Generalizing Chaitin's Algorithm: Graph-Coloring Register Allocation for Irregular Architectures

Johan Runeson, Sven-Olof Nyström

Department of Computer Science,
Uppsala University
{jruneson,svenolof}@csd.uu.se

Abstract

We consider the problem of generalizing Chaitin-style graph-coloring register allocation to the irregular architectures used in embedded systems. A class of formal machine descriptions is introduced, capable of describing a wide range of irregular architectures. We extend Chaitin's register allocation algorithm to handle any architectural constraints that can be expressed using these machine descriptions. The generalized algorithm is applicable to a wider range of architectures and systems than any other adaptation of Chaitin's algorithm found in the literature. We argue that the modifications to the original algorithm can be combined with most important extensions to Chaitin's framework, for example coalescing and optimistic coloring.

1 Introduction

From a compiler-writer's point of view, embedded processor architectures have two striking features: they are often irregular, and there are a lot of them. It is therefore important to find retargetable techniques that are able to handle irregular architectures.

In this paper we focus on register allocation, which is one of the most important optimizations in a modern optimizing compiler [9]. The main task is to decide, for each point in the program, which values to store in registers.

Most modern compilers use register allocation techniques based on graph-coloring. Several variants exist, but the most popular methods are based on Chaitin's algorithm [6]. In this algorithm, the target architecture is characterized only by the number of registers, k, and it is further assumed that the instruction set architecture is orthogonal, i.e. that there are only general-purpose registers. We will call such architectures *regular*.

An *irregular* architecture, on the other hand, places more constraints on register allocation than what can be described by a single number. For example, some instructions may have operands that can only be from a subset of the registers, or registers can be combined into pairs or other clusters which are used as operands. The run-time system can also add constraints on allocation, by reserving certain registers for system use, or by specifying that function parameters are passed in particular registers.

The goal of our work has been to develop a register allocation algorithm that will work for arbitrary architectures, regular or irregular. Our basic idea is that the characterization of the target architecture in Chaitin's algorithm is too inflexible for our purposes.

Our first contribution is that we generalize Chaitin's register allocation algorithm so that it correctly takes into account the constraints imposed by the target architecture and the run-time system, as expressed in a formal machine description. The generalized algorithm is applicable to a wider range of architectures and systems than any other adaptation of Chaitin's algorithm found in the literature. We argue that the modifications to the original algorithm are orthogonal or nearly orthogonal to well-known extensions of Chaitin's framework, for example coalescing and optimistic coloring.

As a second contribution, we present a safe and efficient approximation to the colorability test, which is prohibitively expensive in the generalized algorithm. We show how the parameters that control the approximation can be derived automatically from the machine description. Consequently, the approximation works with any architecture that can be characterized by the machine descriptions. The approximation is precise for regular architectures. For irregular cases, the precision is similar to or better than the precision of previously published methods for handling irregular architectures in Chaitin's algorithm.

The rest of this paper is organized as follows. Sec-

tion 2 gives a brief background on graph-coloring register allocation, and some previous attempts at adapting it to handle irregular architectures. We define our machine descriptions in Section 3, and show how Chaitin's algorithm is adapted to take these into account. The approximation to the colorability test is presented in Section 4, and proven safe. We present the complete generalized register allocation algorithm in Section 5. Section 6 discusses extensions to the algorithm. We give some examples of how our method works in Section 7. Related work is discussed in Section 8, and in Section 9 we present conclusions and future work.

2 Background

We assume that the register allocator is given low-level code, in which the instructions correspond to assembly code instructions, but where the operands are not specific register names, but rather *variables* taken from an unlimited set of names. A variable might correspond to a variable in the source program, or to a compiler-generated temporary. The purpose of the register allocator is to transform the code so that it uses actual registers instead of variables for operands, in the most efficient way. Since the register resources are limited, some variables may have to reside in memory at some points in the program. A good register allocator minimizes the cost of register-memory transfers (spill code), and register-register transfers in the program

2.1 Graph-Coloring Register Allocation

Say that a variable is *live* at a point in the program if it has been assigned a value that may be used later. At each point in the execution of a program, only live variables need to be stored (in registers or in memory).

If two variables are live at the same time, they must be stored in different registers. To represent this information, an *interference graph* is constructed. There is one node in the graph for each variable, and an edge between two nodes if the variables corresponding to those nodes are live simultaneously at some point.

A *k-coloring* of an interference graph is an assignment of one of k colors to each node in the graph such that any two nodes connected by an edge have different colors. Given a regular architecture with k general-purpose registers, a k-coloring of the interference graph corresponds to an allocation of each variable to a specific register.

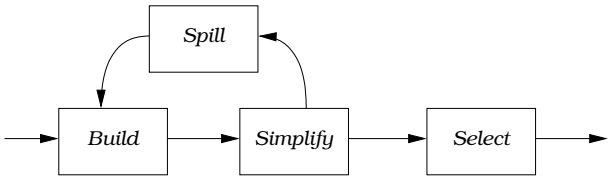


Figure 1: Phases of the basic register allocation algorithm

The problem of determining whether there is a k-coloring for a given graph is known to be NP-complete. Heuristic techniques are therefore used in most practical implementations of graph-coloring register allocation.

2.2 Chaitin's Algorithm

We will say that a node n in an interference graph is *locally colorable*[1] iff, no matter what colors are assigned to the neighbors of n, there is always a free color for n which is not used by any of its neighbors. In a regular architecture, this happens exactly when n has fewer that k neighbors. We will refer to the latter as the *degree* $< k$ *test.*

By removing a locally colorable node from the graph, we can reduce the coloring problem to the simpler problem of coloring the remaining nodes in the graph. Given a k-coloring for the rest of the graph, there is always a free color available for the node which was removed. If all nodes in a graph can be removed in this way, they can be colored in the reverse order from which they were removed.

