. For purposes of discussion let us refer to this methodology as *strong register allocation.* Chaitin's claim is that register allocation and graph coloring are equivalent. He demonstrates this through the use of an example. Consider an arbitrary simple undirected graph, $G = (V, E)$. Let $V = \{v_1, v_2, ..., v_n\}$ be the set of vertices of G and $(v_i, v_j) \in E$ iff there is an edge between vertices v_i and v_j. Using his strong register allocation approach he designs a program template for which the corresponding register interference graph is G. A pseudo-code version of this template is shown in Figure 1. Although Figure 1 is not the exact same template Chaitin presented, the idea is the same. Each vertex, $v_i \in V$, of G has a corresponding program variable declaration. If there is an edge, $(v_i, v_j) \in E$, between two vertices in G then there is an assignment statement involving the two corresponding program variables (i.e. $x := v_i + v_j$) that causes v_i and v_j to interfere with each other. Since the two variables interfere with

```
Program P;

var    x
       v_1,
       v_2,
       ...
       v_n;
begin
x:=v_g + v_h; /* edge 1 */
x:=v_j + v_k; /* edge 2 */
...
x:=v_l + v_m; /* edge i */
end.
```

Figure 1: Chaitin's source program template

each other, or in other words are *alive* at the same time, the register interference graph must have a corresponding edge. From this description, he claims that G is the register interference graph for the program. Therefore, register allocation is equivalent to graph coloring.

The argument given above is convincing. However, it relies on Chaitin's strong register allocation methodology. This methodology leads to arbitrarily complex register interference graphs. Chaitin recognizes this and suggests creating more than one symbolic register for some program variables to reduce the complexity of the register interference graph by "uncoupling distant regions". If this approach is taken to the extreme another register allocation methodology emerges. Let us call this new methodology *relaxed register allocation.*

Under the relaxed register allocation methodology one symbolic register is assigned to each *program variable reference.* For example, consider the assignment statement and corresponding target language code segment in Figure 2. Assume that the target language is the assembly language of a simple load/store architecture and the first operand is the destination of the result for arithmetic operations (e.g. mul a, b is a:=a*b). Figure 3 shows the f:=(a + (b * c)) + (d + (e * e))

ld	s_1,	a
ld	s_2,	b
ld	s_3,	c
mul	s_2,	s_3
add	s_1,	s_2
ld	s_4,	d
ld	s_5,	e
ld	s_6,	e
mul	s_5,	s_6
add	s_4,	s_5
add	s_1,	s_4
st	s_1,	f

Figure 2: Source program statement and target code

register interference graph constructed using the relaxed register allocation methodology for the associated target code in Figure 2.

To contrast the two register allocation methodologies, using the relaxed methodology each *reference* of a program variable is assigned a symbolic register instead of each program variable being assigned only one symbolic register as is done using the strong register allocation methodology (e.g. In Figure 2, using the relaxed approach, the program variable "e" is assigned two symbolic registers, one for each reference of "e"). The relaxation of register allocation simplifies the underlying register interference graph in an important way which will be shown in the next section. This simplification is what Chaitin is alluding to when he says that the register interference graph may be simplified by using more than one symbolic register for some program variables.

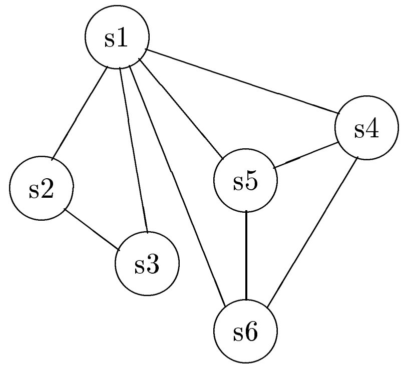

Figure 3: Register interference graph corresponding to Figure 2

3 The Lifetime Property

A mapping from the symbolic register set to the physical register set that satisfies the constraints of the register interference graph in figure 3 is $\{s_1 \to p_1, s_2 \to p_2, s_3 \to p_3, s_4 \to p_2, s_5 \to p_3, s_6 \to p_4\}$. This also happens to be the minimal coloring of the graph. Even though graph coloring is NP-Complete in general, graphs constructed according to the relaxed register allocation methodology can be minimally colored in polynomial time. A proof of this is given below. But first, there are a couple of definitions that have implications in proving this result.

