Graph-Coloring Register Allocation for Irregular Architectures

Michael D. Smith and Glenn Holloway
Harvard University
{smith,holloway}@eecs.harvard.edu

Abstract

The graph-coloring metaphor leads to elegant algorithms for register allocation that have been shown to be quite effective for regular architectures with plenty of registers. Published attempts to make these algorithms applicable to architectures that are irregular in their use of registers have yielded several incompatible extensions that handle only a small subset of the irregularities seen in modern architectures. We propose an approach that is able to extend graph-coloring allocation to a broad class of irregular architectures, and we show that we can do this without destroying the simplicity of the basic algorithm. We have implemented our approach in the Machine-SUIF compiler, and we discuss our initial experiences with it. In particular, we describe how the irregularities of the x86 instruction set architecture are handled in Machine-SUIF's graph-coloring register allocator.

1 Introduction

In the context of global register allocation, graph coloring is a successful and popular technique. The graph-coloring approach developed out of research on RISC architectures [4,5], and thus the vast majority of the literature on this topic assumes that the target processor's instruction set is orthogonal and its register set regular. In this paper, we consider architectures where the problem of graph-coloring register allocation is complicated by non-orthogonal instruction sets and by register files with overlapping registers of several sizes. We refer to these sorts of architectures as *irregular* architectures.

Irregularities appear in many existing microprocessors. *Register pairing* is a commonly cited example of an irregularity that complicates register allocation. The 32-bit instruction set architecture (ISA) of the HP PA-RISC processors and the Sun SPARC processors use register pairing to create double-precision floating-point (FP) registers out of two single-precision FP registers. The Motorola 68K ISA and its ColdFire descendants exhibit a different kind of architectural irregularity. It defines two distinct banks of integer registers: address registers and data registers. In certain operand locations, the register allocator is free to use either an address or data register; in other operand locations, the instruction set restricts allocation to only one of the two banks. As we illustrate later, the Intel x86 ISA exhibits both of these kinds of irregularities.

The literature on graph-coloring register allocation is not entirely devoid of work on irregular architectures. In particular, Nickerson [9] and Briggs [2,3] both describe approaches that adapt graph-coloring-based allocators for regular architectures to a limited class of irregular architectures. These authors realized that the kinds of irregularities described above directly affect a graph-coloring allocator's interpretation of its interference graph. (We expand on this point later in the paper.) To maintain the important invariants of the graph-coloring approach, both authors chose to modify the interference graph by adding extra edges or nodes. Though Nickerson and Briggs show that interference-graph modification can be made to work with a limited class of irregularities, their work also shows that this kind of approach does not present a general solution for dealing with irregular architectures.

In contrast, we present an approach that does not require any modification to the interference graph and is able to handle a very wide range of irregular architectures. We are able to handle this wide range because our approach maintains separate data structures for summarizing the program and architectural constraints pertinent to the problem of register allocation. We find that we can implement this approach in an existing graph-coloring-based allocator by making only a few minor modifications to the source code. These changes essentially comprise the replacement of a couple of constants and architecture-independent functions with architecture-specific functions.

We begin in Section 2 with a quick review of the key phases involved in register allocation by graph coloring and continue in Section 3 with a discussion of those aspects of the algorithm affected by irregular architectures. Section 4 introduces the concept of a weighted interference graph (WIG), and it describes how we can use a WIG to hide the unpleasant architectural constraints of irregular architectures without breaking the graph-coloring abstraction. Section 5 presents the details of our data structures and enumerates the small number of modifications we make to the basic graph-coloring algorithm. It also describes an actual implementation of our approach for the x86 ISA. We modified the register allocator distributed with Machine SUIF [8], which is based upon the work of George and Appel [6], to allocate for the x86 ISA using our new approach. In Section 6, we relate our experiences with this implementation, and we describe our on-going work exploring the effects of modeling precision. Finally, Section 7 discusses related work, and Section 8 offers our conclusions.

2 Allocation by Graph Coloring

Figure 1 decomposes the graph-coloring register allocator proposed by Briggs [3] into its constituent phases and illustrates the algorithmic flow between these phases. We focus our discussion on optimistic allocators because, as Briggs argues, the optimistic coloring heuristic produces better allocations than the pessimistic heuristic used by Chaitin [5]. Furthermore, the truth of this statement is independent of the issue of the regularity in the machine architecture.

