Retargetable Graph-Coloring Register Allocation for Irregular Architectures

Johan Runeson and Sven-Olof Nyström

Department of Information Technology
Uppsala University
{jruneson,svenolof}@csd.uu.se

Abstract. Global register allocation is one of the most important optimizations in a compiler. Since the early 80's, register allocation by graph coloring has been the dominant approach. The traditional formulation of graph-coloring register allocation implicitly assumes a single bank of non-overlapping general-purpose registers and does not handle irregular architectural features like overlapping register pairs, special purpose registers, and multiple register banks. We present a generalization of graph-coloring register allocation that can handle all such irregularities. The algorithm is parameterized on a formal target description, allowing fully automatic retargeting. We report on experiments conducted with a prototype implementation in a framework based on a commercial compiler.

1 Introduction

Embedded applications are growing larger and more complex, often reaching more than 100.000 lines of C code. To develop and maintain such an application requires a fast compiler. However, due to constraints on memory space, power consumption and other system resources, the compiler must also produce high-quality code. State-of-the-art optimization techniques from high-end RISC compilers are not always applicable, because embedded processor architectures are often irregular. Furthermore, the large number of different architectures means the compiler techniques must also be retargetable.

In this paper we focus on global register allocation, one of the most important transformations in a modern optimizing compiler [1] (page 92). For RISC-machines, Chaitin-style graph-coloring [2] is the dominant approach, as witnessed by its prominence in modern compiler construction textbooks [3–5]. It gives high-quality allocations, runs fast in practice, and is supported by a large body of research work (e.g. [6, 7]). Unfortunately, the algorithm assumes a *regular* register architecture consisting of a single, homogenous set of general-purpose registers.

We propose a generalization of Chaitin's algorithm which allows it to be used with a wide range of *irregular* architectures, featuring for example register pairs or other clusters, and non-orthogonal constraints on the operands of certain instructions. The generalized algorithm is parameterized by an expressive formal description of the register architecture, allowing fully automatic retargeting. It has the same time complexity as the original algorithm and is provably correct for any applicable architecture. The changes compared to the original algorithm are modest, so most existing improvements and extensions can be incorporated with little or no work.

2 Background

We assume that the register allocator is presented with low-level intermediate code, where the instructions correspond to target assembly language instructions, but where *variables* (taken from an unlimited set of names) are used instead of registers.

The goal of register allocation is to determine where to store each variable — in a particular register or in memory — in the most cost-effective way, and to rewrite the program to reflect these decisions. *Local* register allocation works in the scope of a single basic block. *Global* register allocation considers a whole function at a time.

Register allocation for a regular architecture can be formulated as a graph-coloring problem. A variable is *live* if it holds a value which may be used later in the program. Two variables which are live simultaneously are said to *interfere*, since they can not use the same register resources. Using liveness analysis, an *interference graph* can be built, where each node represents a variable, and where there is an edge between two nodes if their variables interfere. A *k-coloring* of a graph is an assignment of one of at most k colors to each node, such that no two neighbors have the same color. For a regular architecture with k registers, a k-coloring of the interference graph represents a solution to the register allocation problem, where all nodes with the same color share the same register.

Graph coloring is known to be an NP-complete problem, so heuristic techniques are used to perform register allocation in practice. Chaitin et al. [2] presented the first heuristic global register allocation algorithm based on graph coloring. Although it has a worst-case time complexity of $O(n^2)$, experiments in [6] indicate that in practice it runs in less than $O(n \log n)$ time. Due to space limitations, we can not give the full algorithm here. For the interested reader, we refer to the description by Briggs [6], or the more elaborate presentation in our technical report [8].

3 Retargetability through Parameterization

In modern retargetable compilers, *target descriptions* are often used to parameterize code generation and optimization passes in order to achieve retargetability [9, 10]. We use the same approach for our register allocator. For simplicity, our target descriptions deal only with architectural features that affect register allocation. They can easily be incorporated in or derived from more extensive target descriptions.

