Submitted for review to: *The 2008 International Symposium on Code Generation and Optimization*

The top-down regional register allocator for irregular register file architectures

Vladimir. N. .Makarov
RedHat
vmakarov@redhat.com

Abstract

This paper presents a register allocator which performs graph coloring on a top-down traversal of nested regions. Register coalescing, live range splitting, and choosing of better hard registers for architectures with irregular register files are done in an integrated way through a register preferencing technique based on dynamically changing hard register costs. As a regional allocator, the top-down regional allocator is simpler than the Callahan-Koblentz allocator based on bottom-up and top-down passes.

The proposed register allocator is compared with the Chaitin-Briggs and Callahan-Koblentz register allocators. The same infrastructure and register preferencing technique are used to implement all of these register allocators to ensure a fair comparison. The SPECInt2000 results show that the top-down regional register allocator works best for x86 in the GCC environment.

Introduction

Modern processors have several levels of storage with different access speeds. The faster the storage, the smaller its size. This is the consequence of a trade-off between the storage's speed and its price. The fastest storage units [1] are registers or *hard-registers*. 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. Such an approach permits to manage the complexity of the optimizations but it requires a special pass to map pseudo-registers onto hard-registers and memory. This pass is called *register allocator*.

Nowadays, optimizing compilers implement aggressive optimizations (like inter-procedural or whole program optimizations) which tend to create register pressure[2] which is higher than the number of available hard-registers. The number of available hard-registers cannot be increased because it is a part of architecture. The situation is even worse because some widely used architectures (e.g. x86 and ARM) have few hard-registers.

[1] Registers usually can be addressed only directly.

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

Therefore efficient usage of hard-registers for storing pseudo-register values is very important and, as a consequence, a good register allocator is a significant component of a modern optimizing compiler.

The most popular approach for assigning hard registers to pseudo-registers is based on the *graph coloring* algorithm initially proposed by Chaitin [4] and significantly improved by Briggs [2]. Besides assigning hard registers or memory to pseudo-registers, the register allocator performs other transformations like *register coalescing* and *register live range splitting*.

Register coalescing is to remove unnecessary moves of non-conflicting pseudo-registers by using just one pseudo-register. Usually such moves are generated by previous optimizations and during live range splitting. Too aggressive coalescing can worsen assigning hard registers to pseudo-registers. Live range splitting is based on the idea that if the live range of a pseudo-register is split into several parts, the pseudo-register in each part will conflict with fewer other pseudo-registers; less hard-registers will be needed for all the pseudo-registers. But there is the danger that moves generated to split pseudo-register live ranges overweight better register assigning. As we can see, register coalescing and live range splitting are tightly connected to assigning hard-registers to pseudo-registers based on graph coloring because both can affect the graph's colorability.

Approach used in classical Chaitin-Briggs [4, 2] and George-Appel [6] register allocators consists of different passes solving separate tasks such as register coalescing, coloring and live range splitting, usually in an *iterative* manner. Although such low integration of the major register allocation tasks makes the abovementioned register allocators conceptually simple, better integration is still needed to improve the register allocator. Probably a good integration of register coalescing and coloring has been achieved through *optimistic coalescing* [12], this is not true for register live range splitting despite some attempts [1].

Advanced regional register allocators like those based on graph fusion [9] and the simpler Callahan-Koblenz [3] allocator solve the major register allocation tasks in a more integrated but complicated manner.

One more regional register allocator solving the major tasks of register allocation in a more integrated way is proposed in this paper. The proposed register allocator is based on coloring nested regions on *top-bottom* traversal of them. The proposed register allocator is simpler than the Callahan-Koblenz allocator. It generates better code (at least in the GCC environment) than Callahan-Koblenz and Chaitin-Briggs ones.

For the purpose of graph coloring, all hard-registers are usually divided into non-intersecting *register classes*, e.g. integer registers or floating point registers. All pseudo-registers are then assigned to register classes and a hard register of a given class can only be assigned to a pseudo-register of the same class. Some of the most widely used architectures like x86, however, have *irregular register files*. Irregularity means that different registers of

one class may have different characteristics, such as access time, ability to be used as part of a memory address or to be used in certain instructions etc. The register file's irregularities complicate register allocation. Although the problem has been discussed in the context of assigning hard registers based on graph coloring (e.g. in [13]), the problem is rarely discussed in the context of other register allocation tasks.

Let us consider how the register file irregularities of x86 can affect the decision to do register coalescing and live range splitting. If we have a move instruction `A` to `B` where `A` and `B` are non-conflicting pseudo-registers assigned to the x86 general purpose register class and the best hard-registers for `A` and `B` are correspondingly `EAX` and `EDX`, the decision to coalesce the move would be dependent on the cost of the move, the possibility and profit of using hard register `EAX` for `A` and `EDX` for `B` in comparison to using the best hard register for the pseudo-register resulting from the coalescing. Analogously, the decision to split a pseudo-register at given point would be dependent on the cost of the move at this point, the possibility and profit of using the best hard-registers for the two created live ranges in comparison to using the best hard-register for the whole pseudo-register live range.

