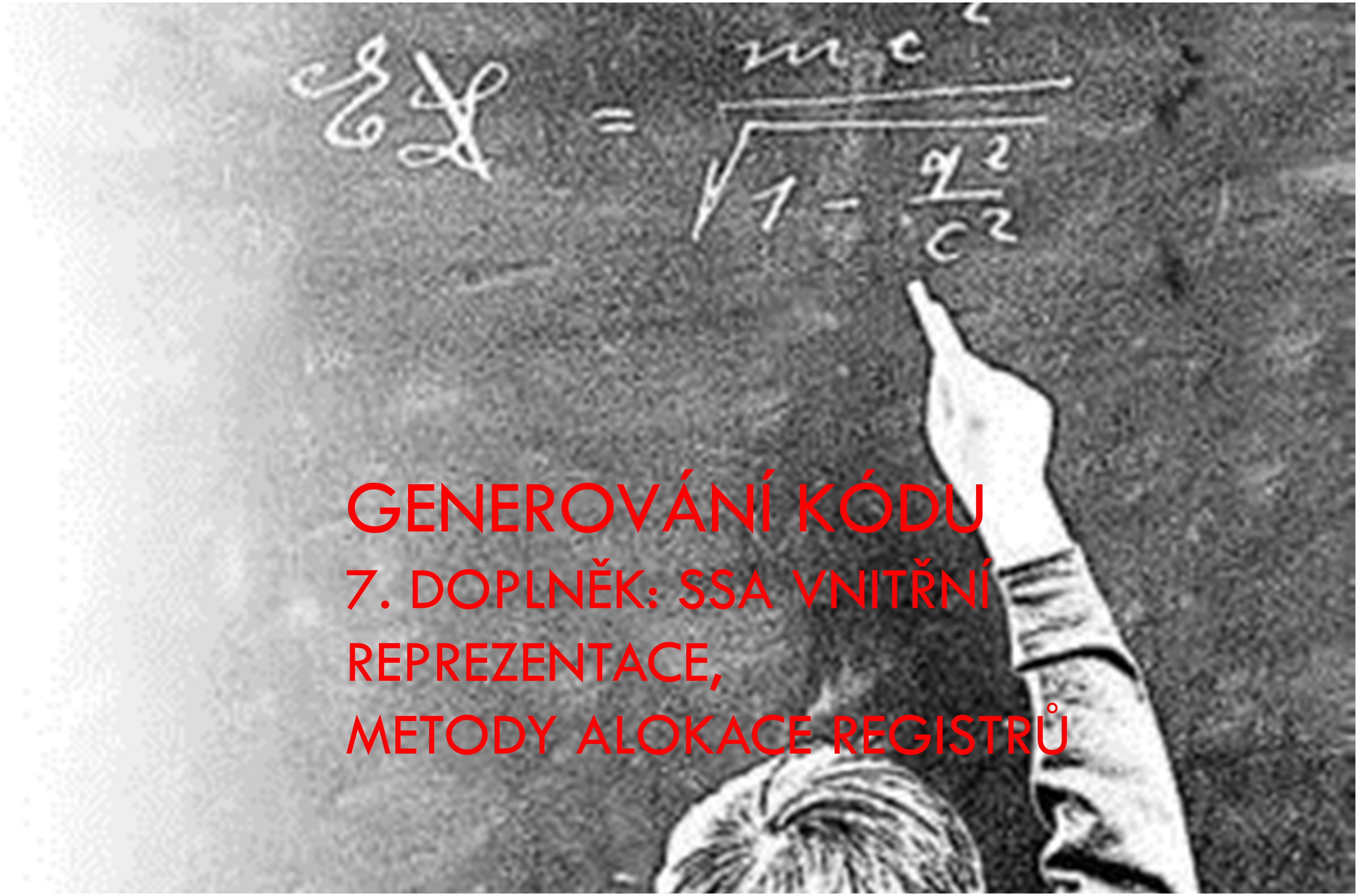


GENEROVÁNÍ KÓDU
7. DOPLNĚK: SSA VNITŘNÍ REPREZENTACE, METODY ALOKACE REGISTRŮ

2011 Jan Janoušek
MI-GEN

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

NOTE: SSA BASED 3AC IR

SSA – static single assignment representation

- 3AC code, where each temporary variable is assigned exactly once,
- first introduced at IBM in the 1980s, many variations´and extensions
- see paper: R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.
- not normally used for direct execution, used for **efficient program optimizations,**
- used in many current compilers: GNU, LVVM, Mini in Mono, Open64, …

SSA – static single assignment representation

- multiple assignments to the same variable create new versions of that variable:

source	SSA based IR
x=1	x_1=1
x=2	x_2=2
y=x	y=x_2

- flow of control makes it impossible to determine the most recent version of a variable (a new special function PHI):

```
if (...)
      a_1 = 5;
    else if (...)
      a_2 = 2;
    else
      a_3 = 13;

    # a_4 = PHI <a_1, a_2, a_3>
    return a_4;
```

REGISTERS ALLOCATION

The problem

- Instructions operating with registers are significantly faster than those operating with memory

- IRs use as many temporaries as necessary
  - This complicates final translation to assembly
  - But simplifies code generation and optimization

- The allocation of registers
  - can be used in various back-end phases,
  - can be performed by various methods

- General rule: certain values, such as stack pointers, base registers,… are typically held in registers. Maximum of other values must be allocated to the remaining registers.

History of register allocation problem

- Register allocation is as old as intermediate code

- Register allocation was used in the original FORTRAN compiler in the '50s
  - Very crude algorithms

- A breakthrough was not achieved until 1980 when **Chaitin** invented a register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

An Example of register allocation

- Consider the program

```
a := c + d
e := a + b
f := e - 1
```

  - with the assumption that `a` and `e` **die after use**

- Three states of temporaries:
  - **Unallocated** – temporary has not been assigned yet
  - **Live** – temporary allocated and will be used in future
  - **Dead** – temporary will not be used anymore

An Example of register allocation

- Consider the program

```
a := c + d
e := a + b
f := e - 1
```

  - with the assumption that `a` and `e` **die after use**

- Temporary `a` can be "reused" after `e := a + b`

- The same - Temporary `e` can be reuses after `f := e – 1`

- Can allocate `a`, `e`, and `f` all to one register (r_1):

$$r_1 := r_2 + r_3$$
$$r_1 := r_1 + r_4$$
$$r_1 := r_1 - 1$$

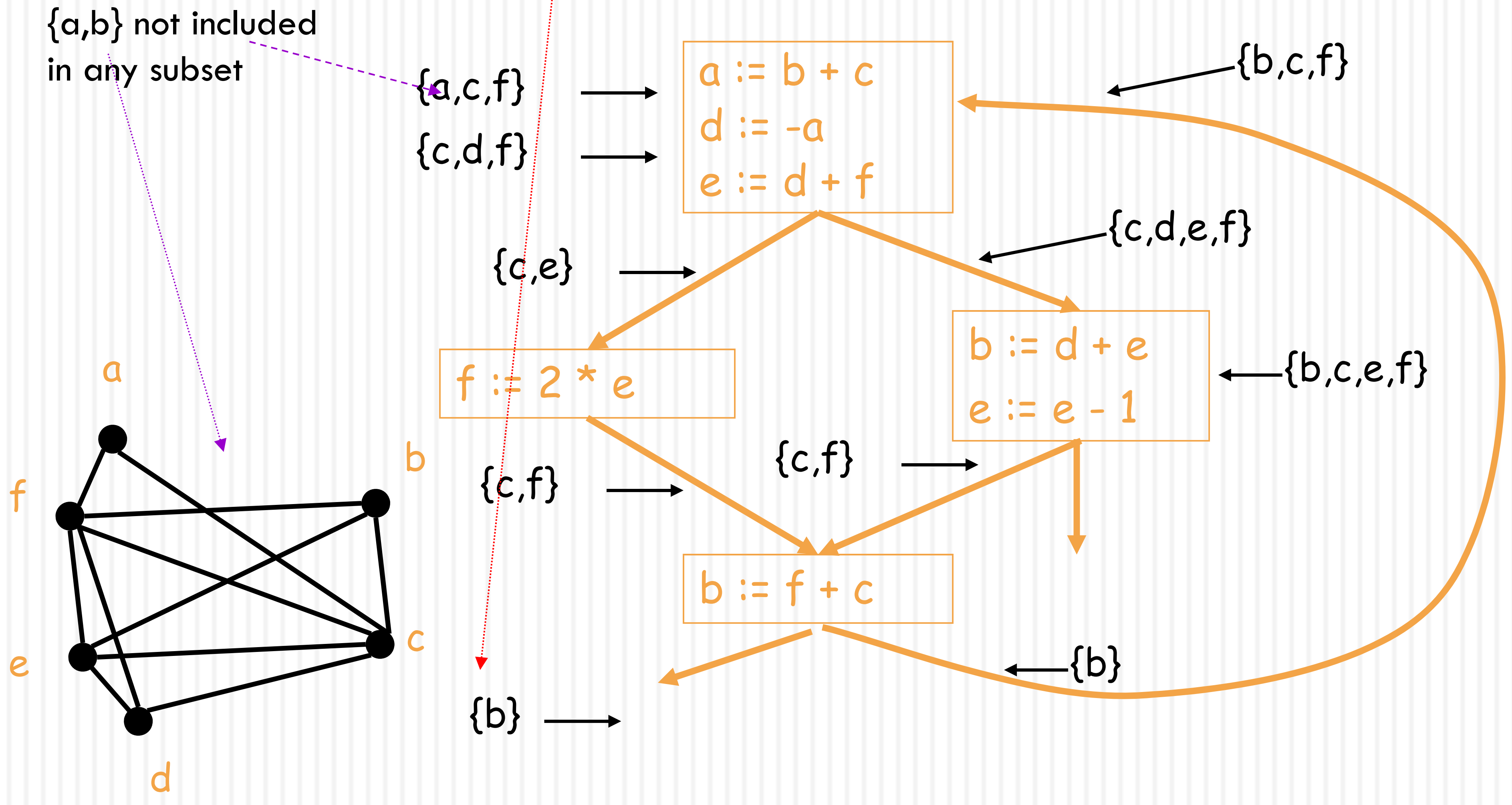
Basic Register Allocation Idea

- The value in a dead temporary is not needed for the rest of the computation
  - A dead temporary can be reused

