RETROSPECTIVE:

Register Allocation by Priority-based Coloring

Fred Chow
Key Research Inc.
fchow@keyresearch.com

John Hennessy
Stanford University
hennessy@stanford.edu

1. ORIGIN OF THE IDEAS

The idea that register allocation can be modeled as a graph color problem has been noted earlier by Cocke, Yershov, Schwartz [13] and others. Chaitin and his colleagues first demonstrated the feasibility of applying the coloring technique to register allocation in the experimental PL.8 compiler [3, 2]. When we were working on a register allocator for the Stanford Ucode compiler, we felt there was a need to develop an alternative register allocation algorithm that takes a more practical view towards register allocation in real programming situations without being too centered on the interference graph. We were motivated by our observation that graph coloring by itself does not take the spacial dimension of the program into account. This results in lack of linkage between the coloring and the actual occurrences of the variables in the code.

We came up with the view that a register allocator should strive to make maximal use of the available registers over each region of the code, and that the process should continue until either all registers are used up or there is no more good candidate to allocate. This region-driven view lead us to live range splitting. As long as there are available registers, splitting a candidate allows part or all of the live range to be allocated to register, even though not the same register is available throughout its range. Because splitting only occurs late in the coloring process, candidates colored earlier have the advantage of being allocated without splitting and thus handled more efficiently. Thus, it is crucial to order the allocation process so that important candidates are colored first. This lead us to the use of priority functions to order the allocation. Hence, we called this approach *priority-based coloring*.

2. IMPLEMENTATION ENVIRONMENTS

Priority-based coloring was originally developed as the global register allocator for the Stanford Ucode compiler. At that time, the Ucode Optimizer, UOPT [4], was the only component in the Ucode compiler that performs its work at the global scale. Thus, the register allocator was implemented as the last phase of the Ucode Optimizer, and the allocation was performed at the intermediate representation level. By making it operate based on a set of machine parameters, the same register allocator can work for a variety of machines.

The above implementation approach has served us well in the research setting, but industrial-strength compilers by and large have departed from this approach. Compilers nowadays usually perform register allocation late in the code generation phase. This allows more customization of the register allocator towards the target instruction set and program linkage, and enables the allocator to take care of candidates generated by the earlier phases of the code generator. At the same time, it promotes more tight coupling between the register allocator and other target-dependent optimization phases like instruction scheduling.

Modern compilers also implement a register promotion phase [11]. By taking into account potential aliasing and side effects, register promotion computes the ranges where a variable can be promoted to register, and replaces the variable over those ranges by a symbolic register. At the same time, it identifies the optimal points in the program to load the variables to their symbolic registers or update their memory locations by their values in the symbolic registers. As a result, the register allocator only has to work on symbolic registers instead of any general program data value.

3. HEURISTIC NATURE

Priority-based coloring is a heuristics-based approach to register allocation. It may not find the optimal allocation even for interference graphs that can be colored without spilling. Instead, it is geared towards finding good allocation solution for programs with non-trivial interference graphs.

To ensure that candidates that are trivial to color would not affect the quality of the overall allocation, the algorithm classifies a live range as *unconstrained* if its number of neighbors in the interference graph is less than the number of registers available. Unconstrained live ranges are colored after all constrained live ranges have been worked on. This classification into unconstrained live ranges is conservative, because there could still be enough registers if more than one of its neighbors in the interference graph could be assigned the same register. By performing the allocation in the order of decreasing priority, imperfect allocation will be incurred (via spilling and splitting) only for the less important candidates, so that the overall performance of the program will be least compromised.

Our original implementation incorporated a local register allocation phase for identifying obvious candidates that should occupy registers over each basic block. The purpose was to guard against situations where global competition for registers results in no more register being left for candidates that are most qualified for register residence from the local perspective. However, such a situation seldom arises because basic blocks are usually small. We have since simplified the algorithm by removing this local allocation phase. Larus and Hilfinger have come to similar conclusion in their implementation in the SPUR Lisp compiler [10].

4. VARIATION IN IMPLEMENTATION

Because priority-based coloring is a region-based approach to register allocation, it keeps track of the live range of each candidate via the basic blocks covered by the live range. Based on this rep-

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

resentation of live ranges, it is easy to determine if two live ranges interfere with each other by checking if the intersection of their two sets of basic blocks is non-empty. In coloring implementations, the interference graph provides a quick means to iterate through the neighbors of each node in the graph. For example, we can determine if live range A is unconstrained by counting its number of neighbors in the interference graph. Alternatively, we can nd its interfering live ranges by looping through the basic blocks belonging to A and collect the live ranges that cover any of these blocks. This method was used in the SPUR Lisp compiler, and the number of neighbors and number of colored neighbors are cached in each live range node, since they do not change frequently but are frequently referenced. Thus, the creation of the interference graph is not required in the implementation.

5. VARIATION IN HEURISTICS

In [14], Sorkin advocates doing away with the interference graph by using a different de nition of unconstrained and constrained live ranges. In his view, a candidate is colorable as long as different colors can be assigned to it over different parts of its live range. Thus, a live range is unconstrained if over each of its basic block, the number of live ranges competing for registers is less than or equal to the number of registers available. A live range can be colored by assigning different colors to it at different parts of its range, with appropriate register moves inserted at cross-over points. This avoids the overhead of splitting the live range before allocating the split parts to different registers, thus speeding up the allocation process. In this scenario, the interference graph becomes totally irrelevant. Live range splitting is carried out as usual to achieve the effect of spilling the candidate over basic blocks where there is no register available, and coloring the remaining parts.