> Definition: A **complete graph** is a simple undirected graph $G = (V, E)$ where for each $v_i, v_j \in V, (v_i, v_j) \in E$.
>
> Definition: A **clique** in a simple undirected graph is a complete subgraph. If a clique of size k exists in a graph, we say it contains a k-clique.

It should be clear that a graph, G, with a k-clique cannot be colored in less than k colors since there are k vertices in G which are all neighbors of each other (i.e. they all share edges).

The problem with strong register allocation is that symbolic registers do not have well-defined lifetimes. For a symbolic register there is no restriction on other symbolic registers it may interfere with. In contrast, the relaxed register allocation methodology insures that a symbolic register is not reused after the variable reference it was created for. That is, the symbolic register has a well-defined lifetime. It is born, lives during the variable reference, and dies. It may not interfere with any other symbolic registers after its death. If we apply this register allocation methodology to the program of Chaitin, the register interference graph becomes trivial. In fact, the relaxed register interference graph associated with any instance of his program template may be colored in just two colors!

The key to solving the relaxed register allocation problem is the lifetime property. This property was implied above when we stated that symbolic registers have well-defined lifetimes. The property can be stated more precisely.

> Definition: A simple undirected graph $G = (V, E)$ exhibits the **lifetime property** iff there exists a permutation $\pi : N \to V$ so that if $(\pi(i), \pi(k)) \in E$, then $(\pi(i), \pi(j)) \in E$, $i < j < k$.

Note that finding this permutation π, if it exists, for a graph would take exponential time in general, so this does not solve the graph coloring problem for all graphs for which the lifetime property holds. However, in the case of register interference graphs constructed using the relaxed register allocation methodology, the permutation is provided. It is simply the order in which the symbolic registers were allocated.

A straightforward inductive proof exists for the minimal coloring of a simple undirected graph $G = (V, E)$ with the lifetime property. The induction is done on the number of vertices in V.

> **Claim**: The minimum number of colors required to color a simple undirected graph, $G = (V, E)$, exhibiting the lifetime property can be determined in polynomial time.
>
> **Proof**: (by induction on the cardinality of V)

Base case: $\mid V \mid = 0$. Clearly, G can be colored in 0 colors.

Inductive case: Assume for all $G' = (V', E')$ exhibiting the lifetime property with $\mid V' \mid = n-1$ that G' can be minimally colored in polynomial time. Since G exhibits the lifetime property, there exists a permutation of its vertices for which the lifetime property holds. This property was stated above. Without loss of generality, assume $V = \{v_1, v_2, ..., v_{n-1}, v_n\}$ is the set of vertices of G where the vertices are numbered according to the lifetime permutation. Then let $G' = (V', E')$ where $V' = \{v_1, v_2, ..., v_{n-1}\}$ and E' be the set of edges of E such that $(v_i, v_n) \notin E' \; \forall i$. Clearly G' is a simple undirected graph for which the lifetime property holds. By the induction hypothesis, G' may be minimally colored in polynomial time. Without loss of generality, assume G' may be colored in k colors. There are three cases to be considered for the graph G.

Case 1: v_n has less than k neighbors. Clearly, v_n can be colored one of the k colors of G' that is not one of the colors of the neighbors of v_n. The neighbor colors of v_n can be determined in polynomial time by examining the coloring for G' and the neighbors of v_n in $E - E'$. Since no new color is needed, G is still minimally colored in k colors.

Case 2: v_n has k neighbors. As stated in case 1, the neighbors of v_n can be determined in polynomial time. Then, since the lifetime property holds for G, if $(v_m, v_n) \in E$ then $(v_i, v_j) \in E$, for $m \leq i < j \leq n$. This means that v_n forms a $(k+1)$-clique with its k neighbors in V. But we observed earlier that a simple undirected graph with a $(k+1)$-clique cannot be colored in less than $k+1$ colors. Therefore, G is still minimally colored with $k+1$ colors.

Case 3: v_n has more than k neighbors. As stated in case 1, the neighbors of v_n can be determined in polynomial time. Then there exists at least a $(k+1)$-clique in G' by the same argument as case 2. But this means that G' cannot be colored with k colors. This contradicts our induction hypothesis, so this case is invalid.

Therefore, G can be minimally colored in polynomial time if G' can minimally colored in polynomial time. By induction it follows that all simple undirected graphs exhibiting the lifetime property may be minimally colored (assuming we know the permutation of the vertices of V) in polynomial time. □

The proof given here is the central idea exploited in the algorithm given in the next section.

