An Overview of More Efficient Register Allocation by Graph Coloring Applications

ALLISON TURNER*, Wellesley College

A review of the history of and improvements to register allocation algorithms based on graph coloring applications. Consideration is given to target computer architecture, program structure, particular data types, and empirically discovered features.

Additional Key Words and Phrases: graph coloring, register allocation

1 THE PROBLEM

Near the end of the back end phase of compiling, around the code generation and optimization phases, the compiler writer must decide how to allocate registers to the various instructions in the compiling program. There are a very limited number of registers relative to the memory capacity of the rest of the computer, but these registers are in fact the fastest layer of the memory hierarchy in the vast majority of computer architectures. Any unnecessary load or store instruction is more time wasted on secondary memory. Therefore, it is in the programmer's best interest to find a way to execute programs that uses registers as often as possible and secondary memory as little as possible. Unfortunately, finding the algorithm for optimal register allocation is an NP-complete problem. So, compilers use various heuristic methods to optimize register allocation as well as possible. [1]

2 CHAITIN'S ALGORITHM

In 1982, Gregory Chaitin, a researcher at the IBM T. J. Watson Research Center, published an algorithm for register allocation that significantly optimized the allocation process. His approach unified the register allocation problem with the node coloring problem in graph theory. In graph theory, the chromatic number of a graph is the number of colors the graph needs if each node is a different color than its neighbors, every node is colored, and the minimal number of colors are used. Chaitin's algorithm associates each color with a different register. So, for a system with, for example, 8 registers, the algorithm must find an 8-coloring of the graph in question.

The graph in question is called an interference graph. Construction of this graph is the first step of Chaitin's algorithm. First, the algorithm determines the nodes of the graph. This is done by using "...well-known optimizing compiler techniques to do a global data-flow analysis..." and then "indicating at the beginning of each basic block which computations are live going into it, and by marking each instruction...to indicate if it goes dead."[2] Each variable in all of the sets of live variables becomes a node. These all represent the values living in an unlimited number of imaginary registers - what Chaitin calls virtual registers. [1] Next, all variables that are members of the same set of live variables, or that are "...live at a definition point of the other"[2], are connected by an edge. Members of the same set are said to "interfere", because their live ranges overlap. They cannot occupy the same register in the allocation, because they would overwrite each other. Chaitin recommends representing the interference graph with both a bit matrix and adjacency vectors, since a bit matrix is an efficient choice for random node accesses but an adjacency vector is better for sequential access to all neighbors.[2]

Next, extra copy instructions are eliminated, by coalescing the nodes that represent the source and target of copy instructions. This can only be done if the source's and target's usages do not

*Undergraduate

interfere with one another's definitions.[2] Chaitin also refers to the coalescing step as subsumption of virtual registers. After this coalescing, the algorithm calculates the cost of spilling each node. This calculated cost has to do with the number of usages of that variable, and so the required total of loads and stores from secondary memory should that node be spilled. Costs are weighted higher if the variable has usages within a loop, and even higher if that loop is nested within others.[1]

Then, a color ordering is determined. This is done by removing nodes, highest degree first, and placing them on a stack. Nodes of a degree higher than the number of registers are marked for spill one at a time, and the interference graph is re-evaluated in hopes that the spill made the remaining graph colorable. The graph is re-evaluated by inserting the spill code and rebuilding the graph from the new set of basic blocks. Once the graph is colorable with a chromatic number less than or equal to the number of available registers, a coloring is finalized, and each distinct color is associated with a distinct register, and the code generation proceeds with replacing references to each variable with the register associated with its node's color.[1]

3 IMPROVEMENTS TO CHAITIN'S ALGORITHM

Chaitin's algorithm was a revolutionary approach to the register allocation problem. However revolutionary it was, the algorithm as a heuristic solution, could be improved upon in areas where it failed to consider some important details of the problem. These details include target computer architecture, program structure, particular data type implementations, and empirically discovered features of a graph based approach. The following sections detail these considerations and examine how various computer scientists since Chaitin have improved graph coloring-based register allocators in light of these minutiae.

3.1 Making Spill and Coloring Decisions Based On Program Structure

3.1.1 Chow, Hennessy. Frederick Chow and John Hennessy saw three key issues with Chaitin's approach in the late 1980s. They did not think that Chaitin's cost and benefit analysis for spilling was accurate or detailed enough, and especially did not take into account particular architectures, caller/callee save protocols, variable clustering, or execution frequencies[3]. They thought that spilling an entire live range was wasteful and inefficient compared to spilling a subsection of a live range, and that coloring and spilling decisions should be made simultaneously[3]. They also thought that the termination and inefficiency of Chaitin's algorithm wasn't well established, given his contingency for backtracking on a spill and recoloring[3]. In short, in Chow and Hennessy's view, a heuristic method that did not account for particular program structure or computer architecture was a poor heuristic method overall.