To do register coalescing, live range splitting, and choosing better hard register for architectures with irregular register files in an integrated way, the *register preferencing* technique based on dynamically changing hard register costs is also proposed in this paper.

The first section describes the proposed regional register allocator and its implementation in GCC: its passes and algorithms. The second section briefly describes the Callahan-Koblenz approach to register allocation, its implementation in GCC, and drawbacks of the approach. The third section gives a comparison of the GCC implementation of the proposed register allocator, Callahan-Koblenz and Chaitin-Briggs on SPECInt2000 benchmarks. The fourth section contains a summary of the paper.

1. The proposed register allocator

The proposed register allocator (or *RA* for abbreviation's sake) implements two coloring algorithms: Chaitin-Briggs coloring and regional top-down coloring. This section describes only regional coloring, as it is the most general case. Chaitin-Briggs coloring is a special case of regional coloring where only one region (the entire function) is used. Figure 1 shows the major passes of the RA and their order. Each pass is described in detail in subsequent sections.

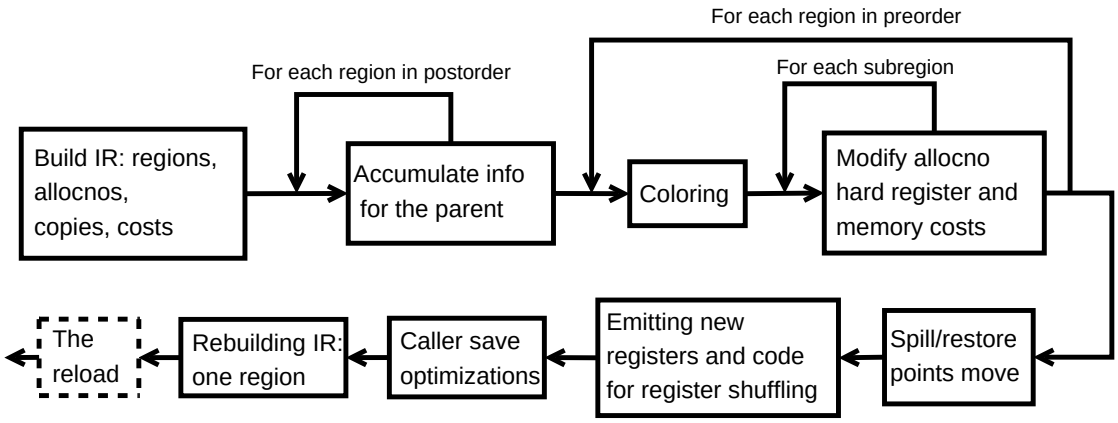

Figure 1. Passes in the RA.

1.1 Building the internal representation

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

GCC has a very powerful model for describing the target processor's register file. This model is built around the notion of register class. A register class is a set of hard-registers. As many registers classes can be described as necessary. 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 a register class should be defined 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 coloring, the register allocator uses a set of non-intersecting register classes. This set is called *the cover class set*. Register classes from the cover set should not intersect and should together contain all of the hard-registers available for register allocation. These are obligatory conditions which the RA checks. Usually there are many possible sets which satisfy these conditions. Experience shows that the register allocator generates the best code when classes for the cover set are chosen in such a way that 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 consisting of one class containing all the hard registers is an extreme example). This is possible because the register allocator assigns hard-registers to pseudo-registers based on their usage costs, and if memory is more profitable then the register allocator will use memory for a pseudo-register.

The register allocator works with several data structures. The central data structure of the RA is the *allocno*. An allocno represents the live range of a pseudo-register in a region. Besides the obvious attributes like the

corresponding *pseudo-register number*, *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. The RA can only assign a hard-register to an allocno from its cover class [3]. 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. The RA calculates and uses the cost of each cover class hard-register available for the allocation. First of all, the RA calculates the costs of allocnos for all register classes[4], and defines the best cover class for the allocno from this information. The costs of register classes define the costs of hard-registers from the cover class.

The cost of a *call-used*[5] 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. Initially, 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 has just received a hard-register, the cost of the hard-register is decreased. Before choosing a hard-register for an allocno, the allocno's current costs of the hard-registers are modified by the conflict hard-register costs of all of the 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, the RA uses this vector to calculate the final current cost of the available hard-registers. 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

[3] 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).

