Register Allocation by Graph Coloring

Less is more.

— Ludwig Mies van der Rohe

by Jason Robert Carey Patterson, Sep 2003 (orig Feb 2001)

Modern RISC architectures have quite large register sets, typically 32 general purpose integer registers and an equivalent number of floating-point registers, or sometimes more (IA64 has 128 of each type). Due to the long latencies of memory accesses in modern processors – even assuming primary data cache hits – the effective use of these large register sets for placement of variables and other values is a critical factor in achieving high performance. Register allocation is especially important for RISC architectures, because all operations occur between registers.

Register allocation is the process of determining which values should be placed into which registers and at what times during the execution of the program. Note that register allocation is not concerned specifically with *variables*, only *values* – distinct uses of the same variable can be assigned to different registers without affecting the logic of the program.

There have been a number of techniques developed to perform register allocation at a variety of different levels – *local* register allocation refers to allocation within a very small piece of code, typically a basic block; *global* register allocation assigns registers within an entire function; and *interprocedural* register allocation works across function calls and module boundaries. The first two techniques are commonplace, with global register allocation implemented in virtually every production compiler, while the latter – interprocedural register allocation – is rarely performed by today's mainstream compilers.

It has been shown experimentally that modern RISC architectures provide large enough register sets to accommodate all of the important values in most pieces of real-world code. Unfortunately, however, there will always be some pieces of code that require more registers than actually exist. In such cases, the register allocator must insert *spill code* to store some values back into memory for part of their lifetime.

Minimizing the runtime cost of spill code is a crucial consideration in register allocation. This issue becomes even more interesting (and challenging) if we consider the simple way in which many values are created, because many simple expressions are only worth keeping if registers are available in which to keep them – loading a common subexpression from memory may well be slower than recomputing it. This is called the *rematerialization* problem.

Interference Graphs

Register allocation shows close correspondence to the mathematical problem of *graph coloring*. The recognition of this correspondence is originally due to John Cocke, who first proposed this approach as far back as 1971. Almost a decade later, it was first implemented by G. J. Chaitin and his colleagues in the PL.8 compiler for IBM's 801 RISC prototype. The fundamental approach is described in their famous 1981 paper [ChaitinEtc1981]. Today, almost every production compiler uses a graph coloring global register allocator.

When formulated as a coloring problem, each node in the graph represents the *live range* of a particular value. A live range is defined as a write to a register followed by all the uses of that register until the next write. An edge between two nodes indicates that those two live ranges *interfere* with each other because their lifetimes overlap. In other words, they are both simultaneously active at some point in time, so they must be assigned to different registers. The resulting graph is thus called an *interference graph*.

Although theoretically an arbitrary graph can arise from a code sequence, in practice interference graphs are relatively sparse. Experiments have shown that the number of edges in a graph is normally about 20 times the number of nodes. As a result, a large graph of a few hundred nodes will only have a few thousand edges, rather than the many tens of thousands it theoretically could contain. In addition, these graphs are not structurally random but tend to contain "clumps" of connected areas.

Given an interference graph, the register allocation problem becomes equivalent to coloring the graph using as few colors (registers) as possible, and no more than the total number provided by the target architecture. When coloring a graph, two nodes connected by an edge must be colored differently, which corresponds to assigning a different register to each value...

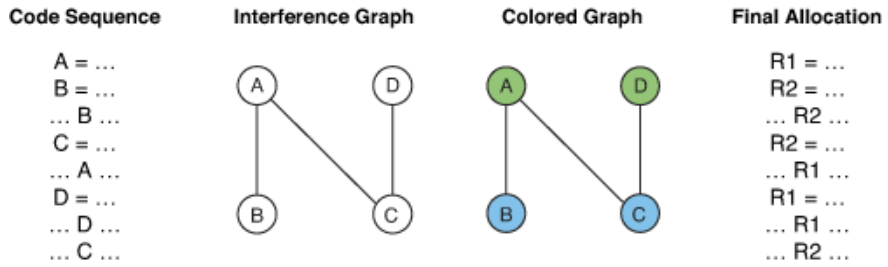

Figure 1 – Register allocation by graph coloring.

Determining the minimum number of colors required by a particular interference graph (the graph's *chromatic number*), or indeed determining if a graph can be colored using a given number of colors (a *k-coloring*), has been shown to be an intractable problem – only a strategy of brute force enumeration of every combination can guarantee an optimal result. Naturally, this approach is completely out of the question for large graphs, which are actually quite common due to optimizations such as function inlining.

