Design and Implementation of a Graph Coloring Register Allocator for GCC

Michael Matz
SuSE Linux AG
matz@suse.de

Abstract

Historically the register allocator used in GCC is a two phase allocator differentiating between local and global pseudo registers, which doesn't itself produce spill code, and therefore is limited in code quality if spilling is needed. This paper describes a new register allocator for GCC based on graph coloring. After a short overview of the concepts of them in general, including some of the improvements (if used in the implementation) we discuss the actual implementation of the allocator including design decisions and justification for them. This includes parts which aren't explained in the usual scientific papers but needed in a real world multi-target allocator.

1 Introduction

While compiling a program often the need arises to have a place wherein to store certain values. One example is the storage for the result of calculating a common subexpression. To actually make use of it in the later occurance it must be remembered somewhere. One possibility would be memory, but as the fastest storage for most real machines are CPU registers, those are the more natural choice. But the CPU registers (also hardware registers, or hardregs) are limited to a comparatively (to the amount of available memory) small set, which makes it unlikely to actually find a hardreg which doesn't yet hold a value.

The traditional solution is the use of pseudo registers (pseudoregs). While generating code for the program (if for initial generation or optimization doesn't matter) the compiler assumes there is an unlimited set of registers, and if it needs a new one it simply creates it. Now we obviously have to create another pass in the compiler (which has to be fairly late in the translating process), which creates a mapping from pseudoregs to hardregs. It is called register allocation for obvious reasons. This mapping must be injective if constrained to all occurring set of pseudoregs which are live at the same time (so that each hardreg only contains the value for one pseudoreg at a time), which means, that it doesn't necessarily exist trivially. In that case the register allocator needs to change the intermediate code to make use of storage in RAM to hold some of the pseudo registers at least during a part of their life time, which we call to spill a pseudoreg to RAM.

1.1 Current Situation in GCC

The traditional implementation in GCC consists of two passes:

- The first one allocates hardregs to pseudoregs which are only defined and used in one basic block (called local-alloc). This constraint makes the creation of the live range for those pseudoregs trivial (it con-

sists of the start and end point of it, which corresponds to the first def and last use in that block), limits the set of pseudo regs to deal with to those which also are used in that block, and leads to efficient algorithms of creating the mapping to hardregs.

- The second (global-alloc) deals with the other pseudoregs, which are defined and used in different basic blocks. Their live range can span multiple blocks, and most often can not be described simply by their borders. This pass allocates hardregs to those pseudos (it also maintains a conflict graph), constrained to the already done allocation for local pseudos. It also can override decisions of local-alloc if it sees fit.

Both of these passes don't change the code. Instead they simply produce a mapping (in `reg_renumber[]`) which simply doesn't contain a hardreg for a pseudo for which it wasn't able to find one. Then follows a pass called `reload`, which uses this mapping to change the instructions accordingly. Pseudos without hardreg get a place on the stack, and the instructions are modified to refer to their memory location. While doing this `reload` also performs a validity check against constraints from the machine description. If this check fails, the operands which were failing are "reloaded" to make them valid (hence the name of that pass). This for instance then also includes creating explicit load and store instructions for those pseudos which have only stack storage, if the insns which used them can't deal with memory operands. That is, the process of spilling pseudos is implicit in forcing instructions to be valid.

Those reload instructions themselves also need register resources. If the reload was caused by a stack reference, there is a high possibility that it was storage for a pseudo which didn't get a hardreg, which further means that it's also probable that there isn't any free hardreg. So `reload` needs to deallocate some of the currently live pseudos in order to free up some hardregs. For instance consider this instruction:

$$p1 \leftarrow p1 + p2$$

Suppose $p1$ and $p2$ didn't get a hardregs, and the add instruction doesn't accept memory operands. Furthermore suppose that there are no hardregs free during that instruction. Now `reload` conceptually creates this instruction internally

$$[sp + 4] \leftarrow [sp + 4] + [sp + 8]$$

notices that it is invalid and creates reload insns for the memory operands. sp here means obviously the stack pointer and $[adr]$ means the memory at address adr. The add instruction here requires registers as operands, so we need to use some, say $h1$ and $h2$. The code now looks conceptually like:

$$\begin{aligned} h1 &\leftarrow [sp + 4] \\ h2 &\leftarrow [sp + 8] \\ h1 &\leftarrow h1 + h2 \\ [sp + 4] &\leftarrow h1 \end{aligned}$$

So we need to deallocate all pseudos live during this insn which formerly used $h1$ or $h2$. This in turn means that some pseudos now get stack storage instead of a hardreg, therefore the process of reload needs to be repeated until it stabilizes (during which more and more pseudos which initially got a hardreg could be spilled again). In an optimizing compilation `reload` actually calls back into global-alloc right before repeating reloading, in the hope,

that some of the newly spilled pseudos could get a different hardreg instead of none at all.

That the emission of spill code is external to the register allocator itself, and that it is done on a per instruction basis leads to non-optimal spill code in some situations. This (and curiosity ;-) lead to the implementation of a more traditional graph coloring register allocator for GCC.

2 Graph Coloring Register Allocators

This section describes graph coloring register allocators in general and introduces some improvements to the naive first versions.

2.1 A First Version

As explained above the problem to which we seek a solution is to find a mapping from a set of pseudoregs into a set of hardregs under the constraint that pseudos simultaneously live must not be mapped to the same hardreg. Or more abstractly the constraint is, that certain pairs of pseudos may not get the same hardreg (for whatever reasons). Such pseudos are called to be in conflict. The set of conflicts forms a relation over the pseudos, which is symmetric and irreflexive. The visualization of a set together with such a relation is simply an undirected graph without loops. The nodes represent the pseudoregs, the edges the conflicts, the graph is called conflict graph. In the context of register allocation we talk about webs, instead of nodes.

Now the problem is to assign each node a hardreg such that no neighbor of the node has the same hardreg. This is exactly the formulation of the graph coloring problem (with hardregs being our colors), which explains the name for the class of register allocators working under this model.