[4] Briefly, all region instructions and their operands are processed and the cost of usage of the considered register class (or memory) for a given operand is summarized for the allocno. For example, if memory is required for the operand, the cost is the load and store cost of the register class for the output and input operand correspondingly. In GCC it is more complicated because an IR instruction in GCC may represent several machine instructions with different operand constraints. Therefore we need two passes for accurate evaluation – for details see [10].

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

above. This is done to show other unassigned allocnos that a given allocno prefers some hard-registers in order to remove the move instruction corresponding to the copy (see below).

Another important RA data structure is the *copy*. Allocnos can be connected by copies. Copies are used to modify hard-register costs for allocnos during coloring. A copy can be used to 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 the given allocno by the copies is changed in order to try to allocate the same hard-register for the connected allocnos. In case of success, the move instruction represented by the copy will be removed. This procedure is analogous to register coalescing in the classic Chaitin-Briggs allocator.

Copies can be created not just for move instructions: because they are important for describing hard-register preference, the RA creates copies for operands of an instruction which should be assigned to the same hard-register due to constraints in the machine description file. A typical example of such an instruction is x86 architecture addition, which requires that the result of the addition should be placed in the same register as one of the operands. The RA creates two copies for the addition because it is an associative operation. One copy connects the allocnos corresponding to the addition's result and the first operand of the addition. Another one connects to the result allocno and the second operand.

The RA 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 a result, the RA 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 coloring*. The RA 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 outer region. A region cap is also created for a subregion cap.

1.2 Allocno information accumulation

Allocnos (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 more. *Thus, attributes for allocnos in a region have the same values as if the region had no subregions*. It also means that attributes for allocnos in the outermost region

corresponding to the function have the same values as though the allocation used only one region which is the entire function.

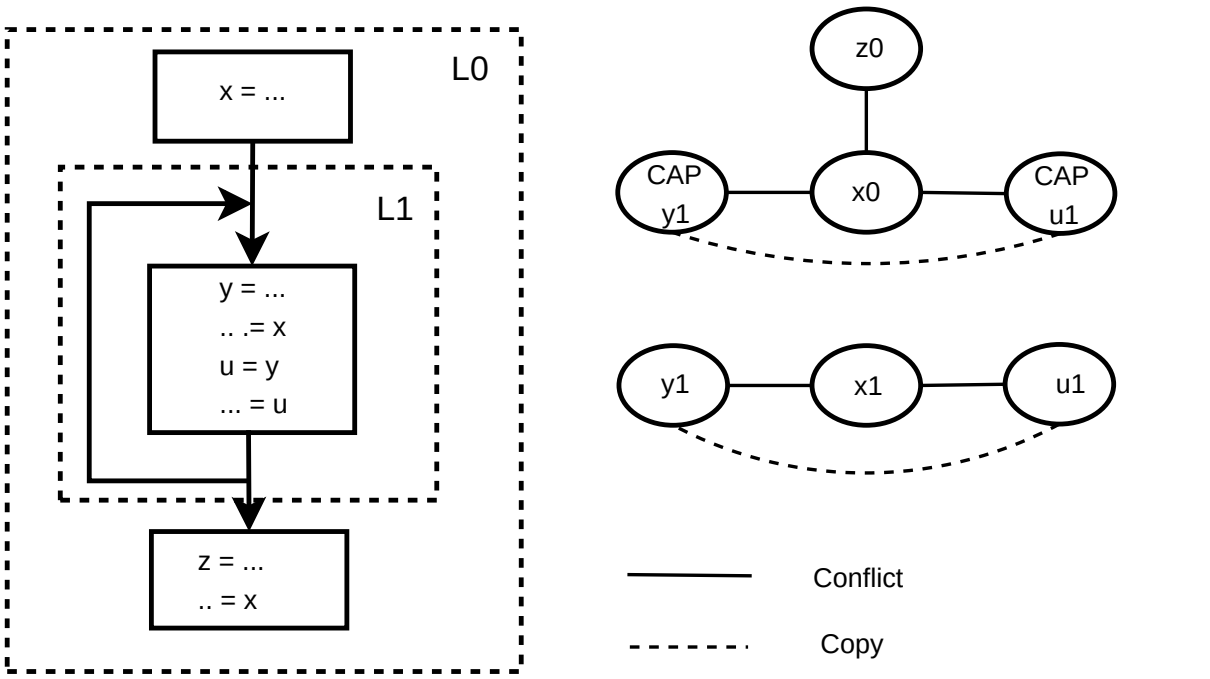

Figure 2. Regions, allocnos, copies.

Figure 2 illustrates the accumulation of allocno information. In this example we have two regions: `L0` corresponding to 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 `y1` and `u1`. We create two caps representing these allocnos in the region `L0`. A move instruction of pseudo-register `y` to `u` is represented by a copy referring to allocnos `u1` and `y1`. This copy is represented on a higher level by a copy referring to the corresponding caps.