The presentation of the algorithm follows Briggs et al. [4]. However, we have omitted several details in order to simplify the presentation, in particular *coalescing* and *optimistic coloring*. These extensions are discussed in Section 6.

The register allocation algorithm is divided into four phases (Figure 1).

1. *Build* constructs an interference graph using liveness information to add edges between variables which are simultaneously live.

2. *Simplify* first initializes an empty stack, and then repeatedly removes nodes from the graph which are locally colorable (i.e. they have fewer that k neighbors), and pushes them on the stack. This

[1] Briggs uses the term "trivially colorable". For an irregular architecture, determining local colorablility is not always trivial.

continues until either the graph is empty, or all remaining nodes fail the local colorability test.

If there are nodes remaining in the graph after this stage, the allocator goes to the Spill phase, otherwise it proceeds with Select.

3. *Select* assigns colors to the nodes in the graph. One by one, the nodes are popped from the stack built in Simplify, and re-inserted in the interference graph. After a node has been inserted, it is assigned a color which is not taken by any of its neighbors.

4. *Spill* is invoked if Simplify fails to remove all nodes in the graph. It selects one of the remaining nodes to spill to memory, and inserts a load before each use of the variable, and a store after each definition. Then, the allocation process is re-started from the Build phase.

The stack guarantees that *Select* receives the nodes in the reverse order from which they were removed from the graph, and the local colorability criterion in *Simplify* guarantees that *Select* will find a free color for each node.

2.3 Handling Irregular Architectures

Chaitin et al. [6] offer a solution to the problem of allocating operands for instructions that operate only on a subset of the available registers. The physical registers are added as nodes in the graph, and each node is made to interfere with those physical registers that it can not be allocated to [6, 5, 7]. The $degree < k$ test in the Simplify stage need not be changed, since the extra edges model the additional constraints on allocation.

Briggs et al. [4] propose a solution to the problem of allocating register pairs. The idea is to extend the interference graph to a multi-graph and have multiple edges between nodes when one node is a pair, expressing the fact that both nodes are more difficult to color. The simplification step now considers the number of edges connecting one node to its neighbors. By adding a sufficient number of edges, we can ensure that a node can only be removed by the simplifiction step when it is safe to do so. Briggs' solution is discussed further in Section 7.2.

Recent work by Smith and Holloway [12], performed independently from our work, has taken a different approach to handling register pairs and non-orthogonal instruction sets. Unlike the methods described above, the interference graph is left unchanged, and instead the interpretation of the graph in the algorithm is modified. The approach is similar to ours in many respects. We discuss the similarities and differences in Section 8.

3 A Systematic Approach

We want our register allocation algorithm to be applicable to arbitrary architectures. Therefore, we build our work around a formal machine description, which contains information about all the constraints that the register architecture or the run-time system can put on register allocation. We modify Chaitin's algorithm so that it takes the machine description into account.

The rest of this section presents the details of the machine descriptions, and how they affect the basic definitions underlying the graph coloring approach to register allocation.

3.1 Machine Descriptions

We define a *machine description* to be a tuple $\langle Regs, Conflict, Classes \rangle$, where

1. *Regs* is a set of register names,

2. *Conflict* is a symmetric and reflexive relation over the registers, and

3. *Classes* is a set of register classes, where each register class is a non-empty subset of *Regs*.

For a given architecture, we include a register name in *Regs* if there is an instruction which accepts that name as a register operand. There does not have to be a one-to-one mapping between register names and physical registers. Some register names may represent register pairs or other clusters, which overlap other registers wholly or partially.

Two register names (r, r') are in *Conflict* if they can not be allocated simultaneously, typically because they overlap. For example, a register pair conflicts with its component registers. The set *Regs* and the relation *Conflict* form a *conflict graph*, which describes how the register resources in the processor interact.

A class C is included in *Classes* if there are operations which restrict a variable to be from the set C only. These restrictions are mostly imposed by the instruction set architecture, which may require, for example, that a particular operand for a particular instruction is an aligned register pair, or that the result of a particular instruction be placed in a particular register or set of registers. The run-time system may also affect the choice of register classes,

by reserving certain registers for system use, or specifying that the arguments to a function are passed in particular registers.

We use register classes to enforce constraints on the operands to certain instructions. If a variable is used as an operand to an instruction which only allows that operand to be from a set $R \subset Regs$, that variable is given a register class which is included in R. A variable which is used in more than one operation must satisfy the constraints from each of those operations, and will consequently be given a register class which is included in the intersection of the register classes required by the operations.

3.2 Generalized Interference Graphs

For a given machine description

$$\langle Regs, Conflict, Classes \rangle,$$

we define a *generalized interference graph* to be a tuple $\langle N, E, class \rangle$, where N and E form an interference graph $\langle N, E \rangle$, and $class : N \rightarrow Classes$ maps each node to a register class.

The register class for a node constrains what registers may be assigned to that node by the allocator. We define an *assignment* for a set of nodes $M \subseteq N$ to be a mapping A from M to $Regs$ such that $A(m)$ is in $class(m)$ for all nodes $m \in M$. Furthermore, we will say that an assignment A for M is a *coloring* iff there are no neighboring pairs of nodes m and m' in M such that $(A(m), A(m'))$ is in $Conflict$.

Given a machine description and a generalized interference graph, the register allocation problem reduces to the problem of finding a coloring for the generalized graph, just like in the regular case.