Note that pseudos not only conflict with other pseudos, but also with hardregs. The reasons can be that due to machine constraints some hardregs are already used in the intermediate representation before register allocation. Or some pseudos only are permitted a certain set of hardregs (which can be modeled by making them conflict with the inverse set). To make this fit into our model we also include a node for each hardreg into the graph, which already are assigned a color; they all conflict with each other.

Now it's well known that graph coloring is NP-complete, so a full solution isn't feasible for a compiler. We have to implement approximate solutions with better runtime behavior.

The first thing is to make use of Kempe's observation (see [Kempe]), namely that nodes with fewer than N neighbors (where N is the number of available colors) can be trivially colored. We can remove such nodes from consideration, which in turn might make other nodes have fewer than N neighbors. The removed nodes are remembered on a stack. The process of pruning the graph in this way is called **simplify**. If we managed to empty the whole graph in this way we can take one node at a time from the top of stack, put it back into the graph and trivially color it (it's guaranteed to have less than N neighbors).

There are two reasons why simplifying the graph might not completely empty it. First it's only a heuristic, and second the graph itself might not be colorable with N colors at all from the beginning. Either way we might end up with an intermediate graph in which all nodes have N or more neighbors (those nodes are called constrained).

To make it simplify-able again we have to change portions of the conflict graph. This is

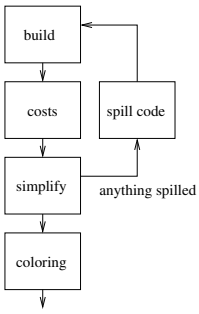

Figure 1: Flow graph of register allocators

done by choosing one of the nodes, the one with the lowest spill cost, remembering it for spilling, and remove it from the graph, in much the same way as if it were trivially colorable. Somewhen this makes other nodes simplifyable again, and in this manner we continue until the graph is empty. If there were spilled node we now add spill code, and repeat the whole allocation process. The next time the conflict graph will be simpler, as all spilled nodes are now split into several nodes, whose conflicts is only a subset of the original ones.

This leads to an allocator like in Figure 1. The **build** phase analyzes the intermediate representation of the program and creates the conflict graph. For choosing which nodes to spill if the need arises, we have to associate a cost for spilling to each node, so we can select the cheapest. Those are calculated by **costs**. The **spill code** phase is only entered if **simplify** had to remove some nodes by marking them as spilled. Otherwise all nodes were simplifyable, and **coloring** is entered, which pops the stack of simplified nodes and colors each one individually. The simplest (and fasted) mean to add spill code is to spill at each reference to a spilled node. Before each use insert a load from, and after each def[1] insert a store to the memory place allocated for the spilled pseudo. See [Cha81], although this includes also a coalescing phase.

[1] definition

2.2 Improvements

There are various improvements to the above simple allocator. Namely in how it deals with copy instructions, in the process of coloring the graph itself, and how spill code is emitted. I'll only describe those which are actually implemented in `GCC`.

After initially building the conflict graph, addition of code often changes it only locally. Therefore it is not necessary to completely rebuild the graph for each colorization round. Instead we **rebuild** the conflict graph incrementally, which is much faster, especially if only few pseudos were spilled.

Coloring and Copies

Copy instructions ensure that the two involved pseudo regs get the same value. Hence they are not a cause for a conflict between those two. To the contrary: if they don't conflict because of other reasons, it even is worthwhile to assign them the same hardreg, as by doing that the copy instruction itself becomes redundant. For instance in a situation like this:

$$\begin{aligned} p1 &\leftarrow \ldots \\ p2 &\leftarrow p1 \\ p3 &\leftarrow p4 + p1 \\ p5 &\leftarrow p4 + p2 \end{aligned}$$

Suppose that $p4$ is defined earlier. Normally $p1$, $p2$, $p3$ and $p4$ all conflict (except $p3$ and $p1$). But the definition of $p2$ is a copy from $p1$, and there are no other defines for it. So $p1$ and $p2$ don't conflict. Furthermore if we could ensure that both get the same color, $p4$ would only conflict with two instead of three nodes.

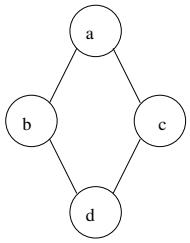

Figure 2: Diamond graph

aggressive coalescing: After building the conflict graph, but before measuring the costs we first try to merge all nodes for pseudos which are involved in one move. Merging them ensures, that they will get the same color. It can only be done if the nodes do not conflict. The resulting conflicts of the merged node are obviously the union of the individual conflicts. As merging nodes may prevent other nodes from being conflict free, nodes associated by the most costly moves should be handled first.

To see a problem in the coloring process look at the graph in Figure 2 and suppose there are only two colors. Here the **simplify** phase doesn't find any node having fewer than N neighbors, and ergo selects one for spilling (the rest is then simplified). Now there is definitely spill code added. But there's a trivial coloring, namely when nodes a and d, resp. b and c get the same color. But we can't know if this holds, until we actually color the nodes, which is only begun when we anyway know, that we succeeded. That is, the decision to spill a node is done too early, which leads us to (see [Briggs94]):

optimistic coloring: Instead of marking a node for spilling in **simplify** we simply also put such nodes on the stack (they are conceptually potentially spilled). No matter if there are such nodes or not, we go to the **coloring** phase. This one works as usual for the stack of nodes. If it colors a simplified node it still is guaranteed to get a color. And if it encounters a potentially spilled node it also tries to find a free color. If it succeeds, good, if not, only then is it actually marked for spilling. It often succeeds, namely in the case, where all the ($\geq N$) neighbors do not need all the N colors at the same time (i.e. some of them are colored equal).

The above mentioned coalescing, which is called aggressive because it tries to coalesce all copies, sometimes results in a much more constrained graph than without coalescing. When nodes are merged whose conflicts are nearly disjoint the resulting node will have much more conflict than the nodes individually. Possibly more than N, which makes it a potential spill candidate instead of a trivially colorable one. It can even make it definitely spill, where without coalescing the individual nodes would not have been spilled (at the expense of leaving a copy instruction around). A solution for this is (see also [GA96]):

iterated coalescing: Two pseudo nodes are only coalesced, if the resulting set of conflicts is smaller than N elements (this is conservative coalescing), and a pseudo to a hardreg node is only coalesced if all conflicts of the pseudo will be colored, or conflict already with the hardreg. This ensures that the graph doesn't become more constrained due to coalescing than it was. To not miss to coalesce too many copies coalescing is tried repeatedly between simplifying and choosing potential spill candidates. There are quite many work lists for nodes and moves, and the exact circumstances when they change their state are a bit involved, so interested readers are referred to the paper, as this is not anymore the method of choice in my implementation.