- **Basic rule:**
  - *Temporaries t_1 and t_2 can share the same register if* **at any point in the program at most one** *of t_1 or t_2 is live !*

Algorithm to minimize the number of registers: Part I
Example of basic blocks and flow graph

Compute live variables for each point:



The Register Interference Graph

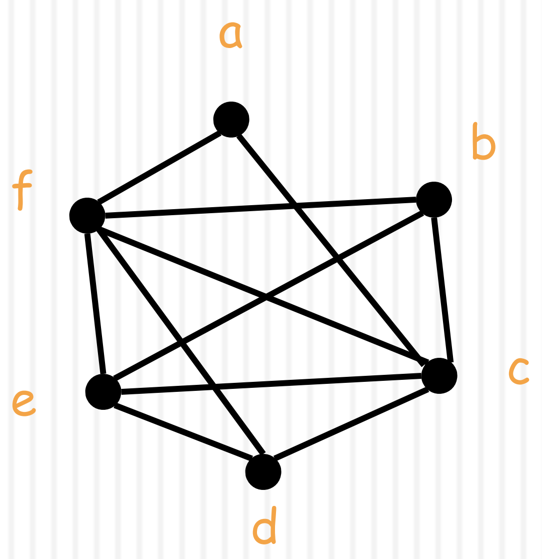
- Two temporaries that are live simultaneously cannot be allocated in the **same register**

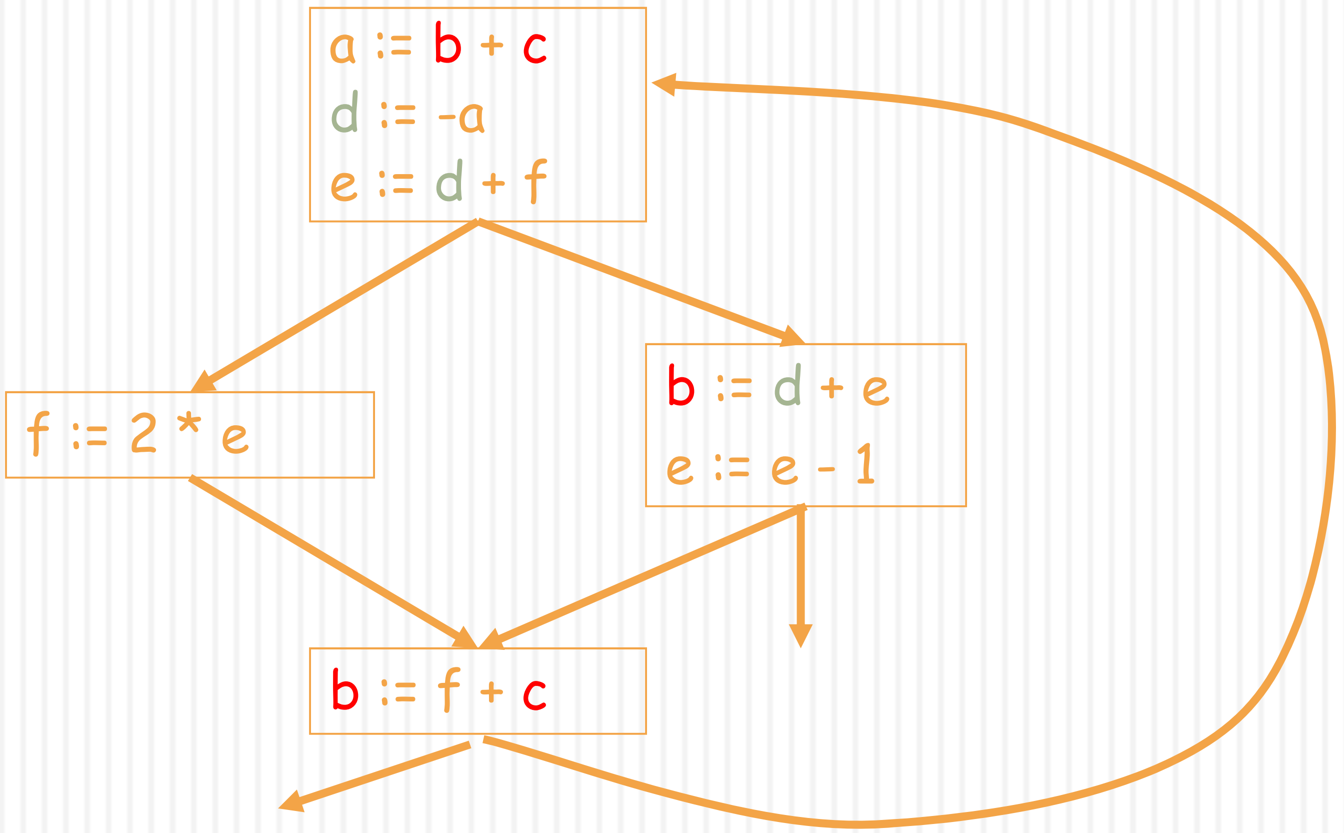
- We construct an undirected graph
  - A node for each temporary
  - An edge between t_1 and t_2 if they are live simultaneously at some point in the program

- This is the register interference graph (RIG)
  - Two temporaries can be allocated to the same register **if there is no edge connecting them**

Register Interference Graph. Example.

For our example:





- E.g., b and c cannot be in the same register
- E.g., b and d can be in the same register

Properties of Register Interference Graph.

- It extracts exactly the information needed to **characterize legal register assignments**
- It gives a **global** (i.e., over the entire flow graph) picture of the **register requirements**
- After RIG construction the register allocation algorithm is **architecture independent**

Graph Coloring. Definitions.

- A **coloring of a graph** is an assignment of colors to nodes, such that nodes connected by an edge have different colors
- A graph is **k-colorable** if it has a coloring with k colors

Register Allocation Through Graph Coloring

- Rewrite the code (generated from IR for example) which uses unrestricted number of registers so that it would use only real machine registers.

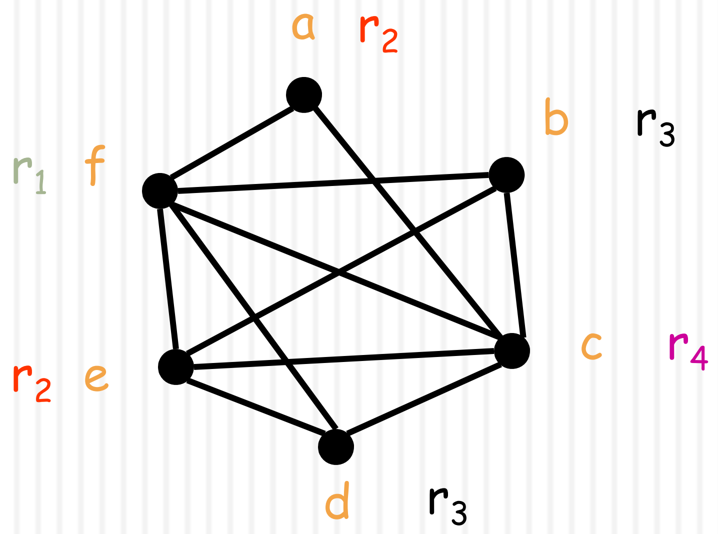
Register Allocation Through Graph Coloring

- In our problem, **colors = registers**
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let k = number of machine registers
- If the RIG is **k-colorable** then there is a register assignment that uses **no more than k registers**

Register Interference Graph

Graph Coloring. Example.

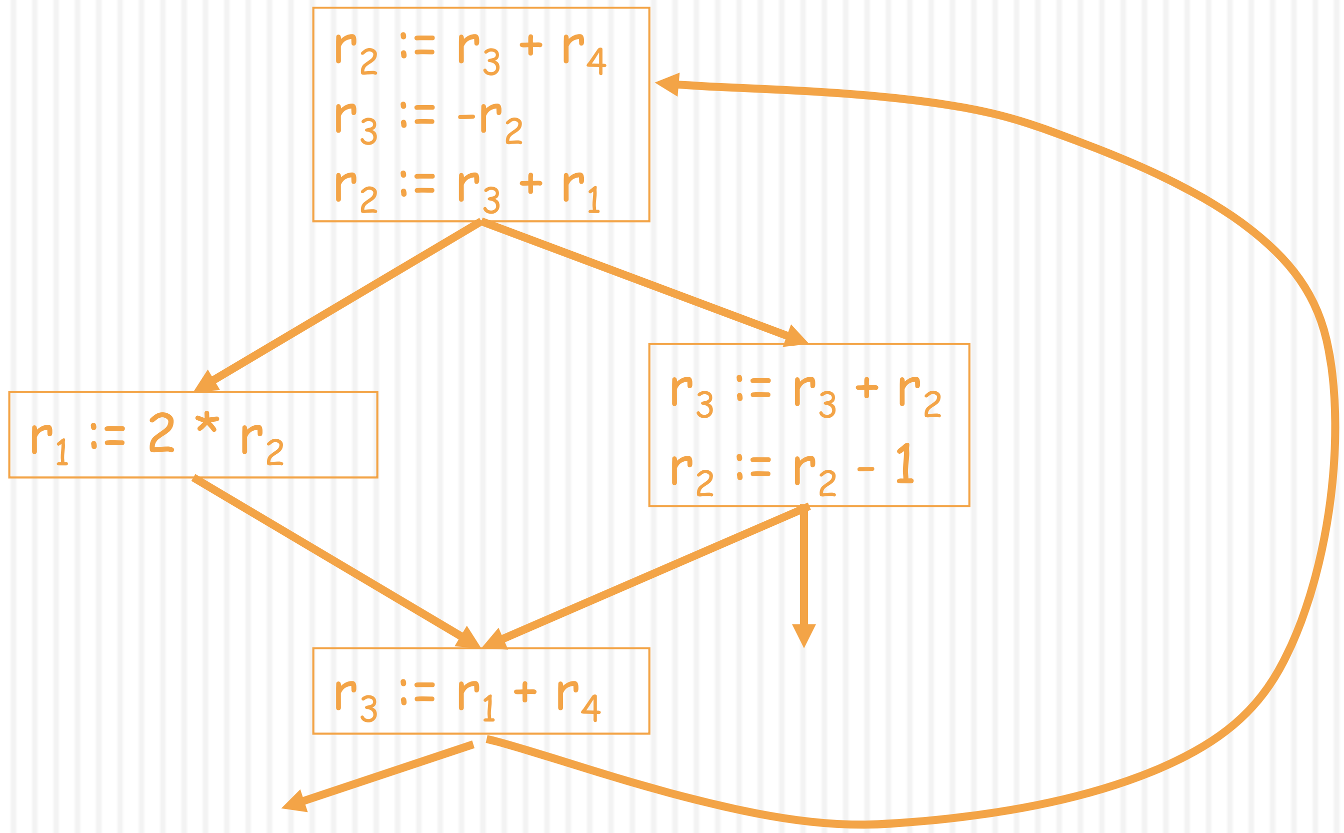
Consider the example RIG



- There is no coloring with less than 4 colors
- There are 4-colorings of this graph

Graph Coloring. Example.

