Fighting register pressure in GCC

Vladimir N. Makarov
Red Hat
vmakarov@redhat.com

Abstract

The necessity of decreasing register pressure in compilers is discussed. Various approaches to decreasing register pressure in compilers are given, including different algorithms of register live range splitting, register rematerialization, and register pressure sensitive instruction scheduling before register allocation.

Some of the mentioned algorithms were tried and rejected. Implementation of the rest, including region based register live range splitting and rematerialization driven by the register allocator, is in progress and probably will be part of GCC. The effects of discussed optimizations will be reported. The possible directions of improving the register allocation in GCC will be given.

Introduction

Modern computers have several levels of storage. The faster the storage, the smaller its size. This is the consequence of a trade-off between the computer speed and its price. The fastest storage units are registers (or *hard registers*). They are not enough to store the values of operations and directly referred variables for any serious program.

It is very hard to force any optimization in a compiler (especially in a portable one) to use the hard registers effectively. Therefore most of compiler optimizations is written as if there is infinite number of virtual registers called *pseudo-registers*. The optimizations use them to store intermediate values and values of small variables. Although there is an untraditional approach to use only memory to store the values. For both approaches we need a special pass (or optimization) to map pseudo-registers onto hard registers and memory; for the second approach we need to map memory into hard-registers instead of memory because most instructions work with hard-registers. This pass is called register allocation.

A good register allocator becomes a very significant component of an optimized compiler nowadays because the gap between access times to registers and to 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 a part of architecture. Even processors with new architectures containing more hard-registers need a good register allocator (although in less degree) because the programs run on these computers tend to be more complicated too.

The register allocator is one or more compiler components that could be considered as ones solving two major tasks (mostly in an integrated way). The first and most interesting one is to decrease register pressure to the level defined by the number of hard registers by different transformations. And the second one is to assign hard registers to pseudo-registers effec-

tively.

So what is register pressure? There are two commonly used definitions. The wide one is the number of hard registers needed to store values of the pseudo-registers at given program point. Another one is the number of living pseudo-registers.

There are a lot of known transformations that decrease register pressure. Some of these transformations generate code which could and should be corrected later. Some transformations are easily and naturally integrated with other transformations, such as the ones decreasing register pressure, assigning hard registers, and fixing the pitfalls of the previous transformations (such as register coalescing in a colouring based register allocator). Some of them are hard to integrate in one pass.

Currently GCC has two register allocators. The new one was written about two years ago and is described in details in [Matz03]. It is based on the Chaitin, Briggs, and Appel approaches to register allocation [Chaitin81, Briggs94, Appel96].

The old register allocator (I will call it *the original register allocator*) has been existing since the very first version of GCC. It was written by Richard Stallman. Some of its important components stayed practically unchanged since the first version. Richard Stallman took the register allocator design from a portable Pastel (an extension of the programming language Pascal) compiler written in Livermore Laboratories [Stallman04]. The design of the Pastel register allocator (which actually was a second version for the Pastel compiler) is very similar to the GCC one [Killian04]—they both have the same separation on a pass assigning hard registers to pseudo-registers and a pass which actually changes the code following the assignment and, if it is not possible, generates additional instructions to reload the registers.

Despite its lack of many modern optimizations present in the new register allocator, the original register allocator can easily compete with the new one in terms of compiler speed, quality of the generated code and size of code. This was a major reason for me to start work on improving the original register allocator. After thorough investigation, I found that the method of assigning hard registers is very similar to the *priority based colouring* register allocator [Chow84, Chow90], although it is more similar to the modifications described in [Sorkin96]. It was confirmed later.

Chow's approach is a real competitor to the Chaitin/Briggs approach. Some advantages of Chow's approach are acknowledged even by Preston Briggs [Briggs89]. Chow's algorithm is used in SGI Pro64 [Pro64] compiler and derived compilers like Open64 [Open64] and ORC [ORC]. For example, as Briggs' optimistic colouring, Chow's algorithm easily finds hard-registers for the diamond conflict graph (see Figure 1).

