Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

The integrated register allocator for GCC

Vladimir N. Makarov
Red Hat
vmakarov@redhat.com

Abstract

The current GCC register allocator, like the classical Briggs and George/Appel register allocators consists of different passes solving separate tasks such as register coalescing and live range splitting. The Integrated Register Allocator (IRA) proposed for GCC does register coalescing and live range splitting on-the-fly during register colouring through a register preferencing technique based on dynamically changed register costs.

IRA design is a result of numerous experiments on Yet Another Register Allocator (YARA) infrastructure. IRA's design and implementation is focused on making it a production register allocator for GCC. Therefore IRA uses the reload pass, which is different from YARA's approach.

This paper describes IRA's colouring algorithms such as Chow's priority based colouring, Chaitin-Briggs colouring and the most powerful: regional top-down colouring, which deals well with high register pressure. Interprocedural register allocation in IRA and different optimizations such as caller register save optimizations and stack slot sharing are also described. The current state of IRA and benchmark results are given.

Introduction

Modern computers have several levels of storage. The faster the storage, the smaller its size. This is the consequence of a trade-off between the memory's speed and its price. The fastest storage units are registers (or *hard-registers*). To manage complexity of the optimizations, most of the optimizations in a modern compiler are written as if there is an infinite number of virtual registers called *pseudo-registers*. The optimizations use them to store intermediate values and values of small variables. As a result of such an approach we need a special pass to map pseudo-registers onto hard-registers and memory. This pass is called a register allocator.

Nowadays, a good register allocator is a significant component of an optimizing compiler because compilers implement more aggressive optimizations (like interprocedural or whole program optimizations). These optimizations tend to create lots of pseudo-registers living simultaneously. The number of available hard-registers can not be increased because it is a part of architecture. The situation is even worse because some widely used architectures (e.g. x86 and arm) are old and have few hard-registers.

GCC community recognized the importance of having a good register allocator long ago. Therefore there were a lot of efforts to improve the current register allocator [Makarov04] whose design was not changed significantly in last 20 years and to implement a new, better register allocator [Matz03, Makarov05]. This task is difficult because of the existence of a very complicated pass in GCC called *reload* which is a consequence of the fact that GCC is the most portable compiler and it must generate effective code practically for all existing architectures. This article describes a new register allocator for GCC called the integrated register allocator. Although the register allocator proposed in this article does not solve all known problems of the current GCC allocator (like the existence of the reload pass), its implementation is focused on making it the next production GCC register allocator.

Modern register allocators can be divided in two classes by their approaches used to solve smaller register allocator tasks. One approach used in classical Chaitin-Briggs [Chaitin81, Briggs94] and George-Appel [Appel96] register allocators consists of different passes solving separate tasks such as register coalescing, colouring and live range splitting, usually in an iterative manner. Another approach used in Callahan-Koblenz [Callahan91] allocators and allocators based on graph fusion [Lueh00] tries to solve these tasks in a more integrated way. The integrated register allocator, as can be seen from its name, uses the second approach: it

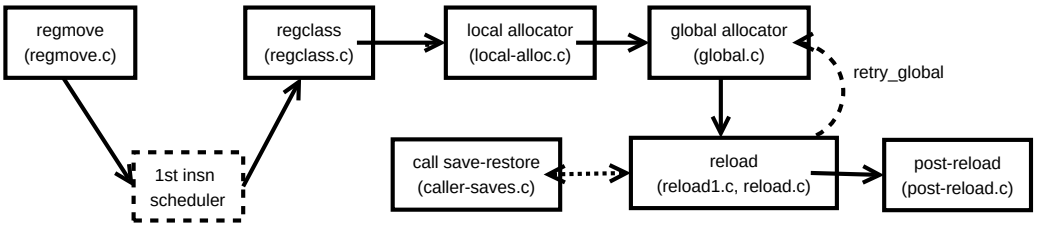

Figure 1: Passes in the GCC register allocator

does register coalescing and live range splitting on-the-fly during register colouring.

This article describes the integrated register allocator proposed for GCC. The first section describes the original GCC register allocator and the different projects to improve the register allocator in GCC that have been tried or are in progress. The second section describes the integrated register allocator: its passes and algorithms. The third section describes inter-procedural register allocation as implemented in IRA. The fourth section gives comparisons of the original register allocator and IRA on different benchmarks. The fifth section describes the current status of IRA, conclusions from the process of IRA's implementation and future directions of the project.

1 Register allocator in GCC

The current register allocator design has been in GCC since its first release without any significant changes. The register allocator consists of the following passes (see Figure 1):

Regmove. The major task of regmove is to generate move instructions to satisfy two operand instruction constraints. The reload pass can perform this task too, but in a less effective manner. The pass also does some register coalescing.

Regclass. GCC has a notion of *register class*, which is a set of hard-registers. The pass mainly finds the *preferred* and *alternative* register classes for each pseudo-register. Briefly but not very accurately, the preferred class is a class with the minimal cost of its usage for the given pseudo-register and the alternative class is a wider class, the usage of which is still more profitable than memory. Pseudo-registers are assigned to hard-registers only from their preferred or alternative classes.

The local allocator assigns hard-registers only to pseudo-registers living inside one basic block. Besides that, the local allocator also does some register coalescing and simple copy and constant propagation.

The global allocator assigns hard-registers to pseudo-registers living in more than one basic block. It can sometimes change assignments made by the local allocator. The algorithm used is very similar to assigning hard-registers in Chow's priority-based colouring [Chow90]. The global allocator does some form of register coalescing through a preference technique: the hard-register will be preferred by the pseudo-register if there is a copy instruction between them.

The reload is a very complicated pass. Its major goal is to transform RTL into a form where all instruction constraints for its operands are satisfied. The pseudo-registers are transformed here into either hard-registers, memory, or constants. The reload pass follows the assignment made by the global and local register allocators, but it can change the assignment if needed.

