Reprinted from Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation

A Generalized Algorithm for Graph-Coloring Register Allocation

Michael D. Smith, Norman Ramsey, and Glenn Holloway
Division of Engineering and Applied Sciences
Harvard University
{smith, nr, holloway}@eecs.harvard.edu

Abstract

Graph-coloring register allocation is an elegant and extremely popular optimization for modern machines. But as currently formulated, it does not handle two characteristics commonly found in commercial architectures. First, a single register name may appear in multiple register classes, where a class is a set of register names that are interchangeable in a particular role. Second, multiple register names may be aliases for a single hardware register. We present a generalization of graph-coloring register allocation that handles these problematic characteristics while preserving the elegance and practicality of traditional graph coloring. Our generalization adapts easily to a new target machine, requiring only the sets of names in the register classes and a map of the register aliases. It also drops easily into a well-known graph-coloring allocator, is efficient at compile time, and produces high-quality code.

Categories and subject descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, optimization, retargetable compilers*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General terms

Algorithms, Languages

Keywords

Graph coloring, register allocation

1 Introduction

The reduction of register allocation to a graph-coloring problem is elegant, effective, and practical. It is warmly endorsed by modern textbooks [Muchnick 1997; Appel and Palsberg 2002; Cooper and Torczon 2003] and widely used in modern compilers. But two assumptions at the heart of the algorithm are invalid for most commercial instruction sets: registers are expected to be *interchangeable* and *independent*. Registers are interchangeable if they are equally suitable in any program context. Registers are independent if writing to one cannot change the value of another.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

This mismatch between the assumptions of the algorithm and realities of commercial architectures forces compiler writers to "augment" Chaitin's [1981] original formulation each time they implement it to handle the peculiarities of each target machine. In this paper, we generalize the graph-coloring approach in a way that addresses this mismatch. Using our generalization, a compiler writer can create a register-allocation pass that is as elegant and practical as the original formulation while also being trivial to target to real machines.

1.1 The need

Machines with completely interchangeable registers are unusual. Even very regular architectures partition registers according to function. For example, the Alpha architecture defines a set of registers for use in floating-point calculations, and these registers are not interchangeable with those used for operating on integers and addresses.

In response, compiler writers have developed the notion of *register classes*, where a class is a set of register names that are interchangeable in a particular role. When all of a target's register classes are disjoint, as they are in the Alpha architecture, a graph-coloring allocator can simply run a separate Chaitin-style allocation pass for each register class. But if a target has register classes that are not disjoint, or if optimizations force non-disjoint register classes, then in order to generate good code, a graph-coloring allocator needs to allocate the *register candidates* of multiple classes simultaneously. (A register candidate may be a source-language variable, a compiler-generated temporary, or a live range.)

Non-disjoint classes occur in machines where source-language data types do not clearly define the target's register classes. For example, in the Motorola 68000 family and its Coldfire descendants, the instruction for adding 32-bit integer data accepts either an address register or a data register as its source operand. But the instruction for multiplying integers only accepts data registers, not address registers, as operands. A similar irregularity occurs in the Itanium instruction set, which has add-immediate instructions that can write only a few of the general-purpose registers.

Non-disjoint classes can also be induced by optimizations performed at compile time. For example, to translate a target-independent, memory-to-memory move, a code generator for an Alpha needs at least one register candidate. If the only register classes in the Alpha are the disjoint integer and floating-point classes, the code generator will be forced to label the register candidate with one of those two classes. The compiler could produce better code, however, if it could postpone the decision of whether to use an integer or floating-point register until it knew the register pressure for each class at the point of the move. We can achieve this postponement by defining a register class that is the union of the integer and floating-point register classes and by having a register allocator that simultaneously allocates multiple non-disjoint register classes. Such a register allocator could also take advantage of instructions in the Alpha architecture that move values directly between the integer and floating-point register banks [Kessler, McLellan, and Webb 1998]. In particular, the allocator might take advantage of these inter-bank move instructions to use registers outside a candidate's class as fast spill space.

Machines that violate the register-independence assumption are also common. When an assignment to one architectural register name can affect the value of another, such register names are said to *alias*. A classic example is the combination of two single-precision floating-point registers to form one double-precision register [Briggs, Cooper, and Torczon 1992]. Early HP PA-RISC and Sun SPARC processors used this design. Some modern embedded and reconfigurable architectures carry it further. The ARM VFP10 unit, for example, has a floating-point register file that can be organized as

- 32 single-precision registers,
- 16 double-precision registers,
- 8 scalar registers and 6 vectors of 4 elements, or
- 8 scalar registers and 3 vectors of 8 elements.

The double-precision registers are typically *aligned* with respect to the single-precision registers, which means that only a pair that starts at an even-numbered single-precision register aliases with a double-precision register. Architectures that use unaligned pairing also exist. Of these, some allow for "wrap around", where the last single-precision register in the hardware register bank is paired with the first. For instance, the ARM VFP coprocessor implements floating-point pairs, quadruples and octuples, any of which may be unaligned and wrap around.

Aliasing is not limited to floating-point registers. The Intel $x86$ family has byte-sized integer registers that alias with 16-bit registers, which in turn alias with 32-bit registers.

In sum, interchangeability and independence are more the exception than the rule. To be able to use graph-coloring register allocation anyway, compiler writers continually invent new workarounds. Michael Matz's [2003] retrospective on what it took to implement a graph-coloring register allocator for GCC is a poignant example of the ad-hoc nature of such workarounds. Matz states clearly that it was difficult to build a multi-target allocator based on graph coloring because of machine features prevalent in the real world but not considered by the traditional formulation. The fact that graph coloring has become popular despite the need for these workarounds is a testament to its appeal.

1.2 Our solution

This paper generalizes the graph-coloring approach to register allocation in a way that eliminates the need for the kinds of workarounds currently employed for modern commercial architectures. Our generalization permits simultaneous allocation of multiple register classes, even when registers alias, while maintaining the efficiency and general structure of the original graph-coloring formulation.

We exploit the fact that compiler writers like to group a machine's registers into potentially overlapping classes. We define a class simply as a set of register names. Registers within a class must be interchangeable but need not be independent. Aliasing (nonindependence) is made explicit by a map $alias(r)$, which takes each register name r to the set of register names with which it aliases. By definition, $alias(r)$ includes r, and for notational convenience, we extend the $alias$ map to sets of registers: $alias(S) = \bigcup_{r \in S} alias(r)$. The alias map and the grouping of register names into classes are the *only* properties needed to target our allocator.

The remainder of the paper is organized as follows: Section 2 presents a brief overview of traditional graph-coloring allocation and identifies the beautiful aspects of the original formulation that we would like to maintain in a new formulation. Section 3 explains how we generalize the graph-coloring heuristic, while Section 4 describes an efficient implementation of that generalized heuristic. Section 5 then presents the highlights of a retargetable allocator that we built based on our generalization of a well-known graph-coloring allocator. Section 6 reports on the practicality of our implementation.

2 Identifying the beauty

A number of published algorithms rely on the traditional formulation of graph-coloring allocation, and we would like these algorithms to work seamlessly with our new generalization. To that end, our generalization preserves three elegant aspects of the traditional formulation: an intuitive interpretation of the interference graph, a simple criterion for computing when a node is trivially colorable, and incremental computation of that criterion as nodes enter and leave the interference graph. We put these aspects in context through a brief review of how a traditional allocator sets up a graph-coloring problem and searches for a solution.

The primary task of a register allocator is to find the most important register candidates and replace them with registers. In general, a program contains more register candidates than the hardware has available registers. To help identify register candidates that can share hardware registers, a graph-coloring allocator builds an interference graph.

To construct an interference graph, an allocator needs to know the register class of each candidate and at what points in the program each candidate is *live* (i.e., holds a value that may be used before it is overwritten). The candidate's class is chosen by intersecting the register-class requirements of all operand locations occupied by the candidate [Briggs 1992]. The set of live points is obtained by running live-variable analysis [Muchnick 1997].

With this information, the construction of an interference graph is straightforward. Each node represents a register candidate. An edge connects two nodes if the register classes of the candidates represented by the nodes alias and at any point in the program the candidates are simultaneously live. In other words, interference graph edges identify those candidates that cannot be allocated to registers that alias.

