

Register Allocation Via Graph Coloring

WHITE PAPER

Table of Contents

Abstract3

Introduction.....4

Graph Coloring Approach.....6

Register Allocation Via Graph Coloring: Pre-requisites.....9

Register Allocation Via Graph Coloring: Implementation10

Conclusion19

References.....20

Abstract

Register Allocation is one of the most important problems in compiler optimization. It is also one of the most studied problems in compiler theory, and a vast number of different algorithms exist to solve it. Register Allocation can happen over a basic block (local register allocation), over a whole function/procedure (global register allocation), or across function boundaries traversed via call-graph (inter-procedural register allocation).

The objective of this technical paper is to give an overview of the Graph Coloring approach, and how it can be applied to perform Local Register Allocation.

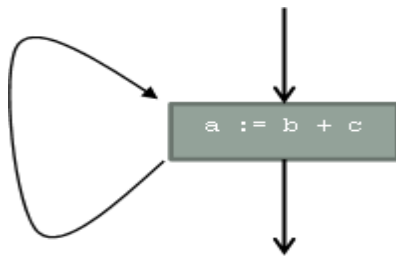
The target audience for this paper would be Computer Science students studying Compiler Development, or Compiler Developers interested in understanding the Graph Coloring Algorithm for Register Allocation.

Introduction

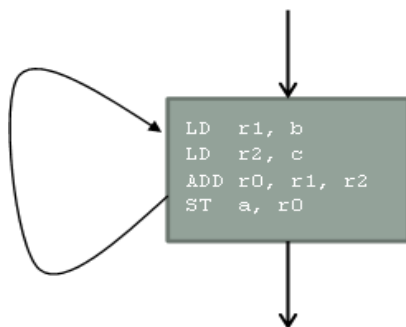
Register Allocation is the process of assigning a large number of target program variables onto a small number of CPU registers.

The code generator generates intermediate code using as many temporaries as necessary. This simplifies and optimizes the code generation process. However, it complicates the final translation to assembly.

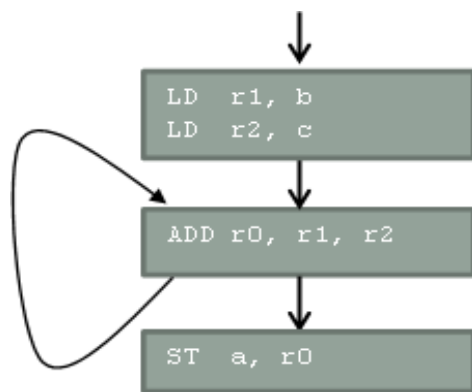
The register allocation problem is to rewrite the intermediate code to use fewer temporaries than there are machine registers. Its main focus is to assign more temporaries to a register without changing the program behavior. It directly reduces running time by converting memory access to register access. Accessing a register's value will require very few clock cycles (most likely, just one). As soon as the memory is accessed, things get more complex. Cache controllers / memory buses get involved and the operation is going to take considerably more time. Therefore, register provides faster access time and thus, improves program efficiency. For example, consider the program



The above program will be expanded to the intermediate code below:



Here, the LD (memory load) and ST (memory store) instructions will be executed for each iteration and hence it slows down the process. Since the value of the variables *b*, *c* and *a* are not changing with each iteration, LD/ST operation can be performed only once; the value will be saved in register for performing the operation as shown below:



Here, LD (memory load) and ST (memory store) will be performed only once during the loop execution, thereby improving program efficiency.