Besides this major task, the reload also does some other tasks like:

- indirect code selection by choosing instruction alternative
- elimination of virtual hard-registers (like the argument pointer) and real hard-registers (like the frame pointer)
- assignment of stack slots to spilled hard-registers and pseudo-registers which finally have not gotten hard-registers

- generation of save/restore code for call-used hard-registers living around the calls and simple optimization in placement of such code
- copy propagation
- simple rematerialization
- address inheritance, which is mostly reuse of calculated addresses to decrease the displacement for architectures with a small possible range of address displacements

Postreload. The reload pass does a reasonably good job for solving many tasks, sometimes in integrated way, but only in a local scope; it generates redundant moves, loads, stores etc. The post-reload pass removes such redundant instructions by a global redundancy elimination technique.

As we see, the current GCC register allocator solves register allocation tasks in adhoc way. There are a lot of passes with poor communication between them. Therefore, there is a strong belief in the GCC community that register allocation should be improved. However, improving register allocation in GCC is a challenging task because it is the most machine-dependent common part of the compiler. There were many projects to improve the register allocator. Only the recent ones are mentioned here.

Michael Matz [Matz03] tried to implement a register allocator from scratch. The project was called *the new register allocator*. It is based on Chaitin, Briggs, and Appel and other modern approaches to solving the register allocation tasks. Although the new register allocator used the reload pass, the project tried to remove unpredictable effect of its work by introducing the *pre-reload* pass before register allocation. The time spent on this project was not enough to allow the new register allocator to compete with the original one, which has been refined for many years. The new register allocator was much slower, generated worse code on average, had poor support and was eventually removed from the compiler.

Vladimir Makarov [Makarov04] tried to improve the current register allocator in GCC. He investigated several algorithms like pseudo-register live range splitting, register rematerialization, and coalescing of registers and memory stack slots. Although part of his work was adopted by the current register allocator, in general the project was not successful because of the addition of more passes, which solved register allocation tasks in a disjoint manner.

After recognizing the drawbacks of the previous approaches, the new project YARA (*yet another register allocator*) [Makarov05] was started. Its goal was to remove all old passes, including the reload pass. To simplify prototyping of register allocation algorithms a very flexible infrastructure was designed. This infrastructure was used to test many algorithms in register allocation including Chaitin-Briggs, Callahan-Koblenz and the FAT heuristic [Hendron93, Morgan98]. The YARA project was not finished. Removing the reload pass is a very challenging task and therefore YARA has been fully implemented only for *x86* and *x86_64* architectures. For these architecture YARA generates better code than the current register allocator.

Andrew MacLeod [MacLeod05] proposed a flexible infrastructure consisting of a set of engines used to solve register allocation tasks. To simplify the register allocator's implementation he proposed performing full code selection before register allocation. This is similar to the pre-reload pass implemented in the new register allocator.

Currently there are several ongoing projects to improve the current register allocator. Peter Bergner is working on improving the global register allocator on svn branch called *ra-improvements*. Michael Matz is working on code selection before register allocation, which is a mostly resurrection of the pre-reload pass of the new register allocator (svn branch *insn-select*). Andrew MacLeod is working on register rematerialization on the tree-SSA level to decrease register pressure, which could help the current register allocator.

YARA's goal to remove the reload pass was too ambitious. It will probably take a few more years to do this. GCC needs a better register allocator right now. That is the reason for starting *the integrated register allocator* (IRA) project. There is a tight relationship between the YARA and IRA projects. IRA was not designed and implemented from scratch: it uses a few of the design decisions from YARA project and the flexible YARA infrastructure was used to try many algorithms including ones finally used in IRA. The major difference between the projects is that the integrated register allocator is focused on being a production register allocator for GCC version 4.4. To achieve this IRA is designed to use the

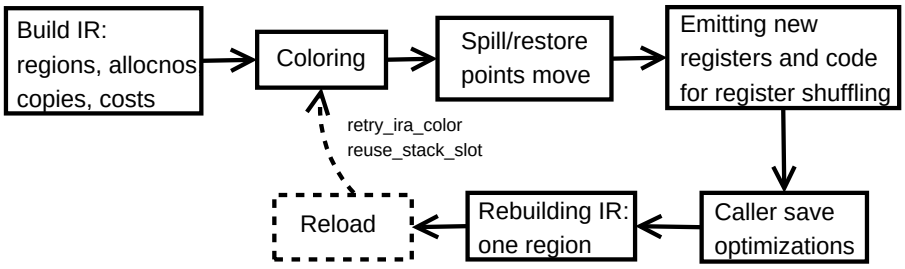

Figure 2: Passes in IRA

reload pass.

2 The integrated register allocator

The integrated register allocator implements a few colouring algorithms: Chow's priority colouring, Chaitin-Briggs colouring and regional top-down colouring. The regional colouring gives the best results for programs whose register pressure[1] is sufficiently large compared to the number of available hard-registers. Otherwise, Chaitin-Briggs colouring is the best. As a rule, it also generates the smallest code. Chow's priority colouring is the fastest algorithm, but this is its single advantage.

This section describes only regional colouring as the most general case. Chaitin-Briggs colouring is a special case of regional colouring when only one region (the entire function) is used. Additionally, Chow's colouring is a special case of one region colouring where hard-registers are assigned to pseudo-registers in order of priority. The priority of pseudos is defined in the same way as in the current register allocator:

$$Priority_p = Size_p \cdot \frac{Freq_p \cdot \log_2 NRefs_p}{Live\_Length_p}$$

$Priority_p$ is the priority of pseudo P, $Size_p$ is number of hard-registers needed for pseudo P, $Freq_p$ is sum of frequencies of all execution points where pseudo P occurs, $NRefs_p$ is number of points where pseudo P occurs, and $Live\_Length_p$ is number of RTL instructions between the first and last occurrences of pseudo P.

[1]There are two commonly used definitions. The wide one is the number of hard-registers needed to store values of the pseudo-registers at a given program point. Another one is the number of living pseudo-registers. In most cases both definitions give the same value for register pressure in optimized code.

Figure 2 shows the major passes of the integrated register allocator and their order. Each pass is described in detail in subsequent sections.

2.1 Building the internal representation

IRA is a regional allocator. It can work on any set of nested CFG regions forming a tree. Currently IRA regions are the entire function for the root region and natural loops for the rest regions.

GCC has a very powerful model for describing the target processor's register file. In this model there is the notion of register class. The register class is a set of hard-registers. You can describe as many register classes as possible. Of course, they should reflect the target processor's register file. For example, some instructions can accept only a subset of all hard-registers: in this case you should define a register class for the subset. Any relationships are possible between different register classes: they can intersect or one register class can be a subset of another register class.