Here's where the coloring metaphor comes in. If the interference graph contains nodes of only a single register class and that class contains k independent and interchangeable registers, then finding a k-coloring of the interference graph solves the allocation problem. The color assigned to each candidate node maps uniquely to an available register, and nodes that are neighbors in the graph never receive the same assignment.

Although k-coloring is NP-complete, Chaitin developed a simple approach that does well in practice for interference graphs containing nodes of a single register class [Chaitin et al. 1981; Chaitin 1982]. Chaitin's approach is based on the observation that when node n has fewer than k neighbors, i.e., $degree_n < k$, it is *trivially colorable*. No matter how colors are assigned to its neighbor nodes, there will be a distinct color left for n. Chaitin's heuristic repeatedly simplifies the graph by removing and stacking trivially colorable nodes. Each removal lowers the degrees of neighbor nodes and may make additional nodes trivially colorable. If the process succeeds in stacking all of the nodes, then it is easy to assign colors by popping one node at a time, restoring its edges to former neighbors already popped, and picking a color not already assigned to one of them.

Of course, the graph-simplification phase may block with no trivially colorable nodes to remove and stack. In that case, Chaitin's method picks a node whose degree in the remaining graph is relatively high (so that its removal liberates as many subsequent nodes as possible) but whose spill cost is relatively low (so that the run-time impact of this spill is low). It removes and stacks that node, and then continues with the simplification phase. This brute-force simplification means that color assignment may later fail for some of the forcibly removed nodes.[1] In that case, the allocator inserts spill code for the occurrences of the corresponding register candidates and starts over on the modified program.

This method can be implemented quite efficiently. The degree of each node is cached and updated incrementally. When the allocator removes a node from the graph, it sets the node's cached degree to zero and decrements the cached degrees of the node's neighbors, but it leaves the edge representation in place for use later, when the node is reinstated in the graph and given a color. The color-assignment algorithm just ignores edges to neighbors whose cached degree is zero.

In summary, the traditional formulation of graph coloring register allocation is based upon an intuitive interpretation of the interference graph, a beautifully simple criterion for computing when a node is trivially colorable, and an incremental method for efficiently computing colorability even as nodes enter and leave the interference graph. Our goal is to provide an equally simple criterion for computing when a node is trivially colorable, even when registers are not completely interchangeable or independent. In addition, our generalization of the colorability criterion should not require changing the structure or interpretation of the interference graph. Finally, this generalization of the criterion must be amenable to an efficient implementation and specifically one that supports incremental computation.

3 Generalizing the Colorability Criterion

We now broaden the scope of the graph-coloring heuristic by developing and illustrating a drop-in replacement for the traditional colorability criterion. Section 3.1 begins with a new formulation of the colorability criterion that is easy to understand, is precise for all architectures, but is expensive to compute. Sections 3.2 and 3.3 present a safe and practical approximation of this formulation based on register classes. Section 3.4 presents an optimality property and shows that our approximation is exact for many commercial architectures.

3.1 Same heuristic, broader scope

Our goals are to allocate register candidates from different register classes simultaneously and allow these classes to contain registers that may alias. From Section 2, we know that two candidates n_1 and n_2 interfere if they are simultaneously live and if a coloring of these nodes may lead to aliasing. The possibility of aliasing may be stated formally as:

$$alias(class_{n_1}) \cap class_{n_2} \neq \emptyset \;\vee\; class_{n_1} \cap alias(class_{n_2}) \neq \emptyset$$

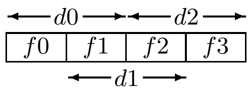
As always, the neighbors of a node n in the interference graph combine to constrain that node's colorability. But our view of colorability is complicated by two problems. Because there may not be a resource limit k that all of the nodes have in common, it no longer makes sense to look for a k-coloring of the interference graph. And the impact of a neighbor on the colorability of n may vary from neighbor to neighbor. Figure 1 presents a simple architecture and example interference graph that illustrate these two problems.

[1] On the other hand, sometimes a node that wasn't trivially colorable turns out to be colorable [Briggs, Cooper, and Torczon 1994].

Architectural definition:

$F = \{f0, f1, f2, f3\}$ $\qquad D = \{d0, d1, d2\}$

$alias(f0) = \{f0, d0\}$ $\qquad alias(d0) = \{d0, d1, f0, f1\}$
$alias(f1) = \{f1, d0, d1\}$ $\qquad alias(d1) = \{d0, d1, d2, f1, f2\}$
$alias(f2) = \{f2, d1, d2\}$ $\qquad alias(d2) = \{d1, d2, f2, f3\}$
$alias(f3) = \{f3, d2\}$



Example interference graph:

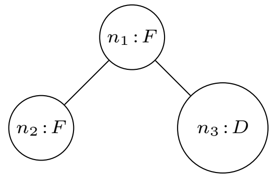


Figure 1. *An example architecture with two register classes, F and D. Class F contains four single-precision floating-point registers, which alias with the three unaligned double-precision registers listed in D. Because the register file does not permit a double-precision register to wrap around the end, the two classes have different resource limits: four single-precision registers create only three double-precision ones. In the interference graph, node n_3 is drawn as a larger circle because it places a greater constraint on the colorability of n_1 than does n_2.*

On the other hand, there are still only finitely many choices for a node n, namely the number of members in the class of n, that is, $|class_n|$. Therefore, to produce a new criterion for trivial colorability, we must find a measure of the constraints imposed on n by n's neighbors, and this measure must be comparable with the number of choices $|class_n|$. We call this measure the *squeeze*.

More precisely, $squeeze^*_n$ is the maximum number of names from $class_n$ that could be denied to n because of an assignment of registers to n's current neighbors. Node n is trivially colorable provided

$$squeeze^*_n < |class_n| \tag{1}$$

This inequality has the same form as the traditional criterion. If $squeeze^*_n$ can be computed efficiently, as is done for $degree_n$ in the traditional graph-coloring implementation, then we can use Equation 1 as a drop-in replacement for the traditional criterion.

So how should $squeeze^*_n$ be defined? Imagine a game in which we want to allocate a register for node n, and an adversary wants to stop us. The adversary gets to move first, by picking a *coloring* of the neighbors of n. This coloring assigns to each neighbor a single register chosen from that neighbor's register class. When it is our turn, we must choose for n a register that does not alias with any of the registers chosen by the adversary. The adversary's best move is to consider all possible colorings and choose one that eliminates as many choices for n as possible. In other words, if we write S for the set of registers in the coloring, the adversary should choose a coloring that maximizes $|class_n \cap alias(S)|$. Therefore, the maximum number of names from $class_n$ that could be denied to n because of an assignment of registers to n's neighbors is

$$squeeze^*_n = \max_{S \in \text{colorings of } n\text{'s neighbors}} |class_n \cap alias(S)| \tag{2}$$

If this ideal number is at least $|class_n|$, the adversary wins. But if $squeeze^*_n < |class_n|$, then node n is trivially colorable, and we win.

3.2 Decomposition by class

We obviously cannot afford to enumerate all colorings of n's neighbors every time we need to determine if it is trivially colorable. We instead define an approximation, $squeeze_n$, that is fast, safe, and good. By "fast," we mean that $squeeze_n$ can be computed quickly. By "safe," we mean that $squeeze^*_n \leq squeeze_n$ always, so if $squeeze_n < |class_n|$ then $squeeze^*_n < |class_n|$, and node n is trivially colorable. And by "good," we mean that $squeeze^*_n = squeeze_n$ for a great many real architectures. In this and the next subsection, we develop the key ideas behind our approximation $squeeze_n$. Section 3.4 gives the complete definition of $squeeze_n$.

The first key idea is to consider the impact of n's neighbors by class, rather than individually. To motivate this idea, let us consider the case in which all of n's neighbors have the same class C. Class C might be different from n's class, $class_n$, which we write N. The worst-case number of elements of N that can be blocked by an adversary's coloring of some number m of n's neighbors is a constant that depends only on the properties of the target architecture. We call this number the *worst-case displacement* of N by C using m nodes, $worst^m(N, C)$:

$$worst^m(N, C) = \max_{S \subseteq C \wedge |S| \leq m} |N \cap alias(S)| \tag{3}$$

Worst-case displacement doesn't depend on the interference graph being colored, which means that we can precompute it for all pairs of classes N and C and for all values of m up to $|C|$. The precomputed values can be stored in a lookup table for use during compilation. For instance, for the example classes in Figure 1, $worst^1(F, D) = 2$ since one double-precision register consumes two single-precision ones, $worst^1(D, F) = 2$ since one single-precision register can block two double-precision ones, $worst^1(D, D) = 3$ since register $d1$ aliases with all of the other double-precision registers, and also $worst^2(D, D) = 3$ since a coloring cannot consume more than the number of registers in n's class.