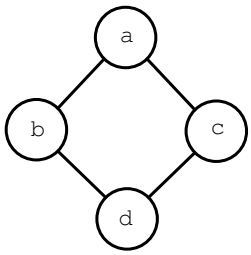

Figure 1: Diamond graph

All that was mentioned above was a major motivation to start work on improvement of the original register allocator. This article is focused on improving the original GCC register allocator. The first section describes the original GCC register allocator. The second section describes the method for decreasing the register pressure for the original register allocator based on register live range splitting. The third section describes decreasing regis-

ter pressure based on the live range shrinking approach. The fourth section describes other possible improvements to the original register allocator. The fifth section gives conclusions from my work.

1 The original register allocator in GCC

The original register allocator contains a lot of passes. Figure 2 describes the major passes and their order.

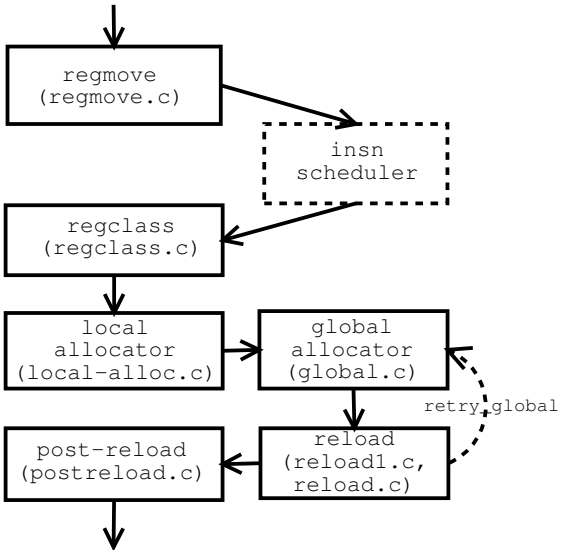

Figure 2: The original register allocator

The regmove pass is usually not considered to be a part of the original register allocator. I included it because the pass solves one task (register coalescing) peculiar to register allocators. The pass removes some register moves if the registers have the same value and it can be found in a basic block scope. Although 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.

If register coalescing and global value numbering (mentioned in Section 4) are a part of GCC, we could try to remove register coalescing from this pass.

The instruction scheduler is not a part of the original register allocator. It is present just to show GCC's major passes starting with the regmove pass. Although the instruction scheduler could solve the task of decreasing register pressure (see section "register pressure sensitive instruction scheduling").

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 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 register or `ALL_REGS` which contains all 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 registers besides the ones in the preferred class).

It is interesting to note that the pass also implicitly does code selection. Regclass works in two passes. On the first pass, it defines the preferred and alternative classes without taking the possible classes of other operands into account. For example, an instruction with two operand pseudo-registers exists in two variants; one accepting classes *A* and *B*, and other one accepting *C* and *D*. On the first pass, the algorithm does not see that the variant with classes *A* and *D* will be more costly because it will require the generation of additional move instructions. On the second pass, the algorithm will take it into account. As a result the preferred or alternative class of a pseudo-register could change. This means two passes are not enough to find the preferred and alternative classes accurately; but it is a good approximation.

The file regclass.c also contains functions to scan the pseudo-registers to find general information about them (like the number of references and sets of pseudo-registers, the first and last instructions referencing the pseudo-registers etc.).

The local allocator assigns hard-registers only to pseudo-registers living inside one basic block. The result of the work is stored in the global array `reg_renumber` whose element values indexed by pseudo-register numbers are hard-registers assigned to the corresponding pseudo-registers.

Besides assigning hard-registers, the local allocator does some register coalescing too: 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 a simple copy and constant propagation. It is implemented in the function `update_reg_equiv`.

Actually all hard-registers could be assigned in the global allocator. Such division between the local and global allocator has historical roots. In my opinion it is reasonable to remove the local allocator in the future because faster allocation of local pseudo-registers does not compensate the cost of an additional pass. If all assigning hard-registers is done in the global register allocator (but we still call `update_equiv_regs`), GCC is in average 0.5% faster on SPEC2000 benchmarks on Pentium 4.

The global allocator assigns hard-registers to pseudo-registers living in more 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 one for the local pseudo-register.