As is the case with any heuristics-based algorithm, variations to the heuristics will result in differences in allocation results. In practice, the best heuristics should be determined via the process of tuning the compiler for a particular processor.

6. SUBSEQUENT DEVELOPMENTS

The Ucode Optimizer was productized in the commercial MIPS Ucode Compiler for MIPS RISC processors [7]. At the time of its release, it was among the rst production compiler that incorporates global register allocation via graph coloring. One essential requirement for commercial use is that it must not be bogged down when performing its work on huge routines where live range splitting dominates the allocation time. By virtue of its coarse-grained approach to computing interferences, priority-based coloring yielded reasonable compilation speed on huge benchmark programs even when compiling on 8 MHz mini-computers of that time. A detailed description and analysis of that implementation is provided in [6].

After its rst release, the register allocator in the MIPS Ucode Compiler was extended to support interprocedural register allocation by building on the priority-based coloring framework [5]. Subsequent generations of SGI compilers have re-implemented priority-based coloring in the code generation phase. In the MIPSpro compiler, a pre-scheduling phase is used to establish the number of registers that needs to be set aside for local usage. A separate local register allocation phase operates closely with the nal instruction scheduling phase to allocate local candidates [12]. This scheme allows for better separation of register resource between the global and local allocators. The MIPSpro compiler was retargeted to Intel's Itanium processor and open-sourced in 2000 as the Pro64/Open64 compiler [8]. In that effort, priority-based coloring was extended to work with Itanium's register stack. After its open source, numerous academic and commercial organizations have adopted the Pro64/Open64 compiler for use in research and product development [1].

Other compilers known to incorporate priority-based coloring include the SPUR Lisp compiler [10] and the Trimaran compiler [9].

REFERENCES

[1] ORC as a research compiler infrastructure. *Birds-of-a-Feather Session, ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[2] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[4] F. Chow. *A portable machine-independent global optimizer — design and measurements*. PhD thesis, Tech. Rep. 83-254, Computer System Lab, Stanford University, December 1983.

[5] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 85–94, June 1988.

[6] F. Chow and J. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[7] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *Proceedings of COMPCON*, pages 132–137, March 1986.

[8] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure. *Tutorial, International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[9] H. Kim, K. Gopinath, and V. Kathail. Register allocation in hyper-block for EPIC processors. In *Parallel Computing: Fundamentals and Applications, Proceedings of the International Conference ParCo'99*, pages 550–557. Imperial College Press, 2000.

[10] J. R. Larus and P. N. Hil nger . Register allocation in the SPUR Lisp compiler. In *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 255–263, June 1986.

[11] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, pages 26–37, June 1998.

[12] S. Mantripragada, S. Jain, and J. Dehnert. A new framework for integrated global local scheduling. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 167–174, October 1998.

[13] J. T. Schwartz. *On Programming: an Interim Report on the SETL project*. Courant Institute of Mathematical Sciences, October 1973.

[14] A. Sorkin. Some comments on "The priority-based coloring approach to register allocation". *SIGPLAN Notices*, 31(7):25–29, July 1996.

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction
SIGPLAN Notices Vol. 19, No. 6, June 1984

Register Allocation by Priority-based Coloring

Frederick Chow† and John Hennessy

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The classic problem of global register allocation is treated in a heuristic and practical manner by adopting the notion of priorities in node-coloring. The assignment of priorities is based on estimates of the benefits that can be derived from allocating individual quantities in registers. Using the priorities, the exponential coloring process can be made to run in linear time. Since the costs involved in register allocation are taken into account, the algorithm does not over-allocate. The algorithm can be parameterized to cater to different fetch characteristics and register configurations among machines. Measurements indicate that the register allocation scheme is effective on a number of target machines. The results confirm that, using priority-based coloring, global register allocation can be performed practically and efficiently.

1. Introduction

The view of global register allocation as a graph coloring algorithm has long been established [7]. A coloring of a graph is an assignment of a color to each node of the graph in such a manner that each two nodes connected by an edge do not have the same color. In register allocation, each node in the graph, called the interference graph, represents a program quantity that is a candidate for residing in a register. Two nodes in the graph are connected if the quantities interfere with each other in such a way that they must reside in different registers. In coloring the interference graph, the number of colors used for coloring, r, is the number of registers available for use in register allocation. The goal is to find the best way to assign the program variables to registers so that the execution time is minimized. In global register allocation, we take into account entire procedures in deciding on the variables to be colored.

The standard coloring algorithm to determine whether a graph is r-colorable is NP-complete. It involves selecting nodes for which to guess colors, and backtracking if the guesses fail [1]. The algorithm takes only linear time when the first trial succeeds. But if the graph is not r-colorable, or is near the borderline cases, an exponential amount of computation is needed to prove that it is indeed so, since it is necessary to backtrack and attempt all possible coloring combinations before reaching the final conclusion. Thus, the standard coloring algorithm works well only when the target machines have a large number of registers. A heuristic procedure with linear running time might be best in practice, and the exponential algorithm may be made only the last resort in critical situations.

