# Algorithms for Register Allocation

## Andrew Pardoe

# The need for register allocation

Older computers were built on complex instruction sets which had instructions that would, for example, pull two operands out of core memory, add them, and write the result back into memory. Registers were few—silicon was used instead to hold instructions—and memory was small enough to address quickly. The instruction set was *orthogonal,* meaning that every instruction provided every possible memory addressing mode. Individual instructions could be tuned so that using a complex instruction was more efficient for a programmer than using a sequence of simpler instructions.

Modern CPUs focus on doing all work in registers. The problem is that while a program has an unbounded number of variables a CPU has a limited number of registers. Anything that doesn’t fit into a register can be *spilled*, that is, copied to memory when not being used and copied back when it is needed, but spilling is slow because memory access is orders of magnitude slower than register access.

CPUs like Berkeley’s RISC project used a combination of hardware and software to solve this problem—it had 128 registers but a program had to limit itself to using only 8 at a time. The CPU would slide a virtual window across the registers to allow different procedures to see their own variables (sharing windows allowed for fast procedure calls.) This technique is extended by Intel’s IA64 to offer an unlimited number of registers available in any procedure’s window. This magic was made possible through a hardware-based Register Stack Engine which allocated physical registers and created virtual registers on demand. Most computer architectures, however, expose a fixed number of physical registers and rely on compilers to allocate them efficiently. *Global register allocation* is the process by which a compiler allocates a fixed set of physical registers to an unbounded number of data across a program’s functions. Newer algorithms, such as the Optimal Register Allocation algorithm, can work with a set of registers that are not regular or fixed in number.

# Register allocation as graph coloring

IBM fellow John Cocke, known as “father of the RISC architecture”, stated in 1971 that global register allocation could be represented as a graph-coloring problem. The statement of the problem of global register allocation as a graph-coloring problem is as follows:

1. During code generation, allocate objects that can be enregistered to distinct symbolic registers.
2. Compute *liveness* for these objects from the dataflow graph. That is, determine the definition point at last reference point of each object through standard dataflow analysis. For any numbering of the program’s code a variable is said to be live in interval *[i, j]* if it is first defined at instruction *i* and there is no reference to the variable after instruction *j*. The trivial liveness for any variable is *[1, N]* for a program with *N* instructions.
3. Construct an undirected *interference graph* whose nodes represent the objects allocated to symbolic registers and whose edges represent objects which are alive at the same time. More precisely, add an edge between two nodes if one of them is alive at the definition point of the other.
4. Color the graph with the *R* colors where *R* is number of physical registers available.
5. Allocate all nodes of the same color to a single register.

There are only two problems with this idea. First, the graph may not be able to be colored with *R* colors. Second, graph coloring is an NP-complete problem for graphs of greater than 3 nodes.

# Chaitin-Briggs algorithm for register allocation

In papers from 1981 and 1982 G. J. Chaitin of IBM Research proposed a method to do register allocation and spilling via graph coloring. In 1989 Preston Briggs proposed improvements to Chaitin’s original algorithm in his PhD. thesis at Rice University. The Chaitin-Briggs algorithm is generally accepted as the best algorithm for register allocation that can be used in a production compiler.

The key insight to Chaitin’s algorithm is called the *degree < R rule* which is as follows. Given a graph *G* which contains a node *N* with degree less than *R*, *G* is *R*-colorable iff the graph *G’,* where *G’* is *G* with node *N* removed, is *R*-colorable. The proof is obvious in one direction: if a graph *G* can be colored with *R* colors then the graph *G’* can be created without changing the coloring. In the other direction supposed we have an *R*-coloring of *G’*. Since *N* has a degree of less than *R* there must be at least one color that is not in use for a node adjacent to *N*. We can color *N* with this color.