The worst-case displacement of N by C is an integral component of our approximation. Whenever node n has exactly m neighbors, all of class C, $worst^m(N, C) = squeeze^*_n$. Since m is the number of neighbors of class C, we write it as $degree_n(C)$.

When n has neighbors of more than one class, we estimate $squeeze^*_n$ by summing worst-case displacements over a set of classes $\mathbb{C}$:

$$\mathcal{W}_n(\mathbb{C}) = \sum_{C \in \mathbb{C}} worst^{degree_n(C)}(N, C) \tag{4}$$

If we sum over all classes $\mathbb{C}_{all}$, the sum $\mathcal{W}_n(\mathbb{C}_{all})$ is at least as great as $squeeze^*_n$, so it is a safe approximation. To understand why, think about the contradiction: if the ideal $squeeze^*_n$ could be greater than $\mathcal{W}_n(\mathbb{C}_{all})$, the contribution from at least one class of a coloring that achieves $squeeze^*_n$ would have to exceed $worst^{degree_n(C)}(N, C)$, contradicting the definition of $worst$. A formal proof of safety relies primarily on the triangle inequality $|A \cup B| \leq |A| + |B|$ and on the interchange rule $\max_S \sum_C f(S, C) \leq \sum_C \max_S f(S, C)$.

3.3 Saturation to avoid overcounting

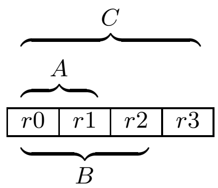
Although $\mathcal{W}_n(\mathbb{C}_{all})$ is safe, it is not always exact. For example, when two register classes use overlapping hardware resources, $\mathcal{W}_n(\mathbb{C}_{all})$ may count the overlapping resources twice. Figure 2 shows an example in which the overlapping resources are registers $r0$ and $r1$.

To improve on $\mathcal{W}_n(\mathbb{C}_{all})$, we observe that the worst the adversary can do with any group of neighbors is to use all the registers in those neighbors' classes. In Figure 2, the worst is bounded by

Architectural definition:

$$A = \{r0, r1\} \quad B = \{r0, r1, r2\} \quad C = \{r0, r1, r2, r3\}$$

$$\forall r \in A, B, C : alias(r) = \{r\}$$



Example interference graph:

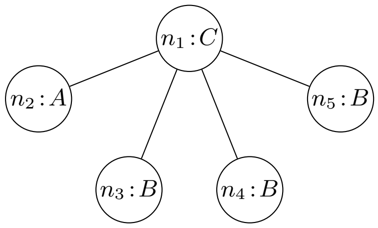


Figure 2. *This example illustrates overcounting of registers. The machine has three register classes A, B, and C. In the interference graph, n_1 has one neighbor of class A and three of class B. The sum $\mathcal{W}_n(\{A, B, C\})$ counts 1 for class A and 3 for class B, for a total of 4, but the actual squeeze imposed by classes A and B together is only 3. Node n is* saturated *with respect to classes A and B. Node n is in fact trivially colorable.*

the number of registers in the set of classes $\{A, B\}$. But the idea applies to *any* set of classes: given a set of classes $\mathbb{C}$, it follows from Equation 2 that the contribution to $squeeze^*_n$ from neighbors of those classes is bounded by the alias sets of those classes. We write this bound as $bound(N, \mathbb{C})$, defined by

$$bound(N, \mathbb{C}) = |N \cap (\bigcup_{C \in \mathbb{C}} alias(C))| \tag{5}$$

For any $\mathbb{C}$, if we consider terms in $\mathcal{W}_n(\mathbb{C}_{all})$ that depend on $\mathbb{C}$, we can replace the partial sum over those terms, $\mathcal{W}_n(\mathbb{C})$, by $\min(bound(N, \mathbb{C}), \mathcal{W}_n(\mathbb{C}))$, and we avoid overcounting the overlapping resources in $\mathbb{C}$. To avoid more overcounting, we can reorder and parenthesize partial sums in $\mathcal{W}_n(\mathbb{C}_{all})$, then cap each parenthesized partial sum by a bound described below. Section 3.4 explains how we structure this bounded sum to best approximate $squeeze^*_n$. Though there are many choices for ordering and parenthesization, Section 3.4 shows that there is only one sensible choice for any real machine. Thus, for the rest of this section, we focus on a fast method for evaluating the bounded sum.

Our method for evaluating the bounded sum is motivated by the observation that the parenthesized expression for $\mathcal{W}_n(\mathbb{C}_{all})$ can be viewed as a tree. We refer to this tree as a *class tree*. Each pair of parentheses in the expression corresponds to a vertex in the tree. Each vertex v has a set of register classes, written $classes(v)$. Each class in $classes(v)$ corresponds to a term in $\mathcal{W}_n(\mathbb{C}_{all})$ that is inside the parentheses for v but not in those of a child of v. The set of child vertices of v is written $children(v)$. Every vertex except the root has a parent vertex, written $parent(v)$. Parents correspond to immediately enclosing parentheses. Finally, each class C appears in exactly one term in $\mathcal{W}_n(\mathbb{C}_{all})$ and thus in one vertex in the tree, which we write $vertex(C)$.

Recall that this class tree is meant to help us avoid overestimating the squeeze on candidate node n by its neighbors. In the simple case, n becomes *saturated with respect to* a set of classes $\mathbb{C}$ when $\mathcal{W}_n(\mathbb{C})$ reaches $bound(N, \mathbb{C})$.[2] For any vertex v in the class tree, the bound we use is with respect to the set of classes in the subtree rooted at v. We write this set $v\Downarrow$, formally defined as:

$$v\Downarrow = classes(v) \cup \bigcup_{v' \in children(v)} v'\Downarrow \tag{6}$$

And we call $bound(N, v\Downarrow)$ the *saturation bound* at vertex v.

Given a class tree containing $vertex(class_n)$, we can estimate the total squeeze on n as the bounded sum at the root R of this tree:

$$squeeze_n = \mathcal{Z}(n, R) \tag{7}$$

The bounded sum $\mathcal{Z}$ is computed using the following recursive function:

$$\begin{aligned} \mathcal{Z}(n, v) &= \min(bound(N, v\Downarrow), raw\mathcal{Z}(n, v)) \\ raw\mathcal{Z}(n, v) &= \sum_{C \in classes(v)} worst^{degree_n(C)}(N, C) \\ &\quad + \sum_{v' \in children(v)} \mathcal{Z}(n, v') \end{aligned} \tag{8}$$

Without $\min$ and $bound$, $\mathcal{Z}(n, R)$ would be exactly equal to $\mathcal{W}_n(\mathbb{C}_{all})$. But $\mathcal{Z}(n, R)$ is a better estimate because it is *filtered*: our estimate of the cumulative number of colors denied by the adversary at the root of any subtree in n's class tree can never exceed the saturation bound at v. At each vertex v of the class tree, we calculate $raw\mathcal{Z}(n, v)$, the "raw" squeeze on node n due to neighbors with classes in $classes(v)$ or to the children of v. The word "raw" is a reminder that this variant of squeeze is not yet filtered. Raw squeeze is simply the total worst-case squeeze from n's neighbors whose classes are in $classes(v)$ plus the filtered squeeze values $\mathcal{Z}(n, v')$ for each vertex v' in $children(v)$. For each vertex, filtering recognizes that the adversary cannot consume more colors than possible by using all registers in classes in the subtree rooted at that vertex.

Safety. To show that $\mathcal{Z}(n, R)$ safely approximates $squeeze^*_n$, we begin by defining $\mathcal{Z}^*(n, v)$ to be the maximum amount the adversary can squeeze node n by coloring neighbors of n whose classes are in $v\Downarrow$. The ideal squeeze $squeeze^*_n$ is exactly equal to $\mathcal{Z}^*(n, R)$. Our approximation is safe because for any vertex v, $\mathcal{Z}^*(n, v) \leq \mathcal{Z}(n, v)$. We prove this fact by induction on the height of the subtree rooted at v. The key lemma is that

$$\mathcal{Z}^*(n, v) \leq \sum_{C \in classes(v)} worst^{degree_n(C)}(N, C) + \sum_{v' \in children(v)} \mathcal{Z}^*(n, v')$$

This lemma is proved using the triangle inequality and interchange rule.