To account for machine architecture, Chow and Hennessy propose a few different cost/benefit heuristics. Most importantly, they base their coloring algorithm on basic blocks, not on individual, full live ranges. They also prioritize basic blocks for coloring based on that block's variables' frequencies of use and load/store costs. They propose a formula to calculate the execution time saved by assigning a given variable to a register, si = LODSAVE x u + STRSAVE x d - MOVCOST x n, where si is a given variable, or "live unit", u is the number of uses, d is the number of definitions, LODSAVE is "[t]he amount of execution time saved for each reference of a variable residing in a register rather than in memory," STRSAVE is "[t]he amount of execution time saved for each definition of a variable residing in register compared with the corresponding store to memory being replaced," and MOVCOST is "[t]he cost of a memory-to-register or register-to-memory move...If the execution times for loads and stores are different in the target machine, they have to be distinguished further. "[3] This benefit is calculated for each live unit in each basic block, and the basic blocks are prioritized for coloring by ordering the blocks by the value of the sums of all live units' benefit divided by the number of live units in the basic block. Chow and Hennessy

also recognize the different memory costs of using caller saved versus callee saved registers, and so propose modifications to the priority function described above to account for such costs, such as adding the time of an extra store instruction to the use of a callee saved register.[3] They also recommend separate interference graphs for registers with different express purposes, such as registers designated for floating point versus integer values.[3] Overall, Chow and Hennessy's register allocation is based on a prioritization of live units that would benefit execution speed the most from residing in registers, as opposed to Chaitin's spill decisions based on lowest spill cost.

Chow and Hennessy base many of their modifications on program structure. They perform colorings on live ranges at the basic block level, not on individual live ranges in the program as a whole. This also makes it easier for their algorithm to split live ranges by their described range splitting algorithm when such splitting is beneficial to execution speed, whereas Chaitin does not consider live range splitting. They base their frequency of use metric on the loop depth of a variable's placement. In their discussion of caller-save and callee-save memory cost differences, they point out that caller-save memory costs are not a concern in a procedure that makes no external calls, or a "leaf" procedure. They also point out that parameters to function calls will be pre-colored based on argument register saving protocols. Chow and Hennessy also mention that their allocator could base priority partially off of measurements of execution frequency from a profile of previous runs of a given program, but they mainly focus on static analysis of variables.

Based on all of these additional considerations, Chow and Hennessy construct a coloring allocator that executes in a single forward pass, as opposed to Chaitin's backward passes of indeterminate repetition. First, they separate live units with a number of neighbors higher than the number of registers from the others. These are known as the constrained live units. Then, for each constrained live unit, until the graph is not further colorable with a chromatic number less than or equal to the number of registers or all live units are assigned, compute the priority value of that unit. When the graph is no longer colorable or the priority value is negative, delete this live unit from the interference graph. Then color each live unit in order of priority value, splitting live ranges as necessary.[3]

Chow and Hennessy's work was another major stepping stone in register allocation by graph coloring. Even in the small sampling of papers in this survey, Chow and Hennessy's work was cited as foundational many times.

3.1.2 Callahan, Koblenz. In 1991, David Callahan and Brian Koblenz similarly found Chaitin's cost/benefit analysis lacking. They point out that Chaitin's analysis does not account for program flow beyond loop depth, and that spilling occurs for an entire live range in the case of spilling for a single variable. They propose a hierarchical tiling system to prioritize certain variables for register allocation, decisions which will be made by each tile based on local usage, after which its individual interference graph is passed to its parent. Callahan and Koblenz note that their algorithm allows for noticeably more compact interference graphs. Their algorithm first traverses the tiles from root to leaves giving an initial tentative register assignment, then back from leaves to root, binding these tentative assignments to physical registers.[4]

Callahan and Koblenz describe their tile tree in terms of the control flow graph G of basic blocks B with flow edges between each block E. G is said to have its starting point at the block in B with no predecessors and its stopping point at the block in B with no successors.[4] They constrain a "tile tree" to be a collection of sets of basic blocks, such that each set is either disjoint or has a parent-child relationship, and that each tile has an entry and an exit edge. The tile is considered the root tile when the starting and ending points of the control flow graph are the same as the entry and exit edges of that tile.[4]