Global allocation of registers by coloring usually does not take into account the cost and saving involved in allocating variables to registers. By cost, we refer to the presence of register-memory transfer operations that put variables in registers or update their home locations to make the registers available for other uses. By saving, we refer to the gain in execution speed due to individual variables being accessed in registers. Variables occur with different frequencies and with varying degrees of clustering, so that the relative benefits of assigning registers to variables differ.

† Present address: Daisy Systems Corp., 139 Kifer Court, Sunnyvale, CA 94086.

The standard coloring algorithm always tries to allocate as many items in registers as possible. It does not recognize that it is sometimes non-beneficial to assign certain variables to registers over some regions in the program, possibly due to the need to save registers before procedure calls or for letting other variables use the same registers. When it is found that an r-coloring is impossible, the decision regarding which variables to be excluded in the coloring (i.e. to be spilled) is difficult to make. The spilling decisions are separate from the coloring decisions, and it is hard to predict the effect of spilling a certain variable on the outcomes of the subsequent coloring attempts. The standard coloring algorithm also does not take into account the loop structure of the program. In practice, variables occurring in frequently executed regions should be given greater preference for residing in registers.

In this paper, we present a global coloring algorithm that overcomes the above problems. The algorithm finds reasonable, though not necessarily optimal, solutions quickly, and it works for most configurations of general-purpose registers in target machines up to and including the grouping into non-intersecting register classes. It involves assigning priorities to all register-residing candidates and ordering register assignments according to this priority. The algorithm does not backtrack, and the running time is proportional to the number of registers and the number of possible live ranges to be allocated to registers.

2. Background

The algorithm we present in this paper is the register allocation algorithm used in the production optimizer UOPT [2]. UOPT is a self-contained, portable and machine-independent global optimizer on a machine-independent intermediate language called U-Code [5]. This intermediate code is output from a Pascal front-end and a Fortran front-end. The optimized versions of U-Code are translated into different target machine code by different back-end code generators.

The global optimizer UOPT performs a comprehensive set of global and local optimizations. Global register allocation is done in UOPT as the last phase of optimization, when the final structure of the code to be emitted has been determined by earlier optimization phases and all potential register uses have been exposed. A fixed number of registers is reserved for use by the code generators. The rest are to be freely allocated by the optimizer. Since the input program is assumed executable without using the global optimizer, all program variables in the input are assumed to have been allocated in main memory. Temporaries generated by the previous phases of the optimizer are also assumed to have been allocated in home memory locations, and they are treated uniformly as variables. Due to these assumptions, it is not necessary to generate spill code for variables not allocated to registers. Instead, all objects have home memory locations and the optimizer attempts to remap memory accesses to register accesses. This contrasts with the approach used in the PL.8 compiler [1] in which the register allocation phase attempts to map the unlimited number of symbolic registers assumed during earlier compilation and optimization phases into hardware registers; if this is unsuccessful, code is added to spill computations from registers to storage and later re-load them.

The register allocation algorithm used is a combination of a local method based on usage counts and the global method based on coloring. The local phase allocates one block to a register each time. The global phase allocates one live range to a register each time. The local register allocation phase is inexpensive and near-optimal for straight-line code, but does little to contribute to the globally optimal solution. The global allocation phase is more computation-intensive. In our approach, the local allocation process is made to do as much allocation as possible so long as the allocation would not have any effect on the outcome of the global allocation phase.

3. Cost and Saving Estimates

In performing register allocation, we divide the program code into code segments, each not longer than a basic block, which represent the smallest extents of program code over which variables are allocated to registers. Assigning a variable to a register involves the loading of the variable from main memory to the assigned register prior to referencing the variable in a register in the subsequent code. If the

value of the variable is changed in the intervening code where it resides in register, the home memory location of the variable has to be updated with the register content at the end of the code segment unless it is dead on exit. These extra move operations between registers and memory represent the execution time cost of the register assignment. The execution time saving of the register assignment refers to how much the code segment is rendered faster due to the variable's residing in a register. Thus, we define the following three parameters, which vary among target machines:

MOVCOST — The cost of a memory-to-register or register-to-memory move, which in practice is the execution time of the U-Code instructions RLOD (load to register) or RSTR (store from register) respectively in the target machine.

LODSAVE — The amount of execution time saved for each reference of a variable residing in register compared with the corresponding memory reference that is replaced.

STRSAVE — The amount of execution time saved for each definition of a variable residing in register compared with the corresponding store to memory being replaced.

4. Local Register Allocation

Local register allocation refers to allocation in a straight-line piece of program code, and it precedes the global allocation phase. The method of allocating registers locally using reference counts is well-established and inexpensive [3]. Locally optimal solutions to the register allocation problem do not necessarily add up to the globally optimal solution. However, it is possible to determine a portion of register allocation locally that also belongs to the global solution, so that the work load of the subsequent, more expensive global allocation phase can be made smaller.