The global allocator forms a bit-vector for each pseudo-register containing hard registers conflicting with the pseudo-registers, builds a conflict graph for pseudo-registers and 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. If the current pseudo-register got a hard-register, the hard-register is added to the hard-register conflict bit-vectors of all pseudo-registers conflicting with the given pseudo-register. 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. In brief, the global allocator is looking for a hard-register to assign to a pseudo-register in the following order:

1. a callee saved hard-register which is in the pseudo-register's preferred class and which is preferred by the pseudo-register while not being preferred by another conflicting pseudo-register.
2. a callee saved hard-register which is in the pseudo-register's preferred class and which is preferred by the pseudo-register.
3. a callee saved hard-register which is in the pseudo-register's preferred class.
4. as in 1-3 but a caller saved hard-register (if it is profitable) instead of callee-saved one.
5. as in 1-4 but the hard-register is in the pseudo-register's alternative class.

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. 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 register of another class, loading or storing a register of the given class. To choose the best pseudo-register for expelling, 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), assigning stack slots for 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 tar-

get processor's register file, permitting 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 in basic block context.

2 Live Range splitting

Live range splitting is based on idea that if we split the live range of a pseudo-register in several parts, the pseudo-register in each live range part will conflict with fewer other pseudo-registers; less hard-registers will be needed for all the pseudo-registers. Figure 3 illustrates this. Pseudo-register *A* conflicts with two pseudo-registers *B* and *C*, but in part 1 and 2 of its live range the pseudo-register conflicts only with one other pseudo-register.

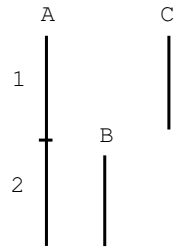

Figure 3: Live range splitting for pseudo-register A.

Live range splitting might require the generation of additional instructions; e.g. instructions storing/loading pseudo-register value into/from memory, moving the pseudo-register into/from a new pseudo-register, or just recalculation of the pseudo-register value. Cost of such additional instructions can outweigh the benefits of reducing the register pressure. So any live range splitting algorithm should take this problem into account.

2.1 Register renaming

Register renaming could be considered as no cost live range splitting because no additional instructions need to be generated. We can change a pseudo-register into several ones if there are multiple independent parts of the pseudo-register's usage. The following is a high level example when register renaming could be used.

```
for (i = 0; i < n; i++) { ... }
for (i = 0; i < k; i++) { ... }
```

After register renaming (the pseudo-register renamed is the variable `i`), the corresponding code could look like

```
for (i = 0; i < n; i++) { ... }
for (i_1 = 0; i_1 < k; i_1++) { ... }
```

This optimization was written independently by Jan Hubicka from SUSE and me. Jan's variant is in GCC mainline now. Earlier it was activated by using `-fweb` (independent part of a pseudo-register is traditionally called web in colouring based register allocator). After solving the problem of generating correct debugging information it is default for `-O2` now. Tables 1 and 2 contain SPEC2000 results for Pentium 4 with and without register renaming. Although the results are not impressive for SPECInt2000 (mainly because of perlbmk), this optimization is a "must be" for any optimizing compiler. In most benchmarks it could considerably increase the performance. The results look much better for SPECfp2000. The reduced register pressure means less instructions for spilling and restoring registers and shorter instructions because hard registers instead of memory are used in more instructions. As the result code size for Pentium 4 is 0.3% and 0.6% less in average for SPECint2000 and SPECfp2000 correspondingly.

Benchmarks	Base ratio	Peak ratio	Change
164.gzip	747	750	+0.40%
175.vpr	531	530	-0.19%
176.gcc	891	897	+0.90%
181.mcf	539	539	+0.00%
186.crafty	800	798	-0.25%
197.parser	649	648	-0.15%
252.eon	663	682	+2.87%
253.perlbmk	1019	939	-7.85%
254.gap	831	838	+0.84%
255.vortex	973	961	-1.23%
256.bzip2	621	628	+1.11%
300.twolf	665	671	+0.90%
SPECint2000	728	726	-0.27%

Table 1: SPECint2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with register renaming.

Benchmarks	Base ratio	Peak ratio	Change
168.wupwise	895	898	+0.33%
171.swim	617	621	+0.64%
172.mgrid	598	597	-0.17%
173.applu	636	637	+0.16%
177.mesa	654	656	+0.31%
179.art	245	250	+2.04%
183.equake	984	988	+0.40%
200.sixtrack	352	406	+15.34%
301.apsi	406	405	-0.25%
SPECfp2000	552	563	+1.99%

Table 2: SPECfp2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with register renaming.