3.4 Alias relationships and the class tree

To get the best possible approximation, we want a class tree that minimizes $\mathcal{Z}$. Since overcounting occurs when classes overlap, a key property of good class trees is that classes with aliases in common appear under the same saturation bound. Under constraints that are satisfied for all machines of which we are aware, we show how to construct class trees for which $\mathcal{Z}$ is as small as possible.

Consider the case in which two classes C_1 and C_2 alias exactly the same registers. We say such classes are *alias-equivalent*, written $C_1 \sim C_2$:

$$C_1 \sim C_2 \iff alias(C_1) = alias(C_2). \tag{9}$$

[2] A node can even be saturated with respect to a set of classes without necessarily being saturated with respect to any of the individual classes in the set.

If $C_1 \sim C_2$, then sets $\{C_1\}$, $\{C_2\}$, and $\{C_1 \cup C_2\}$ all provide the same saturation bound. So to add $worst^{degree_n(C_1)}(N, C_1)$ to $worst^{degree_n(C_2)}(N, C_2)$ and then bound them together is at least as good as to bound them separately and then add. It is therefore not hard to show that to get the best approximation, alias-equivalent classes should always be in the same vertex of the class tree. It is also not hard to show that we get a better approximation if each vertex contains *only* alias-equivalent classes.

A classic example of alias-equivalent register classes is floating-point register pairing as found in the original MIPS, SPARC, and HP PA-RISC microprocessor families: each consecutive pair of single-precision floating-point registers combines to form one double-precision register. But alias equivalence is broader than traditional pairing of registers, as illustrated by the register classes in Figure 1.

Another case in which it is useful to apply the same bound to two classes is one in which the alias set of class C_1 is contained within the alias set of class C_2. We say that C_1 is *alias-contained* in C_2, written $C_1 \sqsubset C_2$:

$$C_1 \sqsubset C_2 \iff alias(C_1) \subset alias(C_2) \tag{10}$$

If the vertex containing C_1 is a descendant of the vertex containing C_2, we can prevent the common registers from being counted in both C_1 and C_2.

Alias-contained register classes are found on such architectures as the Motorola 68K, the Intel x86, and the Intel Itanium. The 68K is a classic example: address registers C_a and data registers C_d form distinct register classes, each of which is alias contained in a third class C_{ad}, which contains *both* the address and data registers ($C_{ad} = C_a \cup C_d$).

When classes can overlap without being alias-equivalent or alias-contained, we know of no way to construct a class tree that always gives the best approximation. But for every real architecture we have studied, classes whose alias sets overlap *are* always alias-equivalent or alias-contained. For these architectures, we can build a class tree that is guaranteed to give the best possible $\mathcal{Z}$ for any interference graph. This tree has three properties:

- For every vertex v, $classes(v)$ is not empty.
- The alias set of every child vertex is contained in the alias set of its parent vertex, where the alias set of a vertex v is the union of the alias sets of classes in $classes(v)$.
- If two vertices have the same parent, their alias sets are disjoint.

Such a class tree always exists, and it is easy to show (by contradiction) that it is unique.

For example, the Motorola 68040, with its on-chip floating-point unit, would have two class trees. The integer class tree would be rooted with the alias set of C_{ad}; this vertex would have two children corresponding to vertices with alias sets of C_a and C_d. The other class tree would contain a single vertex corresponding to the aliases of the floating-point class.

Optimality. To show that the unique class tree above gives the best approximation, we define a "good" vertex as one that alias-contains each of its children, whose children are all disjoint, and whose $classes$ set is not empty. A "bad" vertex violates one or more of these conditions. In the unique class tree above, all vertices are good. Suppose there is some other class tree that provides a better approximation. Then we can find a bad vertex v whose proper descendants are all good. No matter how v is bad, there is a local transformation of the tree that has two properties:

1. It improves $\mathcal{Z}$, or at worst leaves it unchanged.
2. Either it decreases the number of vertex pairs in which alias-containment does not imply ancestry, or it leaves this number unchanged and decreases the number of parent-child pairs in which the parent does not alias-contain the child.

By property 2, we can repeat local transformations until there are no more bad vertices, and this repetition is guaranteed to terminate. By property 1, this sequence of transformations makes $\mathcal{Z}$ no worse. Therefore the "good" class tree produces an approximation that is at least as good as any other class tree.

Exactness. We can show that our approximation is exact for a large number of real architectures. For example, it is not difficult to show that exactness holds when no register classes have alignment restrictions, and the classes with overlapping alias sets are alias-equivalent. In such a case, no overcounting of the aliased register resources is possible in Equation 8, because each class tree has exactly one vertex.

Our approximation remains exact even if we allow some alignment restrictions within a register class; an acceptable alignment restriction simply needs to exhibit a large amount of regularity. By regularity we mean that all "multi-registers" are a power of two in size, when measured in units of singleton registers, and these multi-registers all align on the appropriate power-of-two boundary, as defined by the singleton numbering.

A less regular architecture which demonstrates when our approximation is not exact (though still safe) is the Intel i960. The i960 has not only single-width and quadruple-width integer registers, but also triple-width registers, each of which aliases with three single registers and must align on a quadruple-width boundary. For this machine, our colorability criterion may overestimate the squeeze; for example, if a single-width node has neighbors of both triple-width and quadruple-width classes, our coloring criterion cannot detect that because of alignment constraints, a register numbered 3 modulo 4 may be available.

4 Implementing Our Colorability Criterion

This section describes how we efficiently implement our colorability criterion and, in particular, $\mathcal{Z}(n, R)$, which is the measure of the constraint (or "squeeze") on node n by its neighbors. Just as a traditional graph-coloring register allocator caches the degree of each interference-graph node, we cache the value of $\mathcal{Z}(n, R)$. And as in a traditional allocator, when node n gains or loses a neighbor, we must incrementally update $\mathcal{Z}(n, R)$. Saturation makes this update tricky, since the effect of adding or removing a neighbor of class C depends on whether n is saturated with respect to some set of classes containing C.

A barrier to incrementally updating $\mathcal{Z}(n, R)$ is that the value of $worst^{degree_n(C)}(N, C)$ in Equation 8 may change in nonlinear ways as $degree_n(C)$ changes. But as an approximation, the harm the adversary can do with m registers is no more than m times the harm the adversary can do with one register:

$$worst^m(N, C) \leq m \times worst^1(N, C) \tag{11}$$

Even better, for the vast majority of real commercial architectures, this approximation is exact up to $bound(N, \{C\})$. As a bonus, using this approximation means we have to store only the table of $worst^1(N, C)$ values, not $worst^m(N, C)$ for every m up to $|C|$. We therefore use this approximation of $worst^m(N, C)$.

To update $\mathcal{Z}(n, R)$ incrementally, we observe that adding or removing a neighbor of class C affects the value of $raw\mathcal{Z}(n, v)$ only at vertices v on the path from $vertex(C)$ to the root vertex R. That's because those are the only vertices whose $raw\mathcal{Z}$ values depend on $degree_n(C)$.

We can minimize the amount of recalculation done when adding or removing a neighbor if we cache not only the filtered squeeze $\mathcal{Z}(n, R)$, but also the value of the $raw\mathcal{Z}(n, v)$ for each vertex v in n's class tree. Notice that it is incorrect to cache the filtered $\mathcal{Z}(n, v)$ values, since filtering loses information about how many neighbors of a class were added and this count is needed when neighbors drop out of the graph.

Now consider the sequence of p vertices that starts at $vertex(C)$ and then follows parent links to the root class R: $v_1 = vertex(C)$, $v_{i+1} = parent(v_i)$, $v_p = R$. From this sequence, we can obtain adjustments $\delta_1, \ldots, \delta_p$ for the cached raw squeeze values using the following recurrence relations:

$$\begin{array}{lcl} \delta_1 & = & \pm worst^1(class_n, C) \\ \alpha_i & = & raw\mathcal{Z}(n, v_i) \\ \beta_i & = & bound(class_n, v_i\Downarrow) \\ \delta_{i+1} & = & \Delta(\alpha_i, \delta_i, \beta_i) \end{array}$$

for $1 \leq i \leq p$, where α_i is the cached raw squeeze, δ_i is the change in α_i, and δ_{i+1} is the change propagating along the parent link after filtering by the bound β_i. The initial change δ_1 is positive when a neighbor is added and negative when a neighbor is removed.