For each variable in the local code segment being considered, the local saving that can be achieved by assigning the variable to register can be estimated by:

$$\texttt{NETSAVE} = \texttt{LODSAVE} \times u + \texttt{STRSAVE} \times d - \texttt{MOVCOST} \times n \quad (1)$$

where u is the number of uses of the variable,
d is the number of definitions and
n is either 0, 1 or 2.

n depends on whether a load of the variable to a register (RLOD) at the beginning of the code segment and a store from the register back to the variable's home location (RSTR) at the end of the code segment are to be inserted. If they are both needed, n is 2. If the first occurrence of the variable is a store, then the initial RLOD is not needed. If the variable is not altered, or if the variable is not live at the end of the code segment, then the RSTR is not necessary.

If the local code segment is considered together with its preceding and subsequent code, the term involving MOVCOST represents the uncertainty in cost with regard to NETSAVE that may or may not contribute to the final global solution. This is because if the variable is also allocated to the same register in the surrounding code, then the RLOD and RSTR at the beginning and end of the current code segment are unnecessary, and the actual value of NETSAVE is increased. Thus, for each variable in the local code, we consider two separate quantities:

$$\texttt{MAXSAVE} = \texttt{LODSAVE} \times u + \texttt{STRSAVE} \times d \quad (2)$$

$$\texttt{MINSAVE} = \texttt{LODSAVE} \times u + \texttt{STRSAVE} \times d - \texttt{MOVCOST} \times n \quad (3)$$

The quantity MINSAVE represents the minimum saving in the local code segment gained by allocating the variable to register. The quantity MAXSAVE is the maximum possible saving. The actual saving after all register allocation is performed will range between MINSAVE and MAXSAVE. The parameters MAXSAVE and MINSAVE also apply to variables which do not occur in the code segment, when they are both 0; in such cases, the two parameters are used only in the later global allocation process.

When the surrounding blocks are considered together with the current block, the local allocation may displace some other variable which has been assigned to the same register in the adjacent blocks and which, if allowed to occupy the same register in the current block, would enable the elimination of the RSTR's at the ends of the preceding blocks and the RLOD's at the starts of the succeeding blocks. Thus, the absolute criterion for determining the local allocation of a variable in register can be given as:

$$\texttt{MINSAVE} > \texttt{MOVCOST} \times (p + s) \quad (4)$$

where p is the number of predecessors,
s is the number of successors of the block.

When this condition is satisfied, the variable can be locally allocated in register with certainty regardless of the rest of the program. In computing the above condition, the loop nesting depths of the blocks are used as weights.

The determination of exactly which register to assign is not done in the local allocation pass. It is delayed until the global allocation phase, when the optimizer will look for opportunities to assign the same register to a variable over contiguous code segments to minimize the number of RLOD's and RSTR's.

5. Computing the Live Ranges

A live range of a variable is an isolated and contiguous group of nodes in the control flow graph in which the variable is defined and referenced. No other definition of the variable reaches a reference point inside the live range. Also, the definitions of the variable inside the live range do not reach any other reference point outside the live range. Global register allocation assigns complete live ranges to registers, and if this is not possible, parts of live ranges are assigned. Throughout a procedure, each register is occupied by live ranges or parts of live ranges that do not overlap. Computations for the separate live ranges of the program variables require processing and representation overhead. We circumvent these computations by assuming one live range for each variable in a procedure at the beginning of the global register allocation phase, even though it may have non-adjacent parts. In the course of coloring, the live ranges are broken up into smaller segments when necessary.

By virtue of the contiguity of the blocks in a live range, when the live range is assigned to a register, RLOD's are needed only at entry points to the live range and RSTR's are required only at its exit points (Fig. 1). UOPT supports both the caller-save and callee-save convention regarding registers in procedure calls. In the caller-save context, all registers need to be freed at a procedure call so that they can be used in the called procedure. Thus, live ranges are never allowed to extend over a procedure call. The register allocator is responsible for indicating which

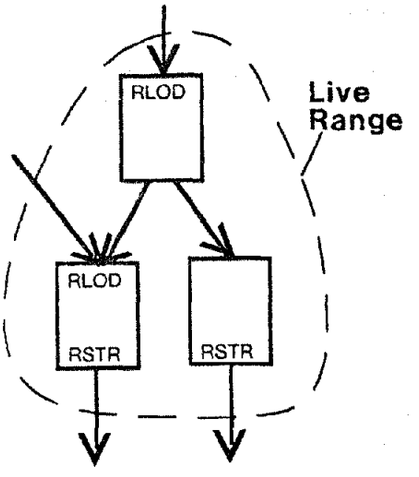

Fig. 1 A live range with associated RLOD's and RSTR's

variable home locations are to be updated from registers before a procedure call, and which variables are to be re-loaded to registers after the call. UOPT has a procedure integration pre-pass that replaces procedure calls by copying their code in-line, thus increasing non-call ranges. In the case of the callee-save convention, live ranges are allowed to extend over procedure calls, and registers are allocated across the calls.

6. The Global Coloring Algorithm

The cost and saving estimates we defined earlier, weighted by the loop-nesting depths of program points at which the variable accesses occur, play an important role in our global coloring algorithm. The coloring process is driven according to the cost and saving estimates. Each iteration assigns one live range to a register by picking the most promising live range according to the cost and saving estimates computed over each live range. By assigning the live ranges with high priorities first, it is hoped that the results of the allocation will be close to optimal. The algorithm terminates when either all live ranges or parts of live ranges have been allocated, or all registers have been used up over all code segments. Thus, the computation time does not deteriorate when r-coloring cannot be achieved. The algorithm allocates register to a live range only when the saving is higher than the cost. Thus, the over-allocation problem does not exist in our algorithm.