The *renumber* phase makes an initial scan of code segment and identifies the live ranges. During this phase, we also classify each live range by the set of hardware resources that would satisfy its allocation needs. Briggs [3] introduces the concept of register classes for this purpose. A *register class* is simply a set of hardware registers. For each class, there is a distinct set of operations

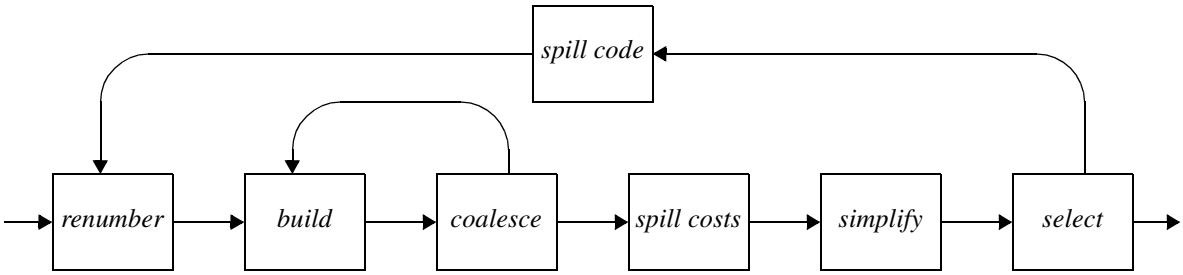


Figure 1. Briggs's Optimistic Allocator [3].

that can act on any member of the class, and these operations are what define the class. For example, the integer registers in the Compaq Alpha ISA define a class, since any Alpha integer operation can use any integer register as an operand. The Alpha FP registers define a separate class, and this class uses a set of register resources completely distinct from the integer class. We revisit the concept of classes when we discuss the problems involved in supporting irregular architectures.

The *build* phase constructs the interference graph. Each node in the interference graph represents a live range that is a candidate for register allocation. An edge connects two nodes (register candidates) if the lifetimes of the two candidates overlap at any point in the program's execution and those nodes compete for the same register resources.

The *coalesce* phase uses the interference graph to identify and remove unnecessary move instructions. The interference graph is rebuilt after successful coalescing. The following phase computes a spill cost for every candidate remaining in the interference graph.

The *simplify* and *select* phases cooperate to color the interference graph or to mark nodes for spilling in case no coloring can be found. If we are able to color the graph, the allocator has found a feasible allocation. Otherwise, we spill the set of marked candidates and start the process again.

3 Effects of Irregularity

The irregular architectures mentioned in Section 1 complicate a graph-coloring register allocator in several, subtle ways. This section discusses each complication in turn.

3.1 Effect on colorability

The interference graph and its interpretation plays a pivotal role in a graph-coloring register allocator. Regularity in an ISA allows *simplify* to use the degree of a node as an accurate indicator of its *colorability*: A node with less than k neighbors is trivial to color given k colors. Irregular architectures affect the interference graph so that a node's degree being less than k is no longer a sufficient condition for determining that node's colorability. A challenge of graph-coloring register allocation on irregular architectures is to find an accurate way to determine the colorability of each register candidate in the interference graph. A complete solution to this challenge must address both of the following two characteristics of irregular architectures.

First, as Nickerson [9] states, register candidates requiring multiple registers destroy the equivalence between "the number of interference relations (edges) in the interference graph and the number of coloring constraints they imply." In other words, a multi-register neighbor of a node n implies more than one coloring constraint on n, and thus the degree of n no longer directly reflects its colorability. Register pairs in the HPPA and SPARC floating-point register banks and the overlapping general-purpose registers in x86 (e.g., AL and AH are two allocable pieces of EAX) are examples of such a complication.

A second complication arises from non-orthogonal instruction sets that allow a set S of register candidates in one operand location and a different set T in another operand location, where $S \neq T$ and $S \cap T \neq \emptyset$. As mentioned earlier, such an irregularity appears in the 68K ISA with its separate but similar address and data registers. The x86 instruction set also exhibits this kind of irregularity in that an x86 register allocator can use the registers EDI and ESI for many but not necessarily all of a program's general-purpose register candidates. This kind of irregularity affects the coloring problem by changing the value of k for a subset of the nodes.

3.2 Effect on register classes

Section 2 introduced the concept of register classes. A graph-coloring allocator uses the class of a candidate to know what register resources are available for use by this candidate and to know when this and another candidate with an overlapping live range are vying for the same resources. The issue here is that irregular architectures require a richer set of register classes than regular architectures. The key to understanding this richness is to remember that classes result from an operational partitioning of the hardware registers. Classes are often confused with what we will call *register banks*, which are a result of an architectural partitioning of the registers resources. For example, the 68K ISA contains three register banks: the address, data, and FP registers. We must define not only a class for each register bank but also a class for the register candidates that can be allocated to either the address or data bank. Such a candidate occurs because some 68K operations (e.g., ADD) can write to either an address or data register. As slightly different example, consider the FP register bank in the SPARC ISA. Here, we define three register classes that each use only the single FP register bank (i.e., a class for single-precision FP operands, one for double-precision, and one for quad-precision).