3.3 Coloring a generalized interference graph

Recall that the problem of coloring an ordinary interference graph could be simplified by removing a node which was locally colorable. Is the same true for a generalized interference graph?

For a generalized interference graph $\langle N, E, class \rangle$, a node $n \in N$ is *locally colorable* iff for any assignment of registers to the neighbors of n, there exists a register r in $class(n)$ which does not conflict with any register assigned to a of neighbor of n.

With this definition of local colorability, the problem of finding a coloring for a generalized interference graph can be simplified by removing a node n which is locally colorable, just like in the regular case. Given a coloring for the rest of the graph, the local colorability property guarantees that we can always find a free register to assign to n. If a generalized interference graph can be simplified to the empty graph by repeatedly removing locally colorable nodes, then by induction it is possible to construct a coloring by considering the nodes in the reverse order from which they were removed.

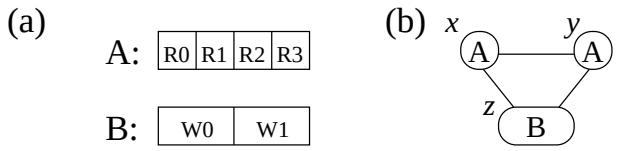


Figure 2: A simple example: (a) machine description diagram, (b) generalized interference graph

4 Approximating Colorability

For some irregular architectures, the $degree < k$ test is not an accurate indication of whether a node is locally colorable. Consider the example in Figure 2(a). The diagram illustrates the machine description for a simple architecture with a class A consisting of four basic registers R0-R3, and a class B with two registers W0 and W1, formed from aligned pairs of the registers in A. The diagram is constructed so that two registers conflict iff they occupy the same column in the diagram. For example, $W0$ conflicts with $R0$ and $R1$, but not with the other registers.

Figure 2(b) gives a generalized interference graph under this machine description. The nodes are annotated with their register classes, for example $class(x) = A$ and $class(z) = B$. It is easy to see that regardless of how we assign registers to y and z, there will be a free register for x. In other words, x is locally colorable, and by symmetry, the same goes for y. Now consider z. If we assign R0 to x, and R2 to y, then we block both W0 and W1, so there is no free register for z, which is therefore not locally colorable.

All three nodes in the example graph have $degree = 2$, but only two of them are locally colorable. Consequently, the $degree < k$ test is not an accurate indication of local colorability in the generalized framework.

If we can not use the $degree < k$ test, what can we use instead? The definition of local colorability in Section 3.3 suggests a test based on enumerating all possible assignments of registers to the neighbors, and checking whether there is a free register in each and every one of them. Since the number of possible assignments is exponential in the number of neighbors, we expect that such a test would be too expensive to use in practice. What we need is an inexpensive test

which safely approximates local colorability.

4.1 The $\langle p, q \rangle$ test

We propose the following approximation of the local colorability test. Given a machine description $\langle Regs, Conflict, Classes \rangle$, let $p : Classes \rightarrow \mathbb{N}$ and $q : Classes \times Classes \rightarrow \mathbb{N}$, written p_B and $q_{B,C}$, respectively, be two functions defined for all classes B and C by

$$p_B = |B|$$
$$q_{B,C} = \max_{r_C \in C} |\{r_B \in B | (r_C, r_B) \in Conflict\}|$$

In other words, p_B is the number of registers in the class B, and $q_{B,C}$ is the largest number of registers in B that a single register from C can conflict with.

The $\langle p, q \rangle$ *test* for a node n in a generalized interference graph $\langle N, E, class \rangle$, with $class(n) = B$, is

$$\sum_{\substack{(n,j) \in E \\ C = class(j)}} q_{B,C} < p_B.$$

The $\langle p, q \rangle$ test consists of a sum over all the neighbors of n, where each neighbor j contributes an amount which depends on both $class(n)$ and $class(j)$. This sum is then compared to a limit which depends on $class(n)$. The test is true if the sum is less than the limit.

In Section 4.2, we prove that the $\langle p, q \rangle$ test is a safe approximation of local colorability in a generalized interference graph, for any machine description.

For a regular architecture with a single class of k non-overlapping registers, we get $p = k$ and $q = 1$, which means that the $\langle p, q \rangle$ test degenerates to the standard $degree < k$ test. Because the $degree < k$ test is precise in the regular case, we conclude that any imprecision in the $\langle p, q \rangle$ test is induced only by the irregular features of the architecture.

Since p and q are fixed for a given machine description, they can be pre-computed and stored in static lookup tables. This makes it possible to evaluate the $\langle p, q \rangle$ test with the same time complexity as the $degree < k$ test.

A straight-forward implementation of a program to compute the p and q values for a given architecture would have a time complexity of $\mathcal{O}(K^2 R^2)$ where K is the number of register classes, and R is the number of registers. This is completely acceptable performance for an off-line register allocator generator tool.

4.2 Proof of Safety

In this section, we prove that the $\langle p, q \rangle$ test is a safe approximation of local colorability in a generalized interference graph.

In all the following, we assume a machine description $\langle Regs, Conflict, Classes \rangle$. We must prove that the $\langle p, q \rangle$ test implies local colorability. It turns out to be easier to show the reverse statement, that in any graph, if a node is not locally colorable, then the $\langle p, q \rangle$ test for that node is false.

First, let $G = \langle N, E, class \rangle$ be an generalized interference graph, and $n \in N$ be a node which is not locally colorable in G. Let $B = class(n)$, and $J = \{j | (n, j) \in E\}$ be the set of neighbors of n in G. Since n is not locally colorable, there must exist an assignment A for J in G such that for all registers $r_B \in B$, there exists a node $j \in J$ where $(A(j), r_B)$ is in $Conflict$.