Under this coloring the code becomes:



Computing Graph Colorings

- The remaining problem is to compute a coloring for the interference graph
- But:
  - This problem is very hard (NP-hard). No efficient algorithms are known.
  - A coloring might not exist for a given number or registers
- The solution to is to use heuristics

Graph Coloring Heuristic

- Observation:
  - Pick a node t with fewer than k neighbors in RIG
  - Eliminate t and its edges from RIG
  - If the resulting graph has a k-coloring then so does the original graph

- Why:
  - Let $c_1,\ldots,c_n$ be the colors assigned to the neighbors of t in the reduced graph
  - Since $n < k$ we can pick some color for t that is different from those of its neighbors

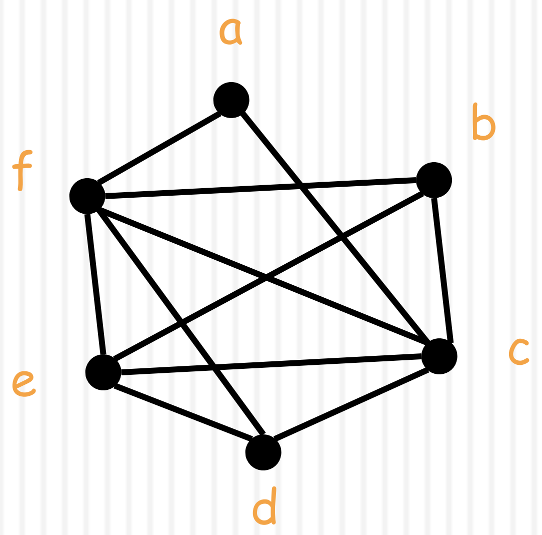
Graph Coloring Heuristic

- The following works well in practice:
  - Pick a node t with fewer than k neighbors
  - Put t on a stack and remove it from the RIG
  - Repeat until the graph has one node

- Then start assigning colors to nodes on the stack (starting with the last node added)
  - At each step pick a color different from those assigned to already colored neighbors

Graph Coloring Example (1)

Start with the RIG and with k = 4:

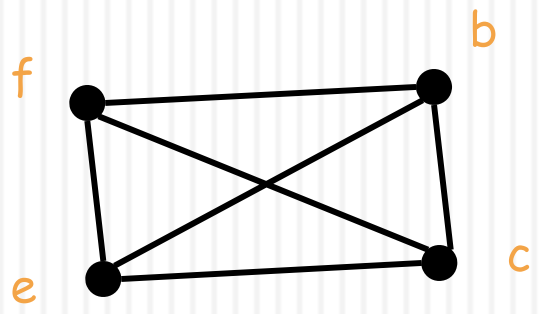


Stack: {}

- Remove a and then d

Graph Coloring Example (2)

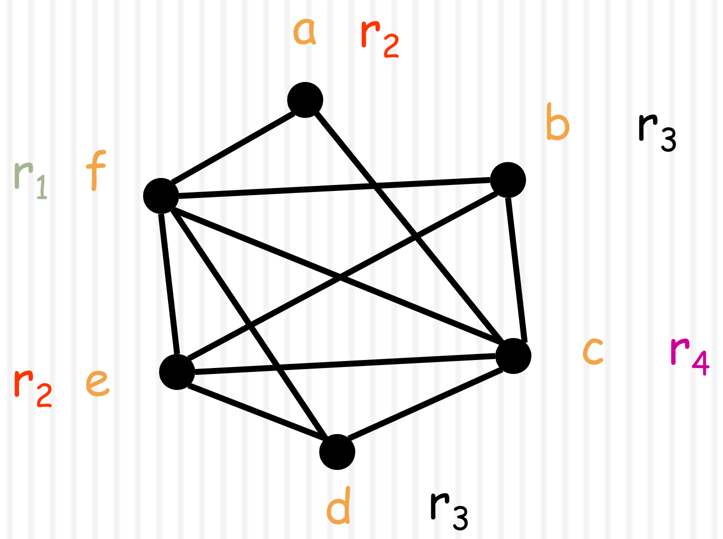
Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f



Stack: {d, a}

Graph Coloring Example (2)

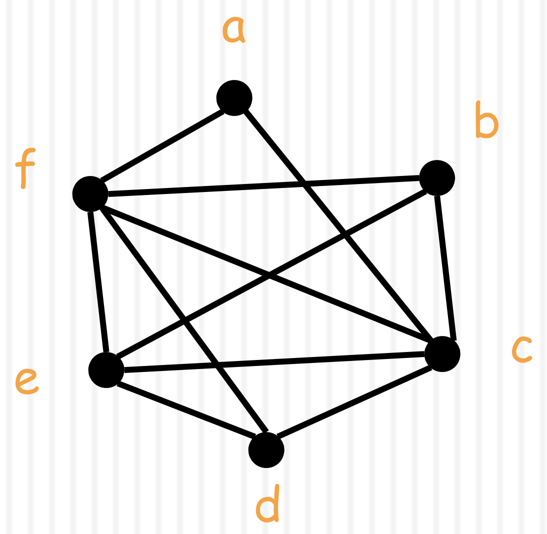
Start assigning colors to: f, e, b, c, d, a



What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors ?

- **Example:** try to find a 3-coloring of the RIG:

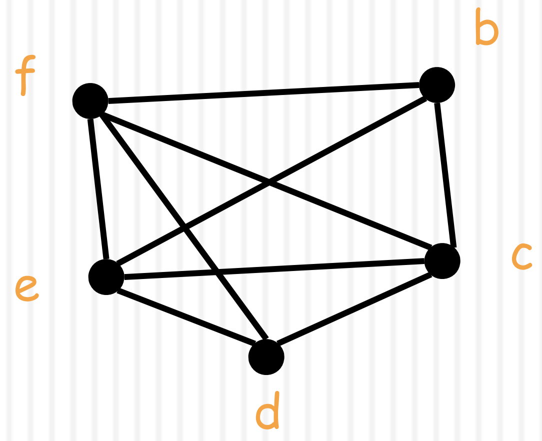


What if the Heuristic Fails?

Remove a and get stuck (as shown below)

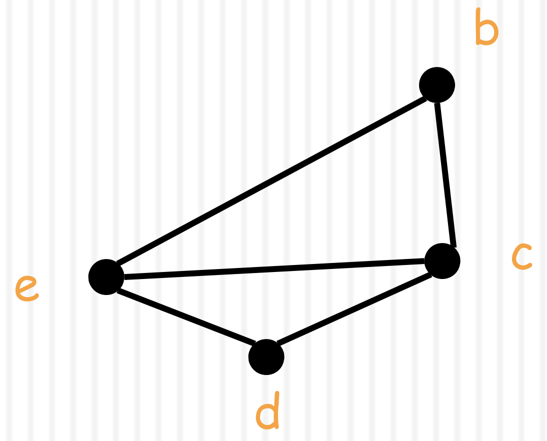
- Pick a node as a candidate for **spilling**
  - **A spilled temporary "lives" in memory**

- Assume that f is picked as a candidate



What if the Heuristic Fails?

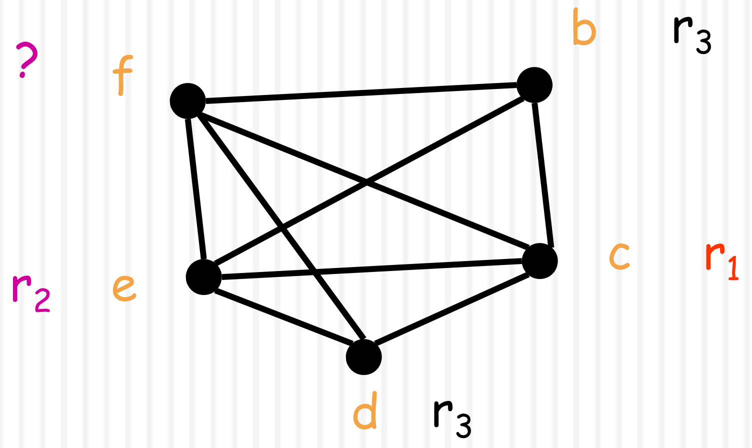
- Remove f and continue the simplification
  - Simplification now succeeds: b, d, e, c



What if the Heuristic Fails?