We assume that all variables have been assigned home locations before register allocation begins. This allows us not only to avoid the problem of having to introduce spill code. By taking into account the cost of register-memory transfer operations, we can factor the effects of not allocating in registers into the coloring decisions. Because the cost and saving estimates are weighted by loop-nesting depths, our algorithm also takes the loop structure of the program into account. Thus, variables in frequently executed regions have higher priority for residing in registers.

We assume a single live range for each variable in a procedure at the beginning of global register allocation. Apart from saving the cost of computing and representing separate live ranges prior to coloring, the interference graph is also made much simpler. The processing cost associated with accessing, manipulating and updating the interference graph during coloring is greatly reduced.

The standard coloring algorithm handles the situation of insufficient registers by spilling variables into main memory. We handle this situation by live range splitting. In the course of performing coloring, when a variable cannot be assigned the same color throughout the procedure, its live range is split into smaller live ranges. The new live ranges are treated the same way as variables as far as the coloring algorithm is concerned, and the interference graph is updated accordingly. As splitting proceeds further along, the split-out parts may not be true live ranges since the original def-use relationships may not be restricted to points inside the subranges. Splitting is repeated until all the split live ranges can be colored or until all the split live ranges consist of single code segments. If a split-out live range is left uncolored at the termination of coloring, the effect is equivalent to not allocating the variable over the region covered. Live range splitting is performed with the emphasis on not creating small live range fragments unless warranted by the situation.

As an illustration, Fig. 2(a) shows a region of code in which variables A, B and C are to be allocated in registers. Although the live range for variable A consists of two separate parts, they are initially taken as a single live range. Assume that a single register is left available to contain the three variables. Fig. 2(b)

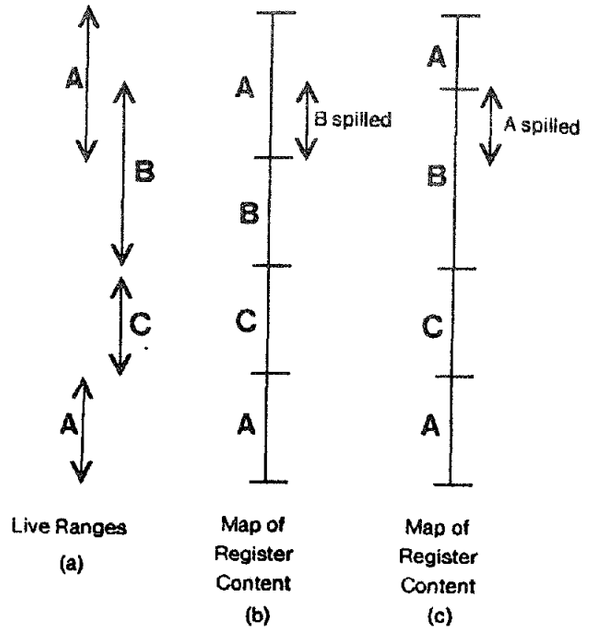

Fig. 2 Possible allocation configurations

and (c) show two possible allocation results, depending on the occurrence frequencies of the variables in the live ranges covered. To arrive at the result of Fig. 2(b), variables A and C are allocated first. The live range of variable B is then split so that one part of it is allocated and the other part spilled. In Fig. 2(c), variables B and C are allocated first. The algorithm splits the live range of A into the two naturally separate parts, and the lower live range of A is then allocated. The top live range of A is then further split so that one part is allocated and the other part that interferes with B is spilled.

In the node coloring algorithm, variables which have a number of neighbors in the interference graph less than the original number of colors available are left uncolored until the very end, since it is certain that an unused color can be found for them. These are called *unconstrained* variables or live ranges. The rest of the live ranges are assigned colors by successive iterations of Step 2 of the algorithm. Each iteration selects a live range and assigns a color to it. New live ranges are formed out of splitting during the iterations, and if any of these are unconstrained, they are added to the unconstrained pool of variables.

Algorithm *Priority-based Node Coloring.*

1. Find the live ranges whose number of neighbors in the interference graph is less than the number of colors available, and set them aside in the pool of unconstrained live ranges.

2. Repeat Steps a to c, each time assigning one color to a live range until all constrained live ranges have been assigned a color, or there is no register left that can be assigned to any live range in any code segment (taking into account registers allocated in the preceding local allocation phase).

   a. Perform Step (i) or (ii) for each live range lr until TOTALSAVE for all original or newly formed live ranges are computed:

      (i). If lr has a number of colored neighbors less than the total number of colors available, assume a color is assigned to it covering all its live blocks. Then compute and record TOTALSAVE for the variable lr as follows:

      1. In each block i of the live range lr, determine whether register load and store is necessary based on whether the adjacent blocks in the flow graph belong to the same live range. Let the number of register loads and stores be n, which ranges from 0 to 2.

      2. Compute $\texttt{NETSAVE}_i$ as

$$\texttt{NETSAVE}_i = \texttt{LODSAVE} \times u + \texttt{STRSAVE} \times d - \texttt{MOVCOST} \times n$$

         where u is the number of uses, and

         d is the number of definitions of the live range variable in block i.