Luckily, heuristics have been found that give good results in a time approximately proportional to the size of the graph, if coloring is actually possible [Chaitin1982]. Over the past two decades, these heuristics have been refined to the point where today's production compilers use quite sophisticated techniques which take into account issues like rematerialization [BriggsEtc1992] and code structure [CallahanKoblenz1991]. A very bold person might even say that global register allocation is now a *solved problem* :-)

Graph Coloring

This introductory paper describes the most widely used variant of these coloring heuristics – the *optimistic* coloring method first proposed by Preston Briggs and his associates at Rice University, which is based on the general structure of the original IBM "Yorktown" allocator described in Chaitin's original papers. The optimistic "Chaitin/Briggs" graph coloring algorithm is clearly the technique most widely used by production compilers, and arguably the most effective.

There are, of course, many other subtle variants of this technique, as well as a significantly different technique called *priority-based coloring* proposed by Fred Chow and John Hennessy, which uses a coarser and less accurate interference graph but can more easily handle live range splitting [ChowHennessy1984], [ChowHennessy1990].

The basic "Chaitin/Briggs" approach is as follows...

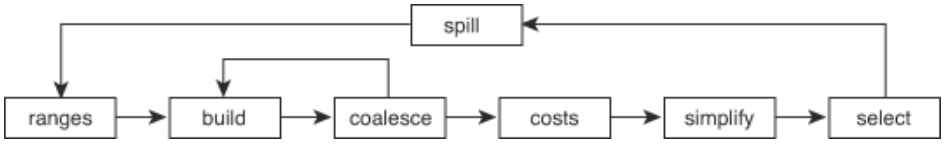

Figure 2 – The graph coloring process.

First, the live ranges must be determined. This is not as trivial as it seems – determining the *exact* live ranges of values requires tricky dataflow analysis to be carried out, carefully following the def-use chains, sometimes called *webs*, through the control flow graph. This is necessary because a value's live range is not necessarily a *contiguous* range of instructions. Even a simple split in the flow of control can cause a live range to become segmented...

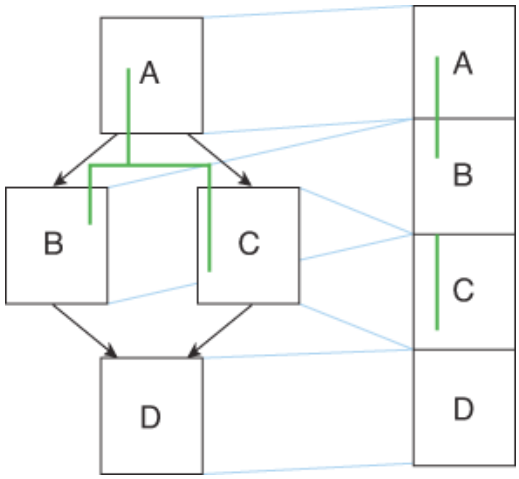

Figure 3 – A non-contiguous live range.

Constructing the interference graph is the next step. Usually, the graph is created in two forms simultaneously – a bit matrix and a set of adjacency lists. This makes working with the graph more efficient, allowing quick tests for the existence of an edge using the bit matrix and quick calculation of the set of neighbors which interfere with any particular node by examining the adjacency lists.

In addition to the nodes for live ranges, the graph also contains nodes for each machine register (normally). This allows the modeling of other types of interferences such as function calling conventions – values live across function calls can simply be made to interfere with all of the caller-saved machine registers, and live range splitting will take care of the rest (see the spilling section below).

After building the graph, unneeded copies (register-to-register move instructions) are eliminated in the *coalesce* phase. This might sound unnecessary, but it is actually an extremely important step, partially because copies are often introduced during optimization, but more importantly because coalescing allows the allocator to achieve *targeting* – where specific registers are used for specific purposes such as argument passing. With luck, those values can be computed directly into the appropriate registers in the final code. Coalescing also allows better handling of self- updating instructions, such as auto-increment addressing, as well as the two-address instructions commonly found in older CISC architectures.

After coalescing, the whole graph must then be rebuilt because eliminating the copies might have changed the interference relationships. A conservative approximation can be used incrementally, to allow multiple copies to be coalesced in a single pass, however a proper rebuild of the graph must be done afterwards for complete accuracy.