This allows us to express B as follows.

$$B = \bigcup_{j \in J} \{r_B | (A(j), r_B) \in Conflict\}$$

By definition, $p_B = |B|$, so we have

$$p_B = |B| = |\bigcup_{j \in J} \{r_B | (A(j), r_B) \in Conflict\}|$$

Now, the size of a union of sets is less than or equal to the sum of the sizes of the individual sets, so we can limit the size of the big union as follows.

$$p_B \leq \sum_{j \in J} |\{r_B | (A(j), r_B) \in Conflict\}|$$

But, for any node j, the number of registers in B in conflict with $A(j)$ can not be more than the maximum number of registers from B in conflict with any register from $class(j)$.

$$p_B \leq \sum_{\substack{j \in J \\ C = class(j)}} \max_{r_C \in C} |\{r_B | (r_C, r_B) \in Conflict\}|$$

By the definition of $q_{B,C}$, we now get

$$p_B \leq \sum_{\substack{j \in J \\ C = class(j)}} q_{B,C}$$

From this follows directly that the $\langle p, q \rangle$ test is false, and that the $\langle p, q \rangle$ test implies local colorability. This proves that the $\langle p, q \rangle$ test is in fact a safe approximation of local colorability. Since we made no specific assumptions about the machine description, the proof is valid for any architecture that can be modelled with a machine description.

5 Putting it all together: The complete register allocation algorithm

For reasons of simplicity, we present the complete register algorithm without coalescing or optimistic coloring. See Section 6 for a discussion of these and other extensions to the algorithm.

Given a machine description

$$\langle Regs, Conflict, Classes \rangle$$

where the target architecture is modelled as a set of registers, a conflict relation between the registers, and a set of register classes, we use the formulae in Section 4.1 to pre-compute p_B and $q_{B,C}$ for all classes B and C. This is simple enough to do manually, but it could also be done by an automatic tool.

The generalized algorithm has the same four phases as the algorithm in Section 2.

1. *Build* constructs the generalized interference graph $\langle N, E, class \rangle$ with a set of nodes N, one for each variable, and with edges in E between variables which are simultaneously live. It also annotates the nodes with register classes based on the operations which the variables are involved in, giving $class$.

2. *Simplify* initializes an empty stack, and then repeatedly removes nodes from the graph which satisfy the $\langle p, q \rangle$ test, that is, when

$$\sum_{\substack{(n,j)\in E \\ C=class(j)}} q_{B,C} < p_B$$

   for a node n of class B. Each node which is removed is pushed on the stack.

   This continues until either the graph is empty, in which case the algorithm proceeds to *Select*, or there are no more nodes in the graph which satisfy the test. In that case, *Simplify* has failed, and we go to the *Spill* phase.

3. *Select* rebuilds the graph by re-inserting the nodes in the opposite order to which *Simplify* removed them. Each time a node n is popped from the stack, it is assigned a register r from $class(n)$ such that r does not conflict with the registers assigned to any of the neighbors of n.

   When Select finishes, it has produced a complete register allocation for the input program, and the algorithm terminates.

4. *Spill* is invoked if *Simplify* fails to remove all nodes in the graph. It picks one of the remaining nodes to spill, and inserts a load before each use of the variable, and a store after each definition. (If the node's class is a subset of another class, one could also replace the loads and stores by moves to a variable from the more general class.) After the program is re-written, the algorithm is re-started from the *Build* phase.

Select always finds a free register for each node, because the the $\langle p, q \rangle$ test in *Simplify* guarantees that the node was locally colorable in the graph which it was removed from, and the use of a stack guarantees that this is the same graph which it is re-inserted into.

5.1 A simple example

As a simple example, we will run the generalized algorithm on the problem in Figure 2. A larger and perhaps more realistic example is presented in Section 7.1.

Based on the machine description illustrated in Figure 2(a), we compute the following parameters: $p_A = 4$, $p_B = 2$, $q_{A,A} = 1$, $q_{A,B} = 2$, $q_{B,A} = 1$, $q_{B,B} = 1$. Computing the $\langle p, q \rangle$ test for all the nodes in Figure 2(b) we see that it is true for x and y (the sum of q:s is 3 which is less than $p_A = 4$), but not for z (the sum, 2, is not less than $p_B = 2$). Recall that x and y are locally colorable in this graph, but z is not, so in this case the $\langle p, q \rangle$ test is precise.

We pick one of the colorable nodes, x, remove it from the graph, and push it on the stack. In the resulting simplified graph, the $\langle p, q \rangle$ test is true not just for y, but for z as well. We therefore remove y and z from the graph, and proceed to the *Select* phase.

The first node to be popped is z. None of z's neighbors have been inserted in the graph yet, so we only have to worry about picking a node from the correct register class. Out of the class B, we select register $W0$ for z. The next node to be popped is y. Since y interferes with z, we can not assign registers $R0$ or $R1$ to it, because these registers conflict with $W0$. Therefore, we select $R2$ for y. Finally, we reinsert x into the graph. The only register available for x is $R3$.

6 Extensions

Previous work on register allocation for regular architectures has resulted in a number of extensions which can be added to the original algorithm to give

significant improvements in the quality of the optimized code. Although we have not yet considered all the popular extensions, we are confident that it will be possible to adapt them to the generalized framework. We discuss how to adapt three important extensions, optimistic coloring, a standard spill heuristic, and conservative coalescing. We also propose an extended test for local colorability, which can increase the precision of the approximation when there are large numbers of registers from small register classes.