      3. Let w_i denotes the loop-nesting depth of block i in the flow graph. Compute TOTALSAVE for the live range lr as:

$$\texttt{TOTALSAVE} = \sum_{i \in lr} (\texttt{NETSAVE}_i \times w_i).$$

      (ii). If the number of colored neighbors of lr is already equal to the number of colors available, then the live range lr has to be split. A new live range lr_1 is split out from lr as follows:

      A new node in the interference graph is created for lr_1. A definition block from lr, preferably one at an entry point to lr, is first added to lr_1. Blocks adjacent to lr_1 that also belong to lr are successively added to lr_1, updating the neighbors in the interference graph until the number of colored neighbors of lr_1 in the interference graph is one less than the number of available colors. The motivation of this is to produce the largest possible live range that can still be colored. This is continued until no more blocks can be added to the new live range lr_1.

      If the newly formed live range lr_1 has a number of neighbors in the interference graph less than the number of colors available, set it aside in the pool of unconstrained variables. Otherwise, add it to the pool of candidates for estimation of TOTALSAVE.

      As a result of the new node in the interference graph, some previously unconstrained live ranges may now become constrained. These have to be transferred from the unconstrained pool to the constrained pool.

   b. For each live range lr, compute ADJSAVE as

$$\texttt{ADJSAVE} = \frac{\texttt{TOTALSAVE}}{(\text{number of nodes in } lr)}.$$

      (The quantities TOTALSAVE and ADJSAVE do not have to be recomputed if the live range has not changed since the previous iteration.)

   c. Looking at the values of ADJSAVE computed for all the uncolored but constrained live ranges in Steps a and b, choose the live range with the highest value of ADJSAVE and assign a color to it.

3. Assign colors to the unconstrained live ranges, each time using a color that has not been assigned to one of their neighbors in the interference graph.

Thus, the algorithm orders the assigning of colors according to which variable currently has the highest value of ADJSAVE (Step 2c). ADJSAVE can be visualized as the total number of occurrences of the variable in the live range, weighted by loop-nesting depths and normalized by the length of the live range. The adjustment by the live range length (the number of nodes in the live range) is needed because a live range occupying a larger region of code takes up more register resource if allocated in register. In the local allocation phase, we have already taken pure occurrence frequencies into account. Thus, when entering the global allocation phase, all the variables that remain unallocated in each code segment have occurrence frequencies that do not differ widely, so the important consideration is whether the allocation enables the same register to be assigned across contiguous code segments so that register loads and stores can be minimized. The value of ADJSAVE comprises a measure of this connectedness. The more connected the code segments in the live ranges of a variable are, the more worthy is the variable to be allocated in register, and the more difficult it will be to find the same register for it throughout; so, it is important to assign a color to it before other variables. The use of the ADJSAVE criterion is justified only if the local allocation phase precedes global allocation.

The determination of n in Step 2a(i) can make use of more information than previously possible in the local allocation phase of Section 4. If the first occurrence of the variable at an entry block is a store, then the RLOD is not needed. If all the predecessors of a block also belong to the live range, then the RLOD is also not necessary, unless any of the predecessor contains a procedure call. By the same token, RSTR's at points internal to live ranges are not always necessary. An RSTR is necessary at the exit blocks of a live range only if the live range contains at least one assignment to the live range variable and the variable is not dead on exit. At blocks internal to live ranges, RSTR's are also generated if any successor node has an RLOD, or contains a procedure call.

The computation time complexity of the above algorithm can be estimated. We are mainly concerned with Step 2 of the algorithm, since this step takes a lot more time compared with Step 3 for the unconstrained live ranges. Let r be the number of registers. Let l be the number of live ranges, and assume that this stays fixed during the course of the algorithm. Also assume that each register is assigned to one and only one live range in the procedure, though in reality this is not always the case. Then there is r iterations for Step 2 of the algorithm. For the first iteration, a live range is to be chosen out of l live ranges. For the second iteration, the choice is to be made out of the $l-1$ live ranges remaining. Summing all the iterations, we get

$$l + (l-1) + \ldots + (l-r+1) = \frac{r(2l-r+1)}{2}.$$

Thus, the algorithm is $O(r(l-r))$. The time of the algorithm proportional to both the number of registers available and the number of candidates to reside in registers.

The algorithm can easily extend to the case of multiple classes of registers. The interference graph will only give interferences between variables of the same class. The algorithm is repeated once for each class of register. In each case, the number of colors corresponds to the number of registers in the class being considered.

The relative importance between the local and global phases can be varied by changing the maximum length of code segments allowed. By setting an option, a limit on the maximum number of variable appearances allowed in a basic block is imposed. If this limit is exceeded, the remaining code is made to belong to a new block. A default value for this option serves to guard against the presence of large blocks which can degrade the output of the register allocator. When blocks are small, the local phase will not be able to allocate as many items to registers based on its allocation criteria, and more work is left to the more expensive global phase. As the limit on block lengths becomes smaller and smaller, the overall allocation also approaches the optimal solution since registers can now be allocated across shorter segments to cater to any irregular clustering of accesses. The processing cost also increases correspondingly because of the larger number of blocks involved and the greater amount of work being performed by the global phase. Thus, the register allocation algorithm has a large amount of built-in flexibility with respect to processing cost and quality of results. In practice, basic