The algorithm is as follows:

While *G* cannot be *R*-colored
 While graph *G* has a node *N* with degree less than *R*
 Remove *N* and its associated edges from *G* and push *N* on a stack *S*
 End While
 If the entire graph has been removed then the graph is *R*-colorable
 While stack *S* contains a node *N*
 Add *N* to graph *G* and assign it a color from the *R* colors
 End While
 Else graph *G* cannot be colored with *R* colors
 Simplify the graph *G* by choosing an object to *spill* and remove its node *N* from *G*
 (spill nodes are chosen based on object’s number of definitions and references)End While

The complexity of the Chaitin-Briggs algorithm is *O(n2)* because of the problem of spilling. A graph *G* will only fail to be *R*-colorable if at some point the reduced graph *G’* only has nodes of degree *R* or greater. When a graph is easily *R*-colorable the cost of a single iteration is *O(n)* because we make two trips through the graph and either remove or add one node each time. But spilling brings in additional complexity because we may need to spill an arbitrary number of nodes before *G* becomes *R*-colorable. For every node we spill we make another trip through the linear algorithm.

This algorithm can be improved through *subsumption* which is the act of coalescing nodes which are the source and target of copy operations into a single node before running the algorithm. This reduces the number of nodes to color but can increase the degree of any coalesced node. This can only be done when the nodes do not interfere with each other, however, and aggressive coalescing can lead to uncolorable graphs. (Briggs’ work introduces safer methods to determine which nodes to coalesce and spill.) The subsumption step is slow and is not done in fast register allocators.

# Greedy algorithms and linear Scan register allocation

Even when the expensive subsumption step is not performed the Chatin-Briggs algorithm is too slow for Just-In-Time (JIT) compilers like those used by Java and .NET or for interpreters as used by programming languages like Perl and Ruby. In situations such as these the time spent compiling code can dominate the time spent executing the compiled code so minimizing throughput can be as important as the quality of code produced. Linear scan, proposed by Poletto and Sarkar of IBM Research in 1999, is the dominant algorithm for register allocation when minimizing throughput is the goal.

Other fast register allocation schemes were in use when linear scan was proposed. One popular greedy algorithm, used in the lcc compiler described in Hanson and Fraser’s book, is to allocate registers to variables which are used most frequently and placing others on the stack. Another, called binpacking, was used in the DEC GEM compiler. It allowed a variable’s lifetime to be split into separate regions so that a single variable may live in a register for part of the code and in memory in other parts. It improves on linear scan by keeping track of lifetime holes—periods in a variable’s lifetime when it is not used—and reusing the register in these places. However, binpacking is slower than linear scan register allocation.

The algorithm for linear scan for *R* registers is as follows:

1. Perform dataflow analysis to gather liveness information (this is the same as step 2 of graph coloring algorithms.) Keep track of all variables’ *live intervals,* the interval when a variable is live, in a list sorted in order of increasing start point (note that this ordering is free if the list is built when computing liveness.) We consider variables and their intervals to be interchangeable in this algorithm.
2. Iterate through liveness start points and allocate a register from the available register pool to each live variable.
3. At each step maintain a list of *active intervals* sorted by the end point of the live intervals. (Note that insertion sort into a balanced binary tree can be used to maintain this list at linear cost.) Remove any *expired* intervals from the active list and free the expired interval’s register to the available register pool.
4. In the case where the active list is size *R* we cannot allocate a register. In this case add the current interval to the active pool without allocating a register. Spill the interval from the active list with the furthest end point. Assign the register from the spilled interval to the current interval or, if the current interval is the one spilled, do not change register assignments.

The complexity of the linear scan algorithm is linear according to the number of variables, *V*. However, the cost of maintaining the list of active intervals can be linear according to the number of registers, *R*, or *O(*log *R)* if a binary tree is used to maintain the list. Thus the complexity is actually *O(V*x*R)*. Given that the number of registers is usually fixed and small (between 8 and 128), however, it is fair to call this algorithm linear.