The function $\Delta(\alpha, \delta, \beta)$ is defined in terms of an initial raw squeeze α, a change δ, and a saturation bound β:

$$\Delta(\alpha, \delta, \beta) = \begin{cases} \max(0,\ \min(\delta,\ \beta - \alpha)) & \text{if } \delta \geq 0 \\ \min(0,\ \max(\delta,\ \delta - (\beta - \alpha))) & \text{if } \delta < 0 \end{cases}$$

When $raw\mathcal{Z}(n, v_i)$ goes from α_i to $\alpha_i + \delta_i$, the resulting change in $\mathcal{Z}(n, v_i)$ is given by $\Delta(\alpha_i, \delta_i, \beta_i)$. First, consider when $\delta_1 > 0$, which corresponds to adding a neighbor. If both α and $\alpha + \delta$ are below the bound β, then the change in squeeze is the same as the change in raw squeeze. If both are above the bound, then there's no change in squeeze. If they bracket the bound, then the change in squeeze is limited to $\beta - \alpha$. The logic for $\delta_1 < 0$ (removing a neighbor) is analogous. When δ_i goes to 0 for some i, it means that a bound somewhere along the path to the root kept the rest of the nodes along the path from being affected by the addition or removal of the neighbor.

By the definition of $raw\mathcal{Z}$, the change in $raw\mathcal{Z}(n, v_{i+1})$ comes from one term in the sum over the children of v_{i+1}, namely $\mathcal{Z}(n, v_i)$. So the change in $raw\mathcal{Z}(n, v_{i+1})$ is $\Delta(\alpha_i, \delta_i, \beta_i)$, i.e., δ_{i+1}. And the net effect on node n of the neighbor change, i.e., the net change in $\mathcal{Z}(n, v_p)$, is $\Delta(\alpha_p, \delta_p, \beta_p)$.

To update n's cached squeeze, we compute $\Delta(\alpha_p, \delta_p, \beta_p)$. Each α_i is available as a cached raw squeeze value, and the implementation must update each cached value that changes due to addition or removal of n's neighbors. Figure 3 presents C++ code for computing $\Delta(\alpha_p, \delta_p, \beta_p)$ given a node n, a vertex v, and a raw change δ_1. During the computation, the code updates n's raw-squeeze cache as necessary. The function `bound(n.class, v)` returns $bound(N, v\Downarrow)$. The computation terminates either when it reaches the root vertex $v_p = R$ or when `filteredChange` becomes zero. When `filteredChange` becomes zero, it means that the addition or removal of n's neighbor has no immediate effect on n's squeeze value, and hence, on the colorability of n. Note that addition or removal of a neighbor will always change at least one cached raw squeeze value, even when the final `filteredChange` value is zero, so that the effect of the change is not lost to the system.

To reflect removal of a neighbor `t` of node `n`, a register allocator might use the function `squeezeChange` as follows:

```
n.squeeze += squeezeChange(n, vertex(t.class),
                           -worst1(n.class, t.class));
```

5 Generalizing a Representative Allocator

The iterated-coalescing algorithm is a textbook example of graph-coloring register allocation [George and Appel 1996; Appel and

```
int squeezeChange(IgNode n, Vertex v, int delta)
{
    int alpha = n.rawSqueeze[v];

    n.rawSqueeze[v] += delta;

    int filteredChange = Delta(alpha, delta,
                               bound(n.class, v));

    if (filteredChange == 0 || parent(v) == noVertex)
        return filteredChange;

    return squeezeChange(n, parent(v), filteredChange);
}
```

Figure 3. *To compute the change in $squeeze_n$ due to adding or removing a neighbor of class C, we start with node n, $vertex(C)$, and δ_1, which is positive to add and negative to remove. Auxiliary function* `Delta` *is Δ. Function call* `parent(v)` *produces the distinguished value* `noVertex` *when* `v` *is the root R of n's class tree. The value finally returned is the change in $\mathcal{Z}(n, R)$.*

Palsberg 2002]. In six pages of well-documented pseudocode, it covers all the important details of a practical implementation of a Chaitin-style allocator, as modified to reflect George and Appel's strategy for coalescing copy instructions and for representing register exclusions.

Before developing the ideas in this paper, we implemented the iterated-coalescing algorithm in C++ following George and Appel's pseudocode, with extensions suggested by Leung and George [1998]. We have derived a second implementation from that first one by substituting our generalization of the colorability criterion and by using class trees to account for saturation. In this section, we describe the effort required to generalize this representative register allocator. In the next section, we show that the result is practical to use in a production compiler.

There are a dozen places where the allocator tests whether a node is trivially colorable. Naturally the allocator makes this test during graph simplification, when it is trying to determine a coloring order by eliminating underconstrained nodes from the interference graph. But it also tests trivial colorability as part of its heuristics for deciding when to try coalescing a copy instruction and whether the decision to coalesce a copy could have negative consequences.

The traditional allocator uses the criterion $degree_n < k$ to test trivial colorability of node n. Our generalization replaces that test with $squeeze_n < |class_n|$. In each case, the test is small (one line of code) and efficient (constant time, no procedure calls).

Strategy for coalescing. Coalescing is affected by more than just the generalization of the colorability criterion. If an allocator decides to eliminate a copy and coalesce the interference-graph nodes representing its source and destination operands, the class of the resulting coalesced node must be the intersection of the operands' classes. A copy instruction can be coalesced only if the result of the intersection is a register class. On every machine of which we are aware, the intersection of two classes is either empty or is one of the two classes, so our implementation prohibits coalescing only when the operands' classes do not overlap. We check for overlap during the construction of the interference graph, at the point where the original allocator identifies copy instructions that might be coalesced.

When two nodes are coalesced, the class of the coalesced node is the smaller of the two original classes. Except for the fact that the allocator must retain the node with the smaller class, coalescing is as implemented by George and Appel.

Should we ever encounter a machine in which two classes could overlap without one being contained in the other, we would have to change our code. First, we would have to add classes as needed to make the set of classes closed under nonempty intersection. Second, to coalesce two nodes, the allocator would have to create a new node to replace the original nodes, making the new node's class the intersection of the original nodes' classes. Transferring edges to the new node would use the same procedure as transferring edges in the original pseudocode, maintaining the integrity of the approach.

Representing register exclusions. In general, there are points in a program where a register and all of its aliases are unavailable for allocation. For example, a caller-saves register is unavailable at a call site. The register allocator needs a way to inhibit allocation of the register at such a point, i.e., the register must be *excluded* from the set of allocable registers of candidates whose lifetimes cross the point of unavailability.

For candidates whose register classes don't contain an excluded register, exclusion is implicit and needs no special representation. For other candidates, it is customary to represent an exclusion by extending the graph with an *exclusion node* that represents the excluded register, and to add an interference-graph edge from the exclusion node to the candidate's node. Such an *exclusion edge* is needed for each member of the candidate's class that is unavailable.

George and Appel use exclusion nodes and edges in their allocator. But because an exclusion node can have a very large number of exclusion edges, they omit neighbor lists from such nodes, and they carefully design their allocator to avoid needing those lists.

We prefer to omit exclusion nodes from the interference graph altogether. Instead, to identify the registers from which a candidate is excluded, our implementation associates an *excluded-register set* with each candidate. When testing for colorability, we still calculate $squeeze_n$ and compare it to the number of register names available to candidate n. The number of names available simply becomes the size of n's class minus the size of n's excluded-register set E, $|N| - |E|$, which is a constant. When the color-selection phase of the allocator computes the set of excluded "colors" for each candidate, the excluded-register set is used to initialize the set of excluded colors. As Section 6 shows, omitting exclusion nodes from the interference graph also leads to space benefits. Omission of exclusion nodes can also improve the cost/benefit estimates used for choosing a candidate to spill. The degree of a candidate's node is traditionally part of such estimates because it approximates the increase in overall colorability to be gained by removing the node from the graph. But counting exclusion edges in such an estimate skews the result, since the colorability of the exclusion nodes that they connect to is not in question.