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
% of var. references in registers	.65	.40	.76	.95	.95	.94	.67
% of var. assignments in registers	.70	.23	.72	.96	.96	.77	.77

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
% of var. references in registers	.91	.78	.87	.87	.62	.71	.77
% of var. assignments in registers	.92	.74	.80	.89	.62	.75	.76

Table 1(a). Static register allocation statistics in the DEC 10

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
% of var. references in registers	.94	.69	.87	.96	.96	.95	.80
% of var. assignments in registers	.95	.54	.88	.95	.95	.78	.80

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
% of var. references in registers	.90	.77	.86	.86	.79	.80	.86
% of var. assignments in registers	.91	.76	.81	.88	.75	.91	.84

Table 1(b). Static register allocation statistics in the 68000

blocks are usually short, and most of the work is done by the global phase.

7. Measurements

The priority-based register allocation algorithm has been implemented in the production optimizer UOPT and tested on a number of target machines. The results have shown that it is effective on a wide range of machines. We now present measurements that give us some ideas about the performance and effectiveness of our register allocation algorithm. The measurements are based on running and optimizing a set of benchmarks consisting of 13 Pascal and Fortran programs. These benchmark programs are standard, compute-bound application programs, with minimal calls to un-optimizable external routines and run-times.

7.1. Statistical counts

Table 1(a) and Table 1(b) display the register allocation statistics for the benchmark programs in the DEC 10 and 68000 respectively. It shows the percentages of variable references and the percentages of variable assignments that are in the global registers. The table does not include register usage by the code generators for code generating purposes. The data are obtained by static counts in the optimized programs. The dynamic counts are expected to be much better, since the register allocator in UOPT takes loop-nesting depths into account.

The percentages of allocation displayed in the two tables are not 100% because both of the two machines are not load/store machines, and infrequently accessed variables will not be allocated. The two machines also have a good set of memory addressing modes, so that a variable will not be allocated unless a payoff is obtained.

The DEC 10 uses the caller-save linkage convention, and the DEC 10 code generator allows UOPT to allocate up to 9 registers out of the 14 available. Programs that have many procedure calls (e.g. Tower)

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
0. No optimization	9.62 (1.0)	1.68 (1.0)	3.95 (1.0)	1.29 (1.0)	1.42 (1.0)	5.22 (1.0)	1.60 (1.0)
1. Only local optimizations	10.92 (1.14)	1.68 (1.0)	4.22 (1.07)	1.10 (.85)	1.23 (.87)	5.24 (1.0)	1.42 (.89)
2. Only local optimizations and reg. alloc.	8.46 (.88)	1.39 (.83)	3.99 (1.01)	1.05 (.81)	1.19 (.84)	4.85 (.93)	1.25 (.78)
3. All except register alloc.	8.87 (.92)	1.36 (.81)	3.76 (.95)	.65 (.50)	.78 (.55)	3.74 (.72)	1.60 (1.0)
4. Full global optimization	7.44 (.77)	1.27 (.75)	2.67 (.68)	.42 (.33)	.55 (.38)	2.47 (.47)	1.30 (.70)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
0. No optimization	3.69 (1.0)	1.01 (1.0)	2.85 (1.0)	5.09 (1.0)	.719 (1.0)	4.71 (1.0)	(1.0)
1. Only local optimizations	3.79 (1.03)	1.05 (1.04)	1.82 (.64)	5.22 (1.03)	.703 (.98)	3.89 (.83)	(.95)
2. Only local optimizations and reg. alloc.	2.04 (.55)	.91 (.90)	1.68 (.59)	3.30 (.65)	.487 (.68)	3.67 (.78)	(.79)
3. All except register alloc.	4.60 (1.25)	1.08 (1.07)	1.40 (.59)	5.86 (1.15)	.807 (1.12)	3.17 (.67)	(.87)
4. Full global optimization	2.33 (.63)	.93 (.93)	1.07 (.37)	3.52 (.69)	.572 (.80)	2.36 (.50)	(.61)

Running times in Seconds
(Ratio to un-optimized running times in parentheses)

Table 2. Running times to show effects of register allocation on the DEC 10

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
0 registers	8.87 (1.0)	1.36 (1.0)	3.76 (1.0)	.65 (1.0)	.78 (1.0)	3.74 (1.0)	1.60 (1.0)
2 registers	8.25 (.93)	1.28 (.94)	3.62 (.96)	.64 (.98)	.78 (.99)	2.56 (.68)	1.48 (.93)
4 registers	7.44 (.84)	1.28 (.94)	3.29 (.88)	.58 (.89)	.71 (.91)	2.54 (.68)	1.42 (.89)
6 registers	7.44 (.84)	1.27 (.93)	2.68 (.71)	.43 (.66)	.56 (.72)	2.54 (.68)	1.42 (.89)
All 9 registers	7.44 (.84)	1.26 (.92)	2.68 (.71)	.42 (.65)	.55 (.71)	2.47 (.68)	1.30 (.81)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
0 registers	4.60 (1.0)	1.08 (1.0)	1.40 (1.0)	5.86 (1.0)	.807 (1.0)	3.17 (1.0)	(1.0)
2 registers	3.71 (.81)	.96 (.89)	1.24 (.89)	4.02 (.69)	.724 (.90)	2.89 (.91)	(.88)
4 registers	3.84 (.83)	.93 (.86)	1.14 (.81)	3.99 (.68)	.669 (.83)	2.75 (.87)	(.84)
6 registers	2.33 (.51)	.93 (.86)	1.09 (.78)	3.53 (.60)	.621 (.77)	2.40 (.76)	(.75)
All 9 registers	2.33 (.51)	.93 (.86)	1.05 (.75)	3.25 (.55)	.572 (.71)	2.35 (.74)	(.73)

Running times in Seconds
(Normalized running times in parentheses)

Table 3. Effects of the number of registers available for register allocation (DEC 10)

tend to diminish the percentage allocated because the numerous instances of register saves and re-loads around procedure calls tend to increase the cost of the allocations.