We must deal with the rich variety of register classes in irregular architectures directly. To see this, recall that register allocation by

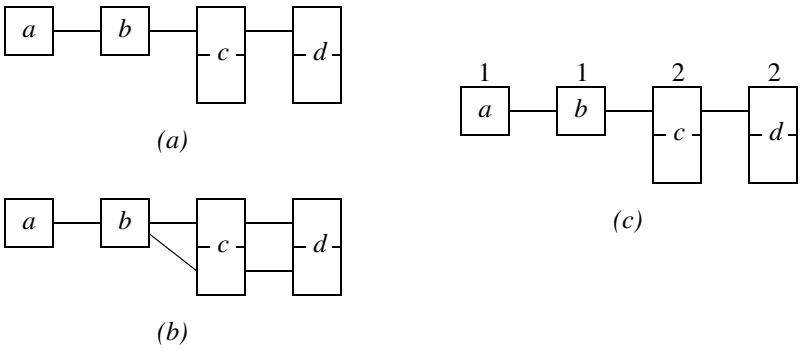


Figure 2. An example with 4 interfering live ranges. The interference graph in (a) does not correctly model the effects of the double-precision candidates. The interference graph in (b) is Briggs's solution, while the WIG in (c) is our solution. For (c), the numbers at the top of each node indicate the weight of the node. This example was taken from Briggs [3].

graph coloring is done on a bank basis: We consider all of the candidates for a register bank in single allocation pass in order to achieve the best allocation possible for the candidates in this bank. Since the register resources in a single bank may be available to the candidates from several different classes, any proposed solution must be able to deal with an interference graph containing nodes of different classes. Furthermore, as in the 68K example above, a class may include register resources from multiple different banks, and thus a complete solution must be able to allocate multiple banks simultaneously from a single interference graph.

4 Weighting the Interference Graph

Our solution to the complications enumerated in the previous section is to weight the nodes in the interference graph, resulting in a data structure we call a WIG. Sections 4.1 and 4.2 gently introduce the details and interpretation of this data structure by focusing initially on the problem of register pairs. We borrow two examples from Briggs's thesis [3] and present the WIGs that are equivalent to his interference graphs. However, as we point out at the end of each section, our approach is not limited to register pairs and in fact works for multi-registers of all sizes. Section 4.3 completes the overview of our solution by addressing the issue of coloring with nodes whose classes contain resources in multiple banks.

4.1 Aligned multi-registers

Consider the interference graph in Figure 2a. It contains two single-precision register candidates (nodes *a* and *b*) and two double-precision register candidates (nodes *c* and *d*). Let us assume $k = 4$ (i.e., we have four single-precision registers) and that the double-precision registers consume an aligned and adjacent pair of single-precision registers. Since no node in this interference graph has a degree larger than 2, a graph-coloring allocator would incorrectly assume that all nodes are trivially colorable. This is not true since the coloring of *c* could be blocked an appropriate coloring of *a*, *b*, and *d*.

Briggs [3] presents an approach that addresses the problem of register pairs. His solution is to modify the interference graph so that the degree of a node in his interference graph always reflects its colorability, independent of whether the node is a register candidate that requires a single register or a register pair. Figure 2b shows the interference multigraph proposed by Briggs to solve the problem with Figure 2a. His graph correctly models the colorability of each node: a double-precision candidate consumes two colors, and a single-precision candidate can block any neighboring double-precision candidate from two colors (due to alignment restrictions).

Figure 2c shows our WIG for this same example. The first item of note is that we do not add or remove edges from the interference graph. Two register candidates are never connected by more than a single edge in the WIG, and thus the existence of an edge between two nodes simply indicates that the lifetimes of the register candidates represented by those nodes overlap. In other words, the edges in a WIG summarize only the program constraints on register allocation.

The second item of note in Figure 2c is that we have associated a weight *w* with each node *n* in the graph. For this simple example, single-precision candidates get a $w = 1$ and double-precision a $w = 2$. (Below, we explain exactly how weights are chosen.) To determine the colorability of a node *n* in the WIG then, we evaluate the following equation:

$$\left(\sum_{j \in adjacent(n)} \left\lceil \frac{w_j}{w_n} \right\rceil \cdot w_n \right) < k \qquad \text{(E.1)}$$

The function *adjacent* takes a node *n* and returns a set corresponding to the neighbors of *n*. We organize this calculation so that it is not much more expensive than the calculation performed in today's graph-coloring allocators to determine the degree of *n*.[1]

The reader can verify that the left-hand side of Equation (E.1) when evaluated for each node in Figure 2c produces the same value as the degree of that node in Figure 2b. As a result, a graph-coloring allocator could use the WIG and Equation (E.1) in place of Briggs's multigraph and the equation $degree(n) < k$. We can

1. At the start of *simplify*, we calculate and store the left-hand side of Equation (E.1) for each node in the interference graph. As *simplify* removes a node *n*, it decrements each neighbor of *n* by the appropriate weighted factor (in contrast to decrementing by 1).