To simplify Chaitin-Briggs colouring, IRA uses set of non-intersected register classes. This is set is called *the cover class set* and defined by a new machine dependent macro `IRA_COVER_CLASS`. Register classes from the cover set should not intersect and should contain all of the hard-registers available for register allocation. Those are the obligatory conditions which IRA checks. Usually there are many possible sets satisfying the conditions. Experience shows that IRA generates the best code when classes from the cover set are chosen in a way when any move between two registers of a cover class is cheaper than a load or store of the registers. Decent results are also achieved when a cover class is a union of such classes (using a cover class set containing only class `ALL_REGS` is an extreme example).

That is possible because IRA assigns hard-registers to pseudo-registers based on their usage costs, and if memory is more profitable then IRA will use memory for the pseudo-register.

IRA works with several data structures which are simplified versions of the ones from YARA. The central data structure of IRA is *allocno*. It represents the live range of pseudo-register in a region. Besides the usual attributes like the corresponding *pseudo-register number* and *mode*, *conflicting allocnos* and *conflicting hard-registers*, there are a few allocno attributes which are important for understanding the allocation algorithm. These attributes are

Cover class. This is a class from the cover class set. IRA can assign a hard-register only to an allocno from its cover class. This is different from the current GCC register allocator, which can assign a hard-register from at most two register classes (the so called *preferred* and *alternative* classes). All allocnos corresponding to the same pseudo-register always have the same cover class. As my experience shows, it is important for good allocation results to decrease the cost of hard-register shuffling on the borders of regions. This is the reason for such a requirement.

Hard-register costs. This is a vector of size equal to the number of available hard-registers of the allocno's cover class. IRA calculates and uses the cost of each cover class hard-register available for the allocation. First of all, IRA calculates the costs of allocnos for all register classes analogously to the way the regclass pass does, and defines the best cover class for the allocno from this information. The cost of register classes defines the costs of hard-registers of the cover class.

The cost for a *call-used*[2] hard-register for an allocno is increased by the cost of save/restore code around calls through the given allocno's life. If the allocno is a move instruction operand and another operand is a hard-register of the allocno's cover class, the cost of the hard-register is decreased by the move cost.

When an allocno is assigned, the hard-register with minimal *current cost* is used. Originally, a hard-register's current cost is the corresponding value from the hard-register's cost vector. If the allocno is connected by a *copy* (see below) to another allocno which just got a hard-register, the cost of the hard-register is decreased. Before choosing a hard-register for an allocno, the current cost is modified by the conflict hard-register costs of all conflicting allocnos which are not assigned yet.

Conflict hard-register costs. This is a vector of the same size as the hard-register costs vector. To permit an unassigned allocno to get a better hard-register, IRA uses this vector to calculate the final current cost of the available hard-register. Conflict hard-register costs of an unassigned allocno are also changed with a change of the hard-register cost of the allocno when a copy involving the allocno is processed as described above. This is done to show other unassigned allocnos that a given allocno prefers some hard-registers to remove the move instruction corresponding to the copy (see below).

Allocnos can be connected by *copies*. Copies are used to modify hard-register costs for allocnos during colouring. The copy can represent a move instruction between the corresponding allocnos. If an allocno is assigned to a hard-register, the cost of the hard-register for other allocnos connected to given allocno by the copies is changed to try to allocate the same hard-register for the connected allocnos. In case of the success, the move instruction will be removed. This is an analog of register coalescing in the classic Chaitin-Briggs allocator.

Copies can be created not only for move instructions: because they are important for describing hard-register preference, IRA creates copies for operands of an instruction which should be assigned to the same hard-registers according to their constraints in the machine description file. Typical example of such an instruction is x86 architecture addition, which requires that the addition result should be in the same register as one of the operands. IRA creates two copies for the addition because it is an associative operation. One copy refers to allocnos corresponding to the result allocno and the first operand of the addition. Another one refers to the result allocno and the second operand.

IRA also creates copies referring to the allocno which is the output operand of an instruction and the allocno which is an input operand dying in the instruction. As

[2] A call-used hard-register can be used in a function without saving and restoring it in the function's prologue and epilogue.

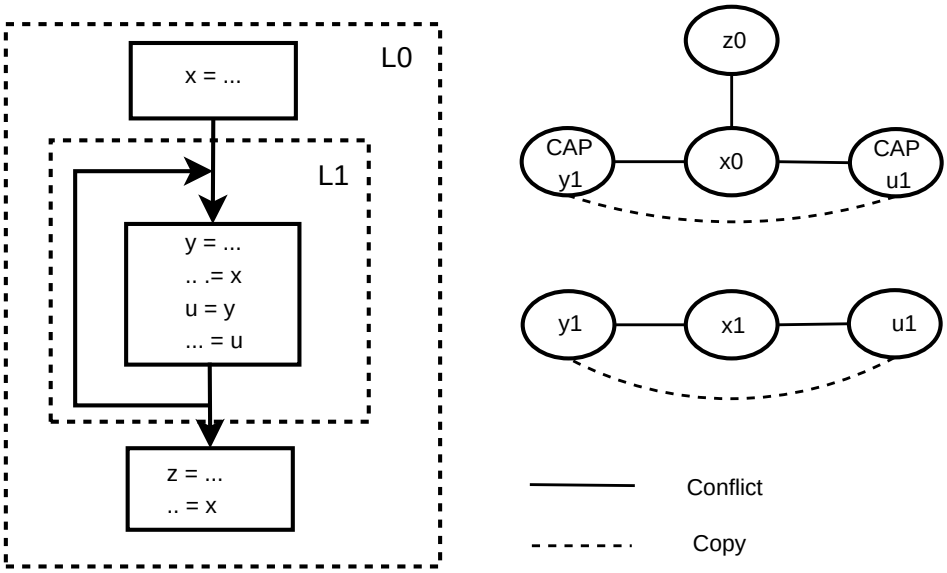

Figure 3: Regions, allocnos, copies

a result, IRA will try to reuse the same hard-register for the output allocno as the one used for the input allocno. My experience shows that such an heuristic is even more important than Briggs *biased colouring*.

IRA does not create copies between the same register allocnos from different regions because we use another technique for propagating hard-register preference on the borders of regions.