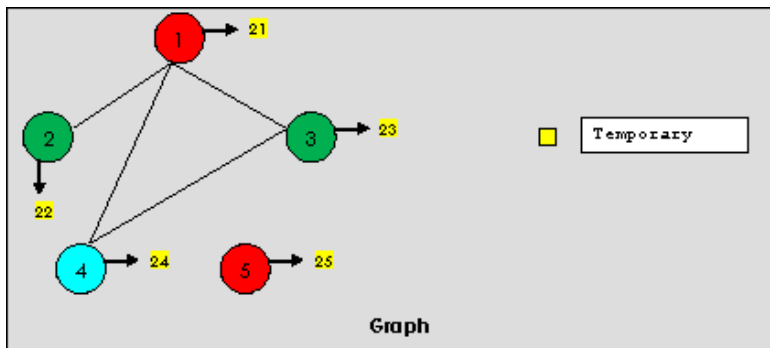
Graph Coloring Approach

The problem of Register Allocation can be viewed as a graph coloring problem. Suppose that we have 'n' available registers: r1, r2... rn; if we view each register as a different color, then the register allocation problem is equivalent to the graph coloring problem where we try to assign one of the 'n' different colors to the graph nodes. This way, no two adjacent nodes will have the same color.

If we view each node as a temporary and if there is no edge between the two nodes of a graph, then we can assign the same physical register to both of them.

On the other hand, if there is an edge between two nodes, then we should not assign the same physical register to the temporaries defined by them. This is because the values of both temporaries will be needed at the same point of time.

For example, as shown in the diagram below, there is no edge between node 2 and 3, thus both the nodes are colored with same color. Whereas, there is an edge between '1' and '2', thus both the nodes are colored differently.



Note:

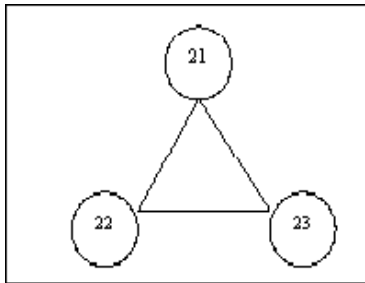
The number of colors available is equal to the number of physical registers. Each color represents a physical register.

Let 'N' be the number of available physical registers. The approach is based on the following observation:

"If we remove a node from the graph that has a degree less than 'N', no matter how its neighbors are colored, there will be at least one color left for the node"

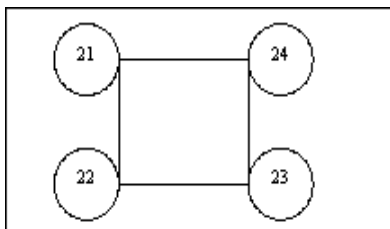
It means that if a virtual register 'u' has a degree less than 'N', then we will always have a physical register for it. But if a virtual register has a degree more than or equal to 'N', then there may or may not be a physical register available for it.

For example, consider the following interference graph:

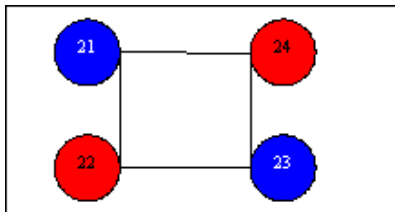


Let's assume we have only '2' physical registers available. In this graph, the degree of each node is equal to the number of available physical registers, i.e. '2'. But this graph is not colorable with '2' colors.

Now, consider one more graph:



Here too, the degree of each node is equal to the number of available physical registers i.e. '2'. But this graph is colorable with '2' colors as shown below:



Thus, it is not necessary that if a node has a degree equal to or greater than the number of available registers, then the node is un-colorable.

So, first we find out all the nodes with a degree less than 'N' because these nodes will be 100% colorable and then remove these nodes from the graph. Removing a node from a graph will reduce the degree of all its adjacent nodes by '1'.

But, if a node has a degree more than or equal to 'N', then it can be or cannot be colorable, so we simply mark that its degree is more than 'N'. We try to color the graph with these nodes, while also giving priority to the nodes having degrees equal to or more than 'N' registers.

Thus, we remove all the nodes with degrees less than 'N' one by one and push it on the stack. We use a stack because these are the nodes which will be a 100% colorable, so we can allocate registers to them

in the end. We give priority to those registers having degrees equal to or more than 'N'. If we are able to color them, it means we can color the complete graph.

Thus, we allocate physical registers to the virtual register in the inverse order they were removed from the graph.