- On the assignment phase we get to the point when we have to assign a color to f

- We hope that among the 4 neighbors of f we use less than 3 colors $\Rightarrow$ **optimistic coloring**



Spilling

- Since optimistic coloring failed we must spill temporary f

- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address `fa`

- Before each operation that uses f, insert

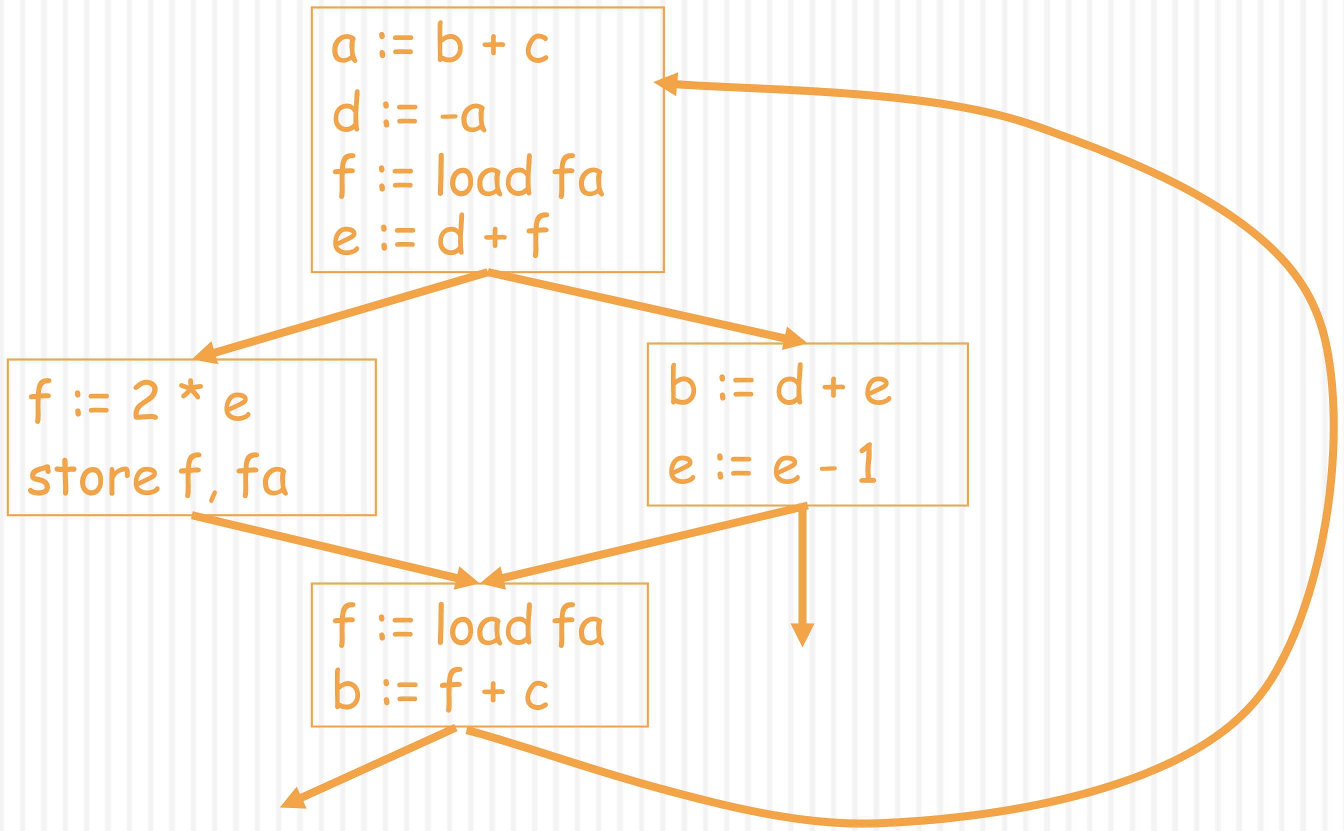
  -
    ```
    f := load fa
    ```

- After each operation that defines f, insert

  -
    ```
    store f, fa
    ```

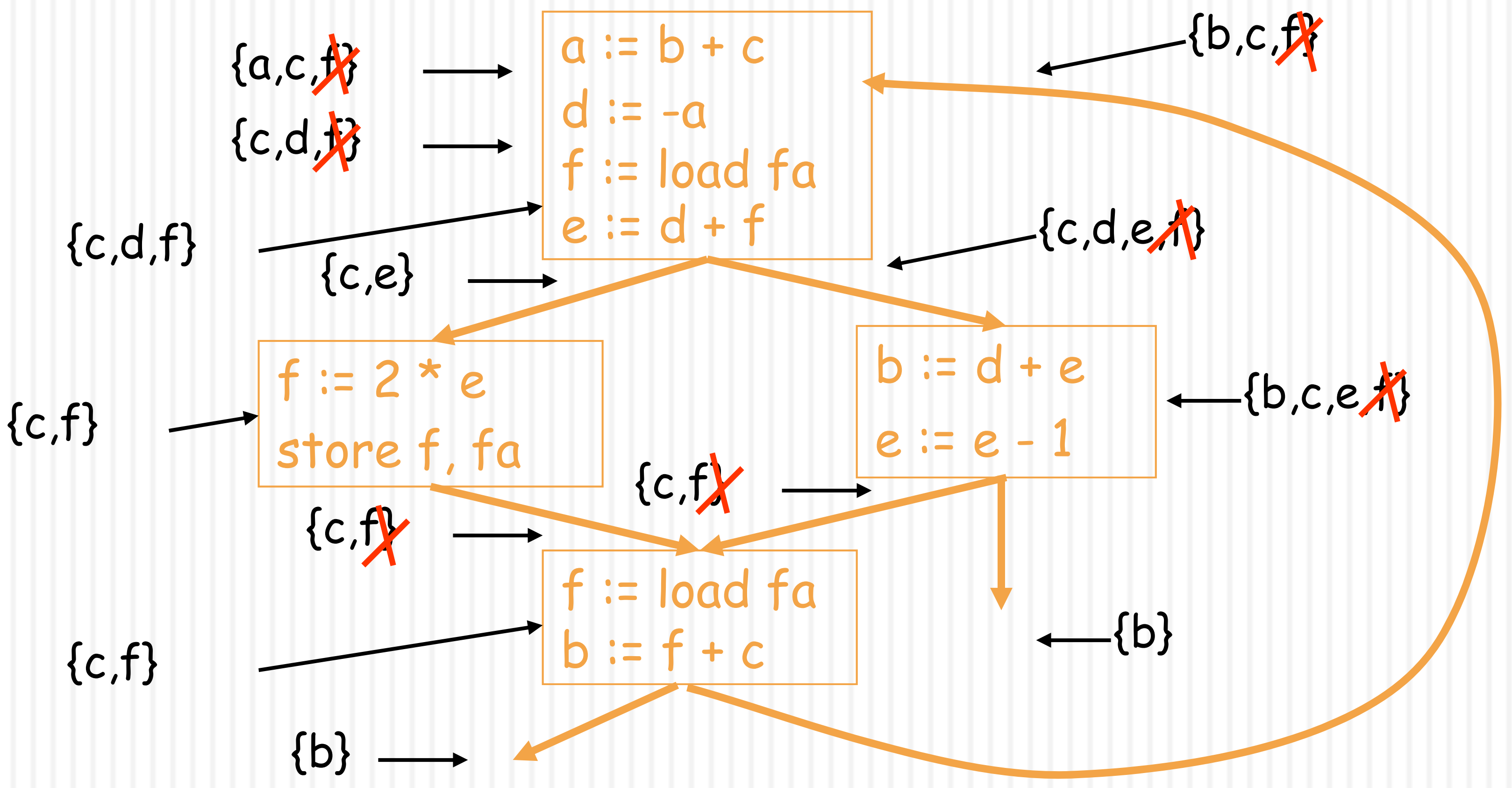
Spilling. Example.

This is the new code after spilling f



Recomputing Liveness Information

The new liveness information after spilling:

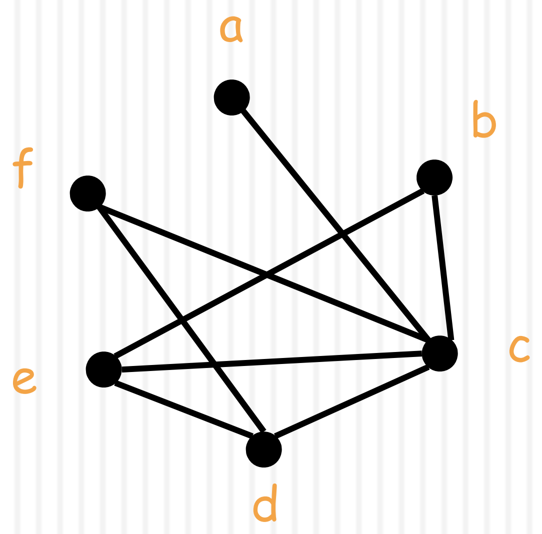


Recomputing Liveness Information

- The new liveness information is almost as before
- f is live only
  - Between a `f := load fa` and the next instruction
  - Between a `store f, fa` and the preceding instruction.
- Spilling reduces the live range of f
- And thus reduces its interferences
- Which result in fewer neighbors in RIG for f

Recompute RIG After Spilling

- The only changes are in removing some of the edges of the spilled node
- In our case f still interferes only with c and d
- And the resulting RIG is 3-colorable



Spilling (Cont.)

- Additional spills might be required before a coloring is found

- The tricky part is deciding what to spill

- Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - **General rule: try to avoid spilling in inner loops**

- Any heuristic is correct

Linear scan register allocation

- simpler but faster

Linear scan algorithm

Linear scan works on a linear representation of the program. Live ranges must be known for all values.

The algorithm scans live ranges from first to last. Whenever there are less than *K* values live at the same time, they are all put in registers. When all registers are allocated and a new value becomes live, one of them must be spilled. The one whose live range ends last is systematically chosen.

Linear scan example

Live ranges

a	b	c	d	e
🟥				
🟥	🟥			
🟥	🟥	🟥		
	🟥	🟥		
	🟥	🟥	🟥	
		🟥	🟥	🟥
		🟥	🟥	
		🟥		

Allocation

R1	R2	
a		
a	b	
a	b	c is spilled
	b	
d	b	
d	e	
d		

Linear scan and spilling

When values are spilled to memory, some registers must be available to operate on them – at least on modern processors that cannot operate on values stored in memory.

There are two ways to make sure that these registers are available:

1. reserve them in advance, which can be sub-optimal if no values are spilled,
2. perform the allocation without reserving them; if spilling turns out to be required, reserve spilling registers and redo the allocation.

Current state in common compilers

- In gcc
- In llvm

Proceedings of the GCC Developers’ Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Yet another GCC register allocator

Vladimir N. Makarov
Red Hat, Inc.
vmakarov@redhat.com

Abstract

The current GCC register allocation is divided into many passes: regclass, regmove, local and global allocators, reload, and post-reload. Such deep division on the passes results in difficulties of evaluation of the final result of optimization decisions practically on each pass and, as a consequence, a bad register allocation which became a bigger problem after introducing the new IL (Tree-SSA) to GCC and the optimizations performed on it.

