A Parametric View of Retargetable Register Allocation

Kelvin S. Bryant
ksb@cs.umd.edu
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742
301-405-2703

Jon Mauney
mauney@csc.ncsu.edu
Dept. of Computer Science
N.C. State Univ., Raleigh, NC 27695

January 24, 1995

Abstract

We discuss the problems involved in building a retargetable register allocator for use in an optimizing compiler. While the popular "register coloring" method is machine-independent, the allocator as a whole must implement numerous machine-dependent decisions. We present the kinds of information that must be parameterized in order to include register allocation in a retargetable compiler back-end, and discuss a sample solution.

1 Introduction

Register allocation is an important compiler optimization, and numerous strategies for allocating registers have been described in the literature. The register allocation approach is usually published as an algorithm that can be applied to any target architecture. However the allocator as implemented in a compiler must deal with machine-specific details and apparently must be hand-tuned to fit the target. In order for a code-generator generator system to include register allocation, the allocator must be implemented as a general algorithm, with the target-machine details suitably parameterized for easy retargeting.

For example, we will deal with what is probably the most popular register allocation technique, graph coloring, which was first described by Chaitin [CAC+81] and subsequently improved many others [CH84, BGG+89, BCKT89]

Graph coloring is used to solve resource allocation problems where resource usage must be mutually exclusive. Nodes in the graph represent clients that use resources, edges link clients that need resources during overlapping time intervals, and finally, colors represent the resources. The problem is to assign a color to each node using a maximum of k colors while not assigning the same color to nodes that are connected by an edge. The register allocation equivalent, termed register coloring, attempts to color a graph called an *interference graph.* In the interference graph, nodes represent uses of a value or variable (known as a live range), edges connect live ranges that are live simultaneously, and allocatable registers represent the the limited colors. Since the graph coloring problem is known to be NP-complete, Chaitin proposed a heuristic (shown in figure 1).

The register coloring method is elegant and apparently quite general. However, in practice, register set configurations and register usage conventions are highly idiosyncratic. A practical register allocator must accomodate the details of the machine, which complicates retargeting register

```
(to allocate k registers, by finding a k-coloring of a graph)
Repeat
   Build the interference graph.

   Repeat
      if there is an unconstrained node (a node with degree less
      than k) left in the graph then
         - Remove the node and all its edges from the graph.
         - Push the node onto coloring stack.
      else
         - Remove a constrained node and all its edges from the graph.
         - Mark the node for spilling.
         - Push node onto coloring stack.
   Until the graph is empty.

   Add spill code for the nodes marked for spilling.

Until no nodes need to be spilled.

Repeat
   - Pop node from coloring stack.
   - Add node and all its edges back into graph.
   - Assign a color that differs from all its neighbors.
Until coloring stack is empty.
```

Figure 1: Chaitin's Coloring Heuristic

allocators to different architectures. For example, consider the following register set configurations and assembly usage conventions:

MIPS R4000: A RISC architecture that contains 32 32-bit general purpose registers that store integer, char, and pointer values. There are also 16 64-bit floating point registers that store float and double values. Function actual parameters are passed using a combination of registers and the program stack depending on the data types and number of parameters.

MC680x0: A CISC architecture that contains 8 32-bit data registers that store integer, char, float and double values. Addresses are stored in 8 32-bit address registers. Although address registers can also store integers, only the addition and subtraction arithmetic operations use address register operands.

VAX: A CISC architecture containing 16 32-bit registers that store integer, char, pointer, float and double values. Scalar function results are returned in register *r0*. Parameters are passed using the program stack (or main memory) and a special assembly *call* instruction.

These examples illustrate the machine-dependencies that a register allocator must accomodate. If the register allocator is to be part of a retargetable code generator, then the allocator will have to deal with the following problems:

- Different machines will have different k values for the different register sets.
- A value of a particular data type may be handled differently on different architectures. For example, the MIPS has a distinct register set to store float values. But on the VAX, there is only one register set to store all data types, so integers and floats will interfere.