The method of iterated coalescing still is a bit too conservative. It effectively ensures that the graph remains at least as colorable after coalescing, but misses the positive effect which coalescing can sometimes have one coalescing.

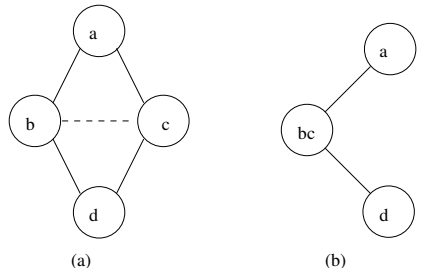

Figure 3: Diamond graph with b and c connected by a move

For instance referring to figure 3(a) if nodes b and c were coalesced the resulting graph (in 3(b)) is trivially colorable without any potential spill. But b and c wouldn't be coalesced under any conservative scheme (when N is two). In general it holds, that if two nodes are coalesced, those nodes which conflict with both have one conflict less after merging. This is the positive impact.

The problem that iterated coalescing (and conservative coalescing) are trying to solve is to prohibit coalesced nodes from becoming spilled. They do this by limiting merging before the fact, but that isn't necessary. It would be better to only act if the danger of spilling a merged node has become real (see [Park]):

optimistic coalescing: All moves are aggressively coalesced before **costs**. Then the normal **simplify** and **coloring** phases are run. When a node which is a merged node now definitely gets no color (i.e. would be spilled) we first split the merged nodes into its ingredients again, and try to color them individually. All parts which still need spilling are spilled. From the parts which get a color only the most costly will be colored right away, the other parts are put under the stack (so they are tried to be colored after all the other nodes), as the building of the color stack expected to only color one node. This splitting of the merge is simply an undo of the merge operation, i.e. all conflicts again point to their initial nodes. Conceptually instead of spilling the node we actually have split it. But compared to general splitting we know already good split points (namely the original copy instructions) and don't even need to insert them.

The example of figure 3 shows, that it sometimes is good for colorability if nodes are merged. It isn't necessary that there actually is a copy instruction. This idea is used by:

extended coalescing: After aggressive coalescing we also try to merge other nodes if it looks feasible. The candidate pairs are those, whose one pseudoreg is target and the other is source in the same instruction, and which do not conflict. Being mentioned in the same instruction makes it probable that the two sets of conflicts have many elements in common, so the merged node will not have that many more conflicts. If we then are unlucky and can't color it we unmerge the nodes again and go on.

Shrinking the Spilled Set

One of the parameters which influences the outcome of our graph colorizer in any way is the heuristic for choosing the next potential spill candidate among a set of remaining nodes (which are all constrained) (the other parameter is which color to choose for a node among those which are still free). The heuristic best for one graph may be bad for another one.

To become a bit more independent from that heuristic Bernstein et.al. ([Bernstein]) proposed a **best-of-three** strategy. For a set of heuristics the graph is colored each time from the beginning with one heuristic, and the overall cost of all spilled nodes is measured. Then finally that colorization with the lowest such cost will be used.

The other parameter is the choice of color

among the free ones for a certain node. The usual choices are first-fit and rotating. Another more complex one is **biased coloring** ([Briggs92]): the number of choices depends on how many colors are used by the neighbors, so one goal for coloring a node would be to not unnecessarily enlarge this set for the still uncolored neighbors of it. For that we look at the colored neighbors of all out yet uncolored neighbors. Those colors are anyway already unavailable to them, so would be a good choice for us.

Cheap Spill Code

The improvements up to now had the goal to make the set of spilled nodes as small as possible. The next few items deal with emitting the cheapest spill code once this set is fixed after one colorization round.

First notice that some pseudo regs contain constants (also values loaded from argument stack slots count as constants for this purpose), or values which are provably constant over the lifetime of that pseudo. This make spilling them easy ([Cha81, Briggs92, Briggs94]):

rematerialization: Such pseudos are called rematerializable as the expression calculating their value at each point during their lifetime is known, and hence, once they are overwritten could easily be "rematerialized." To spill such nodes instead of inserting load from stack instructions, one inserts the rematerialization instructions (depending on the value, for instance load with a constant). Stores are not needed for these nodes (as we know their value). Rematerializing a node is worthy if it's cheaper to create these value-load instructions than the mem-loads and mem-stores. More advanced methods of rematerialization also detect expressions over other pseudoregs, like in this example:

$$p3 \leftarrow p1 + p2$$
... code not changing $p1$ or $p2$
$$p4 \leftarrow p1 + p3$$
$$p4 \leftarrow p4 + p2$$

If $p3$ is spilled and $p1$ and $p2$ are not, and an add instruction on registers is cheaper than a load from memory, then we can instead recompute the value of $p3$ before its use. If we operate on SSA form the required analyzation to prove that $p1$ and $p2$ are not changed during the lifetime of $p3$ are relatively easy. Before actually doing such rematerialization it needs also to be ensured that the lifetime of the operands are not extended, i.e. that all operands are live during the lifetime of the spilled node.

Now we look at this code:

code defining $p1$ and $p2$
$$use \leftarrow p1$$
$$use \leftarrow p2$$
$$use \leftarrow p1$$
no further use of $p2$

Suppose that $p1$ is spilled and before the first use up to its definition are no instruction in which a pseudoreg dies. Naively there would be a load instruction added before each use of $p1$. But adding it before the first use doesn't help colorability at all. As there are no deaths between that use and the def the number of used hardregs remains constant there. Inserting a load is not going to help colorability of $p1$.

Therefore we only **spill at deaths**, we only insert loads if we encounter a death of another non-spilled pseudo. For inserting loads we walk backwards the instruction stream, note which nodes need a load, and emit all loads as soon as we reach a def (or the basic block border).