Register renaming also improves instruction scheduling by removing some anti-dependencies. So it could be useful even for architectures with many hard registers like IA-64. Table 3 contains SPECfp2000 results for Itanium 2 with and without register renaming. The code size for SPECfp2000 was also 0.24% less.

Andrew Macleod from RedHat also implemented this optimization in the transformation of SSA into normal form (this is made very easy on this pass because the independent parts

Benchmarks	Base ratio	Peak ratio	Change
168.wupwise	383	385	+0.52%
171.swim	388	395	+1.80%
172.mgrid	229	230	+0.44%
173.applu	293	297	+1.37%
177.mesa	660	658	-0.30%
179.art	1605	1583	-1.37%
183.equake	315	315	0.00%
200.sixtrack	157	161	+2.55%
301.apsi	266	267	+0.38%
SPECfp2000	373	375	+0.53%

Table 3: SPECfp2000 for Itanium2 GCC with `-O2` without and with register renaming.

of a variable usage are present naturally in SSA). He reported about 2% improvement for SPECint2000 for Pentium 4. When tree-SSA branch becomes GCC mainline, Jan's implementation probably should probably go away because register renaming is made easier and faster during the translation of SSA into normal form.

2.2 Live range splitting

The idea of this approach is to store a pseudo-register living through a region but not used in the region right before entering the region and reload its value right after leaving the region. It decreases register pressure in the region by one.

I have implemented practically the same algorithm described in [Morgan98]. Morgan's algorithm works as a separate compiler pass. It starts work on the topmost loops with the register pressure higher than the number of available hard-registers. It searches for pseudo-registers living through the loop but not being used there. It chooses a pseudo-register living through a maximal number of loops (and basic blocks) which are neighbors of the loop being processed. Then it spills the pseudo-register before the loop(s) and restore the pseudo-register after the loop(s). After processing the

loops the algorithm recursively processes sub-loops. When all sub-loops are processed, the algorithm tries to decrease register pressure inside basic blocks. Figure 4 illustrates how the algorithm works.

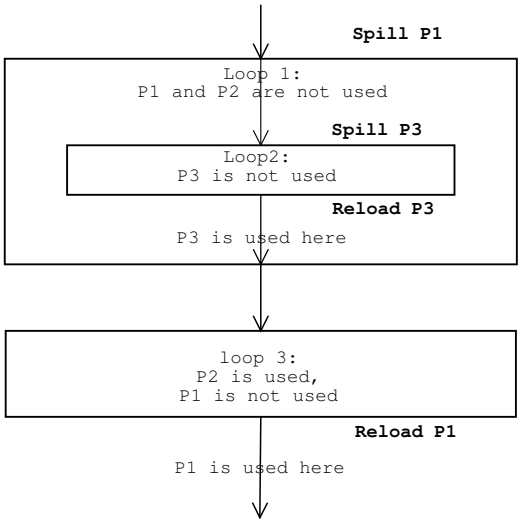

Figure 4: Illustration of Morgan's algorithm of live range splitting.

The current implementation is different from Morgan's in the following:

- Although our implementation also works on loops, it could be easily modified to work on any nested regions instead.
- Instead of spilling the pseudo-register into memory before the loop(s) and reloading it we create a new pseudo-register living only in the loop(s) and inserting instructions shuffling the two pseudo-registers. If both pseudo-registers get memory or hard-registers (it really can happen in the reload pass), the move instructions are coalesced (see the section on coalescing later in this article). If one pseudo-register gets a hard-register and another one gets memory, the move instructions will be transformed into memory store and load instructions.
- GCC has a complicated description model for registers. A hard-register can belong to more one register class. A pseudo-register can get a hard-register from two different classes (see the description of the original register allocator above). To calculate register pressure we consider a pseudo-register belonging to the smallest register class containing the two pseudo-register classes (preferred and alternative ones).
- We do not decrease register pressure inside the basic blocks. We found that on most benchmarks this is not profitable.

The current SPECInt95 results for the optimization usage for Pentium 4 are given in Table 4. The improvement can be even more for some benchmarks. For example, Fast Fourier Transform became 6% faster for Pentium 4 with this optimization, a linear-space Local similarity algorithm [Huang91] became 14% faster, and fftbench [Ladd03] became more 30% faster.