If a pseudo-register does not live in a region but lives in a nested region, we create a special allocno called a *cap* in the upper region.

Allocno (including caps) for the upper region in the region tree *accumulate* all the information from allocnos with the same pseudo-register from nested regions. This includes hard-register and memory costs, conflicts with hard-registers, allocno conflicts, allocno copies and some others. Thus, attributes for allocnos on the highest level corresponding to the function have the same values as we used for the allocation only one region which is the entire function.

Figure 3 illustrates the accumulation of allocno information. In this example we have two regions `L0` corresponding the whole function and `L1` corresponding to a loop. Pseudo-register `z` does not live in the loop and only one allocno `z0` representing it in region `L0` is created. Pseudo-register `x` lives in the two regions and two allocnos `x1` and `x0` are created to represent live ranges in the regions. Pseudo-registers `u` and `y` live only in the loop. Allocnos in the loop corresponding to the two pseudo-registers are named as `y1` and `u1`. We create two caps representing these allocnos in the region `L0`. Move instruction of pseudo-register `y` to `u` is represented by a copy referring to allocnos `u1` and `y1`. This copy on a higher level is represented by a copy referring to the corresponding caps.

2.2 Colouring

After building the data structures needed for the allocation, IRA does colouring. It starts with the root region and does colouring for allocnos (including caps) in the region, than goes recursively to immediately nested regions and does the same.

We use Briggs *optimistic* colouring which is a major improvement on Chaitin's colouring. The colouring consists of two passes. On the first pass we put allocnos onto the *colouring stack*. On the second pass we take allocnos from the stack and assign hard-registers to them.

IRA supports two *buckets*: one for trivially colourable allocnos and another one the other allocnos in the region. An allocno is *trivially colourable* if the number of conflicting hard-registers plus the number of hard-registers needed for a given allocno and for allocnos from the same cover class conflicting with a given allocno is not greater than the number of hard-registers of the cover class available for the allocation.

On the first pass IRA takes the first allocno[3] from the colourable allocno bucket, puts it on the stack, conditionally removes it from the conflict graph and moves any conflicting allocnos from the uncolourable bucket to the colourable one if the conflicting allocno becomes trivially colourable after removing the given allocno from the conflict graph. If there are no allocnos in the colourable allocno bucket, we choose an allocno from the uncolourable allocno bucket with the minimum following cost:

$$Cost_{mem} - Cost_{reg} + \begin{cases} 0 & \text{for cap} \\ Cost_{move} & \text{for hard-register} \\ -Cost_{ldst} & \text{for memory} \end{cases}$$

$Cost_{mem}$ and $Cost_{reg}$ are cost of usage of memory and a hard-register of the allocno's cover class for an allocno inside the region. If the corresponding allocno in the father region is a cap, we do not change the cost. If the corresponding allocno is not a cap and is assigned to a hard-register, the cost is increased by $Cost_{move}$ which is the cost of move instructions on the entry and exit region's edges on which the corresponding pseudo-register lives. If the corresponding allocno in the father region is not a cap and is assigned to memory, the cost is decreased by $Cost_{ldst}$ which is the cost of load instructions on the entry region's edges and the costs of store instructions on those of the exit region's edges on which the pseudo-register lives.

An allocno taken from the uncolourable allocno bucket is processed in the same way as one from the colourable bucket. This allocno is chosen to be spilled to colour allocnos still represented in the conflict graph, but it still can get a hard-register on the second pass.

On the second pass, we pop allocnos from the stack and assign hard-registers to them. If this process fails[4] or memory usage is less costly, we assign memory. We assign to the allocno the first available hard-register with minimal cost:

$$Cost_{ra} - ConflictAllocnoCost_{ra} + CopyAllocnoCost_{ra}$$

where

$$ConflictAllocnoCost_{ra} = \sum_{c \in Conflict(a)} ConflictCost_{rc}$$
$$CopyAllocnoCost_{ra} = \sum_{c \in Connect(a)} ConflictCost_{rc}$$

$Cost_{ra}$ is the cost of using the hard-register r for the allocno a. $Conflict(a)$ is a set of unassigned allocnos conflicting with allocno a. $ConflictAllocnoCost_{ra}$ reflects the preferences of the unassigned conflicting allocnos. $Connect(a)$ is a set of unassigned allocnos connected to allocno a by copies. $CopyAllocnoCost_{ra}$ reflects the preferences of the unassigned allocnos connected by the copies.

Right after assigning a hard-register to an allocno, cost and conflict cost of the hard-register for the allocnos connected to the given allocno by copies is decreased by $Freq_{copy} \cdot Cost_{move}$. Such modification results in the hard-register becoming more preferable, and a real or potential move corresponding to the copy will be removed. Thus, the effect will be the same as pseudo-register coalescing in the Chaitin-Briggs allocator.

After assigning hard-registers to allocnos in the region, IRA modifies the costs for allocnos in the immediately nested regions to propagate the preferences of allocnos and minimize register shuffling on the nested region border. We have two cases: in one case the allocno was assigned to a hard-register R and in another case the allocno was assigned to memory. The new costs of an allocno in the nested region for memory and the hard-register R correspondingly will be following

$$Cost_{mem} + \begin{cases} Cost_{st} \cdot Freq_{enter} + Cost_{ld} \cdot Freq_{exit} & \text{for register} \\ -Cost_{ld} \cdot Freq_{enter} - Cost_{st} \cdot Freq_{exit} & \text{for memory} \end{cases}$$

$$Cost_R + \begin{cases} Cost_{move} \cdot (Freq_{enter} + Freq_{exit}) & \text{for register} \\ 0 & \text{for memory} \end{cases}$$

Here, $Cost_{mem}$ and $Cost_r$ are costs of using memory and a hard-register R for the allocno in the nested region before their modifications. $Freq_{enter}$ and $Freq_{exit}$ are the execution frequencies of, correspondingly, all enter edges into and exit edges from the nested region. $Cost_{st}$ and $Cost_{ld}$ are costs of, correspondingly, storing and loading a hard-register from the allocno's cover class. $Cost_{move}$ is the cost of moving hard-registers of the allocno's cover class.