Of course we don't emit the loads directly after the death, but instead right before the in-

struction which most recently used the spilled pseudos. Otherwise we could end up with code like:

$$
\begin{aligned}
p1 &\leftarrow p1 + p2 \\
p3 &\leftarrow [sp + 4] \\
p4 &\leftarrow [sp + 8] \\
p5 &\leftarrow [sp + 12] \\
p1 &\leftarrow p1 + p3 \\
p1 &\leftarrow p1 + p4 \\
p1 &\leftarrow p1 + p5
\end{aligned}
$$

$p3$, $p4$ and $p5$ were spilled, and $p2$ died at the first shown instruction, where we also inserted all loads together. So the register pressure at the second add instruction is still four. The correct position of the load is right before their uses, but actually emitting them is only triggered by encountering a death, which then leads to this code:

$$
\begin{aligned}
p1 &\leftarrow p1 + p2 \\
p3 &\leftarrow [sp + 4] \\
p1 &\leftarrow p1 + p3 \\
p4 &\leftarrow [sp + 8] \\
p1 &\leftarrow p1 + p4 \\
p5 &\leftarrow [sp + 12] \\
p1 &\leftarrow p1 + p5
\end{aligned}
$$

The next improvement is **interference region spilling** ([Bergner]): if we don't find a color for a node (i.e. it's spilled) we up to now totally removed that node from the graph (by placing it into memory everywhere except for very small ranges around the instructions which needed it). But we also could simply assign any hardreg to this node, and only *remove the edges* to any now really conflicting neighbor.

Practically this is done by choosing a color for all spilled nodes. While emitting the spill loads we also track all hardregs which are currently in use. Remember that we walk backwards. If we encounter a use of a spilled web whose

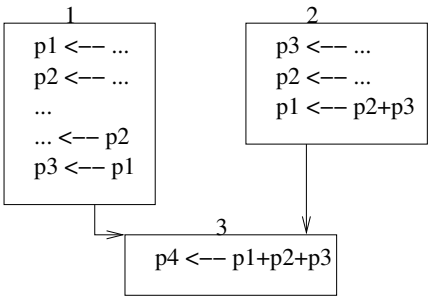

Figure 4: Example for interference region spilling

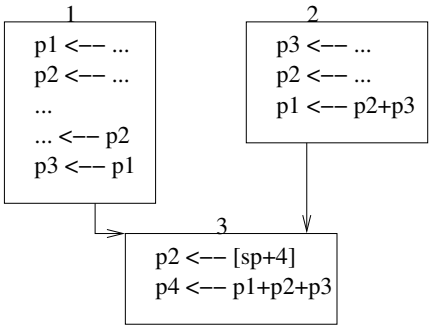

Figure 5: Example for interference region spilling (after inserting loads)

color is in use, we deal with it like described above (i.e. waiting for a death and then inserting a load before the using instruction). If on the other hand its color is currently *not* in use we mark it specially as potentially needing a load. If we go further up and notice a definition for a node marked in this way, and its color didn't become used meanwhile, we simply remove that mark (it's not live before the definition anyway). If we encounter the use of another (non-spilled) node we set its color as used. If we currently have some potential load candidates, whose color now is used, we emit loads for those. This process effectively only adds spill instructions if there is danger that two nodes with the same color are live at the same time.

To better see the effect look at figure 4 for

an example situation. $p1$, $p2$ and $p3$ all conflict, and we only have two hardregs. $p1$ got hardregs 0, $p3$ hardreg 1 and $p2$ was spilled. We choose hardreg 1 for it. We begin with the use in block 3, both colors are needed ($p1$ and $p3$ are used), so we need to insert a load (we reached the block border). Then we analyze block 1. Initially hardreg 1 is not used ($p3$ is not live), so we only mark $p2$ as potentially needing a load. While we go upward hardreg 1 doesn't become used, but we encounter a def of $p2$. So we simply forget about it. In block 2 hardreg 1 is used during lifetime of $p2$, but we don't encounter a death until the def, so no load is added here. We end up with the code in figure 5.

If we had spilled by the former method we also had inserted a load into block 1 (if there is any death in the "…"). With interference region spilling we need to insert stores for each definition which reaches one of the uses for which a load was added. In the above case after both defs.

To further reduce cost of spill code we also do **web splitting** ([Mass]). If we can't find a color for a web, i.e. we are going to split it, we first try if we can split this web around other webs, or other webs around that one, in a cheaper way than splitting. Look at this code:

$$
\begin{array}{c}
p1 \leftarrow \ldots \\
p2 \leftarrow \ldots \\
\text{... code1 without using } p1 \\
\ldots \leftarrow p2 \\
\text{... code2 without using } p1 \\
\ldots \leftarrow p2 \\
\text{... code3} \\
\ldots \leftarrow p1
\end{array}
$$

Suppose $p2$ is spilled (there are other uses of $p1$ which makes it more costly to spill $p1$ than $p2$) and $p1$ already colored. Now instead of doing that, we notice that during the lifetime of $p2$ there are no references to $p1$ (which requires something like a containment graph, which can also be used to implement the conflict graph). This makes it possible to completely split $p1$ around $p2$, so that it isn't anymore live during $p2$. This even guarantees, that the number of conflicts for $p2$ reduces, something which normal spilling can't do generally. The result would then look like:

$$
\begin{array}{c}
p1 \leftarrow \ldots \\
[sp+4] \leftarrow p1 \\
p2 \leftarrow \ldots \\
\text{... code1 and code2} \\
\ldots \leftarrow p2 \\
p1 \leftarrow [sp+4] \\
\text{... code3} \\
\ldots \leftarrow p1
\end{array}
$$

Note that the load of $p1$ is not for the later use of it (like in spilling), but rather because the lifetime of $p2$ ended. That is, generally stores for split webs are created before each def of webs around which they are split. Loads for them are created right after each death of the split around webs. A web can also die over a certain edge, not only explicitely at a use.

One minor improvement is **spill coalescing** ([GLnew]): It can happen, that there are uncoalesced copy instructions remaining, where both pseudos of the copy insn are spilled, but do not conflict. This would create a memory-memory move which often is less than desirable. Therefore we can run another aggressive coalescing pass for just the spilled webs in order to remove such copies. This also reduces the needed frame size a bit.