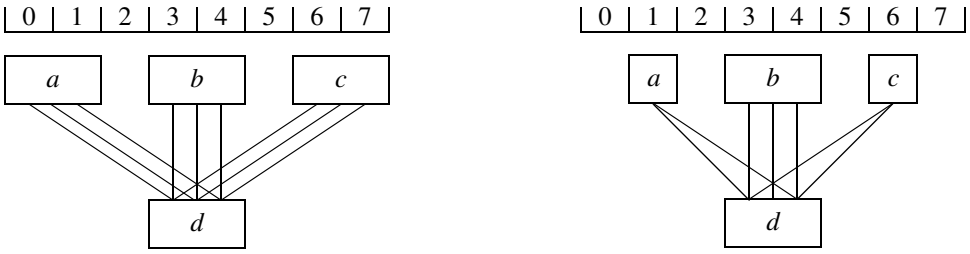


*Figure 3. Two example colorings of three register candidates (*a, b*, and* c*) that block the coloring of the remaining register pair* d *in an unaligned architecture with an 8-entry register file. The register candidates connected by edges have overlapping lifetimes. Briggs models the colorability constraints of* d *by inserting multiple edges between* d *and each of its neighboring nodes. The left-hand example was taken from Briggs [3].The right-hand example follows his methodology but under-constrains* d*.*

see why this approach works by realizing that, in this example, w_n equals the number of colors (registers) required by node (register candidate) n. Furthermore, the term $\lceil w_j / w_n \rceil$ represents the number of color groups (placements in the register file) for n that its neighbor j can exclude n from, and thus multiplying this term by w_n simply expresses this number in terms of colors.

Our solution is not limited to register pairs; it also works for the aligned multi-register problem described by Nickerson [9]. We simply have to choose an appropriate weight for each multi-register candidate (or more accurately multi-register class). In general, for an aligned architecture, we define the weight w_n of a multi-register candidate n of size s to be the smallest power of 2 greater than or equal to s. Recall that a multi-register of size s is defined to be aligned if it begins on a register whose index is 0 or is divisible by the smallest power of 2 greater than or equal to s. For example, the quad-aligned triples from the 8-register example in Nickerson [9] would have $w = 4$.

This approach is precise with respect to the colorability calculation for multi-register candidates and their neighbors whose sizes are all a power of 2. Unfortunately, this simple approach leads to an imprecise answer for colorings with multi-register candidates whose sizes are not a power of 2. In particular, the weight given to such a node may overly constrain the colorability calculation of its neighbors that are smaller than s. For example, when calculating the colorability of a node n of size 1 that has two quad-aligned triples as neighbors, *simplify* will assume that n is constrained (i.e., cannot be trivially colored) in a register file of size 8.

This issue of preciseness is not just a characteristic of aligned architectures, but as we will see, it also occurs with the other architectural irregularities considered below. To our benefit, we err on the right side (i.e., we never incorrectly answer yes to a colorability question), and we mitigate some of this conservatism through our use of an optimistic allocator. Still, it would be interesting to understand how much this conservatism costs us, since it is always possible to calculate the worst-case placement of a node n's neighbors and determine if that worst-case placement blocks the coloring of n. We explore one instance of this issue in Section 6 with the x86 ISA.

Finally, we will find it useful later in our discussion to talk in terms of a worst-case number of placements p for a register candidate of size s. We define p to be the minimum number of candidates of size s required in a worst-case coloring (or placement) to block the coloring of another candidate of size s. In an aligned architecture, $p = k / w$ where k is the total number of singleton registers in the architecture.

4.2 Unaligned multi-registers

Briggs [3] also presents a solution for handling unaligned register pairs. At first glance, the reader might assume that it would be easier to allocate register pairs in an architecture that allows unaligned pairs than in one that forces the pairs to be aligned. But as Briggs points out, greater freedom in register placement does not translate into greater colorability. In fact, the colorability of a multi-register in an unaligned architecture is worse than it is in an aligned architecture. The left-hand side of Figure 3, from Briggs [3], illustrates how to color three register pairs (a, b, and c) so that a fourth d cannot be colored, assuming a register file with eight individual registers. This observation led Briggs to an approach for unaligned pairs that requires more edges than his solution for aligned pairs.

We solve this problem more directly. The issue here is not that we need to adjust the weight associated with a register pair, but that p (the minimum number of pairs required to block the placement of another pair) has decreased. For an 8-register file, $p = 4$ in an aligned architecture, but $p = 3$ in an unaligned architecture. In general, the worst-case number of placements for a multi-register candidate of size s in an unaligned architecture is determined by the following equation:

$$p = \left\lceil \frac{k}{s + (s - 1)} \right\rceil \qquad \text{(E.2)}$$

In Equation (E.2), k is the number of total registers in the register file. This equation is based on the observation that a worst-case placement is achieved by repeating the pattern: consume s registers for the first multi-register of size s, skip $s - 1$ registers, consume s registers for the second multi-register, skip $s - 1$ registers, etc. An example of this pattern occurs on the left-hand side of Figure 3.