In Chaitin's algorithm, the target is characterized only by the number of registers, k. It is assumed that the architecture is regular, i.e. that all registers are interchangeable in every situation. This assumption does not hold for irregular architectures. In our generalized algorithm, the target is characterized by an expressive *target model*, defined below, which allows features like overlapping register pairs, special purpose registers, and multiple register banks to be described. No further assumptions are made, so any architecture which can be described by a target model is applicable.

3.1 Target Models

We define a *target model* to be a tuple $\langle \mathit{Regs}, \mathit{Conflict}, \mathit{Classes} \rangle$, 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 register 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 $R \subseteq \mathit{Regs}$, 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 those operations.

As an example, consider a simple architecture with four basic registers `R0`–`R3`, which some instructions use as pairs `W0` = `R0:R1` and `W1` = `R2:R3`. In the target model for this architecture, *Regs* is the set $\{\mathtt{R0}, \mathtt{R1}, \mathtt{R2}, \mathtt{R3}, \mathtt{W0}, \mathtt{W1}\}$. The *Conflict* relation is defined so that each register in *Regs* conflicts with itself, and the pairs conflict with their components: W0 with R0 and R1, and W1 with R2 and R3, respectively. We define two register classes A and B, where A is $\{\texttt{R0}, \texttt{R1}, \texttt{R2}, \texttt{R3}\}$ and B is $\{\texttt{W0}, \texttt{W1}\}$. These two classes make up the set *Classes*.

The diagram in Fig. 1(a) illustrates this target model. Each box is a register, and each row gives the name and members of one register class. Furthermore, the boxes are arranged so that two registers conflict if they appear in the same column. More examples of target models can be found in Sect. 6, and in [8].

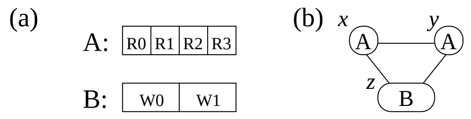

Fig. 1. A simple example: (a) target model diagram, (b) generalized interference graph

3.2 Generalized Interference Graphs

For a given target model 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 the function $class : N \rightarrow Classes$ maps each node to a register class. The nodes in N correspond to variables, and there is an edge in E between two nodes if their variables are simultaneously live at some point in the program.

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

Given a target model and a generalized interference graph, the register allocation problem reduces to the problem of finding a coloring for the graph.

Register allocation for regular architectures is a special case of the more general problem, with a target model consisting of a single class of k registers and an identity conflict relation. It follows that the problem of finding a coloring for a generalized interference graph is NP-hard.

Figure 1(b) shows a generalized interference graph under the target model in (a). The nodes x, y and z are annotated with register classes (A, A, and B, respectively), and from the interference edges we can see that the variables corresponding to the nodes are all live simultaneously.

4 Local Colorability

Chaitin's graph-coloring algorithm is based on a concept which we call *local colorability*[1]. In a generalized interference graph $\langle N, E, class\rangle$, a node $n \in N$ is

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

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 neighbor of n.

The coloring problem can be simplified by removing a node n which is locally colorable: 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 we can recursively simplify the graph until it is empty, then by induction it is possible to construct a coloring by assigning colors to the nodes in the reverse order from which they were removed.

4.1 Approximating Colorability

In a regular architecture with k registers, a node n is locally colorable iff it has less than k neighbors in the interference graph. Chaitin's algorithm therefore removes nodes with $degree < k$.

For irregular architectures, the $degree < k$ test is not always a good indicator of local colorability. Consider the example in Fig. 1. It is easy to see that regardless of how we assign registers to y and z, there is always 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 there is no free register for z, which is therefore not locally colorable.

All three nodes in the example 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 this case.

If we can not use the $degree < k$ test, what can we use instead? The definition of local colorability suggests a test based on generating and checking all possible assignments of registers to the neighbors of a node. Since there is an exponential number of possible assignments, we expect that such a test would be too expensive to use in practice.

Fortunately, the coloring algorithm does not require a precise test for local colorability. In order to guarantee that it is possible to color the nodes in the reverse order from which they were removed from the graph, it is enough if the test implies local colorability. What we need is therefore an inexpensive test which safely approximates local colorability with minimal inaccuracy.