1.3 Coloring

After building the data structures needed for the allocation, the RA does coloring. The process starts with the root region and performs coloring for allocnos (including caps) in the region, then goes recursively to immediately nested regions and does the same.

We use Briggs *optimistic* coloring which is a major improvement over Chaitin's coloring. The coloring consists of two passes. On the first pass we put allocnos onto the *coloring stack*. On the second pass we

take allocnos from the stack and assign hard-registers to them. The RA supports two *buckets*: one for trivially colorable allocnos and another one for the other allocnos in the region. An allocno is *trivially colorable* 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 the given allocno is not greater than the number of hard-registers from the cover class available for the allocation.

On the first pass the RA takes the first allocno[6] from the colorable allocno bucket, puts it on the stack, conditionally removes it from the conflict graph and moves any conflicting allocnos from the uncolorable bucket to the colorable one if the conflicting allocno becomes trivially colorable after removing the given allocno from the conflict graph. If there are no allocnos in the colorable allocno bucket, an allocno is chosen from the uncolorable allocno bucket with the minimum following cost:

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

Here, $Cost_{mem}$ and $Cost_{reg}$ are, respectively, the costs of usage of memory and of a hard-register of the allocno's cover class for an allocno inside the region. If the corresponding allocno in the parent 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 parent 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 uncolorable allocno bucket is processed in the same way as one from the colorable bucket. This allocno is chosen to be spilled in order to color allocnos still represented in the conflict graph, but it can still 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[7] or using memory is less costly, we assign memory. We assign to the allocno the first available hard-register with minimal cost[8]:

[6] 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 coloring.

[7] Assigning a hard-register to an allocno usually fails if we put the allocno from the uncolorable allocno bucket on the stack. It can still fail for an allocno taken from the colorable bucket, if the allocno needs more than one hard register.

[8] Calculations of the allocno costs for each hard-register might seem expensive. In reality, this code takes an insignificant amount of time even for the Itanium with its huge register file. Also the calculations can be easily vectorized by optimizing compilers.

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

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 the set of unassigned allocnos connected to allocno a by copies. $CopyAllocnoCost_{ra}$ reflects the preferences of the unassigned allocnos connected by the copies.

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

1.4 Modification of the costs in subregions

After assigning hard-registers to allocnos in the region, the RA modifies the costs for allocnos in the immediately nested regions to propagate the preferences of the allocnos and minimize register shuffling on the nested region's 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 for hard-register R (the costs for other hard-registers are not changed) correspondingly will be the 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 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 out of the nested region on which the corresponding pseudo-register lives. $Cost_{st}$ and $Cost_{ld}$ are the 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.

The RA does not modify the costs for allocnos from the corresponding caps in the parent region because there will be no move instructions on the region's borders. Although such an allocno can get a different (usually

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 parent region. Experiments show the importance of such special treatment for the caps.

1.5 Spill/restore code moving

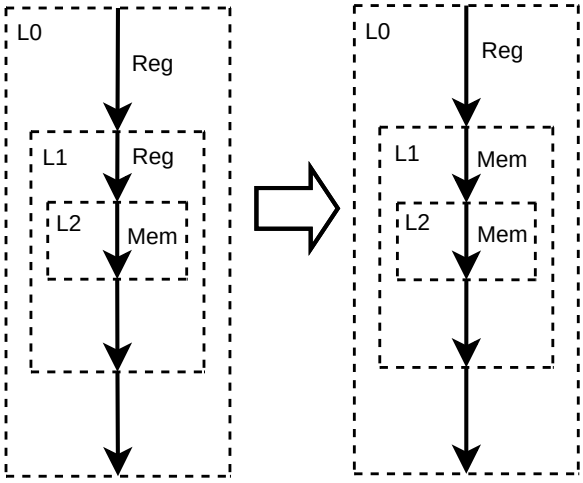

Figure 3. Spill/Restore code movement.

When the RA performs allocation traversing regions in top-down order, it does not know what happens below in the region tree. Therefore, sometimes the RA misses opportunities to perform a better allocation. Figure 3 illustrates this. Let's look at allocnos in different regions for the same pseudo-register. After assigning a hard-register to an allocno in the region `L0`, The RA assigns the same register to an allocno in the region `L1` to remove register shuffling on the border between regions `L0` and `L1`. In the region `L2`, the RA 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 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.

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

1.6 Emitting code for register shuffling

Two allocnos representing the same pseudo-register allocnos outside and inside a region respectively 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. This is done during top-down traversal of the regions.