Another approach to GCC register allocation is proposed in the article. The approach is to support the correct code during all register allocation after getting an initial allocation. Such an approach permits accurate evaluation of each transformation cost. The central notion of the proposed register allocator is *allocno*, which mainly corresponds to a pseudo-register live range. The register allocator implements the following major *transformations*: assignment of a hard register or stack memory to an allocno, coalescing a pair of allocnos, register elimination, allocno value rematerialization, and instruction code selection. Execution of one transformation can result in execution of other transformations to maintain the code correctness. Any chain of transformations may form a *transaction*. Each transaction can be undone. All transformations are stored in the local memory of the register allocator. It means that the RTL is not changed until the very end of the register allocation.

The framework implementing the transformations can be used to implement different register allocation algorithms. Design of a few of them is discussed. The ultimate goal of the project is to remove all of the numerous existing GCC register allocation passes and to use the proposed register allocator instead of them. The current state of the project is reported.

Introduction

Modern processors have a small number of fastest storage units called registers (or *hard registers*). Their number is not enough to store the values of all the operations and directly referred variables in any serious program. The lack of hard registers is the consequence of a trade-off between the processor's speed and its price. Moreover register sets frequently have an irregular structure. The irregularity means that different registers may have different characteristics such as their access time, allowance to be part of a memory address or to be used in some instructions etc.

The small size of the register set and the irregularity of the target processor make it practically impossible to effectively implement any optimization in a compiler (especially in a portable one) whose intermediate representation only refers for the hard registers. Therefore many of the compiler's optimizations are written for an abstract machine which has an infinite regular set of virtual registers called *pseudo-registers*. The optimizations use them to store intermediate values and the values of small variables. For this approach we need a special pass (or optimization) to map abstract machine code into one close to the target machine code which contains only the hard registers of the target processor. This pass is called *register allocation*. Correspondingly, the *register allocator* is one or more compiler components that transform the abstract machine code into the code containing only hard registers.

A good register allocator becomes a very important component of an optimizing compiler nowadays because the gap between access times to registers and to the first level memory (cache) widens for the high-end processors. Many optimizations (especially interprocedural and SSA-based ones) tend to create lots of pseudo-registers. The number of hard registers is the same because it is part of the architecture. Even processors with new architectures containing more hard registers need a good register allocator (although to lesser degree) because the programs run on these computers tend to also be more complicated.

As consequence of its importance, register allocation is probably the most popular area of research in compiler optimizations. The reader can find a brief description of most of the widely known algorithms in [Matz03]. Although there are huge number of articles about different methods of register allocation, practically all of them focus only on a few tasks – register assigning, register coalescing, register spilling, and register rematerialization. In reality there are more tasks solved by the register allocators. As an extreme example, the GCC register allocator additionally solves other tasks like better reloading of operands, instruction transformation into two operand form, dealing with constraints of different class register moves and/or register-memory moves, register elimination, dealing with memory address constraints, exchanging operands in an associative operation, and even partial code selection.

As a consequence of numerous tasks solved by GCC register allocator, it has a lot of passes. Some important components (passes) of the current GCC register allocator have stayed practically unchanged since the first version. Their history is described briefly in [Mak04].

There is a common understanding in the GCC community that we need a better register allocator. It became even more obvious after transition of GCC to *Tree-SSA* infrastructure [Nov03]. This is a huge and very important step in GCC's history. It permitted the implementation of more aggressive optimizations and the generation of better code for targets having many hard registers. Unfortunately, the aggressive optimizations create a bigger register pressure which the current GCC register allocator cannot deal with this problem adequately. As the result, `SPEC2000` scores are worse for architectures with small register files. Very important architectures for GCC like *x86* (and probably *x86_64*) are among them.

> *The major problem of the current GCC register allocator is the* accountability *of optimization decisions, i.e. the final cost of a taken optimization decision in early passes cannot be evaluated. Because the major last pass of the register allocator is reload, which finally removes pseudo-registers, people often blame this component.*

One reason of that GCC has so many register allocation passes is in the powerful (sometimes too powerful) GCC model describing the register set of the target architecture. This model is described by constraints in `define-insn` constructions of the GCC machine description file and by a lot of machine dependent macros.

On one hand, it makes GCC a very portable compiler. On the other hand, this complexity resulted in the use of a typical engineering approach, which is to divide complex task on several smaller subtasks. In other industrial compilers a good register allocation is achieved by combined algorithms. Intel's *x86* global register allocation based on graph fusion [Lueh96] is such an example. Improving GCC register allocation by adding new passes usually only worsens the situation. As the result such projects fail.

As example, let us look at two projects. The new register allocator [Matz03] which removed two passes local-alloc and global-alloc and added two other passes—a colour-based register allocator with live range splitting and coalescing, and pre-reload which is actually the reload moved from after register assignment to before it. The goal of the pre-reload pass was to give the new register allocator a better evaluation of its decisions which early on was not possible because of massive code changes in the reload pass. So the complexity of the new allocator (and its code) became even bigger, and the cooperation of the register allocator passes was not improved much. In my opinion this is one reason why the project has failed.

Another project was to improve the original GCC register allocator [Mak04]. New rematerialization, live range splitting and register coalescing passes, and improved cooperation between reload and global were added. The project was a small success. The biggest improvement was gotten from the better cooperation of the reload and the global allocator. On the contrary, the separate more sophisticated passes gave practically nothing because it is hard to evaluate the final cost of a decision in the passes to figure out the best or better choice. Accurate evaluation is not possible because of the numerous and complex subsequent passes (especially the reload pass).

All that is mentioned above was a major motivation to start work on another approach to the register allocator. The approach is to create an infrastructure which supports all GCC register allocation tasks in a combined way. After getting an initial register allocation, simple code transformations like assigning a hard register to pseudo-register, reloading a value into another location, splitting the live range of a pseudo-register, coalescing registers, register elimination, rematerialization are done. Execution of one transformation can result in the execution of other transformations to maintain the code correctness. Such a chain of transformations is called a *step*. Any chain of transformations may form a *transaction*. Any transaction can be undone. All transformations performed are stored in the local memory of the register allocator. It means that the RTL code is not changed until the very end of the register allocation procedure.

Each transformation step will result in correct code (as it exists currently in GCC after the reload pass), so after each transformation step we can evaluate the cost of the final results. This way the accountability problem can be solved.

This article is focused mainly on description of the infrastructure rather than the register allocation algorithms which can use it. The first section describes the original GCC register allocator and tasks solved by it in more detail. The second section describes the major notions and data structures of the infrastructure of the

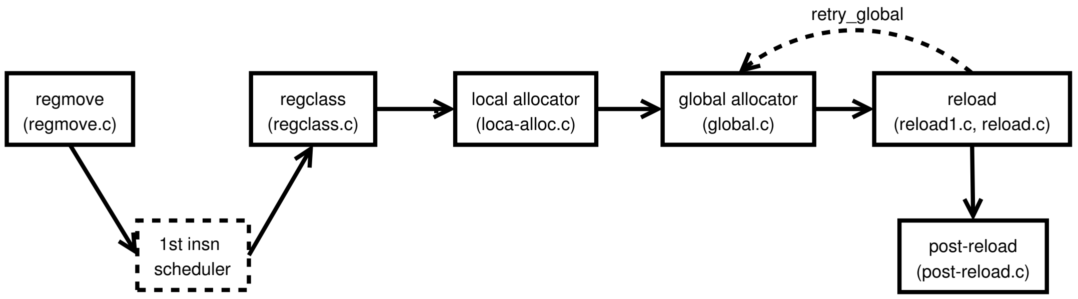


Figure 1: Passes in the current GCC allocator.

proposed register allocator. The third section describes the current status of the project.

1 The original register allocator in GCC

The original register allocator contains a lot of passes. Figure 1 shows the major passes and their order.

Regmove. The major task of regmove is to generate move instructions to satisfy two operand instruction constraints when the destination and source registers should be the same. The reload pass can solve this task too but in a less effective manner. Regmove ignores the fact that sometimes the transformation of an instruction into two operand form is not necessary because there may be another alternative of RTL instruction in three operand form. The pass also does some register coalescing. The regmove pass removes some register moves if the registers have the same value and they can be found in the scope of a basic block.

Regclass. 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 relations are possible between different register classes: they can intersect or one register class can be a subset of another register class (there are reserved register classes like `NO_REGS` which does not contain any hard register or `ALL_REGS` which contains all hard registers).

The pass regclass (file `regclass.c`) mainly finds the *preferred* and *alternative* register classes for each pseudo-register. The preferred class is the smallest class containing the union of all register classes which result in the minimal cost of their usage for the given pseudo-register. The alternative class is the smallest class containing the union of all register classes the usage of which is still more profitable than memory (the class `NO_REGS` is used for the alternative if there are no such hard registers besides the ones in the preferred class).

The weakness of the regclass pass is in that it finds the preferred and alternative classes for whole function. If we implement any pseudo-register live range splitting, this information would be not accurate because the most profitable register class could be different for different live ranges of a pseudo-register.

It is interesting that finding through the preferred and alternative classes, the pass also implicitly does code selection by choosing possible instruction alternatives.

The local allocator assigns hard registers only to pseudo-registers living inside one basic block.

Besides assigning hard registers, the local allocator also does some register coalescing: if two or more pseudo-registers shuffled by move instructions do not conflict, they always get the same hard registers. The global allocator also tries to do this in a less general way. The local allocator also performs simple copy and constant propagation.

The global allocator assigns hard registers to pseudo-registers living in more than one basic block. It could change an assignment made by the local allocator if it finds that usage of the hard register for a global pseudo-register is more profitable than usage for the local pseudo-register.

The global allocator sorts all pseudo-registers according to the following priority:

$$\frac{\log_2 Nrefs \cdot Freq}{Live\_Length} \cdot Size$$

Here $Nrefs$ is number of the pseudo-register occurrences, $Freq$ is the frequency of its usage, $Live\_Length$ is the length of the pseudo-register's live range in instructions, and $Size$ is its size in hard registers.

Afterwards the global allocator tries to assign hard registers to the pseudo-registers with higher priority first. This algorithm is very similar to assigning hard registers in Chow's priority-based colouring [Chow84, Chow90].

The global allocator tries to coalesce pseudo-registers with hard registers met in a move instruction by assigning the hard register to the pseudo-register. It is made through a preference technique: the hard register will be preferred by the pseudo-register if there is a copy instruction with 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.