Another situation which sometimes arises is helped by **spill propagation**: There are three pseudos, $p1$ connected to $p2$ by a copy and $p2$ connected to $p3$ by a copy. They don't conflict, like here:

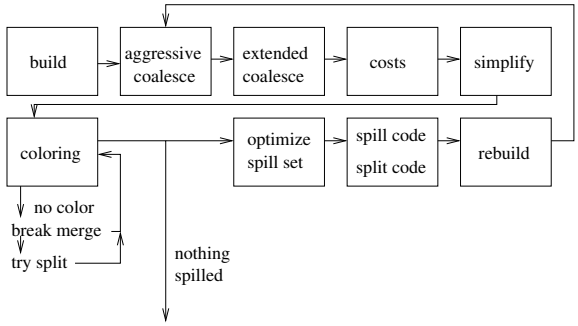

Figure 6: Flow graph of the final allocator

... with def of $p1$
$p2 \leftarrow p1$
...
$p3 \leftarrow p2$
... with use of $p3$

Now suppose, that $p1$ and $p3$ were spilled, but $p2$ colored. It might be better to also spill $p2$ to memory, if then the two copy instructions can be removed by coalescing all three pseudos together. In a sense this "propagates" the spill to a colored pseudo (which initially is counter intuitive).

The last improvement here is **spill coloring**: After the set of spilled webs is finalized a coloring pass is run on the subgraph induced by the spilled nodes, with an unlimited number of colors (i.e. when a node doesn't get a color, the maximum number is simply incremented and it gets the new color). Then a stack slot is allocated for each such color, instead of for each spilled node. This greatly reduces the needed stack frame size for spilling.

The final flow graph of the register allocator can be seen in figure 6.

3 Taking it to GCC

Now we look how to fit all the above descriptions into the framework of GCC. The definite reference is of course the source code (in files `ra*.c, ra.h` and `pre-reload.*`), and to not obsolete the paper as soon as some parts of the allocator are changed we don't follow the source too closely here.

3.1 Constraints Imposed by GCC

Classes and Constraints

GCC not only targets an ideal machine with a set of N completely equivalent registers, whose instruction set is totally orthogonal, which doesn't expect certain conditions from the operands of instructions, but instead it targets real machines with sometimes awkward constraints. The ones which influence the register allocator are described here.

GCC has the concept of **register classes**: The set of all hardware registers for a machine is divided into named smaller set of registers (`ALL_REGS` for the whole set and `NONE_REGS` for the empty set are defined by all machines). They are not disjoint. The register operands of instructions can specify which hard registers they accept by mentioning such register classes.

The instruction templates for a machine can specify **constraints** and can consist of more than one alternative per template. Each of the instructions in the intermediate representation match one template in the machine description. For register allocation purposes each template has many alternatives, where each of them can have a different set of requirements on the operands. For instance it's possible that the generic "add" template has two alternatives, one accepting registers of class `CLASS1` and the other accepting registers of class `CLASS2`.

The are also other types of constraints, for instance to limit the range of constant operands (so as to fit into an immediate field in the

instruction), or matching constraints, which force one operand to be equal to another. In that way two address machines can be implemented. For instance the generic "add" pattern has three operands (one target, and two sources), and if the machine has only an instruction which adds a register to the other source register, this can be specified by constraining the first source operand to be the same as the target operand.

Such matching constraints can easily be made valid by a pass before register allocation, by adding copy instructions for the matching operands (possibly using new pseudo registers). Similar with constraints which don't affect register operands (constants e.g.).

Additionally some hardregs are not available for register allocation at all, as they have special uses (e.g. the static chain pointer for nested functions, or the PIC register on some machines).

The machine descriptions also have the possibility to limit the set of hardregs for a pseudoreg just based on its mode (the `HARD_REGNO_MODE_OK` macro).

Subregs and Wide Regs

Another possibility in `GCC` is the use of **subregs**. Subregs are references to a part of a register (or other values, but in those we aren't interested). This makes such code possible:

$$p1 : [SI + 0] \leftarrow p2$$
$$p1 : [SI + 4] \leftarrow p3 + p2$$
$$p4 \leftarrow p1 + p5$$

Here the notation $p1 : [SI + x]$ means the subreg of $p1$ of mode `SImode` on byte offset x inside $p1$. Suppose that $p1$ is a `DImode` pseudo. The code does define $p1$ piecewise (first the lower half, then the high half), and then uses the $p1$ in its whole. The interpretation of subregs is bitwise.

A special kind of subregs are **paradoxical** subregs. Those are subregs in a wider mode than the inside register provides. I.e. it accesses bits which aren't provided (or are undefined).

Furthermore not all machines allow subregs to be taken from all hardware registers. For instance on Alpha the floating point register can hold 64-bit integers. But it's not possible to access the low or high 32 bit of that value by simply looking at the low or high 32 bit of the register. Therefore some registers are not allowed for references which involve a subreg reference.

And finally there is the notion of **multi word** hardregs. Those are references to hardregs in a mode which is wider than this hardreg. Such references implicitly use the next few adjacent hardregs (as much as needed). For instance a `DImode` reference to hardreg 0 (which for this example shall be SImode maximum) also uses hardreg 1.

3.2 Meaning for the Allocator

The constraints on registers result in a set of allowed hardregs for each register reference. The set of allowed hardregs for a whole web consists of the intersection of the sets for all individual references making up that web.

It's possible that one web consists of references with conflicting constraints, i.e. with disjoint allowed hardregs. For instance a pseudo register used in integer (e.g. bitwise logic) and floating point context (e.g. addition with a float constant). Such a web would have an empty set of possible hardregs. A possible solution is to either fake this set (by ignoring the conflicting reference), and thereby leave the work of fixing up the instructions to `reload`, or to spill

away the conflicting reference while building the web.

As probably already became clear, each web has its own set of allowed hardregs (in the `usable_regs` member of `struct web`). Most often it will not contain all hardregs. This has implications for the predicate `is-trivially-colorable` used in the **simplify** phase. The number N is meaningless here. Instead a web is trivially colorable if the weight of its conflicts is less than the number of registers in `usable_regs`. For that to work there may only be conflict edges between webs whose possible hardregs have a non empty intersection. This of course makes sense, as if it were empty the two webs anyway couldn't get the same color, so conflicts between them are pointless. This also reduces the necessary conflict edges.