6.1 Optimistic coloring

Briggs et al. [5] noted that the *Simplify* phase of Chaitin's algorithm can sometimes be overly conservative, and perform unnecessary spills. Even if a node is not locally colorable, it may still be possible to color it in practice if two or more neighbors get the same color. Briggs suggested that if *Simplify* spills one or more nodes, *Select* could still try to color the nodes in reverse order of removal. If the coloring succeeds, we have obtained a register allocation without spilling. Only if it fails is the Spill phase invoked to place spilled nodes in memory.

It is straight-forward to incorporate optimistic coloring in the generalized algorithm. The modifications to the original algorithm are completely orthogonal to the changes required for optimistic coloring.

The optimistic coloring algorithm never spills more than Chaitin's algorithm, but sometimes less [5]. Since embedded processors often have few registers, optimistic coloring is likely to give improved allocation quality when used with the generalized algorithm.

6.2 Spill heuristic

If the *Simplify* phase can not remove all the nodes in the interference graph using the local colorability test, we must pick one node to spill (or remove optimistically). One of the simpler heuristics used to guide this choice is as follows. Let *cost(n)* be an estimate of the cost of spilling the variable represented by node n. This estimate is computed based on the number of loads and stores that would be necessary in order to spill n. Each load or store is furthermore weighted by its loop-nesting depth (or some other measure of expected execution frequency). For a given graph, we pick the node with the smallest ratio

$$cost(n)/degree(n).$$

The degree measures the current *benefit* of removing n from the graph. Removing a node with high degree is assumed to be preferable, on the grounds that it will reduce the degree of many neighboring nodes, thus making the resulting graph easier to color than if a node with low degree had been removed.

How does this heuristic transfer to the generalized framework? The *cost* function must take into account the varying costs of loading or storing values from different register classes, but is otherwise unchanged. However, the *degree* of a node is no longer a good measurement of the benefit of removing that node from the graph. The following formula can be used instead, where the class of n is B.

$$benefit = \sum_{\substack{(n,j)\in E \\ C=class(j)}} (q_{C,B} \ / \ p_C)$$

For each neighbor j, this computes how much n would contribute to the sum in the $\langle p, q\rangle$ test, divided by the limit in that test. The division allows us to compare the benefit for neighbors from different register classes.

6.3 Coalescing

Chaitin et al. [6] noted that if two variables do not interfere, and there is a copy instruction from one variable to the other, then the copy instruction can be eliminated if the two variables are allocated to the same register. He included a *Coalescing* phase after the *Build* phase, which merges non-interfering copy-related nodes in the graph.

In the generalized framework, coalescing two nodes n and m must take into account the register classes of the two nodes. The resulting merged node must fulfill the restrictions of both n and m, and so its register class must be (a subset of) the intersection of the classes for n and m.

As noted by Briggs [5], Chaitin's coalescing may sometimes cause spills. This can happen, for example, when a variable that could be allocated in a register is merged with a variable that needs to be spilled. Different *conservative* coalescing heuristics have been proposed to ensure that two variables are only coalesced if the resulting variable is guaranteed not to be spilled [5, 7].

The conservative coalescing heuristics usually involve one or more local colorability tests, expressed as $degree < k$ tests. For example, coalescing may be restricted to nodes that are locally colorable when only neighbors which are not locally colorable are considered. In the generalized allocator, the $degree < k$ tests can simply be replaced by $\langle p, q\rangle$ tests.

6.4 The $\langle p, q, b \rangle$ test

Many irregular architectures will require machine descriptions with some very small register classes, sometimes with only one or two members. If we use register classes to specify which physical registers to use for parameter passing, even a more regular architecture could end up having a large number of singleton classes in its machine description.

For a node from a large register class which has many neighbors from the same small class, the $\langle p, q \rangle$ test can be very imprecise. For example, say that class B has 16 registers, and class C only 2, and that $q_{B,C} = 1$. If a node of class B has 16 neighbors from class C, then it will fail the $\langle p, q \rangle$ test. However, since there are only two registers to chose from in class C, those 16 neighbors can never block more than two registers from class B.

We can extend the $\langle p, q \rangle$ test to take this into account. Let $b_{B,C}$ be the maximum contribution from class C to class B.

$$b_{B,C} = |\{r_B \in B | \exists r_C \in C.(r_C, r_B) \in \mathit{Conflict}\}|$$

The maximum contribution is simply the number of registers in B that can be blocked by any register in C. Now, define the $\langle p, q, b \rangle$ *test* for a node n in a graph $\langle N, E, \mathit{class} \rangle$, where $\mathit{class}(n) = B$, to be

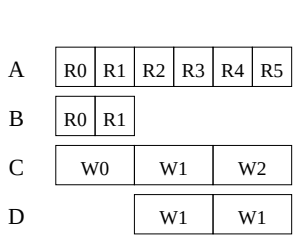
$$(\sum_{C \in Classes} min(b_{B,C}, (\sum_{\substack{(n,j) \in E \\ class(j) = C}} q_{B,C}))) < p_B.$$

The test is exactly the same as the $\langle p, q \rangle$ test, except that we make sure that the total contribution from neighbors of a particular class is never more than the maximum contribution from that class.

For the example given above, the 16 neighbors from class C would only contribute 2 to the sum in the $\langle p, q, b \rangle$ test, so the B-node would easily pass the test. In this particular case, the extended test has much better precision than the ordinary $\langle p, q \rangle$ test.

7 Examples

In this section, we first present a more extensive example of the generalized algorithm in action. Then we give an example of where our approximation of the local colorability test is more precise than the corresponding test in a solution suggested by Briggs for handling register pairs. Finally, we show how the Atmel AVR architecture would be modelled in the generalized framework.



q	A	B	C	D	p
A	1	1	2	2	6
B	1	1	2	0	2
C	1	1	1	1	3
D	1	0	1	1	2

Figure 3: Machine description for the example in Section 7.1.