For example, if the pseudo-register got hard register A in the global allocator but an instruction referring to the pseudo-register requires a hard register of another class, the reload will generate a move of A into the hard register B of the needed classes. Sometimes, a direct move is not possible; we need to use an intermediate hard register C of the third class or even memory. If the hard registers B and C are occupied by other pseudo-registers, we expel the pseudo-registers from the hard registers. In this case, the reload could be considered as a local spiller. The reload will ask the global allocator through function `retry_global` to assign another hard register to the expelled pseudo-register. If it fails, the expelled pseudo-register will finally be placed on the program stack.

To choose the best register shuffling and load/store memory, the reload uses the costs of moving register of one class into a register of another class, loading or storing a register of the given class. To choose the best pseudo-register for expulsion, the reload uses the frequency of the pseudo-register's usage.

Besides this major task, the reload also does 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, copy propagation etc.

The complexity of the reload is a consequence of the very powerful model of the target processor's register file, permitting one to describe practically any weird processor.

Postreload. The reload pass does most of its work in a local scope; it generates redundant moves, loads, stores etc. The postreload pass removes such redundant instructions by a global redundancy elimination technique.

As we can see that one task (e.g. coalescing or assigning) is solved in many passes sometimes differently and in a primitive way, the passes frequently change the decision of the previous passes because they work mainly without cooperation or taking the subsequent or previous passes work into consideration. We cannot be sure what the final cost of our optimization decision is on practically any pass until all the register allocation is finished, because we cannot be sure what the final result of the decision will be. The register allocator proposed in this article tries to solve the problem.

2 The design

The central notion of the proposed register allocator is notion of *allocno*. There are three kinds of allocnos:

- **Instruction allocno** designates an operand in a RTL instruction or an address (or part of it) of memory mentioned in the instruction. Assigning hard registers, memory, or nothing should guarantee that the instruction is valid (there is an alternative where the instruction constraints are satisfied) and the addresses are legitimate.
- **Range allocno** designates a pseudo-register value between instructions in a basic block. Assigning hard registers or memory to it means spilling/restoring the pseudo-register in the basic block.
- **Region allocno** designates a pseudo-register in a region (currently regions are loops). Assigning hard registers or memory to it means spilling/restoring the pseudo-register in the region.

Allocno is an object which should get a hard register, memory or nothing. The later is possible only for an instruction allocno representing an explicit hard register, a constant, or memory. Allocnos have an attribute used to mark that the allocno lives through a *function call*. In this case the allocno will never get a hard register clobbered by function calls. To simplify the implementation different allocnos of the same pseudo-register always get the same memory. It permits not to worry about generation of memory-memory moves.

Allocnos are connected by *copy edges*. A copy edge is potentially one or more move, load, or store instructions to move a pseudo-register

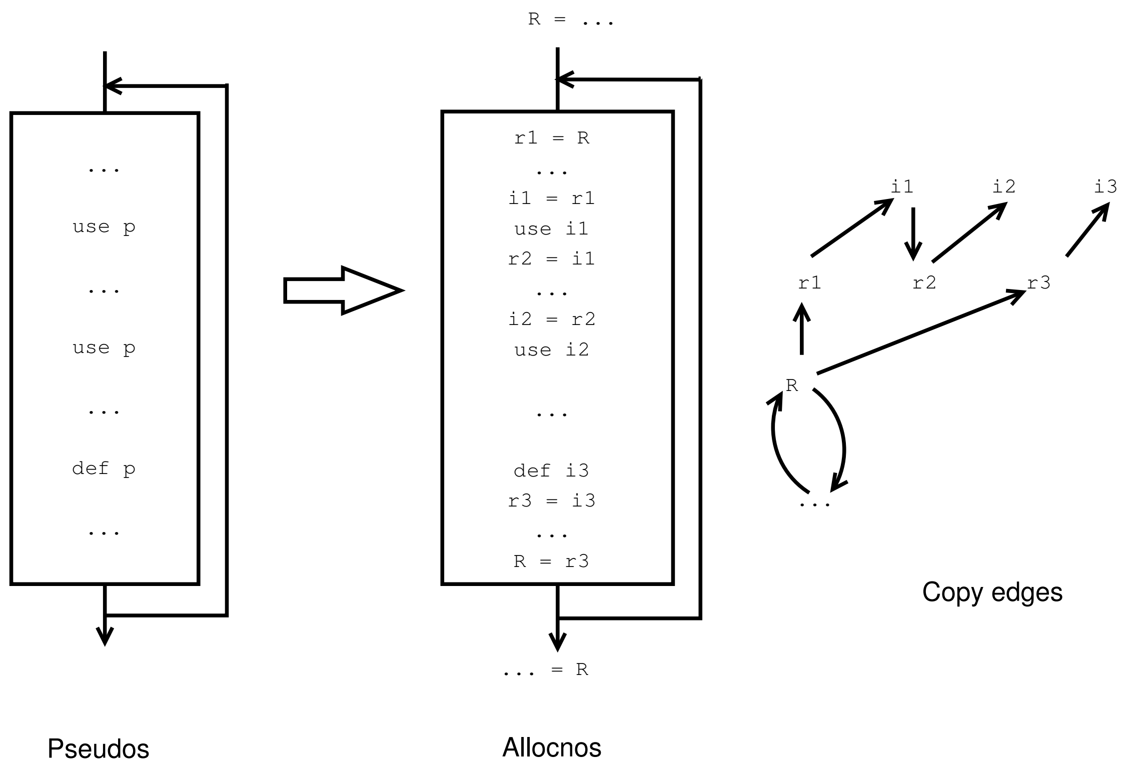


Figure 2: Allocnos and copy edges for a pseudo-register living in a loop

value or reload a non pseudo-register value. For example, if the source allocno of a copy edge gets memory and the destination allocno gets a hard register, the copy edge will be represented by load instruction after the register allocation. If the allocnos get the same memory or hard register, the copy edge will not correspond to any move instruction (in other words the allocnos will be *coalesced*).

Figure 2 illustrates how allocnos corresponding to one pseudo-register `p` are connected by copy edges between the allocnos. Here `i1`, `i2`, and `i3` are instruction allocnos; `r1`, `r2`, and `r3` are range allocnos, and `R` is the region allocno corresponding to pseudo-register `p` in the loop.

Copy edges represent all potential places to split live range of a pseudo-register. The place might be before or after the pseudo-register's references or on entry to or exit from a region or basic block where the pseudo-register is referenced. I call such an approach *pessimistic splitting* as opposed to the famous *optimistic coalescing* [Park98] approach in register allocation. Generally speaking, live range splitting could be anywhere the pseudo-register lives. But such freedom will not permit us to improve code in most cases, it will only make the register allocator too slow because of numerous allocnos and copy edges. Probably even in the current approach there are too many allocnos and copy edges and a more conservative approach to live range splitting might be required in the future.

An instruction allocno has a lot of additional information. The most important information is

- *Type,* which is determining whether the allocno represents an instruction operand, base register, or index register, or other non-operand value which is a part of RTL `CLOBBER` or `USE` clause.

- Reference to another instruction allocno representing another operand of a *commutative* operation.

- Reference to a *tied* allocno. Tied allocnos always get the same hard register or memory. This attribute is used to describe a situation when the two operands in an RTL instruction are actually one operand in a machine instruction, e.g. for an architecture with two operand instructions.

- *Intermediate elimination register*. Sometimes it is impossible to change a hard or virtual register to a hard register with constant displacement without the usage of an intermediate hard register. In such a case, the attribute describes the intermediate hard register. For example, when we eliminate the frame pointer by the stack pointer, the displacement might be not legitimate. If it is the case, we should generate an instruction assigning the stack pointer value plus displacement to the intermediate hard register and use it in place of the frame pointer.

- *Location* in the instruction of the object represented by given allocno. Location of the *container* of the object. It might be used to find `SUBREG` for the register represented by given allocno or `MEM` for the base or index register represented by given allocno.

- *Early clobber* attribute for the corresponding instruction operand. Actually, the clobber flag should belong to the instruction alternative but it is hard to represent and use it for allocno conflicts. Therefore we make it true for all alternatives if there is a flag for at least one alternative.

- Information about *register elimination* for the corresponding allocno.

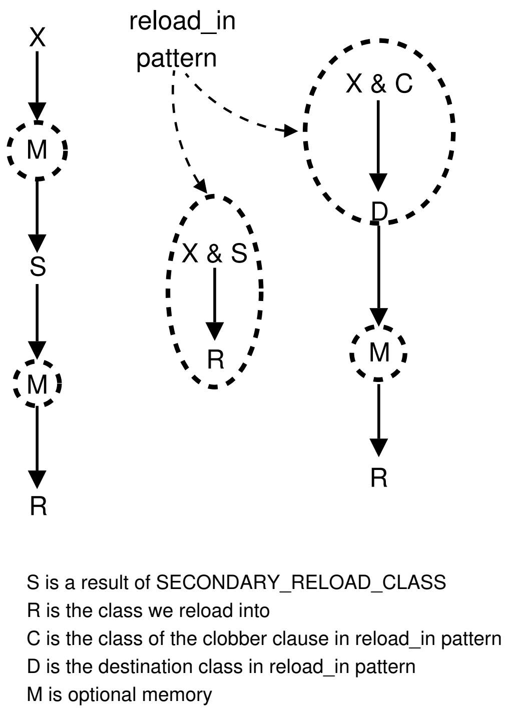

X
M
S
M
R
reload_in pattern
X & S
R
X & C
D
M
R
S is a result of SECONDARY_RELOAD_CLASS
R is the class we reload into
C is the class of the clobber clause in reload_in pattern
D is the destination class in reload_in pattern
M is optional memory

Figure 3: GCC cases of reload value X into hard register of class R.

Copy edges have a lot of attributes. The main reason for this is in the powerful register file model of a target processor in GCC. GCC is based on the suggestion that hard register of any class (also memory or constant of the appropriate mode) can be moved to (from) a hard register of any another class even though it can be impossible in the target processor without using intermediate registers, special instructions described by special `reload_in_*` or `reload_out_*` patterns, or memory (it is called *secondary memory*). A typical example of usage of secondary memory is the movement of general registers into floating point registers for the x86 architecture. Figure 3 illustrates such complicated cases for loading a value `X` into a hard register of class `R`. Here `S` is a register class returned by the GCC target macro `SECONDARY_INPUT_RELOAD`, `C` is a register class of the clobber clause in the corresponding GCC *reload_in* pattern, `D` is a register class of the destination register in the corresponding `reload_in` pattern, and `M` is optional secondary memory.