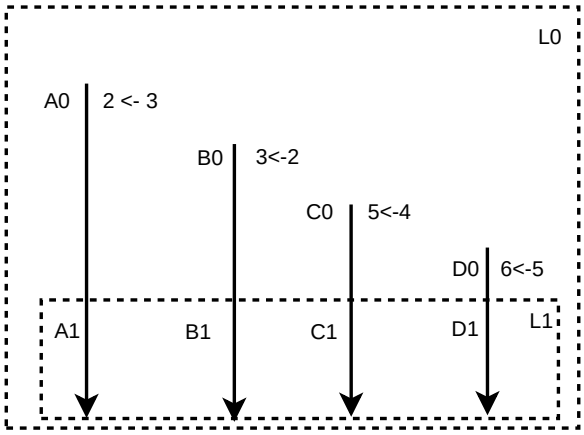

Figure 4. Example for emitting code.

The generation of code moving the same pseudo-register's allocno values on region borders is not so trivial. Figure 4 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 the 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. The reader can find many things in common between this task and the out-of-SSA pass in compilers [11]. Some architectures have a swap instruction, and this instruction could be used for register shuffling without a temporary pseudo-register[9]. The RA will be able use such instructions in the future.

Move instruction on the region exit edges is not created, if the move destination pseudo-register is allocated to memory and the pseudo-register value has not been changed in the region. As we can see, in a general case the move instructions are generated on the enter and exit edges of regions. Usually this results in a lot of duplication on such edges. The RA performs code unification, moving some or all moves into the common destination or source of the edges.

After emitting code, we rebuild the 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 corresponds to all live ranges of the pseudo-register.

The GCC implementation of the proposed register allocator has some GCC-specific features. The major reason for this is a pass called *reload* which GCC runs right after the register allocator. The reload is a very

[9] Actually, a swap operation can be implemented by three *xor* instructions without a temporary pseudo-register: see [7].

complicated pass. The complexity of the reload is a consequence of the fact that GCC is the most portable compiler, working on tens of platforms. The reload's major goal is to transform the GCC back-end's internal representation (*RTL*) into a form where all instruction constraints for its operands are satisfied. Pseudo-register are transformed into either hard-registers, memory or constants. The reload pass follows the assignment made by the register allocator, but it can change the assignment in some complicated cases if needed.

The GCC implementation of the proposed register allocator still creates a pseudo-register and moves on the region borders even when 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, this is better then spilling both allocnos. If the reload does not change the allocation for the two pseudo-registers, the trivial move will be removed by post-reload optimizations. The reload pass can also assign a different hard-register to a pseudo-register and as a result an irremovable move of the hard-registers will be generated. But cases when the reload pass changes the assigned hard-register are 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.

The new pseudo-register and the move code are 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 bigger stack frame because of longer pseudo-register live ranges is less dangerous compared to generating memory to memory moves. This is probably obvious and my experience shows it.

1.7 Caller save optimization

By default the RA performs generation of the save/restore code before the reload pass (the reload can do it by itself but in a less optimal way). RA splits the live range of the 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) which lives between the calls. The new pseudo-registers are assigned to memory. Move instructions connecting the two live ranges (the original and new pseudo-registers) will be transformed into load/store instructions in the reload pass.

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

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

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

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

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

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

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

Here, $Kill_b$ is the set of allocnos referenced in basic block b and $SaveGen_b$ and $RestoreGen_b$ are the sets 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's source and destination correspondingly.

Putting live ranges living through calls into memory means that some conflicting pseudo-registers[10] assigned to memory have a chance to be assigned to the corresponding call-used hard-register. RA does that by using simple priority-based coloring for the conflicting pseudo-registers. The bigger the live range of the pseudo-register living through calls, the better such a chance is. Therefore, the RA moves spill/restore code as far as possible inside basic blocks.

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

1.8 Rebuilding the internal representation

Rebuilding the internal representation at the end of the RA's work is specific to GCC. As mentioned above, GCC runs a pass called reload right after the register allocator. In rare cases when the reload evicts a pseudo-register from a hard-register, it can try to assign another hard-register to the pseudo-register. To do this the reload needs information from the RA (mainly the interference graph of the whole function). This is the reason for rebuilding the RA's internal representation for one region covering the whole function.