- On typical RISC machines, operands only reside in registers which increases the size and complexity of the interference graph. In addition, RISC machine must also reserve registers to store intermediate results.
- The first phases of the code generator construct the control flow graph and live ranges. Different code generators will represent these structures differently.
- Machines have special purpose registers that can be allocated in some portions of a program but not in others. For example, some machines require function results to be returned in a specific register.
- Some machines use register pairs store certain values and instruction results.

In addition, the compiler writer must consider the classic phase-ordering problem between the register allocator and the code selector. If aggressive register allocation is performed before code selection, a CISC code selector may be forced into chosing less than optimal instructions to match the choices made by the register allocator. On the other hand, if code selection proceeds first, the register allocator may be forced into a suboptimal allocation due to restrictions imposed by the selected instructions.

Compiler writers use code generator generators to aid the development of new backends; however, because of the above problems, register allocators are still usually hand-coded. Even with hand-coded allocators, coordinating the actions of the code selector and register allocator to minimize the phase-ordering problem is often overlooked. Previous papers describing register allocation have typically included implementation results, but none have discussed retargetability.

This paper describes an interface for building retargetable register allocators. The perspective will be that of the register allocator. The question is "What would ease the task of building/retargeting allocators?" We wish to completely encapsulate the register allocator. Thus we need to isolate the machine-specific information so that it can be provided as a part of the code-generator system; on the other hand, we also want to be able to replace the register allocator with an improved strategy – or move the allocator to another compiler – so we must also identify the algorithm-specific needs of the allocator. We will use the term "relocatable allocator" to describe a register allocator that is designed to be portable among target machines and among containing compilers. The next section describes the data structures and accessing routines needed by a typical coloring algorithm. Section 3 presents a coloring algorithm designed using our ideas. Finally, Section 4 gives concluding remarks and future work.

2 Parametric Description

Performing register coloring requires several data structures that contain machine independent *and* machine dependent information. These structures are built in the early phases of the code generator and vary in implementation from one compiler to another. If retargeting to multiple compilers and architectures is to be achieved, accessing routines which hide implementation and architectural details must be provided by the compiler writer. We briefly describe the data structures, the associated accessing routines, and the interface between these structures and the coloring algorithm. The first three of these concern the interface between the allocator and the rest of the compiler. The fourth contains the machine-specific information.

2.1 Interference Graph

Discovering interferences between nodes is a machine-dependent operation. For example, integers and floats on the MIPS are stored in two different register sets; therefore, no edges will join integer and float live ranges. Other machines may use the same register set for integers and floats, so there may be conflicts represented in the interference graph. In essence, the set of edges, E, in the graph will vary for the same program on different machines. This difference can be encapsulated into a single function that compares two live-ranges and returns "true" if they interfere. The function, in turn, would use the register-set description information of section 2.4.

The interference graph is apparently independent of the details of the allocation algorithm, and can be constructed by the code generator. However, many coloring routines rebuild the interference graph several times during execution. Therefor, the compiler writer must provide acessing routines to allow incremental modification of the interference graph. The following is a description of the necessary routines:

insert: Given a live range, analyze conflicts and add the node to the interference graph.

delete: Remove a node and associated edges.

traverse: Traverse all nodes in the graph and apply a given operation.

Building the original graph becomes a matter of traversing the list of live ranges and applying the `insert` operation. The `delete` operation should return a reference to the removed nodes to optimize iterative heuristics such as Chaitin's.

2.2 Live Ranges

The vertices of the interference graph are composed of live ranges. Live ranges serve as the atomic unit of allocation for the coloring routine; however, some coloring routines will *split* live ranges as part of the spill process. Splitting live ranges requires the coloring routine to manipulate individual live ranges which are implementation specific. Coloring routines also employ cost functions to help make coloring and spill decisions. These cost functions must be able to access live range information such as live range length, type of reference (read/write), and nesting depth of individual references. Other general access routines are as follows:

- Create a new empty live range.
- Remove a single element from a live range.
- Add a single element from a live range.
- Transfer references from one live range to another.
- Traverse a live range and access individual fields (which ones?)
- Create pointers to live range elements.
- Advance to the next/previous live range.
- Go to beginning of the live range list.