The most important attributes of a copy edge are

- *Position*. This is the place where the corresponding move, load, or store instructions will be placed. The position may be after a given instruction, at a given basic block's start or end, or at the source or destination of a given edge.

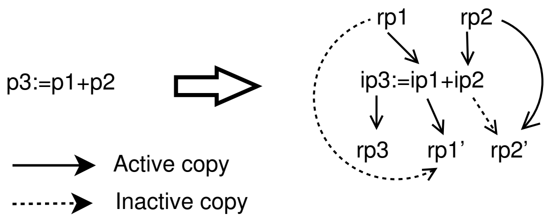


Figure 4: Example of instruction allocnos with inactive copies

- *Active*. A copy edge can be active or inactive. Most copy edges are always active. Sometimes we can load a value into an allocno from different source allocnos. Figure 4 illustrates such a situation when value rp1' can be loaded from rp1 or ip1. In this example, one copy with destination rp1' is active, another one is inactive. If ip1 got a hard register and rp1 got memory, then the active copy might be more profitable if rp1' got a hard register.

- *Intermediate* hard registers mentioned above.

- *Secondary memory* mentioned above. The attribute usually refers to a stack memory slot.

- *Rematerialization attributes*. These attributes contain information about the instruction pattern which can be used for rematerialization and references for allocnos which can be used as the operands of the rematerialized instruction. We can use the rematerialization instruction instead of the move instructions if it is more profitable.

There is an allocno conflict graph. If two allocnos conflict, they cannot get the same hard register or memory. The allocno can be dependent on copy edge activeness. Figure 5 illustrates such case. Allocno `ipm` conflicts with allocno `rp1` only if copy `rp1-rp1'` is active.

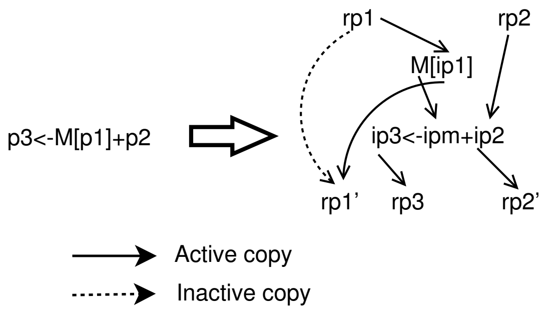


Figure 5: Another example of instruction allocnos and copies

As mentioned above a copy edge can get at most two hard registers and/or memory too. It is needed when secondary reload hard registers and/or secondary memory is needed to move the corresponding allocno values. Therefore there is also a conflict graph for copies and allocnos (copies can conflict with allocnos). If a copy got hard registers, they cannot be assigned to allocnos conflicting with the copy and vice versa.

There is also a dynamic data structure describing stack slots. Freeing a stack slot (e.g. because we assigned a pseudo-register to a hard register instead of memory) could make stack memory smaller and decrease displacements for addressing other stack slots gotten by some other allocnos. Currently the implementation is based on the assumption that shortening the address displacement will not make the legitimate address an illegitimate one.

The register allocator provides the following primitive *transformations*[1]:

- *Assign, Unassign*. The assign transformation assigns a hard register or memory slot to an allocno. It might require the allocation of a stack memory slot, assigning secondary memory and/or intermediate hard registers to copy edges with given allocno. Correspondingly, the unassign transformation might result in freeing a stack memory slot, secondary memory, and/or the intermediate registers of the copies. There are two variants of the transformations: with a specified hard register or a specified hard register class.

- *Tie, Untie*. A pair of unassigned instruction allocnos can be tied. After that, assigning a hard register or memory to one allocno results in assigning the same hard register or memory to another allocno.

- *Commutative exchange*. Two unassigned instruction allocnos representing the operands of a commutative operation can be exchanged. It means that their locations are changed. Such a transformation might result in more profitable code because the operand constraints may differ.

- *Activate, Deactivate* a copy edge. A copy edge connecting two unassigned allocnos can be activated if another copy edge with the same destination range allocno is inactive (see Figure 4).

- *Eliminate, Uneliminate*. An allocno representing a hard register whose value is changed by another hard register plus displacement. The eliminate transformation might require the allocation of a hard register as an intermediate register for the elimination because the substituted expression is illegitimate, e.g. the displacement is too big. There are two classes of eliminated hard registers. The first one is virtual registers (e.g. argument register) which are used for convenience. They should be eliminated in any case. Another class is real hard registers, e.g. the frame pointer register, which can be changed by another hard register, e.g. the stack pointer, to increase the number of hard registers available for register allocation. Registers of the first class may not be uneliminated. Registers of the second class may be uneliminated. Sometimes usage of an eliminated register might be more profitable than the substituted register, e.g. the displacement is smaller, which may result in shorter instructions for some architectures.

A transformation may be failed if it is not possible to perform it. For example, assigning a specified hard register is impossible because the hard register is already assigned to conflicting allocnos or copy edges.

After each transformation is done, the overall cost of the code is modified. The cost of code is evaluated by the following formula

$$\sum_{\forall c \in Copies} Cost_c \cdot Freq_c + \sum_{\forall i \in Insns} Cost_{alt(i)} \cdot Freq_i$$

Here $Cost_c$ is the cost of moves, loads, and stores or rematerialization (whichever ones are cheaper) generated by a copy c, $Freq_c$ and

[1]Practically all transformations have two forms. The second form undoes the first one.

$Freq_i$ are correspondingly the frequency of copy c and the frequency of instruction i, $Cost_{alt(i)}$ is cost of the current alternative of the instruction i.

As a result of such design, we are solving the *accountability* problem of the current GCC register allocator. After getting an initial allocation, we have always the exact cost of the current register allocation and support it correct through all changes to the register allocation.

More complex transformation like allocno coalescing is implemented through a small chain of assign/unassign transformations with the hard registers specified.

Sometimes a long chain of transformations is needed for the implementation of register allocation algorithms. To facilitate rejecting such chains of transformations, e.g. because of one transformation in the chain failed or the chain of transformations is unprofitable, *transactions* of transformations are implemented. Any chain of transformations may form a transaction. Each transaction can be rejected (in this case, we return to the same register allocator state as it was before the transaction) or accepted. Transactions may be nested.

The RTL code will be not changed until the final, pretty simple, stage of changing the code. So all the information needed for this is stored in local data of the register allocator.

3 The current state of the project

Any project for the implementation of a new register allocator for GCC has to be a long project. GCC is an extremely portable compiler because of its powerful register description model for target processors and the notion of sub-registers in RTL[2]. That creates many difficulties for the implementation of register allocators in GCC. The described infrastructure is trying to hide some of these difficulties.

Currently the register allocator is in the early stages of implementation. Most of the infrastructure described above has been implemented. Missing parts are register rematerialization, unelimination, representation of RTL move instructions by copy edges, and evaluation of the cost of instruction alternatives.

The flexibility and power of the infrastructure permits to implement sufficiently easily different register allocation algorithms in GCC, from the ones used in industrial compilers [Muchnick97, Morgan98, Cooper03], to research algorithms [Appel01, Sholtz02]. Most probably, GCC as a portable compiler should use at least two register allocator algorithms (one for irregular and small register file architectures and one for regular and moderate or large register file architectures). Currently two register allocation algorithms are in the process of implementation. They are chosen to justify the project approach.

- Priority colouring [Chow84, Chow90].
  - Initial allocation: all region and range allocnos are in memory, instruction allocnos are assigned taking the optimistic assumption that non-instruction allocnos will be eventually in hard registers.
  - Priority assigning hard registers to range and region allocnos and coalescing them with their corresponding instruction allocnos.
- The current GCC approach which is close to priority colouring.

[2]This model is sometimes too powerful and can and should be simplified.

- Initial allocation: priority assignment of hard registers to region and range allocnos and then assignment to instruction allocnos with possible spilling of conflicting region and range allocnos.
- Coalescing region and range allocnos with their corresponding instruction allocnos.

Results received on some benchmarks for the two register allocation algorithms are promising.

4 Acknowledgments

I am grateful to my company, RedHat, 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, Serguei, for the help in proofreading the article.

References

[Appel01] A. Appel and L. George. Optimal spilling for cisc machines with few registers. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation. ACM Press, 2001.

[Chow84] F. Chow and J. Henessy, *Register allocation by priority-based coloring*, In Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction (Montreal, June 1984), ACM, New York, 1984, pages 222–232.

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

[Cooper03] Keith Cooper, Linda Torczon, *Engineering a Compiler*, Morgan Kaufmann (2003), ISBN 155860698X.

[Lueh96] G.Y. Lueh, T. Gross, and A. Adl-Tabatabai, *Global Register Allocation Based on Graph Fusion*, Ninth Workshop on Languages and Compilers for Parallel Computers, August 1996.

[Mak04] V. Makarov, *Fighting register pressure in GCC*, GCC Summit, 2004.

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

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

[Muchnick97] Steven S. Muchnick, *Advanced compiler design implementation*, Academic Press (1995), ISBN 1-55860-320-4.

[Nov03] D. Novillo, *Three SSA. A new Optimization Infrastructure for GCC*, GCC Summit, 2003.

[Park98] J. Park , S.-M. Moon, *Optimistic Register Coalescing*, Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, October 12-18, 1998.

[Sholtz02] B. Scholz and E. Eckstein, *Register Allocation for Irregular Architectures*. In Proc. of Joint-Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems

(LCTES/SCOPES'02), ACM Press, Berlin, Germany, June 2002.