3.2 Minimizing Spill in Multi-Register Instructions

3.2.1 Briggs, Cooper, Torczon. Preston Briggs, Keith D. Cooper, and Linda Torczon were one group of researchers who raised another concern about Chaitin's algorithm in the early 1990s. They pointed out that Chaitin-style allocators typically over-spilled in situations involving multi-register instructions, and did not have a way to handle values that required more than one register, such as floating point values, without even more gross overspilling, or, even worse, incorrect allocation of these values. Incorrect allocation would entail a multi-register value that must be aligned in adjacent registers being allocated to non-adjacent registers, for example. Briggs, Cooper, and Torczon iterate through several ways to represent multi-register values, highlighting the conflicts between each representation and Chaitin's algorithm. Separating a multi-register value into different nodes by word size over-constrains a complete coloring because of the necessary precoloring of the multi-register value. Transforming the interference graph into a weighted graph is not an option because coloring such a graph is an NP-complete problem. A multigraph, with extra edges representing the extra constraints placed on a coloring by the multi-register values, results in spilling where none is necessary by an unmodified Chaitin's algorithm.[1]

Briggs, Cooper, and Torczon's solution to the overspilling from multi-register values and multi-register instructions involves two modifications to Chaitin's algorithm and a multigraph representation of the interference graph.

For a value that requires n adjacent registers, all interfering live ranges have n edges to that value. If the architecture does not require alignment, an extra edge between any such multi-register values serves to mark one or more of such values for possible spilling. One can already see here the modification made to Briggs, Cooper, and Torczon's allocator, named an optimistic allocator, as opposed to a pessimistic allocator. Instead of immediately spilling a node with a degree higher than the number of registers available, the optimistic allocator simply marks that node as a spill candidate and pushes it onto the color ordering stack as usual. Then, in the coloring selection step, if the algorithm encounters an uncolorable stack, it finalizes spilling and returns to the construction of the interference graph.[1] The implication of these modifications is that Chaitin's allocator was too quick to spill variables, and that his algorithm's interference graph represented program structure too pessimistically in terms of memory usage. Briggs, Cooper, and Torczon's considerations for multi-register values and instructions tune Chaitin's algorithm for more realistic programs and data. Briggs, Cooper, and Torczon's optimistic algorithm is another foundational work of this genre of register allocation.

3.2.2 Nickerson. Brian Nickerson and Briggs, Cooper, and Torczon's work compliments each other well. In 1990, Nickerson was also concerned with Chaitin-style register allocators' performance on multi-register operands, but Nickerson came to an interestingly juxtaposed conclusion about the necessary modifications to the interference graph.

Nickerson considers whether this should be a hardware design issue instead of a register allocation algorithm issue, by either widening instruction sets to reference all of the registers that are arguments to a certain operation, or by eliminating multi-register instructions altogether. He concludes that the former approach is undesirable because of the added instruction width and "the greater complexity of dereferencing the extra register addresses..."[5], and that the latter approach will increase execution time of any program, exactly the opposite of the desired result. He also dismisses pre- and post-coloring of multi-register values as an algorithmic-level solution, explaining that they over-emphasize the importance of multi-register operands and induce unnecessary spill costs.[5]

Nickerson is primarily concerned with data that can be represented as "clusters" of live ranges, not only applying to double precision floating point values but the groups of fields of a struct.

Nickerson posits that "[w]hile one would be tempted to conclude that coloring register clusters requires more detailed information about the nature of interferences, the solution algorithm actually benefits from having less information."[5] Nickerson's algorithm does not look at individual node degree in comparison with the total number of registers, but rather the number of neighbor clusters in comparison with the number of places that cluster could fit, i.e. an aligned cluster requiring three registers being allocated to a set of six registers has two places it could go. Constraints on members of the same cluster are implied in Nickerson's interference graph, such that members of the same cluster cannot occupy the same register. He also recommends only partial use of Chaitin's dual interference graph representation, beginning the algorithm by using the bit matrix, then switching to the adjacency vector representation entirely after register subsumption.

3.3 Accounting for Non-Independence and Non-Interchangeability of Registers

3.3.1 Smith, Ramsey, Holloway. In 2004, Michael Smith, Norman Ramsey, and Glenn Holloway of Harvard University pointed out some characteristics of register architecture that they felt should be accounted for in graph coloring based allocators. They point out that Chaitin-style allocators depend on the strict independence and interchangeability of all of the registers considered in the coloring selected[6], which are not always guaranteed. Some instructions explicitly affecting certain registers can affect other registers' values, an occurrence called aliasing. Some instructions also can only use registers that have been designated for certain types of data, meaning not all unallocated registers are available for any live unit being considered for allocation. To use Briggs, Cooper, and Torczon's example, moving a double precision floating point number into registers affects two registers, but depending on the instruction set, the instructions will only reference one register explicitly, but the adjacent register will be affected as well. A return instruction affects the instruction pointer register in many instruction sets.

