Register Allocation During Compilation: A Graph Coloring Approach

1.0 Introduction:

Graph coloring algorithms serve as the backbone of register allocation during compiler optimization of code. A register is a memory location in the computer CPUs with an access time much faster than that of memory. Typically, two registers can be read and one written within one memory cycle, while accessing memory directly can take multiple memory cycles [2]. Due to the relatively small capacity of registers, as well as their importance in achieving fast computation speeds, optimizing register usage is a great way to increase the speeds of compiled programs.

During compilation, values that are referenced often should be held in easy to access registers in order to maximize the efficiency of compilation. However, programs can be tremendously complex, and even the most up to date hardware has a limited number of registers. It is the job of register allocators to determine what data can be stored in registers, and what will need to be fetched directly from memory.

1.1 Overview of graph coloring for register allocation:

The process of register allocation takes as input intermediate code that references an unlimited number of registers. The intermediate code is assumed to be low level code, like assembly code. These pre-allocation registers are known as “virtual registers”. Registers are used to store *live ranges*, which are composed of multiple *values* connected by common uses [2]. *Values* are composed of a single definition, like an address in memory. When compiling, all values corresponding to the same *live range* are linked to the the same virtual register. It is now the register allocators job to transform the input code in such a way that the number of registers required to compile the code matches that of the hardware [3].

Graph coloring can be used to solve the problem of register allocation by modeling the live ranges and their interferences. A graph *G* can be constructed, with the nodes corresponding to live ranges, and the edges between nodes corresponding to their interference. If two nodes share an edge, that means they are both live at the same time. Since the values in the two seperate live ranges are not connected by common uses, they cannot be allocated to the same register.

The end goal is a k-coloring of this interference graph, such that k matches the number of registers available to the compiler. Since finding a k-coloring of a graph is NP-complete, a general heuristic approach must be taken to find an appropriate coloring. If no k-coloring can be

found using the method below, live ranges are *spilled*. This means that the live ranges are kept in slow-to-access memory, rather than registers. Outlined below is the first register allocator, implemented using a heuristic approach to register allocation.

2.0 Chaitin's Implementation:

Greg Chaitin and his colleagues at IBM developed the first register allocator based on graph coloring, known as the Yorktown Allocator [2]. This heuristic has been modified and adapted over the years, but it is a good introduction to the basics of register allocation via graph coloring.

The allocators heuristic approach to coloring can be broken into a series of steps, outlined below:

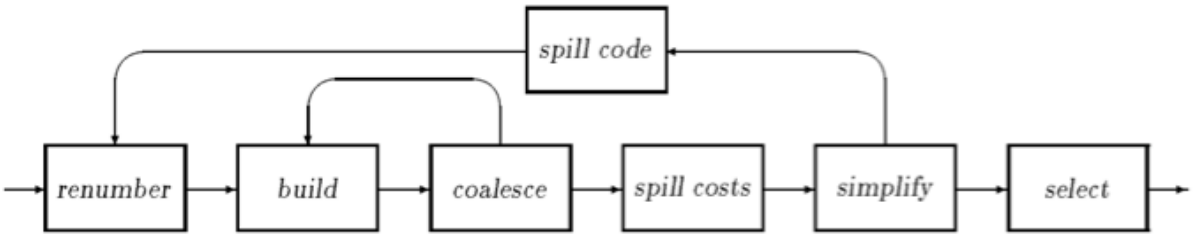

Figure 2.1: Chaitin's Graph Coloring Heuristic [2]

Renumber: All live ranges are identified and given unique numbers.

Build: The interference graph containing the live ranges is constructed.

Coalesce: In this step, unwanted copies are removed by combining multiple live ranges. A copy is eligible for removal if the source and the target of the copies do not interfere. An example of a copy instruction is something like x = y [3]. The entire code is passed through and coalesced before re-building the graph. Rebuilding the graph is necessary after coalescing live ranges since the interference graph will change [2].

An example of removing an unwanted copy:

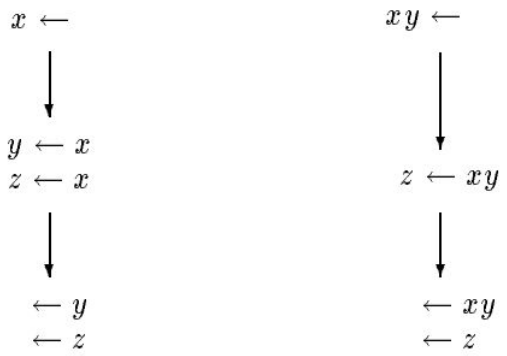

Figure 2.2: Coalescing Example [2]

In figure 2.2, we can see the effects of coalescing in removing an unwanted copy. On the left-hand side, *x* is defined, and then assigned to both *y* and *z*. Both *y* and *z* interfere, since *y* is live across the definition of *z*. However, *x* and *y* do not interfere, since they both have the same value. After one pass of coalescing, the code is condensed to the right side of the figure. The *x* and *y* are combined, removing the redundancy. The resulting interference, is that of *xy* and *z.* However, *xy* and *z* do not actually interfere, since they share the same value. Now that redundancies have been removed, the interference graph must be rebuilt to reflect the updated interference graph.