Effort required. Overall, our original implementation of the George and Appel allocator with extensions by Leung and George took 1215 lines of code. For our generalized version, we changed that code in 25 places. There are 80 lines of new code for defining and maintaining extra fields for *squeeze* caches in the node data structure. The `squeezeChange` function in Figure 3 is typical of this added code. In addition, we replaced 37 lines of original code with 68 lines of generalized but functionally similar code. The twelve places where we generalized the colorability criterion are typical of these replacements. Finally, we wrote 210 lines of code that runs once, when the compiler configures itself to the target machine. This code derives a representation of the register class tree, the worst-case-displacement table, and other static structures that allow the allocator to operate efficiently.

6 Practicality

Section 3.2 talks about an approximation for the ideal $squeeze^*_n$ that is safe, good, and arguably fast. Here we explore exactly how fast it is.

Our generalization of the George and Appel allocator, as discussed in Section 5, is implemented in Machine SUIF. This allocator is not just an experimental prototype; we use it in our everyday research. We measured register-allocation times for the SPEC2000 benchmark suite; each measurement is the average of five runs on an unloaded 2.53 GHz Pentium 4 with 2 GB of memory. Machine SUIF includes back ends for Alpha and x86, and we are able to compile and run all of the C and FORTRAN-77 benchmarks on these two targets. Because SUIF does not support C++, FORTRAN-90, or extended FORTRAN-77, we do not include results for the SPEC benchmarks written in these languages.

Our register allocator can be compiled to use either the traditional colorability criterion or our new, generalized criterion. Reconfiguring the allocator for a new machine requires specifying the register classes, their members, and the register-alias map. In our C++ implementation, this specification takes one line of code for the class enumeration, one line per class for defining class membership, and one line per register name for specifying its aliases. We are working on a scheme to generate this code automatically from a simple specification language.

This section presents three sets of experiments that focus on the practicality of our generalized colorability criterion. Section 6.1 looks at the cost of our approach for an x86 target, which is an architecture that exhibits register aliasing and a non-trivial class tree. Section 6.2 shows how the cost of our allocation approach scales with increased register pressure. Section 6.3 measures the cost of our approach for an Alpha target, which is an architecture that doesn't always need the generality of our approach.

6.1 Cost of doing allocation right

Our first set of experiments measures the cost of allocating integer registers for the Intel x86 target. The x86 architecture is interesting because it includes alias-equivalent 16-bit and 32-bit accumulator registers that also alias with pairs of byte registers. In addition, the architecture includes two alias-equivalent classes of 16- and 32-bit index registers that cannot be used everywhere the 16- and 32-bit accumulator registers can be used. As illustrated in Figure 4, we thus have two register classes—the accumulator registers and the index registers—that are alias-contained in the complete set of integer registers.

In Figure 5, we evaluate the impact on compile time of using our generalized algorithm with the register model illustrated in Figure 4. To help analyze the components of allocation time, we also show two other timings. The first (*Trad-ideal*) uses the traditional graph-coloring formulation and targets an x86-like machine that has a single ideal register class containing six 32-bit registers that are interchangeable and independent. The second (*Gen-noalias*) uses our generalized formulation and the same set of x86 register classes shown in Figure 4 but without any aliasing of their members. The third (*Gen-real*) uses the generalized formulation for the real x86 target, including aliasing. The allocation times are reported, for each benchmark, relative to the allocation time of the (*Trad-ideal*) case. For these experiments, we measure only the cost of the first iteration of coloring, since the algorithms may iterate different numbers of times given the different hardware-resource constraints. Also, because the x86 does not have allocable floating-point registers, we report allocation times for only the integer benchmarks. The times are averages over five runs, with all standard deviations well below 1%. Each bar in the graph breaks the total allocation time into the time spent building the interference graph, time devoted to coalescing copy instructions, and

Register classes for x86:

$$
\begin{aligned}
C_{EX} &: \{eax, ebx, ecx, edx\} \\
C_X &: \{ax, bx, cx, dx\} \\
C_{LH} &: \{al, ah, bl, bh, cl, ch, dl, dh\} \\
C_{EI} &: \{esi, edi\} \\
C_I &: \{si, di\} \\
C_{EXI} &: C_{EX} \cup C_{EI} \\
C_{XI} &: C_X \cup C_I
\end{aligned}
$$

Class tree:

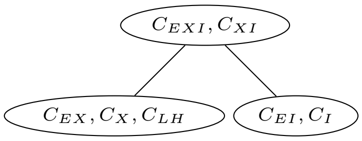


Figure 4. *Class definitions and class tree for x86. Many of the classes are alias-equivalent, e.g., the class C_{EX}, containing the 32-bit accumulator classes, is alias-equivalent with C_X, containing the 16-bit accumulators, and C_{LH}, containing the 8-bit accumulators.*

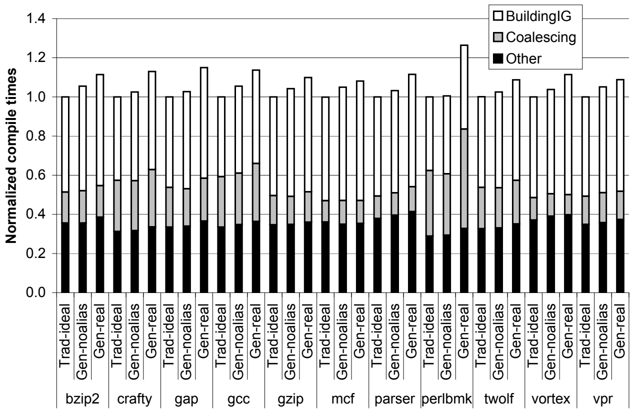


Figure 5. *The cost of our approach when registers aren't interchangeable and independent. The target machine is an x86. Allocation times are presented for the traditional approach (*Trad-ideal*) using a single idealized register class (with interchangeable and independent registers), and for our approach using the real set of x86 register classes (*Gen-real*) and using the same set of x86 register classes but without any aliasing of their members (*Gen-noalias*). The results are scaled for each benchmark to the allocation time of the traditional approach.*

time for other allocation activities (classifying operands, inserting spill instructions, rewriting code to replace register candidates with their assigned registers, etc.).

The results in Figure 5 show that our generalized approach increases allocation times by less than 30% for all benchmarks and less than 15% for all but *perlbmk*. By comparing the numbers for *Gen-real* against *Gen-noalias*, we can quantify the cost of aliasing, and by comparing *Gen-noalias* against *Trad-ideal*, we can quantify the cost of class trees.

Increases in the cost to build an interference graph come from several sources. When registers alias, the generalized allocator must perform an interference check not just for the register defined at a definition point, but also for each alias of the defined register. This repeated work in the inner loop of building the interference graph accounts for much of the difference in allocation time between *Gen-noalias* and *Gen-real*. Most of the rest of the extra time to build the interference graph is due to an increase in the number of edges inserted into the interference graph when register classes alias. This resulting increase in register contention also accounts for increases in the time to complete the allocation tasks in the *Coalescing* and *Other* categories.

In the cost to build an interference graph, the differences between *Trad-ideal* and *Gen-noalias* are due to the cost of running `squeezeChange`. This routine is run twice (once for each endpoint) when adding an edge to the interference graph under *Gen-noalias*. The routine is also run when nodes are removed from the graph, explaining the small change in portions of the allocation times labelled *Other*.

6.2 Cost of increased register pressure

We now present an example of how the compile-time costs of our generalized algorithm scale as registers become scarce. We focus on architectures with aliased registers because, as shown in the previous experiment, aliasing has a noticeable impact on allocation time.

We define an imaginary family of targets based on the Alpha architecture, identical except for the sizes of their floating-point register files. Our primary target in this section is one that creates double-precision registers out of aligned pairs of single-precision registers. To verify that our algorithm can successfully exploit the single-precision register pairs, we also define a baseline machine model with only double-precision registers—so that a single-precision value consumes an entire double-precision register.

For this experiment, we chose to use the benchmark GSM, a speech-compression program from the MediaBench suite [Lee, Potkonjak, and Mangione-Smith 1997]. This program contains a good mix of single- and double-precision floating-point register candidates. All run-time results for GSM were generated using its "`clinton`" input.