IRA does not modify the costs for allocnos from the corresponding caps in the father region because there will be no move instructions on the region borders. Although

[3] Actually, we always take the least frequently used allocno from the bucket. As a result, more frequently used allocnos will be taken from the stack first and assigned first, and they will have more chances to get their preferable hard-registers than less frequently used allocnos. This is a small but important improvement on Chaitin-Briggs colouring.

[4] Assigning a hard-register to an allocno usually fails if we put the allocno from the uncolourable allocno bucket on the stack.

such an allocno can get a better hard-register, the algorithm tends to assign the same hard-register to an allocno as to the corresponding cap because of the modified conflict costs of allocnos not represented by caps in the father region. The experiments show the importance of such special treatment for the caps.

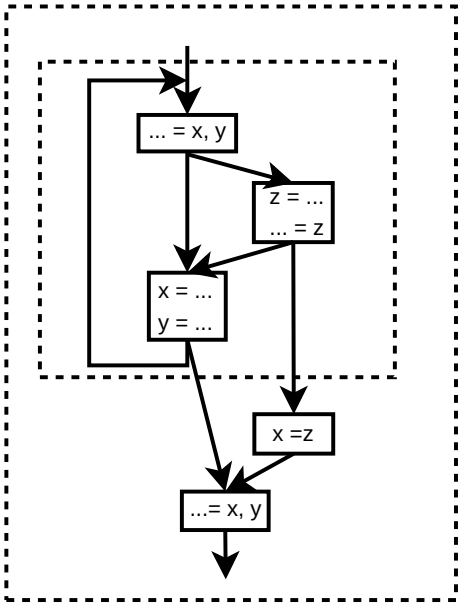

Figure 4: Example showing a drawback of bottom-up colouring

The proposed top-down allocation algorithm used by IRA is simpler and better than the Callahan-Koblenz [Callahan91] algorithm which tries to assign hard-register substitutions on the first bottom-up pass and then maps them into the real hard-registers on the second top-down pass. Figure 4 illustrates the drawback of their approach. There is a possibility that the algorithm assigns the same hard-register for pseudo-registers `y` and `z` when processing the loop on the first bottom-up pass. The right choice would be assigning the same hard-register for `x` and `z` because it removes an unnecessary move instruction outside the loop. The IRA top-down allocation algorithm does not have such a drawback.

2.3 Spill/restore code moving

When IRA does allocation traversing regions in top-down order, it does not know what happens below in the region tree. Therefore, sometimes IRA misses opportunities to do a better allocation. Figure 5 illustrates this. Let's look at allocnos in different regions for the same pseudo-register. After assigning a hard-register to an allocno in region `L0`, IRA assigns the same register to an allocno in region `L1` to remove register shuffling on the border between regions `L0` and `L1`. In the region `L2`, IRA fails to allocate a hard-register to the allocno in `L2`. By using memory for the allocno in region `L1`, we could move the spill/restore instructions generated later from the border between `L1` and `L2` to less frequently used points on the border between `L0` and `L1`. If such a transformation is profitable, it is worth doing.

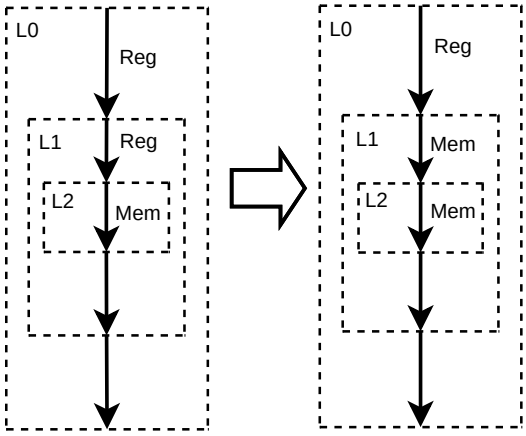

Figure 5: Spill/Restore code movement

IRA has a special pass to move spill/restore or register shuffling code to higher level regions. This pass could be considered an additional, very simple, bottom-up allocation. The pass implements a simple iterative algorithm performing profitable transformations while they are possible. It is fast in practice, so there is no real need for a better time complexity algorithm.

2.4 Emitting code for register shuffling

The same pseudo-register allocnos outside and inside a region may be assigned to different locations (hard-registers or memory). In this case we create and use a new pseudo-register inside the region and add code to move allocno values on the region's borders. It is done in during top-down traversal of the regions.

The new pseudo-register and the move code is not generated when both allocnos (one inside a region and another one outside) have been assigned to memory. Although the reload pass tries to use the same stack slot for different pseudo-registers involved in a move, sometimes it fails. A potentially smaller stack frame because

of shorter pseudo-register live ranges is less important than danger of generating memory to memory moves. This is probably obvious and my experience shows it.

IRA still creates a pseudo-register and moves on the region borders even when the both allocnos were assigned to the same hard-register. If the reload pass spills a pseudo-register for some reason, the effect will be smaller because another allocno will still be in the hard-register. In most cases, it is better then spilling both allocnos. If the reload does not change allocation for the two pseudo-registers, the trivial move will be removed by post-reload optimizations. The reload pass also can assign a different hard-register to a pseudo-register and as a result an irremovable move of the hard-registers will be generated. But the cases when the reload pass changes the assigned hard-register is more rare, especially for a pseudo-register with a long live range. Experience shows that the danger of generation of additional moves with different hard-registers is smaller than that of potentially spilling both allocnos.

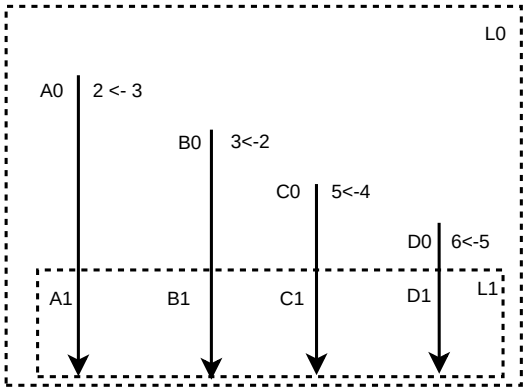

Figure 6: Example for emitting code

The generation of code moving the same pseudo-register allocno values on region borders is not so trivial. Figure 6 illustrates that. First of all, the order of moves is important. For example, we should execute a move for the allocnos `D0` and `D1` before one for `C0` and `C1`, otherwise the value of `D0` in hard-register `5` will be corrupted. There can be a loop in dependencies of the moves, such as for allocnos `A0` and `A1` and `B0` and `B1` whose hard-registers are swapped. For dependency loops a new temporary pseudo-register is generated and assigned to memory. A reader can find many common things of the task with out-of-SSA pass in compilers [Morgan98]. Some architectures have a swap instruction, and this instruction could be used for register shuffling without a temporary pseudo-register.[5] IRA will be able use such instructions in the future.