One difficulty are multi-word pseudos. The webs have an `add_hardregs` member which contains the number of additionally required hardregs (at maximum). To generally ensure that there is a hole of two consecutive hardregs in a block of N, it would be required that there are less than $N/2$ neighbors (which itself wouldn't be allowed to use multiple regs). If we had exactly $N/2$ conflicts all even colors could be taken, leaving no block of size two. But this is clearly an overly conservative heuristic.

Instead the `add_hardregs` member simply is counted as another conflict. So the actual predicate is:

$$trivial_i := |usable_i| > add_i + \sum_{n \in neighbors_i} 1 + add_n$$

This is an optimistic predicate, which means that even webs which were simplified could not get a color (only when they are multi word regs).

The possibility of subregs means for us, that a pseudo may sometimes be live only partially (this is a cause of much of the complexity in the actual implementation). This can also result in partial conflicts, i.e. something like "the lower 32bit of $p1$ conflicts with $p2$." Such conflicts are useful to create a good allocation for multi word pseudos, as now partial overlap is allowed (so that for instance only three hardregs are needed for two pseudos each needing two regs). Partial webs are instances of the normal `struct web` but they have their `parent_web` member set. The `subreg_next` members form a linked list between the whole web and its parts.

3.3 The Conflict Graph

The most important structure in a graph coloring register allocator is obviously the conflict graph but up to now we haven't talked about it, because conceptually it's not very interesting in the context of describing the general methods of register allocation.

It is implemented in `GCC` by these structures:

```
struct conflict_link
{
  struct conflict_link *next;
  struct web *t;
  struct sub_conflict *sub;
};
struct sub_conflict
{
  struct sub_conflict *next;
  struct web *s;
  struct web *t;
};
struct web
{
  ...
  struct conflict_link *conflict_list;
  ...
};
sbitmap igraph;
sbitmap sup_igraph;
```

That is, each web has a linked list of its conflicts. Only whole webs have this list,

subwebs (those corresponding to subregs) don't. The targets of those conflicts (in `conflict_link.t`) are also whole webs. This allows fast iteration over all conflicts without having to care for the details of sub-conflicts. If between web a and b only sub-conflicts occur, then those are remembered in a second linked list, which hangs off of the edge between a and b. I.e. there is one `conflict_link` instance in as conflict list, with `.t` being b, which has its `.sub` member pointing to a list of sub-conflicts which note which parts of a resp. b exactly are conflicting (`.s` points to a part of a or a itself, `.t` to b or a part of it). The bitmaps `igraph` and `sup_igraph` are used to test two webs for conflicts. `igraph` contains the exact conflicts between parts, `sup_igraph` lists for whole webs, if they them-self or any parts of them conflict. This is a bit suboptimal. If we had a mean to go from the indexes of two webs to the corresponding `conflict_link` instance for their connecting edge (a hash table for instance) we wouldn't need `sup_igraph`. If one considers coalescing (which also involved merging conflicts, which we must be able to break up again) such a mean is not totally trivially implemented, though.

Actually **building** the conflict graph is implemented in `ra-build.c`. We use an incremental graph builder which at the same time does an liveness analysis, builds webs and creates (preliminary) conflicts (it's in `build_web_parts_and_conflicts()` and sub-functions). It works use by use. Per use it goes backward the instruction stream (following all edges backward), until it reaches a def for the register we currently analyze. On that way it remembers the defs encountered for the current use (from those the real conflict lists are build later), connects uses and defs of the same reg as that use in a UNION-FIND structure, and fills some house keeping information (for instance if an edge is crossed the use is remembered as live over it).

The currently analyzed use is placed into an instance of **struct curr_use**. Partial liveness is supported by having a bit field (the `.undefined` member) where each bit corresponds to one byte of the use. A bit is set if the byte is still undefined. When a def is encountered the bits which correspond to that def are cleared. If that results in no more left bits we have reached the first def which (partially) defines the use on that path. The set bits also represent the part of the use, which is still live. This is used for creating sub conflicts. Partial liveness could also be represented by a set of ranges, which bits are live. A variation of that scheme is used in [Bitwidth], although they only split the bits into three sections (a set of leading and trailing dead bits, and a section of middle bits, which are live). To correctly represent live information under this scheme we would need to treat some subreg references as read-modify-write, like it's done in the conservative data flow pass in `GCC`. This makes it less attractive again.

The advantage of such a builder compared to a more traditional bit-set based liveness analyzer is the simplicity (we deal with only one use at a time), that it's possible to precisely track partial liveness for subregs (something which is not that easily done with bitmaps) and that we can easily rebuild the graph for only those uses, which need it. After spilling was done not the whole graph needs to be rebuilt, but only those webs, which were changed, and their former neighbors. A bit-set based analyzer also needs to iterate until the solution stabilizes. This is not needed here. And that we can note conflicts already *while* still building webs also is attractive.

With some optimizations (like skipping whole basic blocks if the current pseudo isn't mentioned in them) the part of building webs and

preliminary conflicts actually was nearly as fast as the traditional bitmap based liveness analyzer in `GCC`.

There is a pass necessary which actually creates the `struct web` instances and the conflict lists from the UNION-FIND structure and the preliminary conflicts (which are all based on the defs and uses, for each of whom an instance of `struct web_part` is created. This is done in `make_webs()` and subfunctions.

The rest of initializing the webs is also in `ra-build.c`. Among it are determining the spill cost of a web, if it's rematerializable, collecting the copy instructions and so on. It probably had better been named `ra-analyze.c` ;-)

3.4 Putting it Together

The implementation of the register allocator consists of different files which roughly reflect the structure described in section 2.

Besides `ra-build.c` which builds not only the conflict graph but also most of the other information about webs and program structure (as described above), there is

`ra-colorize.c`
which is all about changing (like in coalescing) and coloring the conflict graph, including optimizations which shorten the set of spilled webs. This includes the work list management. The structure is fairly close to the allocators in the published papers, except for three things:

- **selectable** algorithm: Most of the improvements in the coloring process are selectable at runtime. For instance it can be switched between optimistic or iterated coalescing, or biased coloring can be activated or not.