Benchmarks	Base ratio	Peak ratio	Change
099.go	68.6	67.8	-1.17%
124.m88ksim	72.3	71.8	-0.69%
126.gcc	75.2	74.8	-0.53%
129.compress	55.5	56.4	+1.62%
130.li	78.3	78.0	-0.38%
132.ijpeg	72.5	72.6	+0.14%
134.perl	68.5	79.8	+16.50%
147.vortex	68.1	68.6	+0.73%
SPECint95	69.6	70.9	+1.87%

Table 4: SPECInt95 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with live range splitting.

I see the following possible improvements to the implementation:

- Better utilization of profile information to choose loops with many iterations.

- Forming regions based on the profile information different from the loops for the algorithm of live range splitting.
- Choosing pseudo-registers whose live range splitting does not result in critical edge splitting. As a consequence, no additional branch instructions will be generated. It could be important for live range splitting around loops with few iterations.
- More accurate evaluation of register pressure for register classes to which a living pseudo-register belongs.

2.3 Rematerialization

Instead of reloading a pseudo-register's value we could just recalculate it again if it is more profitable. Such approach is called register rematerialization. Preston Briggs believed that it is a more promising approach than live range splitting. It requires that all the pseudo registers used as operands are live and got hard registers because otherwise we will need to reload the operand value too. Reloading the operand's value usually costs the same as reloading the pseudo-register in question.

My current implementation of the register rematerialization works between global register allocation and reload passes. To rematerialize a pseudo-register we insert an existing instruction setting up the pseudo-register's value. To know what instructions could be inserted we define the partial availability of instruction patterns according to the following equations.

$$P\_PavIn_i = \bigcup_{j \in Pred(i)} P\_PavOut_j$$
$$P\_PavOut_i = (P\_PavIn_i - P\_Kill_i) \bigcup P\_Gen_i$$

Here P_Kill_i is a set of patterns whose defined and used locations (registers or memory) are redefined or clobbered in basic block i or whose clobbered registers are live at the end of basic block. P_Gen_i is a set of patterns in basic block whose defined or used locations are not killed in the basic block after the pattern's occurrence and whose clobbered registers are not live at the end of the basic block.

After we calculated partial availability of patterns, we use it as an initial value to calculate availability of patterns according to the following equation.

$$P\_AvIn_i = \bigcap_{j \in Pred(i)} P\_AvOut_j$$
$$P\_AvOut_i = (P\_AvIn_i - P\_Kill_i) \bigcup P\_Gen_i$$

The algorithm itself looks like

```
foreach insn I defining the
        only pseudo-register D do
  if D got a hard-register then
    foreach pseudo-register operand Op
            of I do
      if Op got memory then
         Pat := a pattern with a minimal
                cost available right before
                I and whose the only
                destination pseudo-register
                is Op and whose all other
                operand pseudo-registers
                got hard-registers;
        if there is Pat and its cost is less
           than cost of loading Op then
          insert insn before I with pattern
              Pat changing Op on D;
          change Op in I on D;
          break;
        fi
      fi
    done
  fi
done
```

e.g. if pseudo-register *A* got memory and pseudo-registers *B*, *C* and *D* got hard-registers, the algorithm will work as follows

```
A <- op1 (B, C)          ...
...                  ->  ...
D <- op2 (A, E)          D <- op1 (B, C)
                         D <- op2 (D, E)
```

If the second instruction in the example is

move, the algorithm together with the dead code elimination will work as

```
A <- op1 (B, C)          ...
...                  ->  ...
D <- A                   D <- op1 (B, C)
```

Table 5 contains results of the optimization for SPECint2000 for Pentium 4.

Benchmark	Base	Peak	Change
164.gzip	838	839	+0.12%
175.vpr	602	598	-0.66%
176.gcc	1137	1146	+0.79%
181.mcf	715	715	0.00%
186.crafty	874	875	+0.11%
197.parser	734	734	0.00%
252.eon	764	763	-0.13%
253.perlbmk	1145	1164	+1.66%
254.gap	954	951	-0.31%
255.vortex	1079	1080	+0.09%
256.bzip2	743	745	+0.27%
300.twolf	757	767	+1.32%
Est. SPECint2000	845	848	+0.36%

Table 5: SPECint2000 for Pentium 4 with `-O2 -mtune=pentium4` without and with register rematerialization.