4 Relaxed Register Allocation

To design any algorithm there are two main problems that must be solved. The first problem is the data representation. A graph is a rather complex object. It requires the representation of two sets, a set of vertices and a set of edges. However, because of the lifetime property, register interference graphs have a simple and eloquent representation. They may be represented as lists of natural numbers. By the lifetime property for a register interference graph $G = (V, E)$, there is a permutation, π, so that if $(\pi(i), \pi(k)) \in E$, then $(\pi(i), \pi(j)) \in E$ for $i < j < k$. The existence of edges from vertex $\pi(i)$ to vertices following it in the permutation π may be represented as a magnitude. Let $e : V \rightarrow N$ such that $e(v) = n$ where n is the number of vertices in V that follow v in the permutation π and are connected to v by an edge in E. Then, G may be represented by $[e(\pi(n)), e(\pi(n-1)), ..., e(\pi(2)), e(\pi(1))]$. It works best if the list is constructed in the reverse order of the permutation π since lists are usually constructed starting with the tail. For purposes of discussion let us refer to this graph representation as the *register list*. So, for instance, the graph shown in Figure 3 can be represented by the register list $[0, 1, 2, 0, 1, 5]$. Vertex s_1, represented by the last position of the list, has edges to every other vertex in the graph. This is indicated by the 5 in its position in the register list. Vertex s_2 has an edge between it and s_3. This is represented by the 1 in the fifth position of the register list.

The algorithm is given below. It is written in Standard ML. Some auxiliary functions are given for generating all possible register interference graphs with a specified number of vertices, and for checking that the coloring algorithm works for all register interference graphs of a given size.

4.1 The Coloring Algorithm

The algorithm for coloring a register interference graph constructed according to the relaxed register allocation methodology is quite simple. It consists of just four ML functions. The function member has two parameters, a natural number, n, and a list, l. It returns true if n is a member of l and false otherwise.

```
fun member(x, nil) = false
  | member(x, y::z) = if (x = y) then true else member(x, z);
```

The function neighborcolors has three parameters. They are a register list, rl, a *color list*, *cl* and a counter. The color list *cl* is a list of the colors assigned to a correct minimal coloring of the register list rl. For example, the graph in Figure 2 is represented by the register list $[0, 1, 2, 0, 1, 5]$ and its color list is $[3, 2, 1, 2, 1, 0]$ If rl is a register list of a graph and *cl* is its corresponding color list then the function neighborcolors returns a list of the colors of the neighbors of vertex x in the graph represented by the register list x::rl. Note that the symbol "::" is the list constructor in Standard ML.

```
fun neighborcolors(nil, y, z) = nil
  | neighborcolors(x, nil, z) = nil
  | neighborcolors(x1::x2, y1::y2, z) = if z < x1
        then y1::neighborcolors(x2, y2, z+1)
        else neighborcolors(x2, y2, z+1);
```

The function firstnotin has two parameters, a color list and a color. It returns the first color not in the color list. This is the minimum color that is not already used in the color list. Note that the color list may have gaps in it.

```
fun firstnotin(x, y) =
       if member(y, x) then firstnotin(x, y+1) else y;
```

The function color has one parameter, a register list x::rl. It operates recursively by first minimally coloring rl. Then it colors x by finding the first color not used by the neighbors of x in the color list for rl. It returns the color list for $x{::}rl$.

```
fun color(nil) = nil
  | color(x::y) =
     firstnotin(neighborcolors(y, color(y), 0), 0)::color(y);
```

4.2 Correctness Verification

The following functions are used to verify the correctness of the relaxed register allocation algorithm. Although these functions do not prove the correctness of the algorithm, they have been used to verify correctness for all register interference graphs for which the lifetime property holds of eight or less vertices. The function max returns the maximum of two numbers. The function maxelm returns the maximum element of a list.

```
fun max(x:int, y:int) = if x < y then y else x;

fun maxelm(nil) = 0
  | maxelm(x::y) = max(x, maxelm(y));
```

The function neighbors has two parameters, a register list rl and a counter. It returns the number of neighbors of x in the register list $x{::}rl$. $1 + neighbors(rl, 0)$ is the size of the maximum clique which includes vertex x in the register list $x{::}rl$.