Spill Costs: In this step, every live range is given a estimate for the spill cost in terms of runtime. The spill cost is the time it would take to load and store the instructions required to *spill* the live range. The Yorktown Allocator quantifies spill cost with the formula 10^d for each instruction, where *d* is the nesting length of the instruction loop [2]. Nesting length is defined as the maximum number of nested statements, like: if, while, for, etc [4]. By quantifying the cost of spilling, we can choose instructions with the lowest spill cost to *spill*.

Simplify: Here, the interference graph is inspected and all nodes with a degree less than *k* (meaning they can be *k* colored) are removed from the interference graph and pushed onto a stack *s*. Their subsequent edges are also removed. This continuous removal of vertices reduces the degree of the remaining vertices. The remaining vertices are all of degree >= k. Vertices are now chosen for spilling. Vertices chosen to be spilled are simply removed from the interference graph.[2]

Here, the next step depends on whether or not there are nodes to be spilled. If nodes have been marked to be spilled, the process continues onto the spill code section. If no vertices are marked for spilling, the process continues to the select section.

Select: Nodes from *s* are popped off of the stack and inserted back into G. When inserted into G, they are given a color such that no two connected vertices are the same color. Because all of the nodes on *s* were chosen because their degree was < k, they can be uniquely colored when inserted into G.

Spill Code: Spill code is generated with the nodes marked in the simplify step. This spill code needs to directly load values into short, temporary variables within the live range. Since new live ranges are defined, a new interference graph needs to be created. The process is then sent back to the build phase.

2.1 The Interference Graph:

The interference graph is the core data structure used in the Yorktown Register Allocator. The interference graph is graph with the following desired possible operations [2]:

new(*n*): Creates and returns a graph with *n* nodes (with no edges).

add(*g*, *x*, *y*): Creates an edge between nodes *x* and *y* in the graph *g*. Returns *g.*

interfere(*g*, *x*, *y*): Returns true if nodes *x* and *y* in graph *g* interfere with each other, meaning they share an edge.

degree(*g*, *x*): Returns the degree of the node *x* in graph *G*.

neighbors(*g*, *x*, *f*): Apply the function *f* to all the neighbors of *x* in the graph *g*.

The graph *G* is stored as both an adjacency bit matrix and list. Although seemingly redundant, this is highly beneficial in regards to efficiency. The bit matrix supports constant time of the *add* and *interfere* methods, while the adjacency list supports easy implementation of the *neighbors* method [2].

Interfere and *add* both require quickly accessing two nodes. The adjacency matrix implementation of the interference graph allows for drastically decreased runtime for these methods, relative to an adjacency list implementation. *Neighbors* runtime is greatly decreased due to the adjacency list implementation, as the neighbors can be accessed in O(n) time, where n is the number of neighbors the node has.

The interference graph is constructed in two passes. In the first pass, the bit matrix is created, initialized to be empty, and the degree of each node is collected. In the second pass, adjacency lists are created, with known size, and the actual interferences are recorded in the bit matrix and adjacency size. The graph is re-constructed during each pass of the coalesce step, as mentioned above.

2.2 The Coloring Heuristic:

The key part of Chaitin's register allocator is the coloring heuristic. Chaitin's graph coloring heuristic runs in O(e + n), where n is the number of nodes, or live ranges, in the interference graph, and e is the number of edges in the interference graph. The main parts of the coloring algorithm are the *simplify* and *select* components. The following attempts to address why the coloring heuristic works.

Select and Simplify:
Simplify removes nodes whose degree is < k, and pushes them onto a stack *s*. Once all nodes that can be removed from the graph are pushed onto the stack, the *select* aspect of the algorithm adds them back onto the graph. When moving a node back onto the graph, we are

adding a node that had degree < k. Once the node is added back to the graph, its degree will be < k. Because there are k colors, there will be a color available for that vertex.

A node is only removed via *simplify* when it is trivially colorable, that is when its degree is < k. When a node is removed, all of its edges are removed as well. This reduces the degree of all nodes it shares edges with. These remaining nodes can potentially become eligible for removal. This removal heuristic ensures that nodes are removed and added to the stack from the graph in increasing difficulty, starting from the easiest (most trivial) to color.