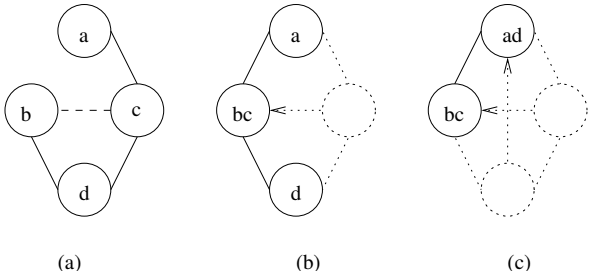

Figure 7: Coalescing of nodes

- **hard trying** to color certain webs: due to irregularities in connection with multi-word pseudos, and with spill temporaries, or other generally difficult webs (which includes those during whose lifetime no death occurs) it's possible that there is no color free for a web which absolutely must have a color (this happens extremely seldom and only on register constrained machines). In that situation it is tried to temporarily mark one of its already colored neighbors as spilled, and try again to find a color. This is done until a color is found or no more colored neighbors are left. After that those temporarily spilled neighbors are tried to be colored again. If they don't get a color they are left in the spilled state.

- **recoloring spills**: after the graph is colorized and the set of spilled webs is determined, each spilled web is tried to be recolored. For this the cost for the spilled web getting a color is measured (it consists of the sum of spill-costs of all neighbors overlapping that color). If the smallest cost is smaller than the web spill cost, this recoloring is done, and the neighbors which now conflict are spilled instead. This can reduce the overall spill cost of the graph.

One particularly ugly problem is how to implement splitting up merged nodes for optimistic

coalescing. Refer to figure 7. Starting with graph (a) first nodes b and c are merged, then nodes a and d. The final graph has only one edge left which was not in the original graph. Now suppose we want to split node bc. We may not yet remove edge $b - ad$, because also the original $b - d$ edge was mapped to it. Only when we also split node ad we remove it, and then we also *have* to remove it in order to not constrain the graph more than necessary.

From that description it becomes clear that the only real solution would be to add reference counters to edges. But that would bloat the size of each edge. That's not desirable as there are potentially very many edges in a conflict graph. The reference counter would also only be ever needed for edges which weren't in the original graph, as only those are candidate for removal.

Currently we don't refcount[2] the edges, but instead "repair" the graph after having split nodes. First we remove all edges incident to split nodes which weren't in the original graph (we have an easy way to test that as each node has a list of those), and then we look for other coalesced nodes that would have added that edge also (in which case we reinsert it into the graph). This process is relatively slow, so we will move to a refcounting implementation eventually (the work has already started for that).

`ra-rewrite.c`
is responsible for actually changing the program to include any spill code. Its behavior is also selectable at runtime, and it can use spill-everywhere (separately for uses and defs), traditional spill at deaths or spill at interference regions. It also implements the code for splitting webs around other webs which can theoretically be used with together with all spill methods. Unfortunately interference region spilling and web splitting use separate data structures and can't currently be used together. They will be usable together once the implementation is done.

Besides the improvements from section 2 for reducing the number of inserted loads during spilling, the actual implementation also has a naïve implementation of optimizing dead stores. It goes backward the insn stream remembering to which locations it wrote to. For each encountered use we delete all locations which overlap that use from the list. If it is about to insert a store it first checks if that location is still in the list, and omits the store if it is.

One thing which should be mentioned is that we defer the creation of real stack slots until the very end of allocation. Until then we create new pseudo regs to hold the value of spilled (or split) webs. These pseudos are not to be confused with normal pseudo regs, as they conceptually represent stack slots or real registers. We do this for two reasons:

- We want to be able to track also liveness for stack slots (in order to merge or color them), and sometimes we are able to actually give them back a hard register. This usually happens when multiple rounds of spilling were needed and a spill method which produces long living temporaries was used.

  In that case it happens that first a web is spilled which then didn't relax the situation as much as hoped, so other webs are also spilled. This in turn can make the spilling of the first web unnecessary, and by creating a web also for stack slots we are able to make use of that. Those *stack-pseudos* or *stack-webs* as we call them in the allocator are handled specially in a number of situations. For instance they are colored after all normal webs. If they don't get a color, they are not spilled again

[2] count the number of references of ... ;-)

(this is implemented by coloring them with an impossible color). This also needs changes in the functions which check validity of constraint, so that stack pseudos are accepted for memory references and for registers.

Some machines have requirements on the addresses they accept, for instance a limited range of offsets from a base registers. Emitting an address reference on them can possibly lead to emitting more than one instruction, which actually constructs the address by doing arithmetics on some new pseudo registers. On those machines we can't defer creating stack slots completely, as creating new pseudo regs means we must redo our register allocation. For those machines we actually emit real stack references for all the stack-webs which did not get a color. I.e. we defer stack slots only by one round, not until the very end.

- The other reason is stack slot coloring as described in Section 2 (as "spill coloring"). When we have webs for all stack slots (i.e. for the stack pseudos) including all conflicts, we can color them easily and reduce the frame size. I.e. we allocate stack space not for each stack pseudo, but instead only for each color needed for them.

The rewriting phase is also responsible for resetting the conflict graph and associated information into a state that is usable as a starting point for the next round. For instance all coalescing has to be undone, and the edges added for that have to be removed (as *all* coalescing is undone this is considerable easier than what was described above). We also need to mark which webs have to be rebuilt (namely those which changed their layout).

The file `ra-debug.c` contains some useful functions for debugging the allocator including a new format of outputting the immediate format (RTL) of `GCC`, which is much more compact and easier to read (although it lacks some information) than the traditional lisp like format. It should somewhen be extended to be usable also for the other passes in `GCC`, and be merged with the format of the scheduler debug dumps (which uses something similar).

To actually scan the instruction stream for all (interesting) references to registers we use functions from `df.c`. For each such reference we build one instance of `struct web_part` which creates an indirection in one of the highly used data structures, so it might somewhen be advisable to do this on our own.