The move instructions are generated on the enter and exit edges of regions. Usually there is a lot of duplication on such edges. IRA makes the code unifications, moving some or all moves into the common destination or source of the edges.

After emitting code, we rebuild IR without taking loops into account. Starting from here, the notion of allocno is the same as the notion of pseudo-register because an allocno's structure corresponds to all live ranges of the pseudo-register.

2.5 Caller save optimization

The current GCC register allocator generates save/restore code around the calls through which pseudo-registers assigned to call-used hard-registers are living. It is done during the reload pass. The code responsible for generating the save/restore code is located in the file `caller-save.c`. The current GCC RA does a simple optimization to remove redundant save/restore code generation in the scope of one basic block.

Although IRA can use the code in file `caller-save.c`, by default IRA performs generation of the save/restore code before the reload pass. IRA splits the live range of pseudo-registers living through calls which are assigned to call-used hard-registers in two parts: one range (a new pseudo-register is created for this) which lives through the calls and another range (the original pseudo-register is used for the range) lives between the calls. Memory is assigned to new pseudo-registers. Move instructions connecting the two live ranges (the original and new pseudo-registers) will be transformed into load/store instructions in the reload pass.

IRA does global save/restore code redundancy elimination. It calculates points to put save/restore instructions according the following data flow equations:

$$SaveOut_b = \bigcap_{p \in pred(b)} (SaveIn_p \bigcap \overline{SaveIgnore_{pb}})$$

[5] Actually, a swap operation can be implemented by three *xor* instructions without a temporary pseudo-register: see [Hendron93]. By the way, the algorithm based on colouring cyclic intervals as it is described in [Morgan98] was implemented and it worked worse in GCC environment.

$$SaveIgnore_{pb} = \begin{cases} \oslash & depth(b) \le depth(p) \\ Ref_{loop(b)} & depth(b) > depth(p) \end{cases}$$

$$SaveIn_b = (SaveOut_b - Kill_b) \bigcup SaveGen_b$$

$$RestoreIn_b = \bigcap_{s \in succ(b)} (RestoreOut_s \bigcap \overline{RestoreIgnore_{bs}})$$

$$RestoreIgnore_{bs} = \begin{cases} \oslash & depth(b) \le depth(s) \\ Ref_{loop(b)} & depth(b) > depth(s) \end{cases}$$

$$RestoreOut_b = (RestoreIn_b - Kill_b) \bigcup RestoreGen_b$$

Here, $Kill_b$ is the set of allocnos referenced in basic block b and $SaveGen_b$ and $RestoreGen_b$ is the set of allocnos which should be correspondingly saved and restored in basic block b and which are not referenced correspondingly before the last and after the first calls they live through in basic block b. $SaveIn_b$, $SaveOut_b$, $RestoreIn_b$, $RestoreOut_b$ are allocnos correspondingly to save and to restore at the start and the end of basic block b. Save and restore code is not moved to more frequently executed points (inside loops). The code can be moved through a loop unless it is referenced in the loop (this set of allocnos is denoted by Ref_{loop}).

We should put code to save/restore an allocno on an edge (p,s) if the allocno lives on the edge and the corresponding values of the sets at end of p and at the start of s are different. In practice, code unification is done: if the save/restore code should be on all outgoing edges or all incoming edges, it is placed at the edge source and destination correspondingly.

Putting live ranges living through calls into memory means that some conflicting pseudo-registers[6] assigned to memory have a chance to be assigned to the corresponding call-used hard-register. IRA does that by using simple priority-based colouring for the conflicting pseudo-registers. The bigger the live range of pseudo-register living through calls, the better such a chance is. Therefore, IRA moves spill/restore code as far as possible inside basic blocks.

The implementation of save/restore code generation before the reload pass has several advantages:

- simpler implementation of sharing stack slots used for spilled pseudos and for saving pseudo values around calls. Actually, the same code for sharing stack slots allocated for pseudos is used in this case.
- simpler implementation of moving save/restore code to increase the range of memory pseudo can be stored in.
- simpler implementation of improving allocation by assigning hard-registers to spilled pseudos which conflict with new pseudos living through calls.

The disadvantage of such an approach is mainly in the reload pass, whose behavior is hard to predict. If the reload pass decides that the original pseudos should be spilled, save/restore code will be transformed into a memory-memory move. To remove such nasty moves, IRA is trying to use the same stack slot for the two pseudos. It is achieved using a standard preference technique to use the same stack slot for pseudos involved in moves. A move between pseudos assigned to the same memory could be removed by post-reload optimizations, but it is implemented in the reload pass because, if it is not done earlier, a hard-register would be required for this and most probably a pseudo-register would be spilled by the reload to free the hard-register.

2.6 Changes in the reload pass

The reload pass is responsible in GCC for assigning stack slots to pseudo-registers not assigned to hard-registers. The reload tries to share stack slots only for pseudo-registers spilled during the reload.[7] The reload does not do it for pseudo-registers assigned to memory by RA. The reload also assigns stack slots to pseudo-registers in the same order as the pseudo-register numbers.

IRA keeps all information about pseudos, including their conflicts, when it calls the reload. The reload for IRA assigns stack slots to most frequently used pseudos first if the frame pointer is used for stack slot addressing. If the stack pointer is used for stack slot addressing, the reload assigns to less frequently pseudos first according to the following heuristics (the first rule has higher priority):

[6] Such pseudo-registers should not live through calls.

[7] Such pseudo-registers were assigned to hard-registers by RA before the reload pass.

- Use the stack slot which results in removing memory-memory move instructions with the biggest overall cost.
- Use the first fit memory slots.

This heuristics result in removing costly memory-memory move instructions and allocation stack slots with smaller displacements for more frequently used pseudo-registers. The later results in smaller instructions for some targets as *x86* and *x86_64* and better code locality as a result.