7.1 A larger example

Consider the machine description on the left in Figure 3. (Refer to Section 4 for instructions on how to interpret such diagrams.) It describes a fictional architecture with six 8-bit registers R0-R5 (class A), which can form aligned pairs W0-W2 (class C). The 8-bit multiplication operation puts its result in either R0 or R1 (class B). Indirect addressing is only allowed on registers W1 and W2 (class D). Figure 3 also gives the p and q values computed from the machine description for use in the $\langle p, q \rangle$ test. The value for $q_{X,Y}$ is found by intersecting the row for X with the column for Y.

We will try to allocate registers for the following code, where the variables are annotated with register classes.

```
        x0:A = 0
        x1:C = 0
10:     x1:C >= 42 ?  l1
        x2:D = x1:A + foo
        x3:A = [x2:D]
        x4:D = x1:A + bar
        x5:A = [x4:D]
        x6:B = x3:A * x5:A
        x0:A = x0:A + x6:D
        goto l0
l1:     output x0:A
```

Variable x6 is the result of a multiplication, so it must be restricted to class B. Both x2 and x4 are used in indirect addressing, and so are constrained to class D. We have computed basic spill costs for each variable as described in Section 6.2. The spill cost for each variable is given in the *cost* column in Table 1.

The first phase of the algorithm constructs a generalized interference graph with nodes for each variable, and edges between simultaneously live variables. Figure 4(a) presents the initial graph for this example.

Next, we begin the simplification. After the name for each node we write the result of the sum in the $\langle p, q \rangle$ test, divided by the p value for the class of that

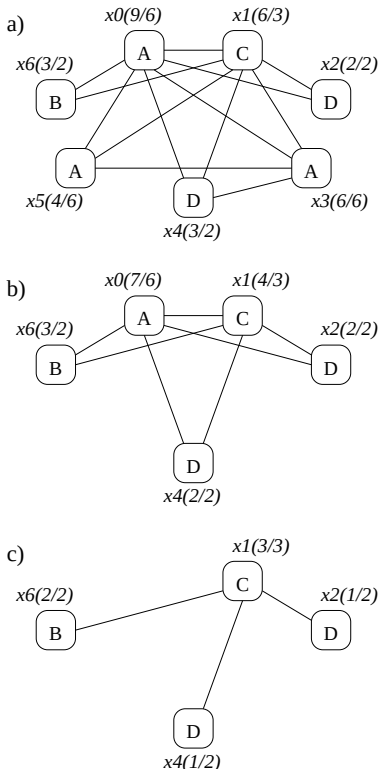


Figure 4: Simplifying a generalized interference graph (Section 7.1).

node. This way, if the resulting fraction is less than one, the node passes the test and can be removed.

In the initial graph, only x5 passes the test. We remove it and place it on the stack. When x5 is removed, the sum for x3 drops below its limit, and it can also be removed. The resulting graph is shown in Figure 4(b).

At this point, none of the nodes in the graph satisfy the $\langle p, q \rangle$ test. Using the spilling heuristic in Section 4, we find that the best node to spill is x0. Instead of immediately proceeding to the Spill phase, rewriting the code to keep x0 in memory, we optimistically remove it and push it on the stack, in the hope that a free register will be available for it when we reach the Select phase.

The graph that results from the removal of x0 is shown in Figure 4(c). We can immediately remove x2 and x4. After that, x1 becomes colorable, and removing x1 we find that the last node, x6, can also be removed. This concludes the Simplify stage.

Now we begin selecting registers for the nodes. The top four nodes on the stack are x6, x1, x4 and x2. Taking each one in turn, we assign the following registers: x6:R0, x1:W1, x4:W2, and x2:W2.

variable	*cost*	*register*
x0	22	R1
x1	42	W1
x2	40	W2
x3	20	R0
x4	40	W2
x5	20	R4
x6	20	R0

Table 1: Spill costs and register assigments for variables from the example in Section 7.1.

After the first four nodes are reinserted and assigned registers, we get to the optimistically removed x0. It interferes with all the nodes inserted previously. Since x4 and x2 were assigned the same register, we find that there is in fact a free register for x0, namely R1.

The rest of the nodes on the stack were all removed after having passed the $\langle p, q \rangle$ test. The algorithm guarantees that there will be free registers for them when they are re-inserted, regardless of how Select has assigned registers to the previously inserted nodes. The final assignment is shown in the *register* column of Table 1.

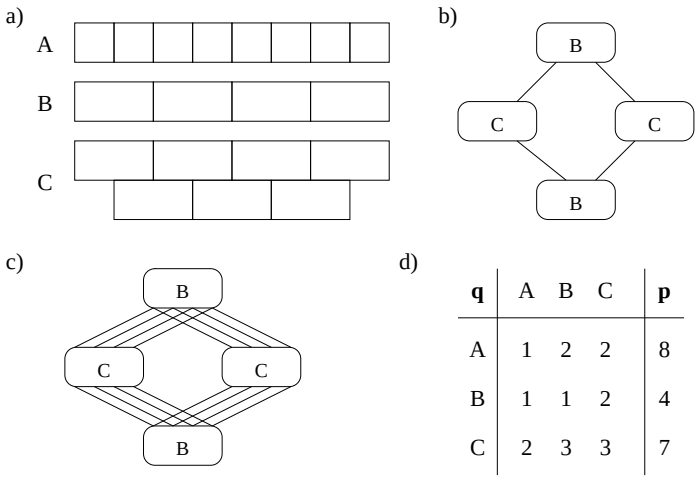
7.2 Mixing aligned and un-aligned pairs

This example shows a case where the $\langle p, q \rangle$ test is a more precise approximation of local colorability than the solution proposed by Briggs et al. in [4].