Smith, Ramsey, and Holloway account for these issues with the concept of register classes, or sets of registers dedicated to a specific purpose. If a piece of data or a particular instruction can use multiple register classes, its register class is denoted as the union of those register classes. This alters the implementation of an allocator, whether pessimistic or optimistic, in several ways. Copy coalescing can only be completed if the intersection of the source's and destination's register classes is a proper register class. To account for aliasing, once a register is allocated, only the complement of the set of all the possible aliases to that register are available. Smith, Ramsey, and Holloway define several quantitative measures to assess the colorability of a given graph under their register class criterion, and so the actual implementations of these concepts is based on various equations using these values concerning the complements of such available sets, the results of which describe what is known as register pressure.

3.3.2 Tavares, Colombet, Bigonha, Guillon, Pereira, Rastello. Andre Tavares, Quentin Colombet, Mariza Bigonha, Christophe Guillon, Fernando Pereira, and Fabrice Rastello built upon Smith, Ramsey, and Holloway's approach in 2011. They noted that separating the spilling and register allocation phases of an allocator simplified the algorithm and made it modular, allowing for better implementation of other program optimizations. Tavares et al. describe a "decoupled" allocator as having the following form:
"1. lower the register pressure at each program point, using any heuristics for variable spilling...until the variables alive at each point can be colored 2. use live range splitting to guarantee [the maximum local register pressure at any program point equals the global register pressure]. In general copies with a parallel semantic are used to split live ranges... 3. assign variables to registers...The register allocator must be able to find a way to assign registers to variables without causing further spills; 4. get rid of [empty set] functions and parallel copies..." [7].

Tavares et al. adapted George and Appel's and Bouchez et al.'s allocator to apply Smith et al.'s measures for register aliasing and uniqueness, and then use one of three tests, tests that were developed as part of the Briggs et al. allocator, Lal and George allocator, and the Bouchez et al. allocator, to determine whether the interference graph is colorable after each coalescence and spill.

3.4 Minimizing Spill by Coalescing Copy Instruction Nodes

3.4.1 Odaira, Nakaike, Inagaki, Komatsu, Nakatani. IBM Researchers Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani point out that it is not always most efficient to allocate entire live ranges. It is in fact more efficient to split live ranges, and allocate parts to one register, other parts to another, and leave still others in secondary memory. Odaira et al.'s implementation of these changes is relatively simple, because it has a very similar structure to Briggs et al.'s optimistic allocator. The difference is that Odaira et al. modify the optimistic allocator to include a primary coloring, which takes place after splitting of live ranges. Coalescence is modified to prioritize the coalescing of nodes representing different sub-ranges of the same live unit. After this color based coalescence, the resultant interference graph is colored for allocation. Odaira et al. claim that this method reduces program execution times by up to 15

ACKNOWLEDGMENTS

Thank you to Professor Ben Wood for his guidance in the research and writing process

References

[1] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1994. Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems 16, 3 (May 1994), 428-455.

[2] Gregory Chaitin. 1982. Register allocation and spilling via graph coloring. ACM SIGPLAN (1982), 66-74.

[3] Fred C. Chow and John L. Hennessy. 1990. The priority-based coloring approach to register allocation. Transactions on Programming Languages and Systems 12, 4 (October 1990), 501-536.

[4] David Callahan and Brian Koblenz. 1991. Register allocation via hierarchical graph coloring. Proceedings of the ACM SIGPLAN âĂŹ91 Conference on Programming Language Design and Implementation (1991), 192-203.

[5] Brian R. Nickerson. 1990. Graph coloring register allocation for processors with multi-register operands. Proceedings of the ACM SIGPLAN âĂŹ90 Conference on Programming Language Design and Implementation (1990), 40-52.

[6] Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A generalized algorithm for graph-coloring register allocation. ACM PLDI '04 39, 6 (2004), 277-288.

[7] AndrÃĺ L.C. Tavares, Quentin Colombet, Mariza A.S. Bigonha, Christophe Guillon, Fernando M.Q. Pereira, and Fabrice Rastello. 2011. Decoupled graph-coloring register allocation with hierarchical aliasing. Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems - SCOPES 11 (2011), 1-10.

[8] Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. 2010. Coloring-based coalescing for graph coloring register allocation. Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10 (2010), 160-169.