2. Implementation of the Callahan-Koblenz allocator

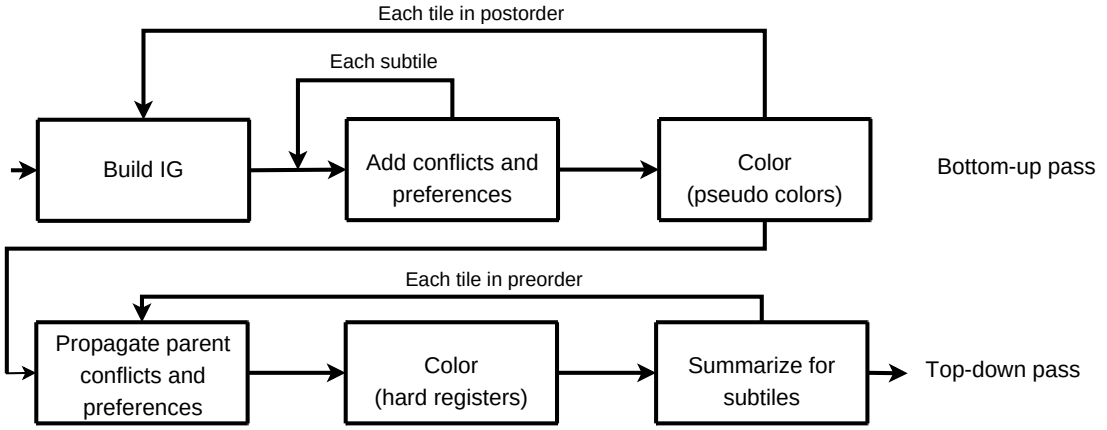

Figure 5. Passes in Callahan-Koblenz allocator.

For comparison purposes, the Callahan-Koblenz allocator [3] was also implemented as it is described in [5]. There are many important details in the Callahan-Koblenz allocator. The most accurate description of these details is probably given in [8]. The Callahan-Koblenz allocator is a two pass regional allocator (the regions are called *tiles*). Figure 5 shows the major passes of Callahan-Koblenz. Here is a very brief description of the allocator [11]:

- On the first bottom-up pass, the interference graph is built only for pseudo-registers referenced in the basic blocks of the tile. This means that we do not consider pseudo-registers that are only mentioned in the subtiles (local pseudo-registers) or pseudo-registers transparent to the tile.
- All local pseudo-registers which got the same register in a subtile are represented in the parent tile by a *tile summary variable* or *TSV*. Conflicts to the created TSVs and non-local pseudo-registers not referenced in the tile and assigned to registers in a subtile are added to the tile interference graph. We change spill costs for the globals living at the subtile's borders in a way to minimize the final spill/restore code on the borders.

[11] The usage of a preference technique to coalesce move instructions is omitted here in order to keep the description simple.

- Graph coloring of global pseudo-registers and TSVs using the preferences is done but only pseudo-colors [12] are assigned. They are used to show a desire to assign the same hard register for pseudo-registers and TSVs on the second pass.
- On the second top-down pass, pseudo-registers assigned to a hard-register in the parent and transparent to the tile are added to the tile interference graph. If a pseudo-register got a hard-register in the parent tile, the pseudo-register prefers the same hard-register in the tile. If a TSV got a hard-register in the parent tile, the pseudo-registers represented by the TSV prefer the same hard-register in the tile. This is done in order to minimize register shuffling on the tile borders.
- Graph coloring using the preferences and real hard-registers is done.
- Hard register preferences are created for the subtiles. This is called the summarization pass.

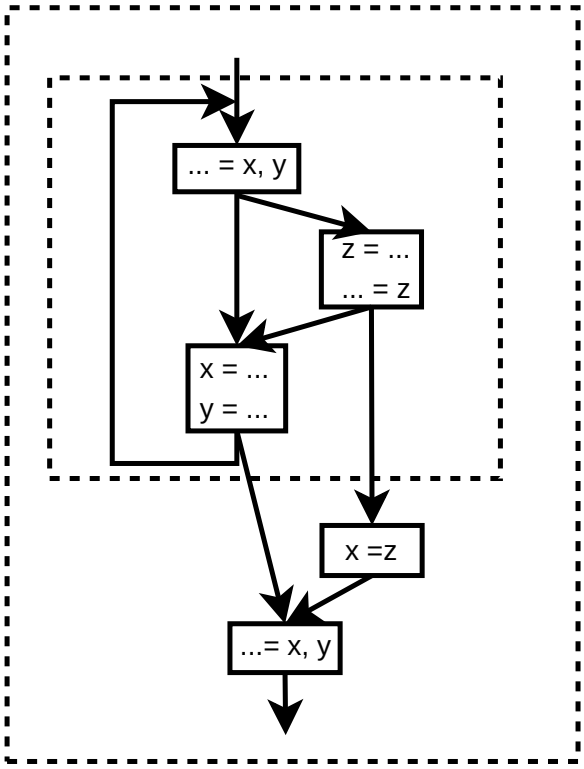

Figure 6. Example showing a drawback of bottom-up coloring.

To make a fair comparison of the allocators, the same infrastructure was used to implement the Callahan-Koblenz allocator. This means that we used the same data structures, passes (see Figure 1), and register

[12] A real hard register can still be assigned on the first pass if there is a move involving the pseudo-register and the hard-register or if the pseudo-register was assigned to the hard-register in a subtile.

preferencing technique as for the top-down regional allocator. The major difference was in the coloring pass where the Callahan-Koblenz algorithm was used instead of the top-down regional algorithm.