Once the graph is constructed, spill costs are calculated for each node (live range) using a variety of heuristics to estimate the runtime cost of storing each value and reloading it around its uses. Typical cost heuristics include loop nesting, reference counting and so on. This is where a degree of "cleverness" is required on behalf of the allocator.

The core of the coloring process itself starts with the *simplify* phase, sometimes called *pruning*. Here, the graph is repeatedly examined and nodes with fewer than *k* neighbors are removed (where *k* is the number of colors we have to offer). As each node is removed it is placed on a stack and its edges are removed from the graph, thereby decreasing the degree of interference of its neighbors. If a point is reached where every node has *k* neighbors (or more), a node is chosen as a possible spill candidate (just because a node has *k* neighbors doesn't necessarily mean it will spill – the neighbors may not all be different colors). Choosing *which* node to spill is, of course, the hard part – often a variant of the ratio of spill cost to degree of interference is used, or sometimes a combination of several different metrics [BernsteinEtc1989]. Once a spill candidate is chosen, the node is then removed from the graph and pushed onto the stack, just like the others, and the algorithm continues on to the next node.

Finally, the *select* phase actually selects a color (register) for each node. This is done by repeatedly popping a node off the stack, re-inserting it into the graph, and assigning it a color different from all of its neighbors. In effect, *simplify* chooses the order of assignment and *select* makes the actual assignments themselves. During the select phase, the allocator might also make assignment decisions to accommodate hardware restrictions such as register pairing (the handling of register pairs can also be done in other ways).

At any time during the select phase, if a situation occurs where no color is available (ie: all *k* neighbors have different colors), the node is marked for spilling and remains uncolored while the algorithm continues on to the next node. If nodes have been left uncolored after this process, the allocator must then generate the necessary spill code (or rematerialization code) and start the whole register allocation process all over again.

An example of the core simplify/select process is shown below...

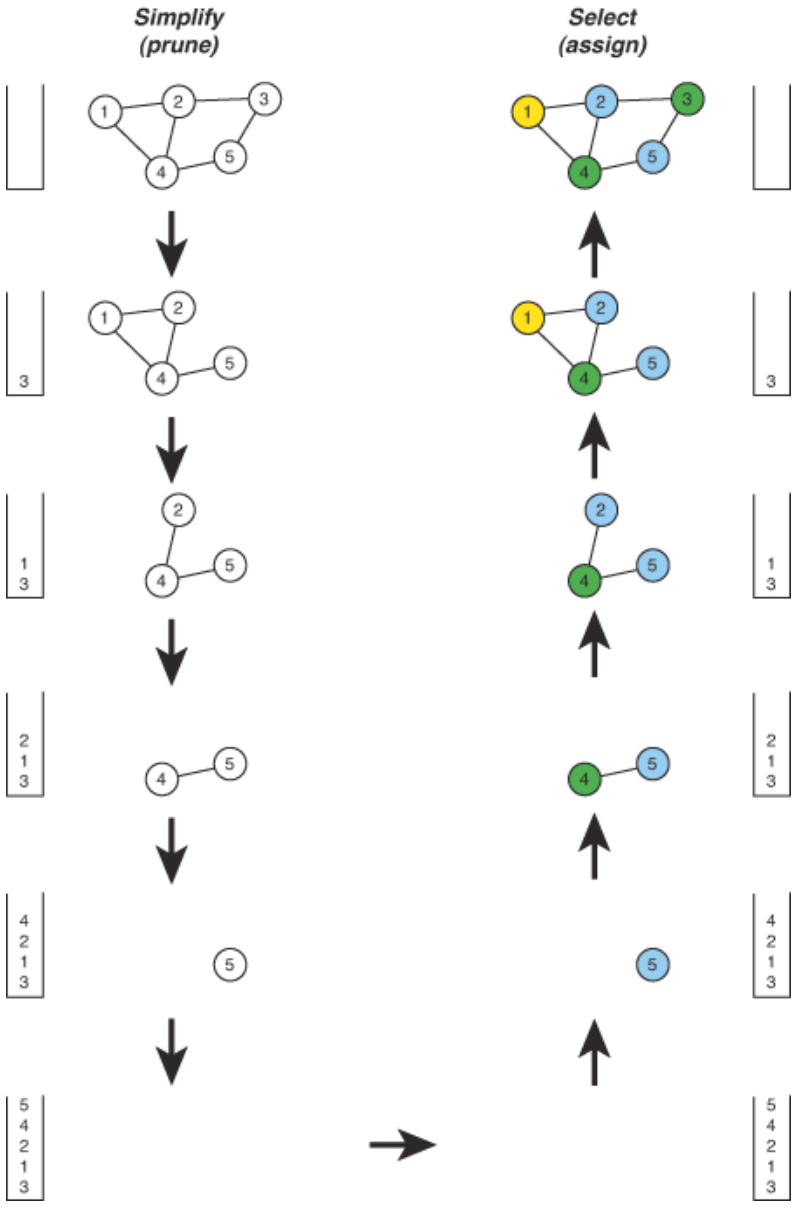

Figure 4 – An example of the graph coloring process.