When the reload evicts a pseudo-register from a hard-register, it tries to reassign another hard-register by calling the function `retry_global_alloc`. The analogous function is implemented in IRA, being called `retry_ira_color`.

The last important change of the reload pass is already mentioned in the previous section. IRA generates moves shuffling pseudo-registers. Although IRA never generates moves for pseudo-registers which are both assigned to memory, such moves can be transformed into memory-memory moves because the reload can spill some pseudos. According to the stack slot assigning heuristic mentioned above, the move will contain the same memory slot and finally will be represented by load and store instructions using a hard-register.[8] Although post-reload optimizations can remove such redundant load and store instructions, it is important to remove memory-memory moves in the reload before the reload provides the hard-register, probably by spilling one more pseudo-register. Removing early on the instructions moving pseudo-registers assigned to the same stack slot has been implemented in the reload pass for IRA.

3 Inter-procedural register allocation

Currently IRA uses a very simple inter-procedural register allocator. Each allocno has information about all of the calls it lives through. If a function and all functions called by the function directly or indirectly have already been compiled, we have information about all actually used hard-registers which can be clobbered by a caller according to ABI. This information permits us to choose a better call-used hard-register and to omit save/restore code around a call in many cases when the assigned call-used hard-register was not clobbered by the call.

To implement inter-procedural register allocation, the GCC call graph infrastructure is used and GCC code was modified in the following way:

- A new field `function_used_regs` was added to the function call graph node. The initial value of the field is all hard-registers clobbered by a call according to ABI.
- If all called functions can be identified in a given function, at the end of function compilation (the GCC final pass) the value of the field `function_used_regs` is set up as union of the call-used hard-registers which are used in the given function and all values of `function_used_regs` from call graph nodes corresponding to all of the called functions.
- In all places in GCC where a call is processed, the value of `function_used_regs` of the call graph node corresponding to the called function is used instead of the hard-registers clobbered by a call according to the ABI.

Although such an implementation of the inter-procedural register allocation improves the code in many cases, it has a drawback. It tends to use call-used hard-registers without generation of save-restore code in functions close to leaf functions in the call graph. Usually such functions are less frequently executed. Thus, there is room to improve the current inter-procedural register allocation.

4 Experimental results

To see how well IRA can work, it is better to use an architecture with a small irregular register file. *X86* is the best known architecture for this. We use the GCC development version between releases 4.2 and 4.3 and the SPEC2000 benchmark suite, which is probably the most widely used benchmark for compilers. Our test machine is an Intel 2.66Ghz Core2 machine with 4GB memory under Red Hat Enterprise Linux version 4.4. Common options used for all runs are `-O2 -mtune=generic -m32 -mmmx -msse -msse2 -msse3` which means

[8] If the target architecture has a memory-memory move instruction, the hard-register will be not necessary.

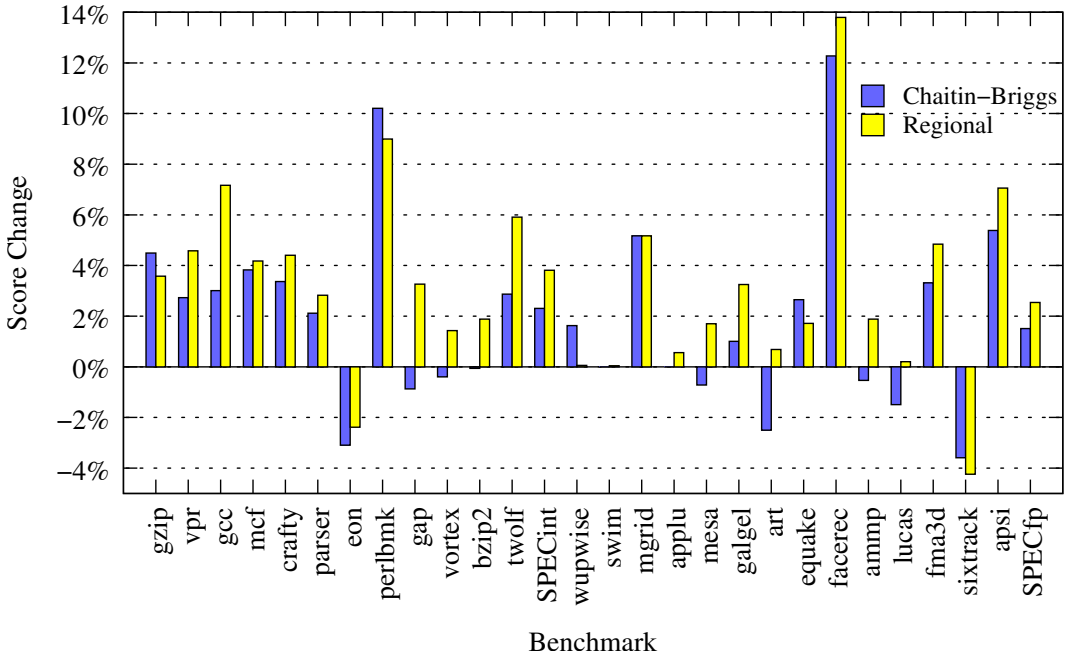

Figure 7: SPEC2000 score change.

that `SSE` registers are used instead floating point stack registers.[9] SPEC2000 results are given on Figure 7 for IRA using Chaitin-Briggs and regional register allocation algorithms without inter-procedural register allocation in comparison to the original GCC register allocator. Text segment sizes of the benchmarks are given on Figure 8.

As we can see, the regional register allocation is the best algorithm generating about 4% better SPECInt2000 code. It deals well with high-register pressure and additional code for register shuffling on the loop borders has less significant effect.

5 Current status and future directions

IRA is now implemented for the following targets: *ARM*, *x86*, *x86-64*, *PowerPC*, *ia64*, *S390*, and *SPARC*. The author continues the work to porting IRA to other architectures. Besides the porting, IRA can be and will be improved in the following directions:

- Better inter-procedural allocation. As mentioned above the current implementation of the inter-procedural register allocator tends to use call-used hard-registers without generating save-restore code in functions close to leaf functions in the call graph. Usually such functions are less frequently executed. There are many possibilities for improving inter-procedural allocation. The simplest would be to use the inter-procedural allocation only for frequently called functions. The most sophisticated way is to implement optimal inter-procedural register allocation as it is described in [Kurlander96].