Consider the machine description in Figure 5(a). When aligned pairs (class B) and un-aligned pairs (class C) are mixed, Briggs' solution requires four edges between a B and a C node. To see this, note that two registers from C can block the allocation of a single register from B. A B-node adjacent to two C-nodes must have a *degree* of at least eight to prevent it from being removed from the graph by Simplify, hence the need for four edges.

The problem comes when we want to simplify a node with class C. If such a node has two neighbors of class B, then it will also have $degree = 8$. However, no matter how we allocate the two B:s, there will always be a free register for C. (In fact, there will be two free registers.) Because of this imprecision, Briggs' solution fails to color the example graph in Figure 5(c).

Our approach uses the parameters in Figure 5(d) to model the same machine. When a node of class B is considered, each neighbor from class C contributes



q	A	B	C	p
A	1	2	2	8
B	1	1	2	4
C	2	3	3	7

Figure 5: Mixing aligned and un-aligned register pairs (Section 5).

$q_{B,C} = 2$ to the sum in the $\langle p, q \rangle$ test. The limit $p_B = 4$ ensures that the test fails for a B-node with two neighbors of class C. On the other hand, in the test for a node of class C, the B-nodes contribute $q_{C,B} = 3$. Since $p_C = 7$, the $\langle p, q \rangle$ test for a C-node with two B-node neighbors will succeed. This allows the graph in Figure 5(b) to be simplified in our framework.

Finally, note that even the $\langle p, q \rangle$ test is somewhat imprecise in this case, since it will not allow a C-node with *three* B-node neighbors to be removed.

7.3 Modelling the Atmel AVR

In the final example, we show a machine description modelling an existing irregular architecture, and the resulting parameters for the approximate colorability test. To simplify the example, we have not considered constraints that may be imposed by the run-time system.

The Atmel AVR [2], illustrated in Figure 6, is a modern 8-bit RISC-like processor which is widely used in the embedded systems market. The AVR has 32 8-bit registers. There are a number of instructions that operate on 16-bit quantities and require their operands in aligned register pairs, so the machine description will have a register class corresponding to the register pairs. Some instructions operate only on a subset of all registers. Such instructions induce a register class in the machine description, thus, we can determine the register classes by looking at the various instructions and the registers they operate on.

First, we have a basic class R consisting of all 32 8-bit registers. Only the upper 16 of these can be used for register-immediate arithmetic, giving us the class *Ri*. There is a 16-bit copy instruction which operates on any aligned register pair, giving the class *W*. Only the top four pairs can be used for 16-bit register-immediate arithmetic, hence the class *Wi*. The top three pairs (called X, Y and Z) also serve as address registers and can used for indirect addressing, giving the class *XYZ*). However, indirect addressing with an offset is only allowed with register pairs *Y* and *Z*, thus class *YZ*. The Z register alone is capable of indirect addressing in the separate code memory space, so we need a singleton class for the Z register. Finally, the 16 bit result of an 8 bit multiply ends up in the first register pair, thus the class *Wm*.

In Table 2, we give the p and q values derived from the machine description for the AVR architecture. These values are used to define the $\langle p, q \rangle$ test used in the *Simplify* phase of the generalized allocation algorithm. The value for $q_{B,C}$ is located in the row labled B and the column labled C. For example, $q_{R,Wi} = 2$, and $q_{Wi,R} = 1$.

8 Related work

Briggs provides an overview of the history of graph-coloring register allocation in his thesis [3].

Starting with Chaitin, the most popular approach to dealing with architectural irregularities has been to modify the interference graph to properly represent the additional constraints [6, 5, 4, 7]. Briggs writes in [5]:

> An attractive feature of Chaitin's approach is that machine-specific constraints on reg-

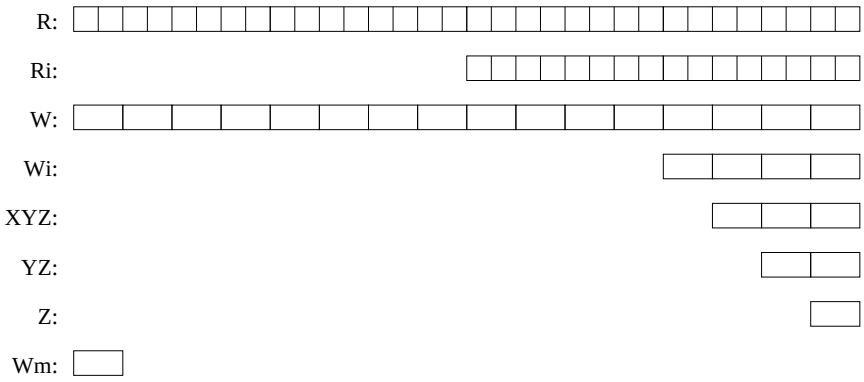


Figure 6: Machine description for the AVR architecture

p	
R	32
Ri	16
W	16
Wi	4
XYZ	3
YZ	2
Z	1
Wm	1

q	R	Ri	W	Wi	XYZ	YZ	Z	Wm
R	1	1	2	2	2	2	2	2
Ri	1	1	2	2	2	2	2	0
W	1	1	1	1	1	1	1	1
Wi	1	1	1	1	1	1	1	0
XYZ	1	1	1	1	1	1	1	0
YZ	1	1	1	1	1	1	1	0
Z	1	1	1	1	1	1	1	0
Wm	1	0	1	0	0	0	0	1

Table 2: The p and q tables for the AVR architecture

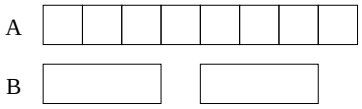


Figure 7: Quad-aligned triples.