Spilling

If spill code is necessary, the spilling can be done in many different ways. One approach is to simply spill the value everywhere and insert loads and stores around every use. There are some advantages to spilling a value for its entire lifetime – it is straightforward to implement and tends to reduce the number of coloring iterations required before a solution is found. Unfortunately, there are also major disadvantages to this simple approach. Spilling a node everywhere does not quite correspond to completely removing it from the graph, but rather to splitting it into several small live ranges around its uses. Not all of these small live ranges may be causing the problem – it might only have been necessary to spill the value for part of its lifetime.

To account for this, a more sophisticated approach is to split the problematic live range into several subranges, only some of which might need to actually be spilled (think about a value entering an *if* statement where one side has high register pressure but the other only has low register pressure). The coloring process will probably require more iterations to find a solution, but the result should be a better final allocation. Some area-based heuristics have been suggested to help the allocator decide where to split live ranges, based on the well-known relationship of *dominance* represented by the dominator tree and control dependence graph [NorrisPollock1994].

Hierarchical coloring is essentially a more structured approach to this type of live range splitting [CallahanKoblenz1991]. With this scheme, the code is first divided into *tiles* which are grouped into a tree representing the hierarchical structure of the code...

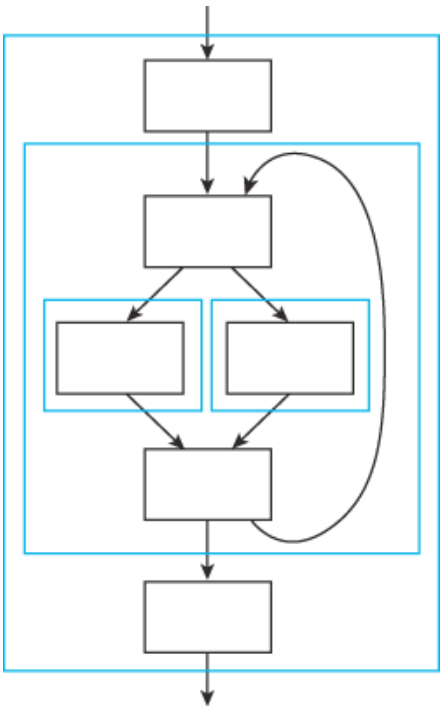

Figure 5 – Tiles used in hierarchical graph coloring.

Registers are allocated for each tile using standard graph coloring, and the conflicts left after each local allocation are handled by the next higher level of tiling. The allocator controls the spilling process by determining which tiles will have spill code added around them. Overall, this approach results in colorings which are more sensitive to the local register requirements within each tile. It can also take advantage of execution profiling information, allowing the allocator to give priority to the most important pieces of code.

Clique Separators

Since graph coloring is a relatively slow optimization, anything that can be done to make it faster is worth investigation. The chief determining factor is, of course, the size of the interference graph. The asymptotic efficiency of graph coloring is somewhat worse than linear in the size of the interference graph – in practice, graph coloring register allocation is something like *O(n log n)* – so coloring two smaller graphs is faster than coloring a single large graph. The bulk of the running time is actually the build-coalesce loop, not the coloring phases, however this too is dependent on the size of the graph (and non-linear).

An allocator can take advantage of this fact by breaking large graphs into parts, as long as this is done carefully. Specifically, good places must be chosen to split the graph so that the resulting two parts are smaller, yet recombining them still gives valid results. Splitting at *clique separators* achieves this goal. A clique separator is a strongly connected subgraph, which when removed from the original graph, completely disconnects it (dividing it into disjoint subgraphs)...