The consequence of Equation (E.2) on coloring is that it is no longer natural to think about a single k that the allocator can use for all register candidates. This is not a problem because we do not need to have a single coloring bound for all candidates, we simply need a unique bound for each class of register candidate. As illustrated by Equation (E.1), the colorability of a node is calculated in

units related to the characteristics of that node; the effects of the neighboring nodes are normalized to those units. Thus, we are free to specify a unique k for each class of register candidate without affecting the correctness of the *simplify* step. This should be easier to see if we rewrite Equation (E.1) in terms of p:

$$\left(\sum_{j \in adjacent(n)} \left\lceil \frac{w_j}{w_n} \right\rceil \right) < p \qquad \text{(E.3)}$$

Notice that the weight w of a register candidate of size s is always equal to s on an unaligned architecture. Notice also that we can derive Equation (E.3) from Equation (E.1) by substituting $\mathfrak{p} \cdot w_n$ for k and simplifying. Recall from our earlier discussion that this is just the equation relating p and k on an aligned architecture. We use Equation (E.3) instead of Equation (E.1) for all of our irregular architectures.

Returning to the example interference graphs in Figure 3, Briggs carefully adds edges so that *simplify* is not able to remove node d in the left-hand interference multigraph before removing one of the other nodes. This guarantees that d is colored by *select* before all of the other three candidates are colored. We achieve the same result without the extra edges.

The right-hand interference multigraph of Figure 3 is built following Briggs's methodology, and it illustrates the danger of an edge-based approach to handling irregular architectures. The node d cannot be colored given the placements of the other three nodes; however the d's degree is less than k (i.e., less than 8). In our approach, singleton's like a and c in the right-hand graph are considered one class with $w = 1$ and $p = 8$, while register pairs comprise a different class with $w = 2$ and $p = 3$.

Finally, it should be clear that Equations (E.2) and (E.3) can handle unaligned multi-registers of any size. As we discussed for aligned registers, these simple equations are sometimes overly conservative. Readers can prove for themselves that the answer given by Equations (E.2) and (E.3) is precise when all of the neighbors of a node of size s are also of size s or greater, and the equations are overly conservative otherwise.

4.3 Multi-bank classes

This section is concerned with register candidates that belong to a class in which they can satisfy their register requirements from more than one register bank. The class comprising the address and data registers in the 68K is an example of such a multi-bank class. We already have all of the analytical tools that we need to handle such candidates, and thus we simply show how we can apply these tools and explain where our results are overly conservative.

For purposes of discussion, let us define C_a to be a class containing only the 68K address registers, C_d to be a class containing only the 68K data registers, and C_m to be a class containing both address and data registers. We assign a weight $w = 1$ to each class, since the 68K does not support multi-registers in its address and data banks. The classes do differ in their p values however. The p for C_m is equal to the sum of the p for C_a and that for C_d, and because $w = 1$, the p for C_a and that for C_d is simply equal to the number of registers in each register bank.

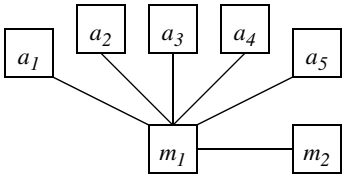


Figure 4. Example WIG involving multi-bank candidates. Assume that we have two single-bank classes C_a *and* C_d *with* p *equal to 4 and 2 respectively. We also have a multi-bank class* C_m *that is a union of the register resources of* C_a *and* C_d*, i.e.* $p = 6$. *The letter of each candidate indicates its class. Equation (E.3) produces an overly conservative answer for* m_1. *Even though* m_1 *has 6 neighbors, no coloring of these neighbors can exhaust all of the available registers in* C_d.

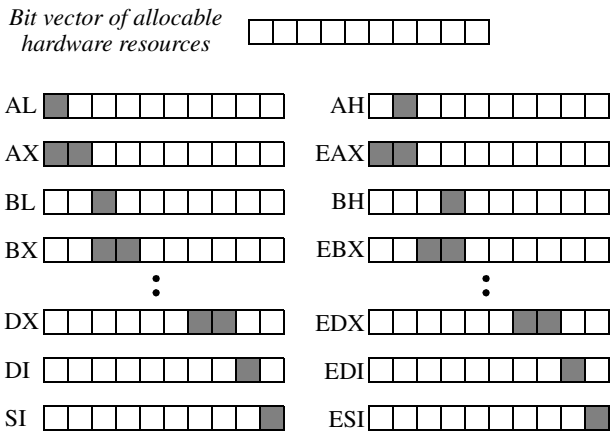
As in Section 4.2, we apply Equation (E.3) when determining a node's colorability, and we sometimes get an overly conservative answer. For WIGs with register candidates of multi-bank classes (multi-bank candidates), Equation (E.3) always provides a precise answer for any node n that is not a candidate of a multi-bank class, even if its neighbors are. Imprecision occurs only when n is a multi-bank candidate and its neighbors are a mix of multi-bank and single-bank candidates. Figure 4 illustrates such a case.