With the above routines, the compiler writer can now manipulate live ranges in an implementation independent way. The spill routine can split live ranges and update fields to reflect the new relationships in the interference graph.

```
new_lr = {};
Find entry point(s) into live range lr and enqueue into queue Q;
while ( (Q is not {}) and
   (Number of colored neighbors in new_lr <= number of allocatable registers) and
   (Number of blocks in lr > 1  /* lr is non-empty */ ) do

   block = head(Q);
   if (block is in lr) then
      Move block from lr to new_lr;
      if (block has 1 successor) then
         Add this successor to head of Q.
      else
         Add successors to tail of Q.
od
```

Figure 2: Live range splitting heuristic

2.3 Control Flow Graph

The Control Flow Graph (CFG) captures the flow of control between the basic blocks of the user program. As mentioned earlier, some coloring routines will split live ranges as part of the spill process. The split routine uses the CFG and the interference graph to locate appropriate split locations. Figure 2 shows the algorithm for a typical split routine [LH86].

The algorithm performs a breadth-first search of the given live range (using the CFG) to build two new live ranges. The first live range will be composed of the live ranges elements traversed in the breadth-first search. The algorithm attempts to make this live range as large as possible while making it colorable (i.e. the number of colored neighbors should remain less than the number of allocatable registers). The second live range is composed of the remaining nodes. To support access to the CFG, the following routines are needed:

- Return the first basic block of the program.
- Given a basic block, return all immediate successor blocks.
- Return the program statements contained in a given basic block.

2.4 Register Set Descriptions

The coloring routine will assign registers to the live ranges represented in the interference graph. These assignments are accomplished by matching the data type of a live range with a register set designed for that data type. A retargetable allocator must be able to access a minimal set of information concerning the registers on the host machine.

- Map a live range to a register set.
- Given a register set, return the number of allocatable registers.
- Given a register set and a particular register number, return the assembly name of the register.
- Return the *forbidden list* of a live range. The forbidden list consists of the register numbers assigned to neighbors.

It is assumed that the code generator will know how to map a live range to a register set using information provided by the compiler writer before compiler compile time. If the register sets are numbered, the register information could be represented by a simple matrix indexed using *(Register set number X register number).*

3 Register Allocator Example

The goal of this section is to demonstrate the flexibility of the interface in building relocatable coloring routines. Figure 3 shows an implementation of the heuristic given in figure 2. The routine relies on the accessing routines described in section 2 to retrieve implementation and hardware specific information. The following is a summary of the important features of the example coloring routine.

Function `buildIntGraph` builds the original interference graph. It makes a call to `insertNode` for each live range in the live range list. The live range accessing routines allow `buildIntGraph` to iterate through the live ranges. In addition, these accessing routines also allow `insertNode` to compare live ranges (nodes) in the graph for conflicts.

The outermost loop executes until a k-coloring is guaranteed. This is accomplished by a series of operations that remove unconstrained nodes from the graph followed by marking (for spilling) and removing one or more constrained nodes. Chaitin's heuristic assumes that the code selector has already executed and generated register references for each operand. Thus, spilling requires the allocator to insert code to load registers from memory at each reference in the spilled live range. Spill code will vary for different machines and is handled by lower-level routines in the allocator.

Function `getConstrained` returns an unconstrained node if one exists. This process is optimized if the data structure containing the live ranges is sorted and indexed by degree [MB81, BCKT89].

Function `getLowCost` returns a pointer reference to the constrained node with the lowest spill cost. There are many adequate methods of computing spill cost but most rely on a combination of the degree of the live range, the number of references in the live range and the nesting depth of the references. The interface allows modification of the function without modifying the coloring routine.