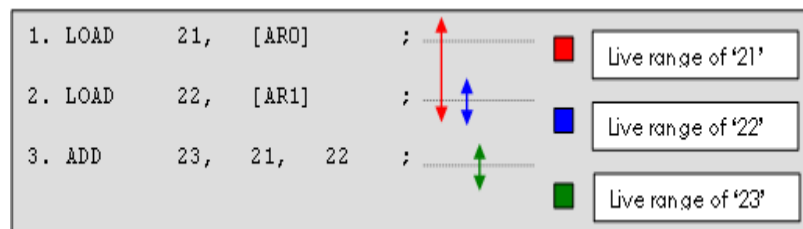
Register Allocation Via Graph Coloring: Pre-requisites

Live Range of a Virtual Register:

If a virtual register is defined at instruction “u” and it is last used at instruction “v”, then its live range will be {u, v-1}.

The start point of the live range of a virtual register is the instruction where it is defined. The end point of the live range of a virtual register is the instruction preceding the instruction where it is last used. It means that the instruction where a virtual register is last used, is not considered as a part of the live interval. This is because it enables the reuse of the register in the same statement that last uses it.

For example, consider the following instructions:



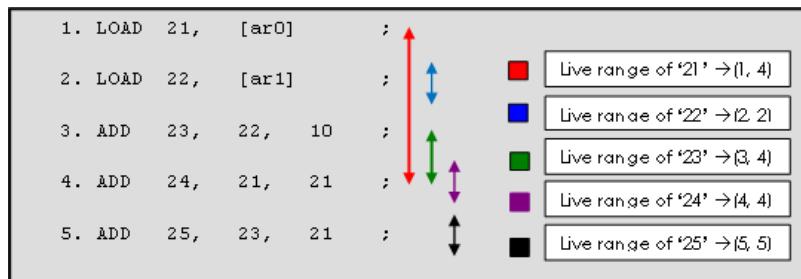
In the above example, the live range of 21 is {1, 2}, the live range of '22' is {2, 2}, and the live range of 23 is {3,3}. The live range of '21' and '22' are not interfering with live range of '23'.

Interference Graph

An interference graph, $Gr = (Vr, Er)$ can be defined as a set of vertices (Vr) and set of edges(Er), where:

- Every vertex 'v' corresponds to an instruction that defines a new virtual register.
- There exists an edge between two vertices ('u' and 'v'), if the live ranges of two virtual registers defined by instructions 'u' and 'v' intersect.

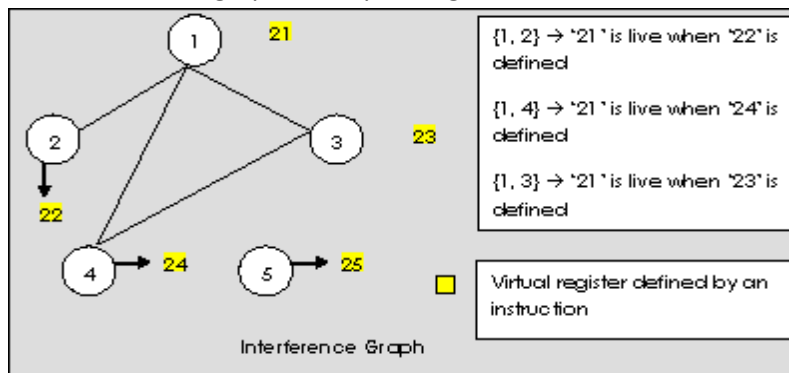
In this graph, the nodes represent virtual registers. An edge exists between two virtual registers if they are live at the same time. The interference graph is used for assigning registers to virtual registers. If two virtual registers do not interfere (i.e., there is no edge between these two virtual registers in the interference graph), then we can use the same physical register for both of them, thus reducing the number of registers needed. On the other hand, if there is a graph edge between two virtual registers, then we should not assign the same physical register to them, since this physical register needs to hold the values of both the virtual registers at one point of time. For example, consider the set of instructions below:



The interference graph will contain nodes corresponding to each instruction which is defining a new virtual register. For example, in the above code, each instruction 1, 2, 3, 4, and 5 are defining a new virtual register '21', '22', '23', '24' and '25' respectively. Therefore, the interference graph corresponding to the above code will contain 5 nodes corresponding to each instruction.