The imprecision demonstrated in Figure 4 results from the fact that the worst-case coloring of the neighbors of m_1 must simultaneously block a coloring of m_1 in C_a and C_d. This conjunctive condition was turned into a disjunctive condition when we summed up the individual p values of C_a and C_d to produce the p for the multi-bank class C_m. Section 6 explores the question of how much this particular kind of imprecision costs in the x86 ISA, and it suggests a slight modification to the *simplify* phase that invokes more precise calculation only when *simplify* is blocked from proceeding using the simpler one.

5 Implementation Details

The previous sections have introduced the main ideas of our approach to graph-coloring allocation on irregular architectures and compared it against existing approaches. In this section, we present the full details of our data structures and describe exactly how these structures are used. Many of the structures we describe already appear in one form or another in a basic graph-coloring allocator for regular architectures.

To aid in our presentation, we will use the general-purpose register file in the x86 ISA as our allocation target. Having a specific target will help to make the discussion concrete by allowing us to describe the contents of each data structure with respect to this target. The x86 is an interesting target because it is a multi-register architecture (e.g., AL and AH are individually allocable pieces of AX) that also has register candidates in a multi-bank class (e.g., register candidates that are used as both a 32-bit and an 8-bit value must be allocated to one of the EAX, EBX, ECX, or EDX registers on the x86, while pure 32-bit candidates can choose among the EAX, EBX, ECX, EDX, ESI, or EDI registers).



(a) Map from register names to hardware resources.

	resource mask	*w*	*p*
Class C_m		2	6
Class C_a		2	4
Class C_b		1	8
Class C_i		2	2

(b) Map from register classes to hardware details.

Figure 5. An allocator's view of the general-purpose register resources in the x86. Each square represents an allocable register resource, and the shaded squares indicate which resources are consumed by a particular register name or are part of a particular register class. We also list the weight w *and worst-case placement value* p *for each class.*

At the heart of every graph-coloring allocator is a color array that represents the individually allocable register resources in the target machine. Our allocator is no different. On the x86, our color array for the general-purpose register file has 10 elements, as shown by the bit vector at the top of Figure 5.

We use this bit vector in two ways. First, it provides a mapping between the target-specific register names and the hardware resources that these names imply, as shown in Figure 5a. Notice that the register AX consumes the two individual resources used by AL and AH, and thus we correctly capture the overlapping nature of these resources. EAX and AX map to the same hardware resources because you cannot name the upper half of EAX.

Second, we use this bit vector as a component of the register class data structure. The register class data structure stores four pieces of architectural information pertinent to register allocation, three of which are shown in Figure 5b. The first of these is the resource mask that indicates which hardware resources are available to candidates of the class. Two register candidates interfere if their live ranges overlap and the intersection of the resource masks of their classes is non-zero (i.e., they compete for the same register resources).

Briggs in his thesis section on multiple register classes describes a small bit vector that performs a similar function [3]. Our approach differs slightly from Briggs in that two register classes may have the same resource mask; in his approach, the bit vectors uniquely identify the register class. Figure 5b shows that class C_a and C_b have the same resource mask but different w and p values. The class C_a represents the 16- and 32-bit variants of the x86 A, B, C, and D registers; the class C_b represents the 8-bit registers AL, AH, BL, BH, CL, CH, DL, and DH. Notice that the union of the hardware resources of these two register sets does indeed yield the same resource mask.

The next two architectural details of the register class data structure were discussed in detail in Section 4: each register class is given a weight w and a worst-case number of placements p. It is these values that differentiate classes C_a and C_b. We use these values, as discussed earlier, during *simplify*. Careful examination of the code in a graph-coloring register allocator shows that we always use k and a node's degree together, and thus we can replace the code comparing these values with code that looks up w and p in the node's register class and evaluates Equation (E.3).

The last piece of the register class data structure is a list of the register names available to the register candidates of the class. The list for each x86 register class is as follows:

C_i : {DI, EDI, SI, ESI}

C_b : {AL, AH, BL, BH, CL, CH, DL, DH}

C_a : {AX, EAX, BX, EBX, CX, ECX, DX, EDX}

C_m : {AX, EAX, BX, EBX, CX, ECX, DX, EDX, DI, EDI, SI, ESI}

We use this list when allocating a "color" to a stacked register candidate during *select*. In particular, each candidate is given a register name when colored. Through the map from register names to hardware resources, we can know what resources a colored neighbor uses. Thus, when *select* pops the next candidate n off the stack, it visits each neighbor of n in the interference graph and builds a list of used (unavailable) colors. It can then intersect a bit vector of these unavailable colors against the hardware resources used by each one of the register names in n's class. If the intersection is the empty bit vector, that register name is an available "color" for n. (In the case of the registers AX and EAX, the actual register name returned is the one whose bit size matches the bit size of the register candidate.)