Since we would like to compile GSM under a range of register pressures, we must identify a proxy for register pressure. The proxy we use is the percentage of spill instructions (loads and stores) executed. Using this measure, Figure 6 shows that our algorithm encounters nearly no register pressure when compiling GSM for a target with 12 double-precision registers (with pairs). The number of executed spill instructions (and thus register pressure) increases steadily as we reduce the number of available floating-point registers.

Figure 6 also reports the percentage of spill instructions executed by GSM when compiled for a target without single-precision pairs. As expected, an aliasing-aware allocator is able to reduce spilling by exploiting the single-precision register pairs.

Figure 7 resumes our focus on the targets with aliased pairs. It shows the total time spent in allocation for all procedures in GSM. As registers become scarce and register pressure increases (reading the bars right to left), total allocation time also increases. Interestingly, as Figure 7 illustrates, the change in allocation time comes in steps. The reason is that when spilling occurs, the allocator is forced to rebuild the interference graph and to launch a new coloring attempt. As register pressure increases, the allocator must discard and rebuild the interference graph more often. In this experiment, the targets with 10 and 12 double-precision registers required the same number of iterations of the build-graph-and-color loop; the targets with 6 and 8 double-precision registers required an extra two iterations due to spilling; and the target with 4 double-precision registers required yet another two rounds. Hence the steps up in time.

It may seem odd that the times tend to be so even within the steps. The reason the "*10 pairs*" case resembles "*12 pairs*" is simply that, for the GSM benchmark, spilling doesn't change between the two. But that's not true for the transition between 6 and 8 pairs. To understand the effect of this difference, consider the time

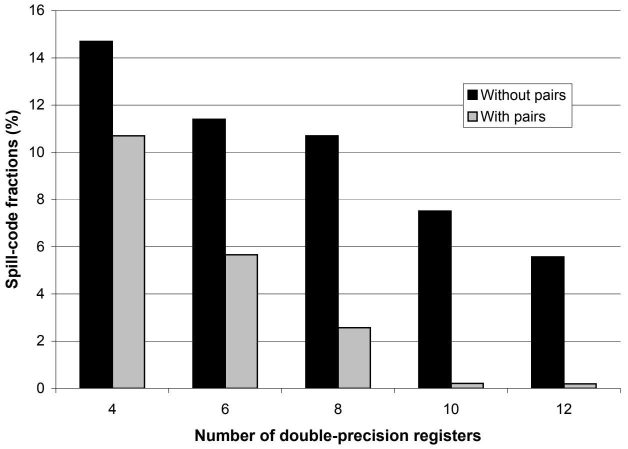


Figure 6. *Dynamic spill-code fractions for the GSM benchmark. Targets labelled* With pairs *create double-precision floating-point registers from aligned pairs of single-precision registers. Targets labelled* Without pairs *store all floating-point values in double-precision registers. The vertical axis shows the fraction of dynamic instructions that are spills.*

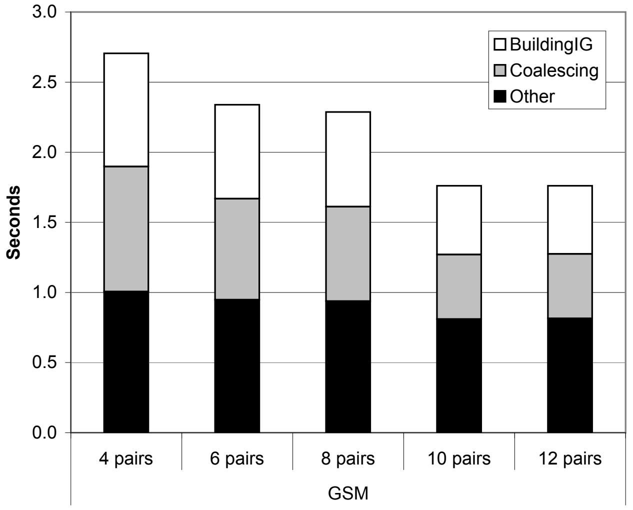


Figure 7. *Register-allocation times for the GSM benchmark, as compiled for an imaginary Alpha family with floating-point register pairs. The horizontal axis shows the number of single-precision pairs comprising the floating-point register pool. The vertical axis shows the total number of seconds required for register allocation of the entire GSM benchmark.*

required to build an interference graph. For a given procedure, the first graph takes as long to build for one target as for another. But when spilling forces a second or subsequent iteration of the build-and-color loop, two kinds of changes affect graph-building time. The addition of spill instructions to the program means that there are more instructions to be scanned while creating the interference graph. However, the removal (by spilling) of highly constrained register candidates means that there are fewer interference-graph edges to be created. The time cost of graph building is very nearly a linear combination of the number of instructions scanned and the number of edges created. For allocations with 6 and 8 pairs, the two terms counterbalance each other almost exactly.

The cost of copy coalescing also depends heavily on the edge density of the graph, because the heuristic that avoids over-

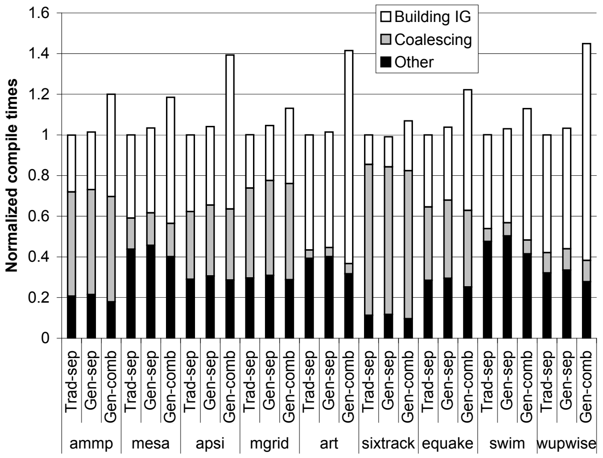


Figure 8. *The cost of our generalization when it isn't necessary. The target is an Alpha. Allocation times are presented for the traditional approach (*Trad-sep*), our approach using separate allocations for each register class (*Gen-sep*), and our approach using a single combined allocation pass for all classes (*Gen-comb*). The results are scaled for each benchmark to the allocation time of the traditional approach.*

		Separate		Combined
Benchmark	Procedures	Trad	Gen	Gen
ammp	179	402	402	218
mesa	1106	2329	2329	1216
apsi	98	256	256	157
mgrid	13	29	29	16
art	26	53	53	27
sixtrack	147	322	322	172
equake	27	55	55	28
swim	7	14	14	7
wupwise	22	50	50	28

Table 1. *Number of procedures compiled per benchmark and total number of coloring iterations required during register allocation under the traditional (*Trad*) and our generalized (*Gen*) approaches.*

	Total edges		Edges/node	
Benchmark	out	in	out	in
ammp	1880.1	2107.7	2.4	3.0
mesa	1958.5	2090.3	1.2	1.6
apsi	8819.6	9339.5	7.3	8.4
mgrid	10163.4	10440.4	9.0	9.7
art	775.6	888.5	1.3	1.6
sixtrack	27413.6	28294.2	3.6	4.2
equake	2419.5	2592.3	1.5	1.7
swim	2755.6	2887.6	3.0	3.3
wupwise	5962.2	6385.1	3.6	4.6

Table 2. *Increase in the number of edges per interference graph and the average number of edges per node in an interference graph when compiling without (*out*) and with (*in*) special register nodes in the interference graph. These numbers are averages over all interference graphs produced during each benchmark's compilation; the edges-to-nodes ratio was computed as a geometric mean.*

coalescing requires visiting the neighbors of potential coalesced nodes. However, the heuristic also entails more bookkeeping when the number of overconstrained neighbors of a coalesced node is higher, which becomes more likely as register pressure grows. So like interference-graph building time, the time for copy coalescing stays about level as the register pool drops from 8 pairs to 6 pairs, because the lower edge density on second and subsequent coloring iterations is counterbalanced by a higher relative population of overconstrained candidate nodes.

The effect of increased spilling on the "Other" time costs of register allocation is smaller, because the extra time to insert load and store instructions and to create short-lived temporaries is a relatively small fraction of the work in this catchall category.

In summary, the allocation-time performance of our generalized algorithm increases slowly and in a predictable manner as register pressure increases. Part of the appeal of traditional graph coloring is that its cost is commensurate in practice with the work it gets done, and our generalization retains that appealing property.