LLVM PROJECT BLOG

LLVM Project News and Details from the Trenches

Sunday, September 18, 2011

Greedy Register Allocation in LLVM 3.0

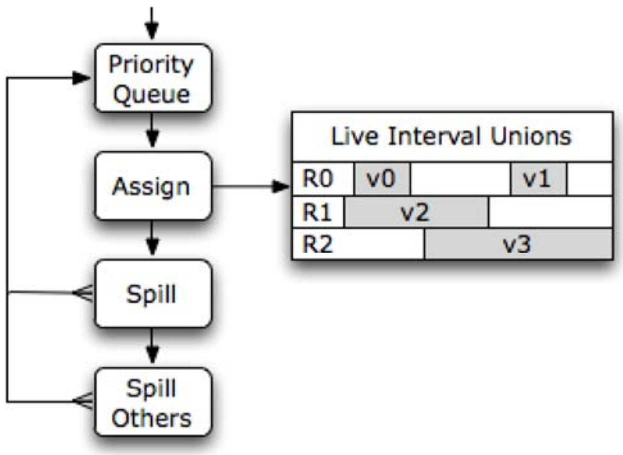
LLVM has two new register allocators: Basic and Greedy. When LLVM 3.0 is released, the default optimizing register allocator will no longer be linear scan, but the new greedy register allocator.
With its global live range splitting, the greedy algorithm generates code that is 1-2% smaller, and up to 10% faster than code produced by linear scan.

Lessons learned from linear scan

Linear scan has been the default register allocator in LLVM since 2004. It has worked surprisingly well for such a simple algorithm. In fact, the simple design made it easier to tweak the algorithm in order to make small improvements to the generated code. More advanced register allocation algorithms often need to build expensive data structures, or they make assumptions about live ranges being invariant. That makes it difficult to, say, commute a two-address instruction on the fly, or rematerialize a constant pool load instead of spilling it to the stack.
A new register allocation algorithm needs to preserve this simplicity. It must be possible to change the machine code while the algorithm is running.
Linear scan depends on the virtual register rewriter to clean up the code after registers have been assigned. In theory, the rewriter should only rewrite virtual registers to their assigned physical registers, but it knows many other tricks. When linear scan does something silly like reloading a register from a stack slot twice, the rewriter will clean up the code by reusing the

first register and eliminating the second reload. The algorithm is local, and it cannot clean up messes that extend beyond a single basic block. The rewriter always saves the day by removing obvious mistakes. It comes at a high price, though. It accounts for about half of the linear scan compile time, and its large collection of tricks makes the code very hard to maintain.
A new register allocator should avoid making obvious mistakes so the rewriter can concentrate on rewriting registers.
As the name implies, linear scan works by visiting live ranges in a linear order. It maintains an *active list* of live ranges that are live at the current point in the function, and this is how it detects interference without computing the full interference graph. The active list is the key to linear scan's speed, but it is also its greatest weakness.
When all physical registers are blocked by interfering live ranges in the active list, a live range is selected for spilling. Live ranges being spilled without being split first cause the mess that the rewriter is working so hard to clean up. We would much rather split them into smaller pieces that might be assignable, but this would require the linear scan algorithm to backtrack. This is very expensive, and full live range splitting isn't really feasible with linear scan.

Basic allocator



The new basic allocator does away with linear scan's dependence on visiting live ranges in linear order. Instead, it uses a priority queue to visit live range in order of decreasing spill weight. The active list used for interference checks is replaced with a set of *live interval unions*. Implemented as a B+ tree per physical register, they are an efficient way of checking for interference with already assigned live ranges. Unlike the active list, live interval unions work with any priority queue order.

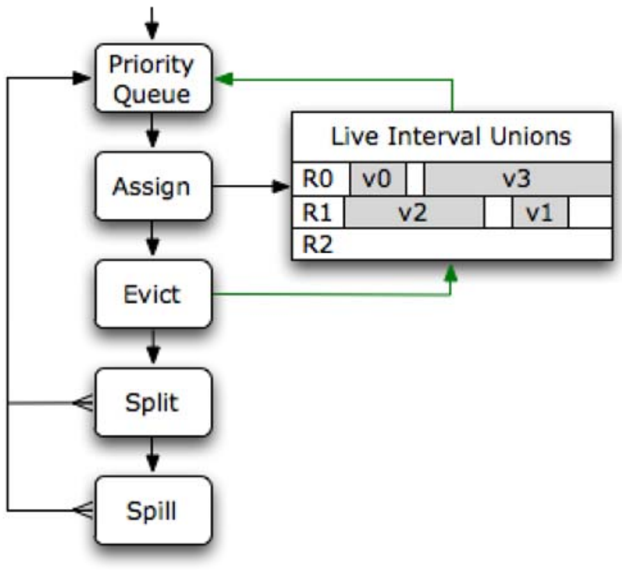
When a live range cannot be assigned to any physical register in its register class, it is spilled. Because live ranges are assigned in order of decreasing spill weight, all the interfering live ranges in the live interval union have a higher spill weight. It is not necessary to look for a better spill candidate.

On CISC architectures, the spill slot memory accesses can often be folded into existing instructions. On RISC architectures, explicit load and store instructions must be inserted. This will also create new tiny live ranges between the spill code and the original instructions using the spilled live range. These new live ranges are put back on the priority queue with an infinite spill weight—they cannot be spilled again.

Technically, these small live ranges with high spill weight should have been assigned first, but the basic allocator never backtracks. Therefore, it can happen that such a live range is blocked by already assigned live ranges with smaller spill weights. In that case, the allocator picks a physical register and spills the interfering live ranges assigned to that register instead.

The basic allocator produces code very similar to linear scan's output, and it also depends on the virtual register rewriter to clean up the code for good results. It doesn't offer significant advantages over linear scan, and it is intended mostly for testing the framework of priority queues and live interval unions. The basic algorithm is very simple, and it offers many opportunities for tweaking. Greedy does just that.

Greedy allocator



The first thing to notice about the basic algorithm is that its priority queue order doesn't work well for coloring registers optimally. Spill weights are computed as use densities, and small live ranges tend to have high spill weights. This means that all the tiny live ranges are allocated first. They use up the first registers in the register class, and the large live ranges get to fight over the leftovers. Most of them end up spilling.

Greedy avoids this problem by allocating the large live ranges first. This makes the full register class available for the large ranges, and the small ranges can often fit in the gaps. Some functions have too many large live ranges, so there is not enough room for all the small live ranges. It would be really bad to spill small live ranges with high spill weights, so instead already assigned live ranges with lower spill weight can be evicted from the live range union. Evicted live ranges are unassigned from their physical register and put back in the priority queue. They get a second chance at being assigned somewhere else, or they can move on to *live range splitting*.

When a live range cannot find interfering live ranges it is allowed to evict, it is not spilled right away. If possible, it is split into smaller pieces that are put back on the priority queue. This is a very important optimization. A large live range may be idle a lot of the time, but used intensively in a hot loop. By creating a separate live range covering the hot loop, there is a good chance it will be assigned a register. The remaining live range may spill outside the loop where it was idle anyway. A live range is only spilled when the splitter decides that splitting it won't help. That usually happens after all the busy regions have been separated, and the remaining live range only has a few copies to and from the busy registers.

The interaction between live range splitting and eviction creates a process of gradual refinement. As live ranges are split around busy regions, they get a higher spill weight. This may allow them to evict older live ranges that are less busy in that region. The evicted ranges are split, and so on.

The gradual process of splitting usually terminates before the live ranges become tiny, and the end result is a set of live ranges covering multiple instructions, or even multiple basic blocks. This means that there is nothing for the rewriter to clean up, and indeed greedy uses a completely trivial rewriter that is 85 lines of code compared to 2600 in the old rewriter.

The code generated by the greedy algorithm is almost always better than what linear scan can do. Usually this is because live range splitting was able to eliminate spill code from loops. Greedy does know some more tricks, though.

Tweaks

It was an important design goal to make the algorithm as flexible as possible, and to avoid introducing arbitrary constraints. It is possible to change machine code and live ranges at any time. Simply evict the relevant live ranges, make the change, and put them back on the queue.
This flexibility allows many tweaks to the register allocator:

- Register preferences. Function arguments are passed in specific physical registers defined by the ABI. LLVM represents this with copies between physical and virtual registers before and after function calls. The register allocator tries to assign the virtual registers to the same physical registers, so the copies can be eliminated. Linear scan was never really good at this—the preferred physical register had often been occupied by an earlier assignment. Greedy can simply evict the earlier assignment when that happens.
- Prefer small encodings. On architectures like ARM Thumb2 and x86-64, some registers require a larger instruction encoding. Greedy will evict less important live ranges from the cheap registers before it assigns an expensive register. This means that the larger instruction encodings are used less often, and overall code size decreases.
- Dead code elimination. Optimizations like rematerialization cause live ranges to be shorter, or even completely unused. Greedy will recompute the live ranges exactly, and recursively eliminate dead code.
- Register class inflation. Live range splitting creates virtual registers that are used by fewer instructions. This sometimes lifts a constraint, so the virtual register can be moved to a larger register class. Depending on the architecture, this can double the number of registers available to the new live range.

The greedy register allocator still has lots of room for improvement. That was the whole point of replacing linear scan.

Posted by Jakob Stoklund Olesen at 9:34 PM

Labels: new-in-llvm-3.0, optimization

Newer Post Home Older Post

LLVM Logo

Labels

- Clang (13)
- optimization (11)
- C++ (8)
- LLVM-IR (5)
- meta (5)
- new-in-llvm-2.7 (5)
- MC (4)
- llvm-users (4)
- codegen (3)
- devmtg (3)
- new-in-llvm-3.0 (3)
- LLDB (2)
- sanitizer (2)
- GHC (1)
- KLEE (1)
- SelectionDAG (1)
- asip (1)
- dragonegg (1)
- eda (1)
- new-in-llvm-2.8 (1)
- new-in-llvm-3.1 (1)
- new-in-llvm-3.3 (1)
- tce (1)
- testing (1)
- tta (1)
- vliw (1)

Blog Archive

- ▶ 2013 (3)
- ▶ 2012 (4)
- ▼ 2011 (11)
  - ▶ December (2)
  - ▶ November (2)
  - ▼ September (1)
    - Greedy Register Allocation in LLVM 3.0
  - ▶ May (5)
  - ▶ April (1)
- ▶ 2010 (17)
- ▶ 2009 (6)

About The LLVM Blog

This blog is intended to be a news feed for information about the LLVM Compiler Infrastructure and related subprojects.

We welcome new contributors. If you'd like to write a post, please get in touch with Chris.