The last big part in the allocator is implemented in
`pre-reload.c`.
As written at the very begin the *reload* pass is responsible for actually emitting spill code in the old register allocator, and for fixing up any invalid instructions (those whose operands do not match their constraints). The spilling code we do add ourself now, but we could still produce invalid instructions (for instance operands don't match where they have to, or an operand is in a register which isn't in the required class). This would make reload emit fixup code. As this code is emitted locally without having the big picture of a conflict graph or similar means this often results in spilling some other pseudo registers, and reloads method for adding spill code is undesirable.

Therefore the goal must be to never leave the register allocator with possibly invalid instructions. One requirement is to allocate pseudos to a register which is accepted by all the instructions that reference it. To that end prereload collects the possible register classes for each register reference. Another requirement is to not violate matching constraints, which

is done by pre-reload emitting copy instructions before or after the invalid instruction, and change the operands of it to actually match. It also makes sure that constraints which don't involve pseudo regs are fulfilled, like constants be of a certain range, or decomposing multi-level indirect memory access (i.e. the address is a memref[3] itself) if necessary.

The techniques it uses are heavily inspired by reload itself, but as pre-reload works on pseudo regs, the actual implementation can be quite a bit simpler.

The use of pre-reload can not make totally sure, that no invalid instructions are generated. Which register class is acceptable for one operand can depend on which register another operands was put in and this is only known, once allocation finished, so in some situations we have to give up in the allocator and assume something. This means, that reload will still be needed, but only extremely seldom (we once had only about 10 reloads while building cc1 IIRC). This makes me hope that reload can be implemented in a much simpler way than now, for instance by simply emitting fixup instruction as invalid operands are encountered, instead of first collecting all reloads of all instructions. Reload inheritance probably would also not be useful anymore.

Finally `ra.c` holds it all together and contains some initialization functions plus the main loop.

4 Numbers

For comparing the performance we give some numbers of runs of the SPEC2000 performance test suite, with the old allocator and the new one.

Table 8 shows the result on a 1.53 GHz Dual

[3] memory reference

Name	T_{old}	S_{old}	T_{new}	S_{new}
164.gzip	223	627	223	627
175.vpr	420	334	431	324
181.mcf	864	208	874	206
186.crafty	127	787	129	774
197.parser	439	410	438	411
252.eon	170	766	171	759
253.perlbmk	275	654	274	656
254.gap	205	537	201	546
256.bzip2	371	405	359	418
300.twolf	831	361	819	366
168.wupwise	294	544	290	551
171.swim	973	319	1036	299
172.mgrid	481	374	485	371
173.applu	624	337	599	351
177.mesa	233	602	229	610
179.art	1607	162	1664	156
183.equake	327	398	323	403
188.ammp	707	311	721	305
200.sixtrack	334	329	327	337
301.apsi	925	281	963	270

Figure 8: SPEC2000 results for Athlon 1800+

Name	$100 * \frac{(T_{new}-T_{old})}{T_{old}}$
164.gzip	-1.38889 %
175.vpr	-3.7037 %
176.gcc	-0.465116 %
181.mcf	1.82648 %
186.crafty	3.8835 %
197.parser	1.43885 %
252.eon	5.7971 %
253.perlbmk	-3.15315 %
254.gap	-1.17647 %
256.bzip2	-1.5873 %
300.twolf	-5.66616 %
168.wupwise	0.840336 %
171.swim	0.915751 %
172.mgrid	-4.17755 %
173.applu	-1.42518 %
177.mesa	-5.67686 %
179.art	0.555556 %
183.equake	1.0989 %
188.ammp	0.444444 %
200.sixtrack	0.593472 %

Figure 9: Relative SPEC2000 performance on AMD64

Athlon (Athlon 1800+). The T column shows the runtime in seconds (smaller is better), the S column the SPEC score (bigger is better). Note in particular bzip2, twolf and applu, which show some nice improvements. With the tested version of the allocator there were also some quite severe regressions as shown in the table. I've not yet analyzed them in detail.

Table 9 shows the runtime of the SPEC2000 tests compiled with the new register allocator compared with the old one on an AMD64 machine (i.e. with twice as much general purpose registers as x86). As can be seen crafty and eon regress quite much, but the potential of the allocator can be seen in the other results.

5 Acknowledgments

I wish to thank Daniel Berlin who started the `new-regalloc-branch` and created an initial implementation, and Denis Chertykov who created `pre-reload`, for their help in implementation and fruitful discussions.

I also would like to thank SuSE and AMD for letting me work on the register allocator.

And Cafebar 8006 for keeping me awake ;-)

6 Availability

The current development version of the register allocator is available in the `new-regalloc-branch` in `GCC` CVS. See

`http://gcc.gnu.org/cvs.html`

References

[Bergner] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, *Spill Code Minimization via Interference Region Spilling*, Proc. of the 1997 ACM SIGPLAN Conf. on PLDI, pp. 287–295. June 1997

[Bernstein] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter, *Spill code minimization techniques for optimizing compilers*, SIGPLAN Notices, 24(7):258 263, July 1989

[Bitwidth] S. Tallam, R. Gupta, *Bitwidth Aware Global Register Allocation*, TR, Dept. of Computer Science, U. of Arizona, July 2002

[Briggs92] P. Briggs, *Register Allocation via Graph Coloring*, PhD thesis, Rice University, Houston, Texas, April 1992

[Briggs94] P. Briggs, K. D. Cooper, and L. Torczon, *Improvements to graph coloring register allocation*, ACM TOPLAS, Vol 16, No.3, pages 428–455, May 1994

[Cha81] G. J. Chaitin, et. al., *Register allocation via coloring*, Computer Languages, 6:47–57, Jan. 1981

[GA96] L. George and A. Appel, *Iterated Register Coalescing*, ACM Trans. on Prog. Lang. and Systems, 18(3):300–324, May 1996

[GLnew] Allen Leung, Lal George, *A New MLRISC Register Allocator*, `http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/ra.html`

[Kempe] A. B. Kempe, *On the geographical problem of the four colors*, Am. J. Math. 2, 193–200, 1879

[Mass] MSCP group, *The massively scalar compiler project*

[Park] Jinpyo Park and Soo-Mook Moon, *Optimistic register coalescing*, IEEE PACT, pages 196–204, 1998