4.2 The $\langle p, q \rangle$ Test

We propose the following approximation of the local colorability test. Given a target model as defined in Sect. 3.1, let p_B and $q_{B,C}$ be defined for all classes B and C by

$$\begin{aligned} p_B &= |B| \\ q_{B,C} &= \max_{r_C \in C} |\{r_B \in B | (r_B, r_C) \in \mathit{Conflict}\}| \end{aligned}$$

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.

A node n of class B in $\langle N, E, \mathit{class} \rangle$ is locally colorable if

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

We will call this the $\langle p, q \rangle$ *test*.

The intuition behind the $\langle p, q \rangle$ test is as follows. To begin with there are p_B registers available for assigning to n. Each neighbor may block some of these registers. In the worst case, a neighbor from class C can block $q_{B,C}$ registers in B. If the sum of the maximum number of registers each neighbor can block is less than the number of available registers, then it is safe to say that we will be able to find a free register for n. In Sect. 4.3 we prove formally that the $\langle p, q \rangle$ test is a safe approximation of local colorability in any generalized interference graph, for any given target model.

The $\langle p, q \rangle$ test is efficient: Since p and q are fixed for a given target model, 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 $\mathit{degree} < k$ test.

For a regular architecture with k registers, we get $p = k$ and $q = 1$, which means that the $\langle p, q \rangle$ test degenerates to the precise $\mathit{degree} < k$ test. Any imprecision in the $\langle p, q \rangle$ test is thus induced only by certain irregular features of the architecture.

Note that for two disjoint register classes B and C, we get $q_{B,C} = 0$. Interference edges between nodes from disjoint classes therefore do not contribute to the sum in the $\langle p, q \rangle$ test. Also, for a self-overlapping class B (e.g. a class of unaligned pairs), $q_{B,B} > 1$, since a single register from B can conflict with both itself and one or more other registers in B.

4.3 Proof of Safety

We will show for a given target model $\langle \mathit{Regs}, \mathit{Conflict}, \mathit{Classes} \rangle$ that in any generalized interference graph $G = \langle N, E, \mathit{class} \rangle$, if a node is not locally colorable, then the $\langle p, q \rangle$ test for that node is false.

Let n be a node which is not locally colorable in G. Let B be the register class of n, and J the set of neighbors of n in G. Since n is not locally colorable, there must exist an assignment A of registers to the neighbors of n, such that for all registers r_B in B, r_B conflicts with $A(j)$ for some j in J.

This allows us to express B as follows.

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

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

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

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.

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

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 $class(j)$, which is exactly the definition of $q_{B,C}$.

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

Thus, if n is not locally colorable in G, then the $\langle p, q \rangle$ test for n is false. Conversely, if the $\langle p, q \rangle$ test is true, then n is locally colorable. This proves that the $\langle p, q \rangle$ test is a safe approximation of local colorability, for any graph in any target model.

5 The Complete Algorithm

For simplicity, we present the algorithm without coalescing and optimistic coloring. These extensions are discussed separately below.

Given a target model as in Sect. 3.1, we use the formulae in Sect. 4.2 to pre-compute p_B and $q_{B,C}$ for all classes B and C.

The algorithm is divided into four phases (Fig. 2).

1. *Build* constructs the generalized interference graph.
2. *Simplify* initializes an empty stack, and then repeatedly removes nodes from the graph which satisfy the $\langle p, q \rangle$ test. 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. After the program is rewritten, the algorithm is restarted from the Build phase.

Select always finds a free register for each node, because 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 it is reinserted into the same graph.

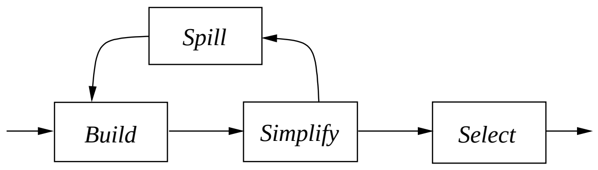

Fig. 2. Phases of the basic register allocation algorithm

In Chaitin's original algorithm, there are no register classes. Nodes are removed in Simplify when their *degree* $< k$, and in Select registers conflict only with themselves. Other than that, the algorithms are identical.