Hardware registers existing in the code before register allocation (e.g., as part of a parameter passing convention) just appear as precolored live ranges in the interference graph. We assign a class c to the WIG node representing a hardware register r by first finding the set of classes that contain r in their name list and then choosing the class in that set with the smallest resource mask. For example, a node representing the hardware register EAX would have class C_a. We choose this class instead of C_m because the node representing EAX does not compete for resources with a register candidate of class C_i.

Note that we never insert a node representing a hardware register into the interference graph for the sole purpose of excluding another live range from that hardware register. Our WIG simply summarizes the program constraints on register allocation, and

Benchmark	Allocator slowdown	Spill code reduction		Reduction in coloring iterations
		Loads	Stores	
espresso	15.7%	3.5%	0.9%	1.2%
m88ksim	7.8%	1.3%	0.8%	0%
perl	11.8%	6.7%	2.7%	1.2%
fpppp	65.3%	0%	0%	0%

Table 1: Experimental results for our investigation into the use of extra precision. This table summarizes the costs (allocator slowdown) and benefits (reductions in spill code and coloring iterations) of a more precise calculation of a node's colorability. These results were obtained by compiling the benchmarks under Machine SUIF.

each node represents one live range. (Actually, a pre-colored node may represent several live ranges.)

The last item in our discussion of the implementation concerns one of the first data structures used by the register allocator. This structure is the map from instruction operands to register classes. We use this map to find the class of each live range. Initially, a live range is assigned the class required by its first reference point. At each later reference point in the code, we check to see if the class C_{rp} of the reference point equals the current class C_{lv} of the live range. If it does, no further action is required. If not, we intersect the resource mask of C_{rp} with the resource mask of C_{lv}. The result of this intersection is the resource mask of either C_{rp}, C_{lv}, or some class that is more constrained than either C_{rp} or C_{lv}. Notice that the resource masks must define a partial order on the set of register classes for the initial scan of the code segment during *renumber* to correctly assign a register class for each register candidate.

6 Buying Extra Precision

We have implemented our support for multi-register allocation and multi-bank classes in the Machine-SUIF compiler from Harvard University [8]. Machine SUIF is distributed with a coloring register allocator that faithfully implements the iterated register coalescing algorithm of George and Appel [6]. This algorithm integrates coalescing with the coloring heuristic, and in order not to let coalescing create more over-constrained nodes than existed anyway, it imposes extra prerequisites for combining two move-related nodes. As in Briggs-Chaitin style, these tests always compare the degree of a node with the coloring bound k, and thus the same well-localized kinds of adjustments described in the preceding section suffice to transform the George and Appel algorithm into one that uses a WIG.

As explained in Section 4, situations exist where our straightforward use of the WIG yields an overly conservative answer to the question of a node's colorability. Figure 4 illustrates the kind of inaccuracy that can occur in our model of the x86. In fact, the classes C_m, C_a, and C_d in Figure 4 correspond directly to the classes C_m, C_a, and C_i in Figure 5b.

To explore how much better we might be able to do with a more precise calculation of colorability, we inserted a hook in the iterated coalescing algorithm at the point where it cannot find any more graph simplifications using its colorability heuristic and just before it is about to make a concession in order to catalyze forward progress. By concession, we mean that it is either going to give up on the possibility of coalescing two nodes, or it is going to mark a node for possible spilling and remove it from the graph. We can use this hook to insert a routine that scans the interference graph and identifies high-degree nodes that belong to a multi-bank class. For each such node n, we invoke a routine that performs a more precise calculation to determine whether a worst-case coloring of those neighbors still thwarts the coloring of n. If we find one or more nodes that are colorable under this more precise calculation, we remove them from the graph, stack them, and return to *simplify*. Otherwise, we proceed as if the hook were not there.

This hook allows us to measure how much benefit the extra precision yields in coloring cycles avoided and spill instructions removed. It also allows us to do so without warping the underlying coloring algorithm. In addition, we also measure how much the work done in the hook costs in compile time.

Table 1 displays the results for a smattering of benchmarks chosen from the SPEC92 and SPEC95 suites. It shows the fractional slowdown of the register allocation pass when extra precision is used, and it shows the reduction in spill overhead that results. By and large, the cost of the extra precision in terms of increased compile time is relatively large, and the reduction in spill code that results is relatively modest. There may well be more sophisticated ways of increasing precision without incurring so much compile-time cost, but the gains to be reaped from going beyond our straightforward WIG-based approach do not appear to be large for the x86.

7 Related Work

There exists a large body of work on global register allocation by graph coloring, starting with the work by Chaitin [4,5]. Of this work, only Briggs [2,3] and Nickerson [9] deal directly with irregular architectures, and both of these authors modify the interference graph to encode the extra constraints in irregular architectures. The unstated goal of these modifications is to maintain the traditional form of the test for colorability, i.e. you simply compare a node's degree against a known coloring bound k.