> ister use can be represented directly in the graph ... Thus, the graph represents both the constraints embodied in the program and those presented by the target architecture in a single, unified structure. This is one of the key insights underlying graph-coloring allocators: The interference graph represents *all* of the constraints.

Our approach is fundamentally different, in that it separates the constraints of the program from those of the architecture and run-time system into different structures. Instead of modifying the interference graph to represent the additional constraints, we change the *interpretation* of the graph based on the machine description for the target architecture. We believe that our approach leads to a simpler and more intuitive algorithm, which avoids increasing the size of the interference graphs before simplification, and where expensive calculations relating to architectural constraints can be performed off-line.

As shown in Section 7.2, there are cases where our algorithm is more precise than the approach proposed by Briggs for handling register pairs [4]. Furthermore, although Briggs' approach can be made to work for both aligned and unaligned register pairs, there are only vague rules for determining how many additional edges are needed between nodes from different register classes in the general case ("add enough edges"). In contrast, the formulae in Section 4 immediately produce parameters that make the $\langle p, q \rangle$ test a safe approximation of local colorability for any machine description.

The scheme proposed by Smith and Holloway [12] is more similar to ours, in that it also leaves the interference graph (largely) unchanged. Their interpretation of the graph is based on assigning class-dependent weights to each node. Rules for assigning weights are given for three common classes of irregular architectures, but no general rules are presented. The resulting colorability test is equivalent to ours in many cases, but is less precise in the case of "quad-aligned triples": For the machine description in figure 7, the formulae in Section 4 give $p_A = 8$, $p_B = 2$, $q_{A,A} = 1$, $q_{A,B} = 3$, $q_{B,A} = 1$ and $q_{B,B} = 1$. These constants accurately reflect the fact that two registers from class B can conflict with at most six registers from class A. Smith and Holloway, however, are forced to equate the tripples with quadruples in order to capture the alignment restrictions, and so are unable to remove an A-node with two B-node neighbors.

Beacuse of the way Smith and Holloway's weight-based test is formulated, two neighboring nodes from disjoint register classes will still affect the local colorability of each other. To avoid this over-approximation, interferece edges between nodes from disjoint classes are removed from the graph in a separate pass before simplification. In our scheme, two disjoint classes A and B simply have $q_{A,B} = q_{B,A} = 0$, so there is no need for special-case modifications of the graph.

There have been some attempts to use precise methods instead of heuristics for register allocation. Goodwin and Wilken [8] formulate the regular graph coloring problem as a 0-1 integer linear programming (ILP) problem, which is optimally solved by a generic solver. Kong and Wilken [10] extend this work to cover some irregular features of the x86 architecture. Appel and George [1] propose a hybrid approach which first makes the spill decisions using ILP, and then allocates the remaining variables to registers using a variant of graph coloring called "optimistic coalescing". While these methods produce allocations of very high quality, they still seem to be too slow to be useful in practice.

Scholz and Eckstein [11] have recently described a new technique based on expressing the register allocation as a boolean quadratic problem, which is solved heuristically. The range of architectures which can be handled by their technique is slightly larger than what can be represented by our machine descriptions. In particular, so called pairing constraints are allowed, for example that index register N0 can only be used with address register A0 or A1, and N1 only with A2 or A3. Another appealing feature of Scholz and Eckstein's technique is that cost irregularities, eg. that it may be cheaper to use R0-R7 than R8-R15 in a particular instruction, are taken into account during allocation.

An anonymous reviewer of an earlier version of this paper claimed that techniques similar to ours are "widely practised" in commercial compilers. To the best of our knowledge, these techniques have never been described in the literature, so it is impossible for us to give a comparison.

9 Conclusions and Future Work

We have introduced a class of machine descriptions, and generalized Chaitin's graph-coloring register allocation algorithm so that it takes into account the architectural constraints expressed in such a description without having to modify the interference graph given by the program being compiled. To make the resulting algorithm practical, we have developed a safe approximation to the local colorability test, whose parameters can be derived automatically from the machine description. Because the allocator makes no assumptions about the target architecture, it can be used with arbitrary regular or irregular architectures.

A modern graph-coloring register allocator should include a number of extensions to the basic algorithm, like coalescing and optimistic coloring. The modifications that we propose are largely orthogonal to these extensions, and it should be easy to combine them with our generalized allocation algorithm.

We intend to implement a parameterized register allocator based on the generalized framework, and an automatic tool capable of generating suitable parameters for any given architecture. Together, these two could constitute the core of a completely re-targetable register allocator. In the context of this work we will also include several extensions to the algorithm, including those discussed in Section 6. Another upcoming project is to compare the graph coloring allocator with a precise allocator based on integer linear programming [10].

10 Acknowledgements

This work was conducted within the WPO project, a part of the ASTEC competence centre.

Johan Runeson is an industrial Ph.D. student at Uppsala University and IAR Systems.

References

[1] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, 2001.

[2] Atmel Corporation. The AVR instruction set. `http://www.atmel.com/atmel/acrobat/doc0856.pdf`.

[3] Preston Briggs. *Register allocation via graph coloring.* PhD thesis, Rice University, April 1992.

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

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

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1982.

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

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

[9] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 1995.

[10] T. Kong and K. D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

[11] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES-SCOPES*, 2002.

[12] Michael D. Smith and Glenn Holloway. Graph-coloring register allocation for architectures with irregular register resources. Unpublished manuscript, `www.eecs.harvard.edu/machsuif/publications/publications.html`, 2001.