As a final note, consider deleting and reinserting nodes in the graph. Spilled nodes are not returned to the interference graph; however, their removal affects neighboring nodes. A good graph data structure should allow quick access between neighboring nodes in the graph. When a node is spilled, neighboring nodes should be incrementally updated to reflect the fact that the spilled node will not be returned to the graph. Other nodes should only be "psuedo" removed (i.e. just marked as removed but remaining in the graph data structure). This process optimizes the subsequent rebuilding phase of the allocator.

4 Conclusions

We have described major issues related to retargeting register allocators to different architectures and different optimizing compilers. The idea is to provide a common set of routines for accessing the data structures containing machine specific and compiler specific information. Such data structures include the interference graph, list of live ranges, control flow graph, and register set information. Now, the compiler writer can immediately concentrate retargeting efforts on improving the allocator output. Because of length limitations, only an overview of our strategy is presented here. Additional details on how the different data structures and accessing routines interact could be presented.

```
intGraph = buildIntGraph( lrangeList );
do {
   rebuild = 0;
   do {
    /* Add unconstrained nodes to coloring stack. */
      if ((nptr = getUnconstrained(intGraph)) != NULL) {
         sptr = deleteNode( nptr, intGraph );
         push( sptr, coloringStack );
      }

    /* Mark lowest-cost constrainted node for spilling */
      else if ( (nptr = getLowCost(intGraph)) != NULL) {
         rebuild = 1;
         sptr = deleteNode( nptr, intGraph );
         markSpilled( sptr );
         push( sptr, coloringStack );
      }
   }
while (!emptyGraph( intGraph ));

   /* Rebuild the Graph if a node was spilled. */
if (rebuild) {
      while (!emptyStack(coloringStack) {
         sptr = pop( coloringStack );
/* spilled nodes are not added back to intGraph. */
         if (spilled(sptr))
            addSpillCode(sptr);
else
          insertNode(sptr, intGraph)
      }
   }
}
while (rebuild)

do {
   sptr = pop( coloringStack );
   assignColor( sptr, intGraph );
   insertNode( sptr, intGraph );
}
while (!emptyStack( coloringStack ));
```

Figure 3: Example Register Coloring Implementation

Even with the interface, there are still difficult issues that must be addressed by the compiler writer. For example, coloring algorithms expect accessing routines to a data structure containing detailed register set information. This information will vary from one machine to another; therefore, the compiler writer must produce this information for each new target. In addition, some CISC instructions have unusual register requirements. This may cause additional spills if the requirements cannot be encoded in a form usable by the coloring routine.

The interface was designed in an effort to combine automatic code generators (such as Twig [AG89] and BURG [FHP92]) with register coloring. Our approach, termed Generic Register Allocation System (GRAS) [Bry93], enhances the automatic code generator to allow the compiler writer to provide general register set information and instruction specific register information. The general register set information is provided using a Register Description File. The automatic code generator tabulates the information in this file so that it can be used by the accessing routines during coloring. The instruction specific register information is described in the associated instruction templates of the automatic code generator. A pre-allocation phase extracts the register information from the templates and embeds it into the live ranges so that it is available for the coloring routine. The specific register information helps guide the coloring routine in dealing with temporary values and mapping the correct register sets to the instructions.

Future efforts will combine different allocators with GRAS code generators. This will test the flexibility of our interface with different architectures. In addition, we plan to focus on some of the RISC related issues addressed by the Marion System [BHE91].

References

[AG89] A. V. Aho and S. W. K. Ganapathi, M. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. on Programming Languages and Systems*, 11(4):491–516, October 1989.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[BGG$^+$89] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *SIGPLAN Notices*, 26(6):229–240, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[Bry93] Kelvin S. Bryant. *A Generic Appoach to Integrating Automatic Code Generation and Register Allocation.* PhD thesis, North Carolina State Univ., September 1993.

[CAC$^+$81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CH84] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[FHP92] C.W. Fraser, R.R. Henry, and T.A. Proebsting. Burg - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[MB81] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. Technical Report CSE-8104, Southern Methodist University, Dallas, TX, July 1981.