The function maxclique has one parameter, a register list $x{::}rl$. It returns the maximum clique of the graph represented by $x{::}rl$. It does this by returning the max of the maximum clique including x and the maximum clique of rl.

```
fun neighbors(nil, z) = 0
  | neighbors(x::y, z) = if z < x
     then 1+neighbors(y, z+1) else neighbors(y, z+1);

fun maxclique(nil) = 0
  | maxclique(x::y) = max(1+neighbors(y, 0), maxclique(y));
```

The function coloring has two parameters, a register list, rl, and a color list, cl. The function coloring checks that cl is a valid coloring for rl. Note that it does not check that it is a minimal coloring. It only checks that the color assigned to a vertex x is not a member of the neighborcolors of x.

```
fun coloring(nil, x) = true
  | coloring(x, nil) = true
  | coloring(x1::y1, x2::y2) =
          ((not (member(x2, neighborcolors(y1, y2, 0))))
     andalso coloring(y1, y2));
```

The functions xlist, ylist, and allgraphs generate all graphs of size $x+1$. Passing x as an argument to the function allgraphs results in a list of all register lists of $x+1$ vertices.

```
fun xlist(x, y) = if x > 0
   then (y@(x::nil))::xlist(x-1,y)
   else (y@(0::nil))::nil;

fun ylist(x, nil) = nil
  | ylist(x, y::z) = xlist(x,y)@ylist(x,z);

fun allgraphs(x) =
        if x > 0 then ylist(x,allgraphs(x-1)) else [[0]];
```

The functions checkit and check take care of checking that the color algorithm produces a correct minimal coloring for all register interference graphs of size x. If a graph fails to be colored minimally and correctly then it is returned by the function check. If all graphs were colored minimally and correctly then nil (i.e. [ ]) is returned.

```
fun checkit(nil) = nil
  | checkit(x::y) =
    let val z = color(x)
    in
        if coloring(x,z) andalso ((1+maxelm(z)) = maxclique(x))
        then checkit(y) else x
```

```
    end;

fun check(x) = if x > 0 then checkit(allgraphs(x-1)) else nil;
```

As stated above, the algorithm given here has been checked for all register interference graphs of size one vertex up to eight vertices. However, a real proof of correctness must be done by induction. This has been done using an automated inductive theorem prover and the proof appears in [Lee99].

5 Conclusion and Future Work

Contrary to popular belief, register allocation and graph coloring are not necessarily equivalent. The differences between register allocation and graph coloring are based on the fact that register interference graphs constructed according to the *relaxed register allocation methodology* are restricted by the *lifetime property*. Therefore, because of the lifetime property, a polynomial time algorithm exists for the minimal coloring of relaxed register interference graphs. In fact, by inspection, the algorithm has $O(n^2)$ complexity and more efficient algorithms are likely possible.

The relaxed register allocation methodology relies on a demand driven approach to register allocation. Symbolic registers are created on demand by the code generator. As stated above, Pittman and Peters [Pittman92] claim that an informed demand allocation scheme gives good results without the complexities of graph coloring. This intuitive insight into register allocation has been formalized in this paper. Formally, the strength of a demand driven approach is the lifetime property.

Future work in this area includes the search for transformations on register interference graphs that preserve the lifetime property while improving the efficiency of generated target code. These optimizations include common subexpression elimination, register spilling, and subsumption as applied to the relaxed register allocation methodology.

6 Acknowledgements

I would like to thank Tom Halverson, Dr. Sriram Pemmaraju, Dr. Ken Slonneger, Eric Van Wyk, and Dr. Hantao Zhang for their help in preparing this paper.

References

[Aho86] Aho, Sethi, and Ullman, Compilers: Principles, Techniques, and Tools (Addison Wesley), 1986.

[Chaitin81] Chaitin, Auslander, Chandra, Cocke, Hopkins, Markstein: Register Allocation via Coloring, Computer Languages, Vol. 6, pp. 47-57, 1981.

[Garey79] Garey and Johnson: Computers and Intractibility: A Guide the the Theory of NP-Completeness (Freeman), 1979.

[Karp72] Karp, R.M.: Reducibility among Combinatorial Problems, in R.E. Miller and J.W. Thatcher (eds.) Complexity of Computer Computations (Plenum Press, New York), pp. 85-103, 1972.

[Lee99] Lee and Zhang: Formal Development of a Minimal Register Alocation Algorithm. Technical Report 99-07, University of Iowa, Iowa City, Iowa, 1999.

[Pittman92] Pittman and Peters: The Art of Compiler Design: Theory and Practice (Prentice Hall), 1992.