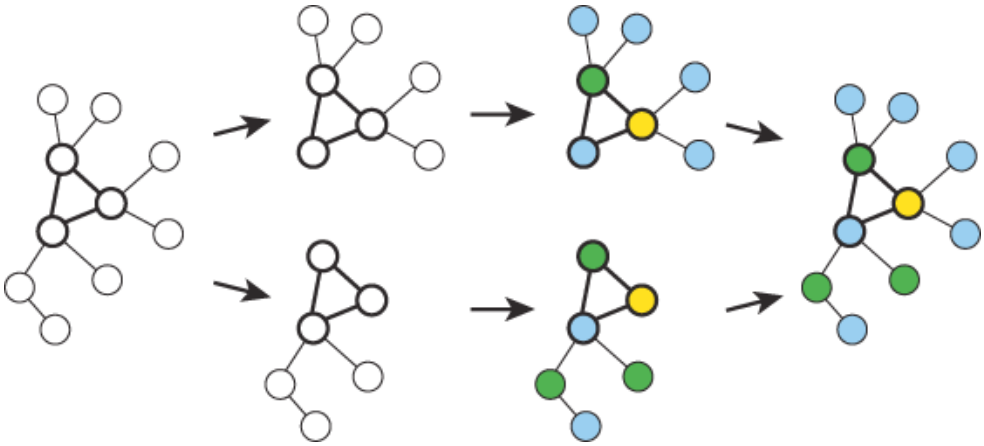

Figure 6 – Faster register allocation using clique separators.

Since the disjoint subgraphs can use the same set of colors (registers), these subgraphs can be colored separately. The original graph can then be reassembled to form the final allocation. The size of each individual coloring problem is now smaller than the original graph, so this approach significantly reduces both the time and space requirements of the graph coloring process [GuptaSoffaSteele1989], [GuptaSoffaOmbres1994].

Fortunately, the basic approach of clique separators can be used almost implicitly, by letting the control flow graph itself guide the subdivision of the "large" graph for each function. In effect, this is a natural consequence of the tiling allocators mentioned above. In a sense, every compiler also uses clique separators at function call boundaries – each function is colored independently and they are all knitted back together by the assignments required by the argument passing conventions.

Linear-Scan Allocation

As a special case, local register allocation within basic blocks can be considerably accelerated by taking advantage of the structure of the particular interference graphs involved. For straight-line code sequences, such as basic blocks or software-pipelined loops, the interference graphs are always *interval graphs* – the types of graphs formed by the interference of segments along a line. It is known that such graphs can be easily colored, optimally, in linear time. In practice, an interference graph need not even be constructed – all that is required is a simple scan through the code. This approach can even be extended to the global case, by using *approximate* live ranges which are the easy-to-identify linear supersets of the actual segmented live ranges.

The linear-scan approach to register allocation is common in two real-world situations. First, this approach is useful in dynamic code generation situations, such as Java virtual machines, where basic blocks are often the unit of translation – and even if they are not, a more complex and time- consuming algorithm such as graph coloring would be too costly. Second, linear-scan allocation can be applied as a fast alternative to graph coloring when compilation speed is more important than execution speed, such as during development situations. Rather than turning off register allocation altogether, linear-scan register allocation is fast enough that it goes virtually unnoticed during compilation, yet produces results that are often not dramatically worse than true graph coloring [TraubEtc1998]. Many compilers for RISC architectures default to having local register allocation always turned on, even during debugging, implemented at the basic-block level using simple linear-scan allocation.

Linear-scan register allocation can also be extended in an attempt to take into account the "holes" in the not-really-linear live ranges. This approach corresponds to the mathematical problem of *bin packing*. Perhaps surprisingly, one production compiler actually used this approach – DEC's GEM compilers for the Alpha architecture (the 'GEM' name used for the Alpha compilers actually relates to the RISC architecture which preceded Alpha at DEC, the PRISM architecture (PRISM and GEM, get it?) – it is often said that the AXP from the name Alpha AXP unofficially stands for Almost eXactly Prism).

More Information?

To learn more about graph coloring register allocation, Preston Briggs' PhD thesis is definitely the best place to start [Briggs1992]. Briggs' thesis gives detailed coverage of the entire graph coloring process, including many important real-world issues such as dealing with register pairs, the rematerialization problem, representations for the required data structures and so on (in his own words, it "basically beat the subject to death"). For anyone actually implementing a graph coloring register allocator, Briggs' thesis is an absolute must-read.

Register allocation is also covered in some detail in the book *"Advanced Compiler Design & Implementation"* by Steven Muchnick [Muchnick1997].