The proposed top-down allocation algorithm is simpler and better than the Callahan-Koblenz algorithm. Figure 6 illustrates the main drawback of the Callahan-Koblenz approach. There is a possibility that the algorithm will assign 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 to assign the same hard-register for `x` and `z` because it removes an unnecessary move instruction outside the loop. The top-down regional allocation algorithm does not have this drawback[13].

3. Experimental results

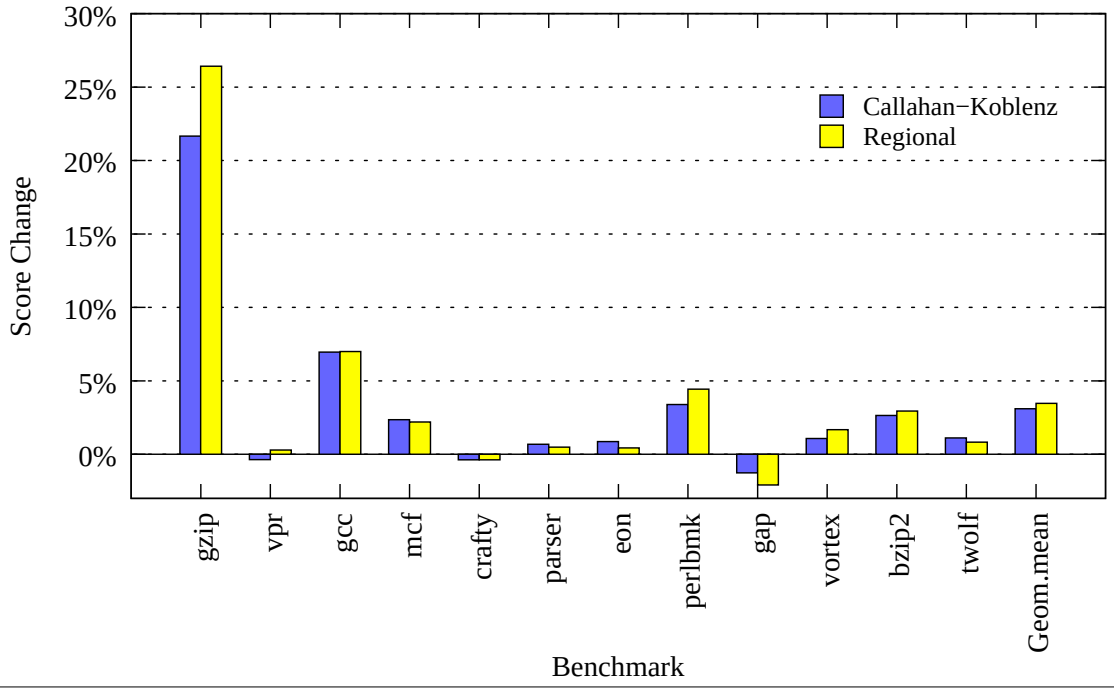

Figure 7. SPECInt2000 score change in comparison with Chaitin-Briggs allocator.

To see how well the top-down regional register allocation algorithm can work, it is better to use an architecture with a small, irregular, register file. *x86* is the best known architecture for this [14]. We use the GCC development version between releases 4.2 and 4.3 and the SPECInt2000 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

[13] Callahan-Koblenz allocator, however, has an advantage too: its conflict graphs are smaller, which can be important for compilation of large functions.

[14] The author found that in many cases GCC still generates code with small register pressure even for x86. The situation may be changed in the future after whole program optimizations with aggressive inlining are implemented in GCC and better escape analysis permits keeping more values in pseudo-registers.

under RedHat Enterprise Linux version 4.4. Common options used for all runs are `-O3 -mtune=generic -m32`. SPECInt2000 results are given in Figure 7 for the Callahan-Koblenz and the top-down regional register allocators in comparison to the Chaitin-Briggs register allocator. Text segment sizes of the benchmarks are given in Figure 8.