Register rematerialization could be done in a separate pass before the register allocation [Simpson96]. In brief, Simpson's algorithm looks like

```
foreach basic block BB do
  while the register pressure is too high
        in BB do
    P := a pattern available and live at
         the end of BB with a result
         pseudo-register is not used in BB
         and its operands are live
         at the end of BB;
    if there is no such P then
      break;
    fi
    put insns with pattern P on edges
        exiting from BB where P are live;
    move the insns to the bottom of CFG as
         far as possible along the paths
         where P is still available, and P
         and its operands are live;
    update the register pressure in BB and
           basic blocks we moved the insns
           through;
  done
done
```

The liveness of a pattern in a CFG point means that a result register of the pattern is used in another point achieved from the given point. Figure 5 illustrates how the algorithm works.

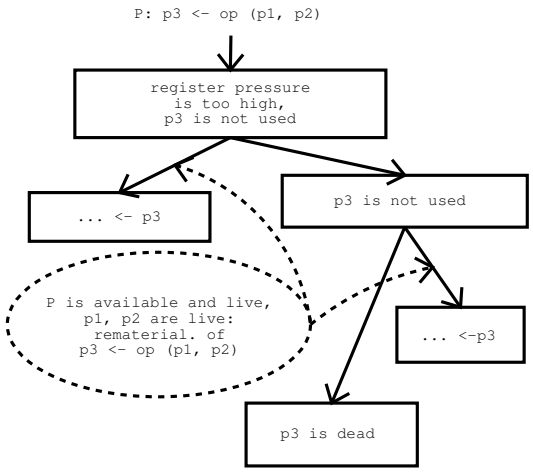

Figure 5: Illustration of Simpson's algorithm of rematerialization.

I have implemented Simpson's approach in GCC. It gave about 1.3% improvement for SPECint2000 for Pentium 4 on the tree-ssa branch. And after deciding to implement Morgan's live range splitting, I rejected Simpson's implementation because I believe that Mor-

gan's live range splitting together with register rematerialization after global register allocation will work better. I see the following reasons for this:

- It is difficult to know which operand pseudo-registers will get hard-registers in the end. Adding instruction rematerializing pseudo-register's value might result in generation of additional load instructions in the reload pass if the operand pseudo-registers do not get hard-registers.

- Morgan's approach to live range splitting works in more cases than Simpson's. The instructions shuffling pseudo-registers generated in Morgan's algorithm are removed by coalescing and, if it is not possible, rematerialized.

- Rematerialization could be done in more cases. The single criterion is a profitability not just high register pressure as in Simpson's approach.

3 Live range shrinking

The live range shrinking approach is to move the definitions of pseudo-register as close as possible to their usages. It decreases the number of conflicts for the pseudo-register and consequently may decrease register pressure. There are few articles devoted this approach (one of them is [Balakrishnan01]). The reason for this is in its constraints for modern pipelined processors. Solving this problem without taking instruction scheduling into account could worsen code in many cases. So live range shrinking mainly became a part of register pressure sensitive instruction schedulers.

3.1 Register Pressure Sensitive Instruction Scheduling

GCC uses a classical two pass instruction scheduling approach: instruction scheduling both before and after the register allocator. It works well for RISC processors with a large enough number of registers.

For processors with few registers, however, instruction scheduling before register allocation creates such high pressure that it actually worsens the code. Therefore it is switched off for x86 and sh4.

One year ago Dale Johannesen from Apple added a new heuristic right after the critical path length heuristic. This heuristic prefers instructions with smaller contribution to register pressure. He reported about 2% improvement for SPECint2000 for PowerPC.

Sanjiv Gupta implemented machine-dependent register pressure-sensitive instruction scheduling for SH4. He reported a big improvement for some benchmarks (Table 6) when the first instruction scheduler with the register-pressure heuristic was switched on. Unfortunately, he did not compare instruction scheduling with and without the heuristic (probably the results would be even better because earlier the first instruction scheduling pass without any register-pressure heuristic was switched off).