If there is an instruction which is not defining any new virtual register, then no node will be created for that instruction. For example, "store" instruction will write the value of previously defined registers to memory. So, no node will be created for "store" instruction.

The interference graph corresponding to the above code will look like:



Here,

- There is an edge between '1' and '2' in the interference graph. This is because the live range of '21' defined in instruction '1' is interfering with live ranges of '22' defined in instruction '2'. Similarly, we get the other edges in the graph.
- The number pointing by an arrow represents the virtual register defined by an instruction.

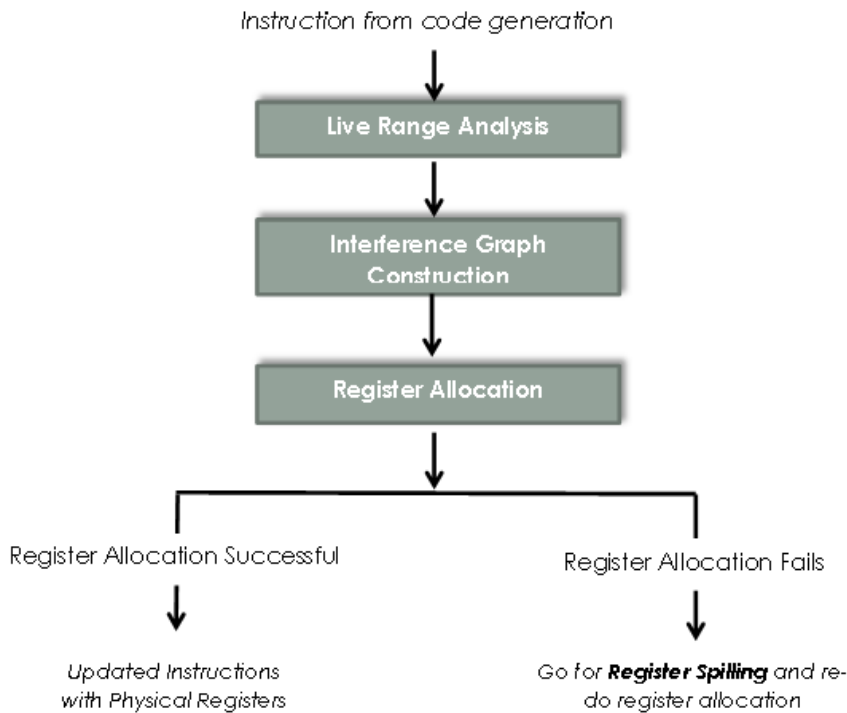
Register Allocation Via Graph Coloring: Implementation

To perform register allocation via graph coloring, we need to complete the following steps:

- Step1: Compute the Live Ranges of the virtual registers defined in a basic block
- Step 2: Construct the Interference Graph
- Step 3: Allocate physical registers to the virtual registers using the graph coloring approach

- Step 4: If the register allocation is successful, then return, else go for register spilling.

The flow of the algorithm will be as below:



Let's understand each step in detail:

Step1: Compute the Live Ranges of the virtual registers defined in a basic block

Algorithm

1. Traverse the instructions one by one, starting from the first instruction of the basic block, till the last instruction of the basic block. For each instruction, repeat the following steps:
 - If an instruction is defining any virtual register, then we will update its Live Range Start as well as its Live Range End as current instruction index.
 - If an instruction is using any virtual register, then we will update its Live Range End as "current instruction index - 1"
 - If an instruction is redefining any virtual register, then we will update its LiveRangeEnd as "current instruction index - 1"

Example

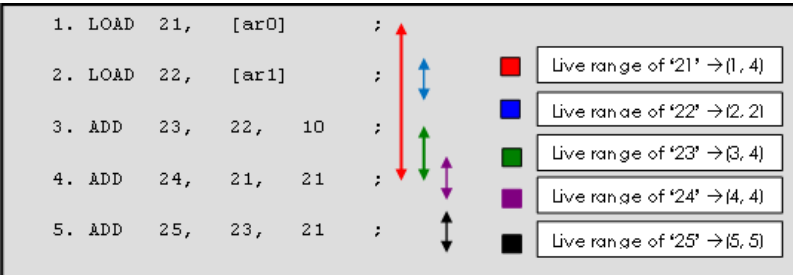
Consider the following set of instructions:

```

1. LOAD 21, [ar0] ;
2. LOAD 22, [ar1] ;
3. ADD 23, 22, 10 ;
4. ADD 24, 21, 21 ;
5. ADD 25, 23, 21 ;