5.1 A Simple Example

As a simple example, we run the generalized algorithm on the problem in Fig. 1. Based on the target model illustrated in (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 of the graph in (b), we see that it is true for x and y, but not for z.

The fact that z is not locally colorable does not mean that it can not be colored – it just means that we should color it before some of its neighbors in order to guarantee that it will be colored. This is fine with the other two nodes: since they are locally colorable we know that we can always color them regardless of how we color z.

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, 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`.

5.2 Extensions

Optimistic coloring [6] is an important extension to Chaitin's algorithm, where spilling decisions are postponed from the Simplify to the Select phase: If Simplify can find no more locally colorable nodes, one node is picked to be removed anyway and pushed on the stack optimistically. When it is popped in Select, it may be possible to color it, for example if two neighbors have been assigned the same color. If so, there is no need to spill. Nodes which are popped later and which were locally colorable when pushed are still guaranteed to find a free color. Optimistic coloring often reduces the number of spills significantly, and can hide much of the imprecision of an approximating local colorability test [6]. It is completely orthogonal to the modifications presented here, and can (and should) be implemented just like in a regular graph coloring register allocator.

Another standard extension is *coalescing* [2], where copy-related non-interfering nodes are merged before the Simplify phase. If nodes n and n' are merged into m, then m must obey the constraints imposed on both n and n'. Therefore, it is given a register class from the intersection of the classes for n and n'. (If the intersection is empty, coalescing is not possible.)

Aggressive coalescing may sometimes cause unnecessary spills, when a node which is simple to color is merged with a node which is hard to color [6]. Therefore, *conservative coalescing* only merges two nodes if it can be guaranteed that the merged node will be locally colorable. It is straightforward to replace the $\mathit{degree} < k$ test with the $\langle p, q \rangle$ test to take register classes into account when doing this.

The *spill metric*, used to determine which node to pick for spilling, also deserves mention. It, too, should take register classes into account. We achieve this by picking the node with the smallest ratio

$$\mathit{cost}(n)/\mathit{benefit}(n).$$

However, rather than using $\mathit{degree}(n)$ as a measure of the benefit of removing that node, we define

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

Dividing $q_{C,B}$ by p_C allows us to compare the benefits for neighbors of different classes.

Figure 3 shows the phases of the register allocator when all the extensions described in this section are included. (The spill metric is used in the Simplify phase to determine which node to push optimistically on the stack.) Some further extensions are discussed in [8], including an alternative local colorability test which is slower, but has higher precision.

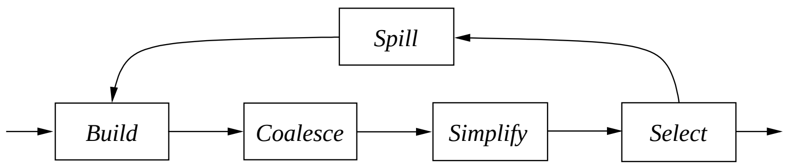

Fig. 3. Phases of the register allocation algorithm with extensions

6 Experiments

There are many factors besides register allocation which affect the quality of the code generated from a particular compiler. To make a fair comparison between different allocators, they must all be implemented in the same compiler. Often, though, there are strong dependencies between the allocator and the rest of the compiler, which could favour one allocator design unfairly over another. A good testbed for register allocation should strive to minimize such dependencies.

We have created a prototype framework for comparing different register allocators based on a commercial C/EC++ compiler from IAR Systems [11]. The framework short-circuits the existing allocator, which is closely tied to the code selection phase of the compiler. The allocator to be evaluated is inserted after the code selection phase, and presented with assembly code where the instruction operands contain virtual registers annotated with register classes. The new allocator is responsible for rewriting the code with physical registers and inserting spill code, after which regular compilation resumes.