6.3 Cost of always doing allocation right

Our last experiment measures the cost of allocating the integer and floating-point registers for a real Alpha target. The Alpha architecture is interesting because if we allocate Alpha's integer and floating-point registers separately, our generalization is not needed—we can use the traditional formulation of graph coloring. Figure 8 compares allocation times for the traditional formulation; for our formulation, but using separate allocation passes for integer and floating-point registers; and for our formulation using a single combined allocation pass for all registers. Since we're interested in compilation involving multiple register classes, this section includes results for only the SPEC floating-point benchmarks.

As the figure illustrates, the allocation times for our approach with separate allocations of the integer and floating-point registers are within 5% of the times of the traditional approach. When we run the allocation of integer and floating-point registers together, however, you can again see the costs of allocating multiple register classes simultaneously, which were highlighted in Section 6.1.

Table 1 may help to explain the consistent nature of the increases in Figure 8. The table lists the number of procedures compiled and the total number of coloring iterations required for register allocation. Notice that there are always slightly more than half as many iterations required in the combined case. This is a result of the fact that spilling in one class requires reallocation of all classes. This extra cost, plus the non-linear cost of interference-graph building with larger sets of live candidates and greater numbers of relevant candidate-definition points, accounts for the majority of the increase in allocation time. Overall, these compile-time costs should be considered when investigating new optimizations that require the simultaneous allocation of register classes, as discussed in the introduction.

Using the Alpha configuration, we also investigated the time and space effects of including special register nodes in the interference graph. In this experiment, we configure our generalized algorithm so that it allocates all register classes in a single pass. We don't present the time effects in detail because there was very little difference between the allocation times (all within 2%). With respect to space savings, Table 2 reports on the change in the average number of edges per interference graph and the average ratio of edges-to-nodes per interference graph. We found that handling register exclusions without introducing special register nodes eliminates about 6%, on average, of the edges in each interference graph.

7 Related Work

Starting with Chaitin et al. [1981], there is a large body of work on global register allocation by graph coloring. But only a handful of authors describe algorithms that extend graph-coloring allocation beyond the assumptions that registers are interchangeable and independent.[3] As we explain below, none of these algorithms is as complete a solution as ours.

Briggs describes an algorithm for coloring aligned and unaligned register pairs [Briggs 1992; Briggs, Cooper, and Torczon 1992]. This algorithm requires that a node's degree accurately reflect its colorability. To make a node's degree reflect its colorability even in the presence of aliasing, Briggs adds "additional" edges to the interference graph in an attempt to model the aliasing constraints. Unfortunately, this edge-focused approach sometimes reports that a node is trivially colorable when it is not, and it is not easy to see how the approach could be extended to handle the simultaneous allocation of multiple, overlapping register classes.

Nickerson [1990] presents an algorithm that handles register candidates requiring two or more adjacent, aligned registers. Candidates requiring two or more registers are called "clusters" and the individual registers of a cluster are called "cluster-mates". In Nickerson's approach, an interference-graph node represents an individual register of a cluster. Nickerson points out that it is not always possible to use the traditional colorability criterion everywhere in his interference graph, even after identifying and removing implicit edges whose interference relation is subsumed by an edge of a cluster-mate. For these cases, Nickerson invents an artificial k to make his model work.

Runeson and Nyström [2003] describe a design for a retargetable graph-coloring allocator for irregular architectures. Their work goes part of the way along the path to a generalized coloring criterion. When overlapping alias sets are alias-equivalent, our generalized colorability criterion simplifies to their $\langle p, q \rangle$ test. Our independently discovered results, however, go quite a bit farther. We show how nested register classes can lead the $\langle p, q \rangle$ test to identify candidates as more squeezed than they truly are, and we show how to avoid this inaccuracy by using saturation bounds.

Koseki, Komatsu, and Nakatani [2002] describe a technique for modifying the selection phase of graph-coloring allocation to increase the likelihood that candidates are given their preferred registers. Preference-directed graph coloring is related to our work in that it handles multiple register classes. However, while we use class information to help determine when candidate nodes are trivially colorable, Koseki et al. use class information only during register selection (i.e., register classes appear only in their register-preference graph, not in their interference graph). Their approach has no notion of saturation and can incorrectly assume that an interference node is not trivially colorable when it actually is. Finally, their algorithm does not seem to support architectures in which registers alias.

A number of researchers have cast register allocation as a mathematical-programming problem, rather than a graph-coloring problem [Goodwin and Wilken 1996; Kong and Wilken 1998; Appel and George 2001; Fu and Wilken 2002; Scholz and Eckstein 2002; Hirnschrott, Krall, and Scholz 2003]. These approaches can handle a wide variety of architectural irregularities, but these benefits come at the cost of significant increases in compile time.

8 Conclusions

Despite decades of research on compiler construction, reusable compiler components remain all too rare. Chaitin's graph-coloring formulation of register allocation has been remarkably robust, but to make it usable for real targets, practitioners have almost always had to augment it in unstructured ways. We maintain the structure and efficiency of the original algorithm while making it extremely simple to target new machines and retrofit existing allocators.

One of our key insights is that coloring constraints on each interference-graph node should be expressed in terms of the set of registers available to it. With this insight we produce a generalization that handles simultaneous allocation of multiple register classes and accommodates register aliasing in an elegant way. Because allocators using our approach know about the set of registers available to each node, they can recognize when overlap between such sets would introduce inaccuracies in the criterion for colorability, and thereby avoid the overcounting inherent in simpler formulations.

[3] Clearly, many others have implemented allocators that go beyond these assumptions, but here, we avoid discussing what would be categorized as "workarounds".

Acknowledgments

We thank João Dias, Greg Morrisett, and the anonymous referees for their comments and suggestions. This work has been funded in part by gifts from Intel and Microsoft, by an Alfred P. Sloan Research Fellowship, and by NSF grants CCR-0310877, CCR-0311482, and ITR-0325460.

References

Andrew W. Appel and Lal George. 2001 (May). Optimal spilling for CISC machines with few registers. In *2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253.

Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.

Preston Briggs. 1992 (April). Register allocation via graph coloring. Technical Report TR92-183, Rice University, Department of Computer Science.

Preston Briggs, Keith Cooper, and Linda Torczon. 1992 (March). Coloring register pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13.

Preston Briggs, Keith Cooper, and Linda Torczon. 1994 (May). Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3): 428–455.

Gregory J. Chaitin. 1982. Register allocation and spilling via graph coloring. In *1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages*, 6(1):47–57.

Keith Cooper and Linda Torczon. 2003. *Engineering a Compiler*. Morgan Kaufmann.

Changqing Fu and Kent D. Wilken. 2002 (November). A faster optimal register allocator. In *35th ACM/IEEE International Symposium on Microarchitecture*, pages 245–256.

Lal George and Andrew W. Appel. 1996 (May). Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324.

David W. Goodwin and Kent D. Wilken. 1996 (August). Optimal and near-optimal global register allocations using 0–1 integer programming. *Software—Practice & Experience*, 26(8):929–965.

Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. 2003 (August). Graph coloring vs. optimal register allocation for optimizing compilers. In *Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 202–213. Springer Press, Klagenfurt, Austria.

Richard E. Kessler, Edward J. McLellan, and David A. Webb. 1998 (October). The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, pages 90–95.

Timothy Kong and Kent D. Wilken. 1998 (December). Precise register allocation for irregular architectures. In *31st ACM/IEEE International Symposium on Microarchitecture*, pages 297–307.

Akira Koseki, Hideaki Komatsu, and Toshio Nakatani. 2002 (June). Preference-directed graph coloring. In *2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 33–44.

Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997 (December). MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *30th ACM/IEEE International Symposium on Microarchitecture*, pages 330–335.

Allen Leung and Lal George. 1998. A new MLRISC register allocator. Standard ML of New Jersey compiler implementation notes.

Michael Matz. 2003 (May). Design and implementation of a graph coloring register allocator for GCC. In *GCC Developers Summit*, pages 151–170.

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

Brian R. Nickerson. 1990 (June). Graph coloring register allocation for processors with multi-register operands. In *1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 40–52.

Johan Runeson and Sven-Olof Nyström. 2003 (September). Retargetable graph-coloring register allocation for irregular architectures. In *Software and Compilers for Embedded Systems (SCOPES)*, volume 2826 of *Lecture Notes in Computer Science*, pages 240–254. Springer Press, Klagenfurt, Austria.

Bernhard Scholz and Erik Eckstein. 2002 (June). Register allocation for irregular architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 139–148.