```

Based on the above algorithm, the live range of these registers will be:



Step 2: Construct the Interference Graph as Adjacency List

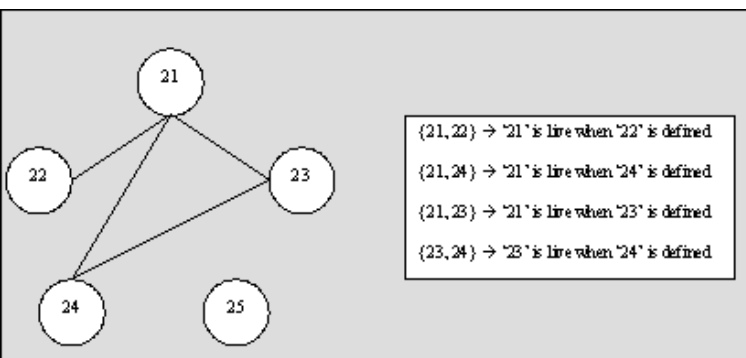
Algorithm

Let 'N' be the number of virtual registers defined or used in a basic block.

1. Create nodes corresponding to each virtual register defined in a basic block
2. For each virtual register 'u' in a basic block:
 - For each virtual register 'v' such that 'v' is greater than 'u'
 - If (Live Range End (u) \geq Live Range Start (v)) and (Live Range End (v) \geq Live Range Start (u))
 - Add an edge between 'u' and 'v'
 - End For
3. End For

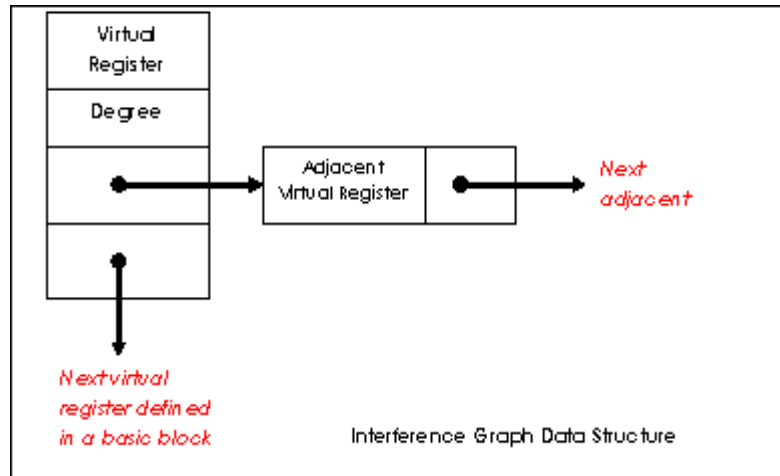
Example

On applying the above algorithm on the live ranges computed in Step1, the interference graph will be as follows:



Representation of the Interference Graph

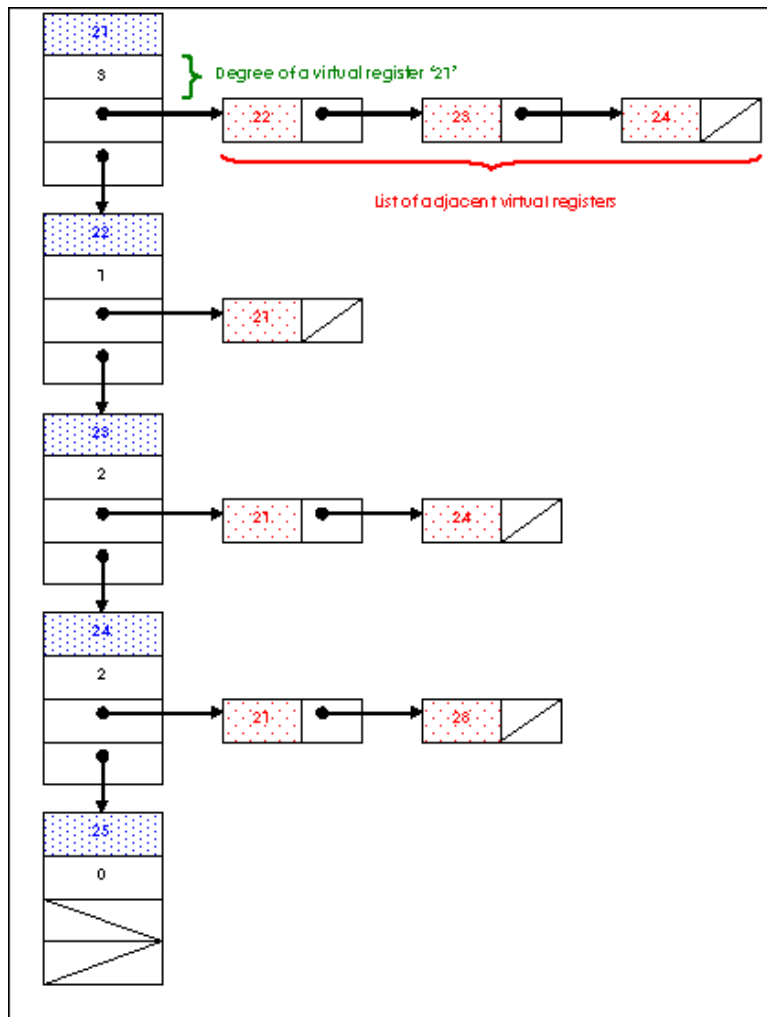
The above graph can be implemented as adjacency list. A node of an adjacency list will be as shown below:



The above node contains the following information:

- **“Degree”** to represent the degree of a node (virtual register) in the interference graph.
- **“Pointer”** points to the list of adjacent nodes (virtual registers) to the current node.
- **“Next Node Pointer”** points to other nodes (virtual registers) in a graph

The number of nodes in the adjacency list is equal to the number of virtual registers used and defined in a basic block. The adjacency list representation of the interference graph constructed in step '2' will be as shown below:



Step 3: Register Allocation via Graph Coloring Approach

Algorithm

1. Start searching in the adjacency list for a virtual register with a degree less than 'N' and greater than '-1'.
2. If such a virtual register is found, traverse the list of virtual registers adjacent to it and decrement the degree in the data structure for each of these adjacent virtual registers. Set the length for the chosen virtual register to '-1'. This helps to signify that it is no longer in the graph. Push the chosen virtual register into the stack.
3. Repeat the above two steps till the degree of all virtual registers is less than '0', or no more virtual registers have a degree less than 'N'.
4. If any virtual register has a degree greater than 'N' or equal to 'N', we mark this node for spilling. Set the length for the chosen virtual register to '-1'. Push the chosen virtual register into the stack.
5. If all the virtual registers have a degree less than "0", then the graph is colored and we can stop. Pop the registers from the stack and allocate them physical registers. While allocating physical

registers, check the physical registers allocated to its adjacent registers, and then choose a physical register that is still free.

6. If no physical register is free, then go for register spilling.

Example

Assume we have '3' physical registers available (GR0, GR1 and GR2)

Consider the interference graph constructed in step '2'.