Although the compiler is retargetable[2], incorporation of the prototype framework requires substantial changes in the target-dependent parts of the backend. Therefore, it currently only generates code for a single target: the Thumb mode of the ARM/Thumb architecture [12]. In ARM mode, the ARM/Thumb is a RISC-like 32-bit processor with 16 registers. In Thumb mode, a compressed instruction encoding is used, with 16-bit instructions. Most instructions in Thumb mode are two-address, and can only access the first 8 registers.

6.1 Implementation

The algorithm from Sect. 5, including optimistic coloring, conservative coalescing and the spill metric from Sect. 5.2, has been implemented in the prototype framework described above. Fig. 4 illustrates the target model that we use, derived from the register classes that the framework generates for us. These classes reflect constraints imposed both by the instruction set and by the runtime system. There are classes for 32-bit and 64-bit data (in unaligned pairs), for individual 32-bit and 64-bit values (used in the calling convention), a larger class of 32-bit registers which can sometimes be used for spilling to registers, and some classes of 96 and 128-bit values used for passing structs into functions. Registers `R13` and `R15` are dedicated by the runtime system. Registers `R8`–`R11` are too expensive to use profitably in Thumb mode. Table 1 shows the p and q values that we compute for the target model in Fig. 4. (The value of $q_{B,C}$ is located in the row for B and the column for C.)

We have implemented three different variants of the allocator.

1. *Full* is the full allocator described above, including the extensions from Sect. 5.2.

[2] Currently, IAR Systems supports over 30 different target architecture families with its suite of development tools.

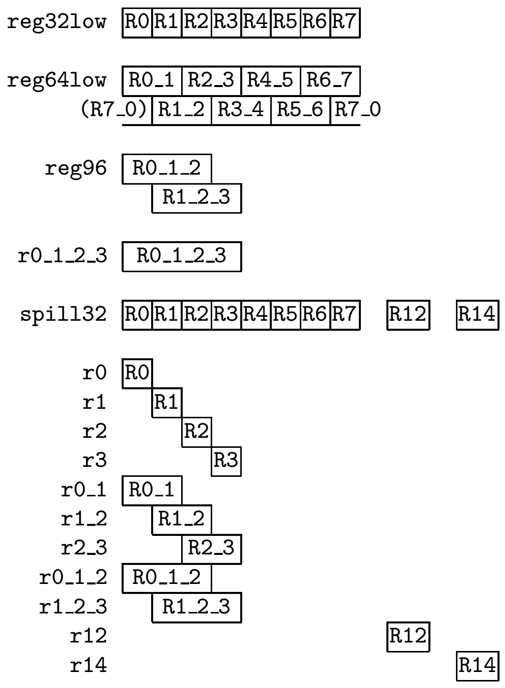

Fig. 4. Target model diagram for the Thumb architecture.

Table 1. Computed *p* and *q* values for Thumb.

class	p	reg32low	reg64low	reg96	r0_1_2_3	spill32	r0	r1	r2	r3	r0_1	r1_2	r2_3	r0_1_2	r1_2_3	r12	r14
reg32low	8	1	2	3	4	1	1	1	1	1	2	2	2	3	3	0	0
reg64low	8	2	3	4	5	2	2	2	2	2	3	3	3	4	4	0	0
reg96	2	2	2	2	2	2	1	2	2	1	2	2	2	2	2	0	0
r0_1_2_3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
spill32	10	1	2	3	4	1	1	1	1	1	2	2	2	3	3	1	1
r0	1	1	1	1	1	1	1	0	0	0	1	0	0	1	0	0	0
r1	1	1	1	1	1	1	0	1	0	0	1	1	0	1	1	0	0
r2	1	1	1	1	1	1	0	0	1	0	0	1	1	1	1	0	0
r3	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	0	0
r0_1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1	0	0
r1_2	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	0	0
r2_3	1	1	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0
r0_1_2	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0
r1_2_3	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	0
r12	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
r14	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1

2. *Local* is the same allocator, but made to spill all variables that are live across basic block boundaries.
3. *Worst-Case* spills all variables.

The Local allocator is intended to mimic heuristic local register allocators such as used in e.g. Lcc [13]. The Worst-Case allocator represents the worst case, and gives a crude base line for comparisons.