# Optimal Register Allocation

While it is not precisely an algorithm itself because it relies on a separate integer program solver to do the algorithmic work, Goodwin and Wilken used 0-1 integer programming to create a novel solution to the problem of register allocation. 0-1 integer programming is a special case of integer linear programming where the variables are required to be assigned a solution value of 0 or 1. While 0-1 integer programming is NP-hard there are commercial solvers which work in *O(n3)* time for limited integer programming problems.

The Optimal Register Allocation algorithm is as follows:

1. Determine points in a function where a decision must be made about register allocation. At any point in the program either a register allocation is made (1) or not made (0).
2. Use information about decision variables, register allocation overheads and program conditions to construct a 0-1 integer program.
3. Use an integer programming solver to solve this 0-1 integer program.
4. Rewrite the program’s intermediate representation based on the values decided by the solver. When the solver assigned a 1 to a variable rewrite the program instruction to use a register. Otherwise, for variables assigned 0, insert spill code to write the variable to memory.

This algorithm works well for architectures which have a uniform set of registers which can be accessed at a constant cost. It has been extended to handle register allocation for irregular architectures which may have instructions with combined source and destination registers or memory operands or may have non-uniform cost for accessing registers. Still, the difficulty of integrating a commercial integer programming solver makes this algorithm more of academic interest.

# Application of register allocation algorithms

As discussed above the real difference between the Chaitin-Briggs algorithm and the Linear Scan algorithm is the complexity of executing the algorithm. There are two key goals in optimization of a compiled or interpreted program: quality of the generated code and throughput of the compiler.

Greedy algorithms, such as linear scan, are ideal for situations where throughput is a key consideration. When code is compiled and optimized at run time, such as for JITted code or interpreted code, greedy algorithms do a “good-enough” job. Graph coloring algorithms, such as Chaitin-Briggs, are better for when the code will be compiled and optimized once and executed multiple times thereafter.

While integer programming can claim to create an optimal allocation of registers it is of little use for real-world scenarios. Neither mass-production code nor standard JIT compilers and interpreted languages can easily make use of a commercial linear program solver and the runtime of the algorithm is worse than Chaitin’s fast graph coloring algorithm.

As register allocation is one of the key optimizations in code generation a fast allocator which led to an efficient allocation would be a great discovery. However, as the problems of static code generation are considered to be mostly solved it’s unlikely that much work will be done in this area. Estimates of linear scan register allocation show it to be within 12% of the ideal allocation and linear performance is ideal for runtime-compiled scenarios.

It is more likely that a non-algorithmic breakthrough will change this field. Registers exist because large memory banks are costly to index and access. Some method of accessing main memory cheaply would obviate this problem. Caches “split the difference” between main memory and registers by placing a small set of quickly-accessible memory between registers and main memory. Possibly a solution similar to the Register Stack Engine in the Intel IA-64 architecture which exposed an unbounded number of registers to program code will be the next big breakthrough in this area.

Regardless of what developments come in this space the problem of global register allocation is one that affects every program that executes on a modern architecture. There have only been three significant algorithmic developments in this space in the last quarter-century. It’s unlikely that research will provide a new solution to this problem but an ideal solution which worked in real-world situations would be a significant breakthrough in computer science.

# References

Chaitin, G. J. Register Allocation and Spilling via Graph Coloring. *Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction,* Boston, MA, published as *SIGPLAN Notices,* Volume 17, Number 6, June 1982.

Hansen, David and Christopher Fraser. *A Retargetable C Compiler.* Addison-Wesley Professional, 1995.

Kong, Timothy and Kent Wilken. Precise Register Allocation for Irregular Architectures. *Proceedings of the 31st Microarchitecture Conference, December 1998.*

Muchnick, Steven S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

Poletto, Massimiliano and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems,* Volume 21, Number 5, September 1999.

Wikipedia article on Register Allocation. <http://en.wikipedia.org/wiki/Register_Allocation>. Article updated for correctness and completeness based on my other research.