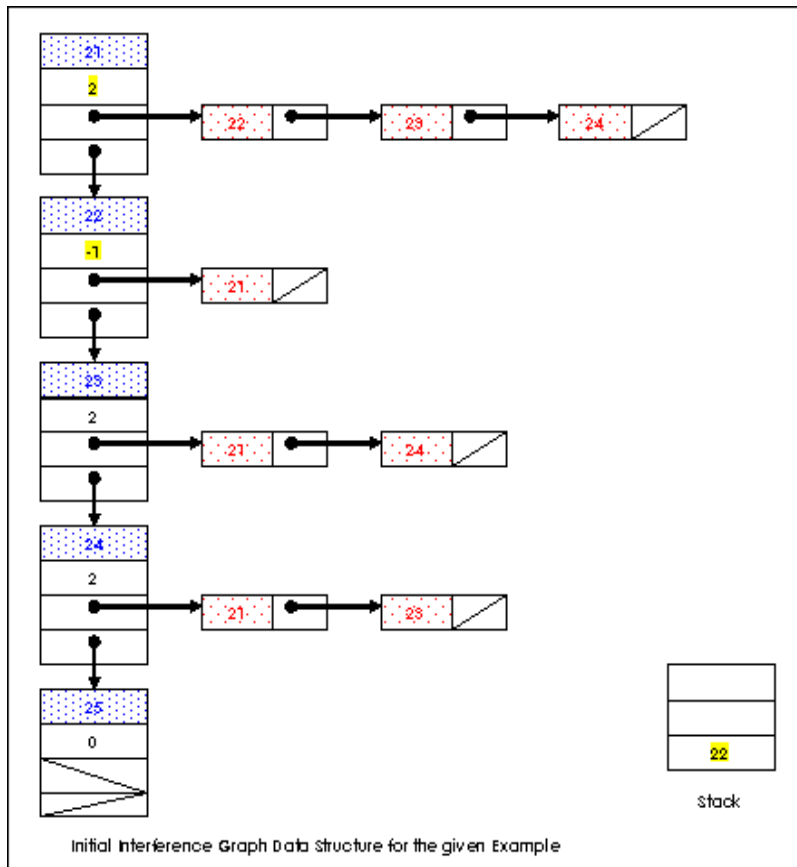
1. Start searching in the adjacency list for a virtual register with a degree less than 'N' and greater than '-1'.

The virtual register '22' has degree '2' i.e. less than '3'.

2. If such a virtual register is found, traverse the list of virtual registers adjacent to it and decrement the degree in the data structure for each of these adjacent virtual registers. Set the length for the chosen virtual register to '-1'. This helps to signify that it is no longer in the graph. Push the chosen virtual register into the stack.

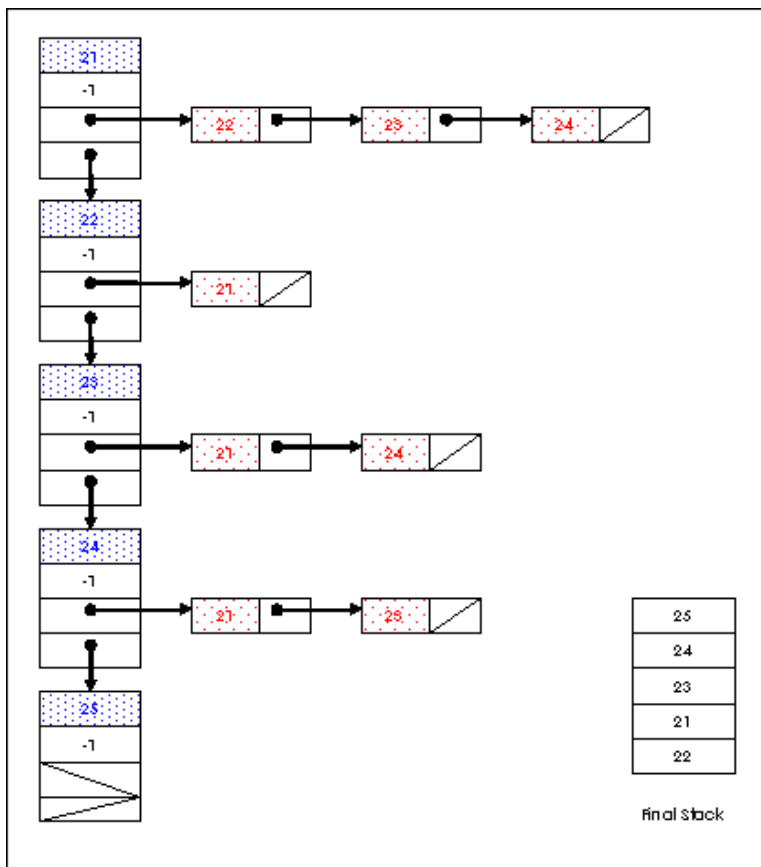
The register adjacent to '22' is '21'. We decrement the degree of '21' and set the degree of '22' as '-1'.