- Rematerialization. Register rematerialization is a way to improve code by recalculating pseudo-register values instead of reloading them from memory. The reload performs register rematerialization for the simple case when the value is a constant. A more common case is recalculating the value from other pseudo-registers which were assigned to hard-registers. Probably it would be better to do it after the reload, because even if IRA decides to assign a hard-register to a pseudo-register the reload pass can reconsider its decision and spill the pseudo-register to memory. In this case, the resulting rematerialization code will be much worse than the original restore code.[10]

[9] Floating point stack is quite specific to `x86` therefore we prefer the regular `SSE` register file to show how IRA works in the general case.

[10] The author may be wrong. The complexity of implementing rematerialization after the reload might not be worth the potential improvement.

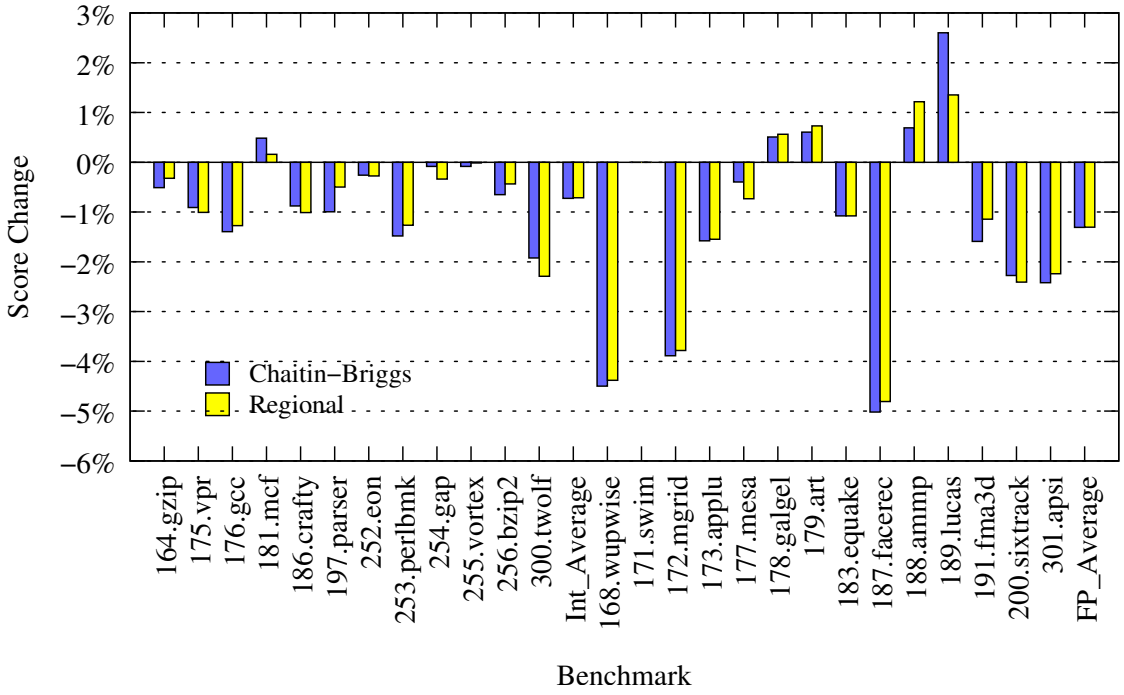

Figure 8: SPEC2000 code size change.

- Register allocation is the most machine-dependent part of GCC. Most dependencies are hidden in the reload pass. The reload pass has evolved and being refined for twenty years. Its code is very complicated and removing it without losing generated code quality is an enormous task. That is what the YARA project has shown. Still, there is necessity to rewrite the pass because there is very little communication (it is also mostly one way communication) with other register allocation passes. It needs to be done if we want to achieve further register allocation improvements.

6 Acknowledgments

I am grateful to my company Red Hat for the attention to improving GCC and for permitting me to work on this project. I would like to thank my colleague Andrew MacLeod for providing interesting ideas and his rich experience in register allocation.

Last but not least, I would like to thank my son, Serhei, for his help in proofreading the article.

References

[Appel96] L. George and A. Appel, *Iterated Register Coalescing*, ACM TOPLAS, Vol. 18, No. 3, pages 300-324, May, 1996.

[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.

[Callahan91] D. Callahan and B. Koblenz, *Register Allocation via Hierarchical Graph Coloring*, SIGPLAN, , Vol. 26, No. 6, pages 192-203, June 1991.

[Chaitin81] G. J. Chaitin, et. al., *Register allocation via coloring*, Computer Languages, Vol. 6, pages 47-57, Jan. 1981.

[Chow90] F. Chow and J. Hennessy. *The Priority-based Coloring Approach to Register Allocation*, TOPLAS, Vol. 12, No. 4, 1990, pages 501-536.

[Kurlander96] S. M. Kurlander and C. N. Fischer, *Minimum Cost Interprocedural Register Allocation*, Proceedings of the 23rd ACM SIGPLAN, 1996.

[Hendron93] I. J. Hendron, G. R. Gao, E. Altman, and C. Mukerji, 1993. *Register allocation using cyclic interval graphs: A new approach to an old problem*. (Technical Report.) McGill University.

[Lueh00] Guei-Yuan Lueh, Thomas Gross, Ali-Reza Adl-Tabatabai, *Fusion-Based Register Allocation*, ACM TOPLAS, Vol. 22, No. 3, pages 431-470, May 2000.

[MacLeod05] A. MacLeod. *RABLE - Register Allocation by Logic Engines. Initial Architecture.* http://gcc.gnu.org/ml/gcc/2005-11/msg00783.html.

[Makarov04] V. Makarov, *Fighting register pressure in GCC*, Proceedings of GCC Summit, 2004.

[Makarov05] V. Makarov, *Yet another GCC register allocator*, Proceedings of GCC Summit, 2005.

[Matz03] M. Matz, *Design and Implementation of a Graph Coloring Register Allocator for GCC*, Proceedings of GCC Summit, 2003.

[Morgan98] Robert Morgan, *Building an Optimizing Compiler*, Digital Press, ISBN 1-55558-179-X.