Due to some simplifying design decisions, the prototype framework generates spill code which is less efficient than what would be acceptable in a production compiler. This exaggerates the negative effects of spilling somewhat, which should be taken into account when looking at the experimental results.

6.2 Results

Finding good benchmarks for embedded systems is hard, since typical embedded applications differ from common desktop applications in significant ways [14, 15]. We have chosen to use the suites Automotive, Network and Telecomm from MiBench [14], a freely available[3] collection of embedded benchmarks.

The benchmark suites were compiled with each variant of the allocator[4]. The first part of Table 2 gives the number of functions (*funcs*) in each suite, and the average number of variables per function (*vars*). The largest number of variables in any function is 1016. For each allocator, we then report the total size of the generated code (*size*), and for Full and Local the number of spilled variables (*spill*). The Full allocator is not optimized for speed, yet. Currently, the average time spent in the allocator is 1.67 seconds per function.

Table 2. Results compiling benchmark suites

			Full			Local			Worst-Case	
Suite	*funcs*	*vars*	*size*	*spill*	*cost*	*size*	*spill*	*cost*	*size*	*cost*
Automotive	29	113	8918	77	5232	12598	175	9452	59076	65722
Network	17	84	3260	8	690	6048	100	4501	17970	25961
Telecomm	130	118	35116	154	6020	70778	1021	51992	329102	322858
Total	176	114	47294	239	11942	89424	1296	65945	406148	414541

Many programs in MiBench rely on the presence of a file system for input and output. Since this was not available in our test environment we were unable to run many of the programs. To give some indication of the performance impact of the different allocators, we give the accumulated spill cost for all spilled variables (*cost*). These costs are weighted by loop-nesting depth, so that spills inside loops are more costly. In Table 3, we compare the accumulated spill costs (*cost*)

[3] See `http://www.eecs.umich.edu/mibench/`.

[4] All files were compiled except `toast.c`, which failed because of a missing header file, and `susan.c`, which failed for unknown reasons.

with cycle counts ($kCycles{*}10^3$) from runs of three programs, one from each benchmark suite. The programs were executed in the simulator/debugger that comes with the compiler [11], using the “small” input sets.

Table 3. Results running benchmark programs

	Full		Local		Worst-Case	
Program	*cost*	*kCycles*	*cost*	*kCycles*	*cost*	*kCycles*
Automotive/qsort	0	136729	280	142556	1990	152005
Network/dijkstra	20	154339	820	188772	7360	979790
Telecomm/CRC32	20	3416	750	12618	3210	30731

7 Related Work

Briggs’ [6] approach to handling multiple register classes (in part suggested already by [2]) is to add the physical registers to the interference graph, and make each node interfere with all registers it can not be allocated to. Edges between nodes from non-overlapping classes are removed. To handle register pairs, multiple edges are used between nodes where one is a pair. Thus, the interference graph is modified to represent both architectural and program-dependent constraints, leaving the graph-coloring algorithm unchanged.

Our approach is fundamentally different, in that we separate the constraints of the program from those of the architecture and run-time system into different structures. Instead of modifying the interference graph, we change the *interpretation* of the graph based on a separate data structure. 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.

For an architecture with aligned register pairs, the solution proposed by Briggs is equivalent to ours in terms of precision. However, Briggs gives only vague rules (“add enough edges”) for adapting the algorithm to other irregular architectures [6]. Our generalized algorithm, on the other hand, works for any architecture that can be described by a target model.

The scheme proposed by Smith and Holloway [16] 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 a handful of common classes of irregular architectures. In contrast, our algorithm covers a much wider range of architectures without requiring classification, we give sufficient details to generate allocators automatically from target descriptions, and we prove that our local colorability test is safe for arbitrary target models.

Scholz and Eckstein [17] have recently described a new technique based on expressing global 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 target models. Practical experience with this new approach is limited, however, and it is not supported by the large body of research work that exists for Chaitin-style graph coloring.

There have been some attempts to use integer linear programming techniques to find optimal or near-optimal solutions to the global register allocation problem for irregular architectures [18, 19]. These methods give allocations of very high quality, but, like other high-complexity techniques, they are much too slow to be useful for large applications.