When *select* moves nodes from the stack back onto the graph, it will have less than k neighbors, making it trivial to k-color. This is best visualized via a graphic example. See Figure 2.3 below:

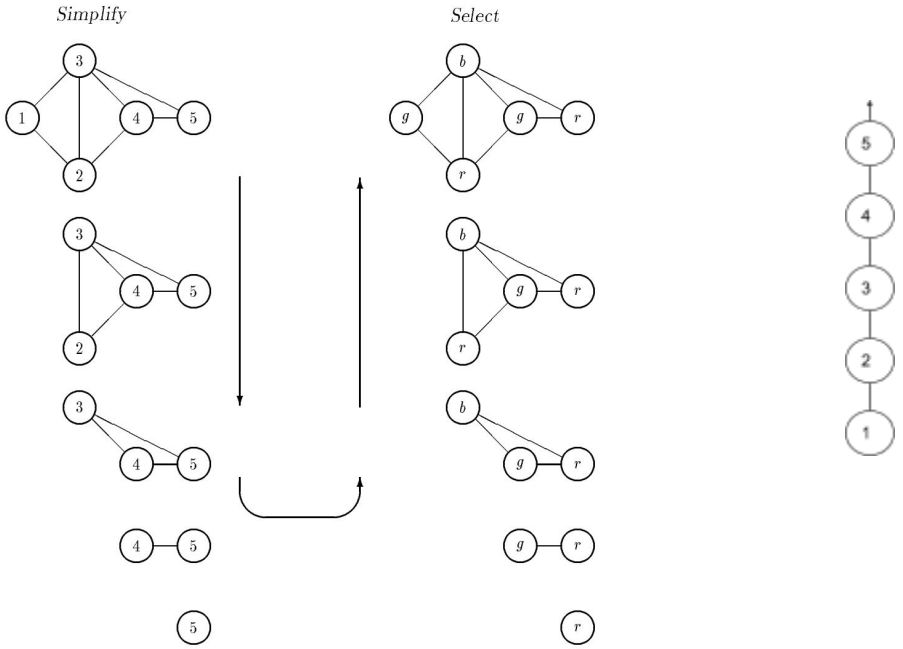

Figure 2.3: Coalescing Example [2]

Say we are trying to find a 3-coloring of the graph on the left hand side of Figure 2.3. In the *Simplify* section, node 1 removed because its degree of 2 is less than 3. Nodes 2 and 3 have had their degree reduce by 1. Next, node 2 is removed, since its degree is also less than 3. This process continues until all nodes that have degree less than 3 have been removed [2].

Now, in the middle of Figure 2.3, we have the *select* phase. The stack *s* created by simplify now looks like the stack on the right in Figure 2.3. Nodes are added to the now empty graph, starting with node 5. Since all these were moved when they were trivially colorable, they can be added back and colored trivially [2].

During the *simplify* phase, if the graph cannot be emptied due to nodes with a degree >= k, those nodes are chosen to be spilled. Spilled nodes are chosen by the criteria outlined below.

Spill Nodes:

The register allocator implementation by Chaitin uses the following as the criterion for which node to spill.

$$M_n = cost_n / degree_n$$

Further research by Bernstein *et al.* at Haifa improves upon Chaitin's initial metric, by proposing the following three [5]:

$$M_n = cost_n / degree^2_n$$
$$M_n = cost_n / degree^2_n area_n$$
$$Area_n = \Sigma\ 5^{depth,i}\ width_i$$

$Area_n$ is Bernstein's way of quantifying the live range n's impact on other live ranges throughout the current routine. $Depth_i$ corresponds to the number of loops containing the specific instruction *i* and $width_i$ is the number of live ranges that are live across the specific instruction [2].

The node with the smallest above ratio is chosen. For different instructions and routines, certain metrics dominate over other metrics. This is why Bernstein and his colleagues use a "best of 3" heuristic when choosing which node to spill, meaning *simplify* is run three times, each time with a different spill metric. The metric which produces the lowest spill cost is chosen. This method is implemented due to the fact that choosing the best spill node is NP-complete [2].

3.0 Concluding Thoughts:

Chaitin's graph coloring heuristic was the first approach to register allocation via graph coloring. Many improvements have been made since his implementation. Research has also shown that the coloring heuristic only has a minor impact on the performance of a register allocating system. Other factors, like architectures with register use constraints, have a large impact on register allocation.

Koes and Goldstein, from Carnegie Mellon, found that optimizing the coloring heuristic had a relatively small effect compared to heuristically improving spill decisions or preferentially allocating certain registers [1]. They did this by replacing the coloring heuristic with an optimal coloring algorithm, by changing the coloring problem into an Integer Linear Program and solving that with a commercial grade optimizer. This only lead to minor improvements in register allocation [1].

References:

[1] David Koes, Seth Copen Goldstein. An Analysis of Graph Coloring Register Allocation

[2] Preston Briggs. Register allocation via graph coloring. PhD thesis, Rice University, Houston, TX, USA, 1992.

[3] Zhong Shao. More on Machine-Code Generation. Yale University, New Haven, CT, USA. 1994-2017.

[4] Semmle 1.17 Documentation Team. Semmle, San Francisco, CA, USA. March 27, 2017.

[5] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill Code minimization techniques for optimizing compilers. *SIGPLAN Notices,* 24(7):258-263, July 1989.