Briggs [2,3] focuses on architectures with aligned or unaligned register pairs. He presents two heuristics for adding edges to the interference graph that he claims correctly models the extra coloring constraints imposed by register pairs. Though it is intuitively appealing, this is a tricky approach that increases the complexity of

the interference graph and is difficult to get right, as illustrated by our example in Figure 3.

Nickerson [9] presents an approach that handles register candidates requiring two or more adjacent, aligned registers. Like Briggs, Nickerson's solution modifies the interference graph, but in a different manner. A node exists in his interference graph for each individual register of a multi-register candidate; he refers to candidates as *clusters* and the individual registers of a cluster as *cluster-mates*. He then presents an approach for identifying and removing "implicit" interference edges, edges whose interference relation is already indicated by another edge between related cluster-mates. Though this approach can use a node's degree to accurately indicate the coloring constraints on some clusters, Nickerson points out that it does not work in all cases. Finally, as we have done, he allows the coloring bound k to vary for clusters of different sizes.

Sander [10] describes a modification made to the IMPACT graph-coloring allocator to support register allocation in the x86. The modification marks live ranges that include byte operations as ones that cannot be colored using the x86 registers EDI and ESI. The idea here is simply another formulation of the register classes idea; however, Sander does not appear to change k for these candidates during *simplify*.

Register allocation on irregular architectures has also been done using approaches other than graph coloring. For example, Kong and Wilken [7] use a 0-1 integer linear programming formulation to produce an optimal register allocator for the x86 ISA. They assign a binary decision variable to each minute decision required for register assignment and spilling at each program point. They describe the constraints on register use using linear inequalities over these decision variables. This approach is able to handle multi-register and multi-bank architectures.

Finally, our use of separate data structures for summarizing the program constraints and the architectural constraints on register allocation is similar in spirit to what is used in instruction scheduling algorithms for pipelined and multi-issue machines. For example, a basic-block scheduler may employ a data dependence graph to summarize the program dependences between instructions and a finite-state automaton [1] to model the machine's pipeline constraints. This separation allows a retargetable compiler to apply the same basic list-scheduling algorithm to a wide range of machine models. We have shown how to apply the same basic graph-coloring allocator to a wide range of irregular architectures.

8 Conclusions

We have presented a practical approach to graph-coloring register allocation for irregular architectures. We have shown that our approach is applicable to a wide range of irregular architectures, and that it fits neatly into the structure of existing graph-coloring allocators. We demonstrated the truth of this statement by adapting the published algorithm for iterated register coalescing [6]. We have implemented this adaptation in the Machine-SUIF compiler, and we use it as our everyday allocator for the x86 target.

The key difference between our work and the previous work is that we allow ourselves to consider an alternate form of the test for colorability. In a regular architecture, a node in the interference graph is trivially colorable if its degree is less than the coloring bound k. As we discussed, architectural irregularities such as interference nodes that require a pair of registers make it so that this simple test is no longer an accurate indicator of a node's colorability. The previous work on graph-coloring allocators for irregular architectures changes the structure of the interference graph so that they can keep using this simple test for colorability. We, on the other hand, keep the simple structure of the interference graph and adapt the test for colorability so that it considers architectural requirements of the node and its neighbors. We feel that this is cleaner, safer, and leads to a more generally applicable solution.

9 Acknowledgments

TBW.

10 References

[1] V. Bala and N. Rubin, "Efficient Instruction Scheduling Using Finite State Automata," *Proc. 28th Annual Intl. Symp. on Microarchitecture*, pp. 46–56, November 1995.

[2] P. Briggs, K. Cooper, and L. Torczon, "Coloring Register Pairs," *ACM Letters on Programming Languages and Systems*, 1(1):3–13, March 1992.

[3] P. Briggs. "Register Allocation via Graph Coloring," Technical Report CRPC-TR92218, Center for Research on Parallel Computation, Rice University, Houston, TX, April 1992.

[4] G. Chaitin, et al. "Register Allocation Via Coloring," *Computer Languages*, 6(1):47–57, 1981.

[5] G. Chaitin. "Register Allocation and Spilling via Graph Coloring," *Proc. SIGPLAN '82 Symp. on Compiler Construction*, pp. 98–105, 1982.

[6] L. George and A. Appel, "Iterated Register Coalescing," *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[7] T. Kong and K. Wilken. "Precise Register Allocation for Irregular Architectures," *Proc. 31st ACM/IEEE Intl. Symp. on Microarchitecture*, pp. 297–307, December 1998.

[8] Machine SUIF. See http://www.eecs.harvard.edu/machsuif.

[9] B. Nickerson, "Graph Coloring Register Allocation for Processors with Multi-Register Operands," *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp. 40–52, June 1990.

[10] B. Sander. "Performance Optimization and Evaluation for the IMPACT x86 Compiler," MS thesis, Dept. of Computer Science, U. of Illinois, Urbana, IL, May 1995.