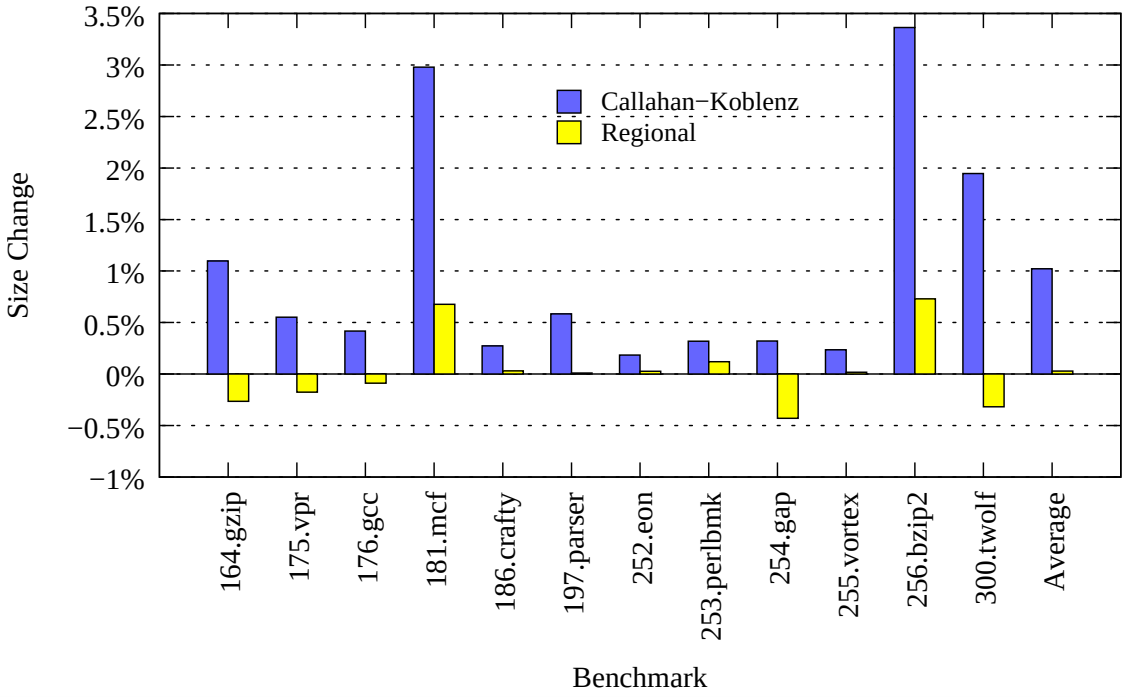

Figure 8. SPEC2000 code size change in comparison with Chaitin-Briggs allocator.

As we can see, the top-down regional register allocation is the best algorithm, generating faster code than Chaitin-Briggs one and smaller code than Callahan-Koblenz one. It deals well with high register pressure and the additional code for register shuffling on loop borders does not have a significant effect on performance.

4. Summary

The proposed top-down regional register allocator is simpler than the famous Callahan-Koblenz allocator and yet generates better code, at least in the GCC environment. The proposed *register preferencing* technique based on dynamically changed hard register costs simplifies the implementation of register coalescing, register live range splitting, and hard register selection for architectures with irregular register files. It also permits to solve all the three tasks in more integrated way.

References

[1] BERGNER, P., DAHL, P., ENGEBRETSEN, D., AND O'KEEFE, M. T. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation* (1997), pp. 287–295.

[2] BRIGGS, P., COOPER, K. D., AND TORCZON, L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 428–455.

[3] CALLAHAN, D., AND KOBLENZ, B. Register allocation via hierarchical graph coloring. *SIGPLAN 26*, 6 (1991), 192–203.

[4] CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., , AND MARKSTEIN, P. W. Register allocation via coloring. *Computer Languages 6*, 1 (1981), 47–57.

[5] COOPER, K., DASGUPTA, A., AND ECKHARDT, J. Revisiting graph coloring register allocation: A study of the chaitin-briggs and callahan-koblenz algorithms. In *Workshop on Languages and Compilers for Parallel Computing (LCPC05)* (2005).

[6] GEORGE, L., AND APPEL, A. W. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems 18*, 3 (May 1996), 300–324.

[7] HENDRON, I., GAO, G., ALTMAN, E., AND MUKERJI, C. *Register allocation using cyclic interval graphs: A new approach to an old problem*. (Technical Report) McGill University, 1993.

[8] KOBLENZ, B., AND CALLAHAN, D. Register allocation methods having upward pass for determining and propagating variable usage information and downward pass for binding; both passes utilizing interference graphs via coloring. *United States Patent 5,530,866* (1994).

[9] LUEH, G.-Y., GROSS, T., AND ADL-TABATABAI, A.-R. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems 22*, 3 (2000), 431–470.

[10] MAKAROV, V. Fighting register pressure in gcc. In *GCC Summit* (2004), pp. 85–104.

[11] MORGAN, R. *Building an Optimizing Compiler*. Digital Press, 1998.

[12] PARK, J., AND MOON, S.-M. Optimistic register coalescing. In *IEEE PACT* (1998), pp. 196–204.

[13] SMITH, M. D., RAMSEY, N., AND HOLLOWAY, G. A generalized algorithm for graph-coloring register allocation. In *ACM Transactions on Programming Languages and Systems* (2004), pp. 277–288.