Push '22' into the stack. The modified data structure will be as shown below:



3. Repeat the above two steps till the degree of all virtual registers is less than '0', or no more virtual registers have degrees less than 'N'.

The final data structure will be as shown below:



4. If any virtual register has a degree greater than 'N' or equal to 'N', we mark this node for spilling. Set the length for the chosen virtual register to '-1'. Push the chosen virtual register into the stack.
This step is not valid as there are no more virtual registers left with a degree greater than '3'.

5. If all the virtual registers have a degree less than "0", then the graph is colored and we can stop. Pop the registers from the stack and allocate them physical registers. While allocating physical registers, check the physical registers allocated to its adjacent registers, and then choose a physical register that is still free.

Since all the virtual registers have a degree less than "0", we will pop the virtual register from the stack and allocate registers one by one keeping their adjacent virtual register in mind.

Virtual Register	Physical Register
25	GR0
24	GR0
23	GR1
21	GR2
22	GR0

6. **If no physical register is free then go for register spilling.**

This step is not applicable as the given interference graph is colorable using 3 colors.

However, if any node is marked for spilling, we need to generate a store instruction (to store its value into memory) after its definition and a load instruction (to read its value from memory) before its usage. Spilling is a costly process. Therefore, spilling should be done optimally.

Register spilling is in itself a vast topic to be discussed and hence will be considered in subsequent papers.

Conclusion

Register allocation is one of the most important problems in compiler optimization. It is also one of the most studied problems in compiler theory, and a vast number of different algorithms exist to solve it.

However, Graph coloring can be used to optimally perform register allocation. Each variable becomes a node in the graph. If two variables can't reside in the same register, we add an edge between them. The coloring algorithm assigns colors so that no neighboring nodes receive the same color. If there is no node left uncolorable, then it will be marked for spilling.

References

- 1) Compilers - Principles, Techniques & Tools (Addison Wesley - Aho, Sethi, Ullman)
- 2) Advanced Compiler Design Implementation – Steven S. Muchnick
- 3) G. J. Chaitin. Register allocation & spilling via graph coloring. In Proceedings of the 1982 SIGPLAN symposium on Compiler construction pages 98–101. ACM Press, 1982
- 4) Preston Briggs. Register allocation via graph coloring. PhD thesis, Rice University, Houston, TX, USA, 1992

A Note of Thanks

Special thanks to Pushpendra Maharya (pmaharya@hcl.com), Yash Agarwal (yashc@hcl.com) and Manav Manchanda (manavm@hcl.com) for their support in developing this paper.

Author Info



Meena Jain (meena.j@hcl.com), is a Technical Specialist in HCL's Engineering and R&D Services Group. She has more than 10 years of industry experience working primarily in Compilers. Her areas of interest have been compiler back-end, assembly-to-assembly translations and porting of compilers across CISC, RISC and embedded platforms. Her major work has been the development of the "C Compiler targeted for the 16 bit DSP Processor" and "GCC porting to VLIW DSP Architecture".



Hello, I'm from HCL's Engineering and R&D Services. We enable technology led organizations to go to market with innovative products and solutions. We partner with our customers in building world class products and creating associated solution delivery ecosystems to help bring market leadership. We develop engineering products, solutions and platforms across Aerospace and Defense, Automotive, Consumer Electronics, Software, Online, Industrial Manufacturing, Medical Devices, Networking & Telecom, Office Automation, Semiconductor and Servers & Storage for our customers.

For more details contact: ers.info@hcl.com

Follow us on Twitter: <http://twitter.com/hclers> & LinkedIn: <http://lnkd.in/bt8hDXM>

View our blog-site: <http://www.hcltech.com/blogs/engineering-and-rd-services>

Visit our website: <http://www.hcltech.com/engineering-rd-services>

HCL