Some people argue that longer compile times are justified for certain embedded systems with extremely high performance requirements [20]. This has prompted researchers to look into compiler techniques with worse time complexity that what is usually accepted for desk-top computing, often integrating register allocation with scheduling and/or code selection. For example, Bashford and Leupers [21] describe a backtracking algorithm with either $O(n^4)$ or exponential complexity, depending on strategy. Kessler and Bednarski [22] give an optimal algorithm for integrated code selection, register allocation and scheduling, based on dynamic programming. Still, with embedded applications reaching several 100.000 lines of C code, there is a need for fast techniques such as ours, for compilers in the middle of the code-compile-test loop.

8 Conclusions

With our simple modifications, Chaitin-style graph-coloring register allocation can be used for irregular architectures. It is easy to incorporate well-known extensions into the generalized algorithm, allowing compiler writers to leverage the existing body of supporting research. The register allocator is parameterized on a formal target description, and we give sufficient details to allow automatic retargeting. Our plans for future work include comparisons with optimal allocations, incorporation of more extensions, and creating a free-standing implementation of the allocator to better demonstrate retargetability.

9 Acknowledgements

This work was conducted within the WPO project, a part of the ASTEC competence center. Johan Runeson is an industrial Ph.D. student at Uppsala University and IAR Systems. The register allocation framework used for the experiments in this paper was implemented by Daniel Widenfalk at IAR Systems. The register allocator itself was implemented by Axel Burström as a part of his Masters' thesis project. The authors wish to thank Carl von Platen for fruitful discussions and comments on drafts of this paper. We also thank the anonymous reviewers for valuable comments and suggestions for improvements.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, Second Edition. Morgan Kaufmann Publishers (1996)
2. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages **6** (1981) 47–57
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
4. Morgan, R.: Building an Optimizing Compiler. Digital Press (1998)
5. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
6. Briggs, P.: Register allocation via graph coloring. PhD thesis, Rice University (1992)
7. George, L., Appel, A.W.: Iterated register coalescing. TOPLAS **18** (1996) 300–324
8. Runeson, J., Nyström, S.O.: Generalizing Chaitin's algorithm: Graph-coloring register allocation for irregular architectures. Technical Report 021, Department of Information Technology, Uppsala University, Sweden (2002)
9. Ramsey, N., Davidson, J.W.: Machine descriptions to build tools for embedded systems. In: LCTES. Springer LNCS 1474 (1998) 176–188
10. Bradlee, D.G., Henry, R.R., Eggers, S.J.: The Marion system for retargetable instruction scheduling. In: PLDI. (1991)
11. IAR Systems: EWARM (2003) `http://www.iar.com/Products/?name=EWARM`.
12. Jagger, D., Seal, D.: ARM Architecture Reference Manual (2nd Edition). Addison-Wesley (2000)
13. Fraser, C.W., Hanson, D.R.: Simple register spilling in a retargetable compiler. Software - Practice and Experience **22** (1992) 85–99
14. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization. (2001)
15. Engblom, J.: Why SpecInt95 should not be used to benchmark embedded systems tools. In: LCTES, ACM Press (1999)
16. Smith, M.D., Holloway, G.: Graph-coloring register allocation for architectures with irregular register resources. Unpublished manuscript, `www.eecs.harvard.edu/machsuif/publications/publications.html` (2001)
17. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES-SCOPES, ACM Press (2002)
18. Kong, T., Wilken, K.D.: Precise register allocation for irregular register architectures. In: Proc. Int'l Symp. on Microarchitecture. (1998)
19. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: PLDI. (2001)
20. Marwedel, P., Goosens, G.: Code Generation for Embedded Processors. Kluwer (1995)
21. Bashford, S., Leupers, R.: Phase-coupled mapping of data flow graphs to irregular data paths. In: Design Automation for Embedded Systems. Volume 4., Kluwer Academic Publishers (1999) 1–50
22. Kessler, C., Bednarski, A.: Optimal integrated code generation for clustered VLIW architectures. In: LCTES, ACM Press (2002) 102–111