The 68000 code generator allows UOPT to use up to 6 data registers and 4 address registers, out of the 8 data registers and 8 address registers available. The register allocation statistics for the 68000 is markedly different from that for the DEC 10, which is due to the use of the callee-save linkage convention in the 68000. The percentages of variable accesses allocated in registers in the 68000 are always greater than those in the DEC 10, since register saves and re-loads do not occur around procedure calls unless there are side effects. Tables 1(a) and (b) show that the linkage convention concerning the handling of registers does affect register allocation.

7.2. Selective application

Another method to study the effectiveness of our register allocation is by comparing the running times of the benchmarks with and without register allocation. Table 2 displays the running times of the benchmarks on the DEC 10 for different extents of optimization. Rows 1 and 2 show between them the effects of adding the register allocation phase if the optimizer performs only the minimal local optimizations. Rows 3 and 4 show between them the effects of leaving out the register allocation phase when the optimizer performs its full set of optimization. The data show that the register allocation is very effective, especially when the optimizer performs other global optimizations. Without register allocation, the benefits of the other global optimizations cannot be fully exposed, because the cost of saving intermediate quantities in main memory is high enough in some cases to cancel out the benefits that can be derived from the optimizations.

7.3. Varying the number of registers

In Table 3, we investigate the effects of allowing different numbers of registers to be allocated by UOPT out of the 14 available in the DEC 10. The results displayed in Table 3 show that the optimized running times always improve when a larger number of registers are available to UOPT. The 5 registers normally used by the code generator is enough for most practical purposes, and increasing the number used by the code generator (i.e. decreasing the number used by UOPT) does not cause appreciable improvement in execution speed.

As expected, different programs require different numbers of registers for optimal register allocation. In the programs Perm, Tower and Tree, 4 registers seem to be all that are needed; for others, increasing the number further yields better execution speeds. In the programs Puzzle and Sieve, just 2 registers can dramatically improve the program running time. Different programs have different cut-off points regarding the number of registers they need for optimal register allocation. The cut-off number of registers required is related to the *chromatic numbers* of the interference graphs — the numbers of colors needed to color the graphs.

8. Concluding Remarks

In this paper, we have shown that, by using a priority-based coloring algorithm, the traditional register allocation problem can be approached practically and efficiently. Moreover, it can be performed in the machine-independent context using a few machine parameters. Among the parameters we use are characterizations of the benefits of register accesses over memory accesses. The performance and efficiency of the algorithm are not affected by the number of registers available in the target machines.

There are possibilities for further enhancement to the register allocation scheme we have presented. In the global coloring phase, register allocation priorities are computed by assuming that the register-memory move instructions are at fixed positions. The results can be improved if the priority ordering takes into account the possibility of moving the register-memory transfer instructions to positions that can minimize execution time cost.

The problem of allocating overlapping registers of different sizes have not been considered in this paper. It would be interesting to see to what extent the priority-based coloring scheme can be adapted to such situations.

Acknowledgement

This work represents part of the research performed for the S-1 project, under Contract 2213801 from the Lawrence Livermore National Laboratory. The development of the S-1 computer is funded by the Office of Naval Research of the U. S. Navy and the Department of Energy.

References

[1] G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *ACM SIGPLAN Notices, 17,* 6 (June 1982), *(Proceedings of the SIGPLAN 82 Symposium on Compiler Construction),* pp. 201 - 207.

[2] F. Chow, "A Portable Machine-independent Global Optimizer — Design and Measurements," Ph.D. Thesis and Technical Report 83-254, Computer System Lab, Stanford University, Dec. 1983.

[3] R.A. Freiburghouse, "Register Allocation Via Usage Counts," *Comm. ACM 17,* 11, Nov. 74.

[4] B. W. Leverett, "Register Allocation in Optimizing Compilers," Ph.D. Thesis and Technical Report CMU CS-81-103, Carnegie-Mellon University, February 1981.

[5] D. Perkins and R. Sites, "Machine-independent Pascal Code Optimization," *ACM SIGPLAN Notices, 14,* 8 (August 1979), *(Proceedings of the SIGPLAN 79 Symposium on Compiler Construction),* pp. 201–207.

[6] R.L. Sites and D.R. Perkins, "Machine-independent Register Allocation," *ACM SIGPLAN Notices, Vol. 14, Number 8 (August 1979), (Proceedings of the SIGPLAN 79 Symposium on Compiler Construction),* pp. 221–225.

[7] J.T. Schwartz, "On Programming: An Interim Report on the SETL Project," Courant Institute of Math. Sciences, New York University, 1973.