Sanjiv's implementation is very similar to the Hsu and Goodman approach [GooHsu88] to register pressure sensitive instruction scheduling: when the pressure becomes high, it uses register pressure heuristic as major one instead of the critical path length heuristic. I have implemented Hsu's approach in a machine independent way. My goal was to improve x86 code by switching on the first instruction scheduling pass. Although GCC with the register pressure sensitive approach in the first pass generated a more 1% better code for

Benchmark	Base	Peak	Change
Gsm compression	31.83	26.16	+17%
Gsm decompression	17.72	16.94	+4.4%
cjpeg -dct int	2.30	2.34	-1.7%
cjpeg -dct float	2.12	2.19	-3%
djpeg -dct int	1.53	1.45	+5%
djpeg -dct float	1.69	1.42	+15%
gzip	225	222	+1%
gunzip	17.30	16.69	+3.5%
Mpg123	1.29	1.26	+2%

Table 6: Benchmarks for SH4 GCC with `-O2` without the 1st instruction scheduling and with the 1st register pressure sensitive instruction scheduling.

SPECfp95 than with the standard first pass, the results are disappointing in comparison with GCC without any first instruction scheduling. Table 7 contains SPEC95 results for the programs compiled without the first instruction scheduling pass (default in GCC for x86) and with Hsu's approach in the first instruction scheduler. I used Athlon MP because GCC still has no pipeline description for Pentium 4.

The most interesting result is for *fpppp*: the code became practically 3 times slower (SPECfp95 results would be very close without *fpppp*). The hot point of *fpppp* is the function with one huge basic block. The register pressure reaches several hundred there for x86 GCC. It looks to me like the basic block was optimized manually to minimize the register pressure. Any rearrangement of the instructions results in a higher register pressure, especially for x87 floating point top stack register. So in my opinion, to make a successful register pressure sensitive instruction scheduler for x86, we need a more sophisticated approach than Hsu's on-the-fly approach. These approaches should be based on the evaluation of all data flow graphs like a parallel interference graph [Norris93] or a register reuse graph [Berson98].

Benchmarks	Base	Peak	Change
099.go	68.9	68.2	-0.73%
124.m88ksim	52.8	51.8	-1.89%
126.gcc	57.5	57.1	-0.70%
129.compress	28.0	27.9	-0.36%
130.li	58.9	59.0	+0.17%
132.ijpeg	53.1	50.6	-2.82%
134.perl	79.2	76.3	-3.66%
147.vortex	50.3	50.5	+0.40%
SPECint95	54.0	53.3	-1.30%
101.tomcatv	74.1	75.7	+2.16%
102.swim	139	139	0.00%
103.su2cor	22.4	21.8	-2.68%
104.hydro2d	24.5	24.3	-0.82%
107.mgrid	47.7	50.6	+6.08%
110.applu	28.2	27.4	-2.84%
125.turb3d	53.2	51.4	-3.38%
141.apsi	32.5	33.3	+2.46%
145.fpppp	148	54.0	-63.51%
146.wave5	77.3	73.7	-4.66%
SPECfp95	52.2	47.0	-9.96%

Table 7: SPEC95 results for Athlon MP with `-O2 -mtune=athlon` without the first instruction scheduler and with Hsu's register pressure sensitive first instruction scheduling.

4 Other improvements of the GCC original register allocator

4.1 Coalescing

Live range splitting tends to create unnecessary move instructions. As I mentioned above, we generate additional pseudo-registers and instructions shuffling them instead of the traditional approach generating instructions spilling registers to memory and restoring them. Even if the live range splitting optimization is not run, there are still unnecessary move instruction generated by the previous optimizations. To remove them, pseudo-register coalescing is run after the global register allocator. If the pseudo-registers in a move instruction do not conflict we could use one pseudo-register and remove the move instructions. It is done if both pseudo-registers got hard registers or

both pseudo-registers were placed in memory (it means that the move would have been transformed into instructions moving the memory). The following example describes the two situations (the number in the parentheses is the hard register number given to the pseudo-register):

```
p256 (1) <- p128 (2)
     or
p256 (Memory) <- p128 (Memory)
```

Sometimes, removing a pseudo-register move instruction when one pseudo-register gets a hard-register in the global register allocator and another one gets memory could be profitable too. The resulting pseudo-register will be placed in memory after coalescing the two pseudo-registers. Profitability is defined by the execution frequency of the move instruction and the reference frequency of the pseudo-register which got a hard-register. A typical situation when it is profitable is given on figure 6. The pseudo-register *p128* got the hard register number 2 and *p256* was placed in memory.

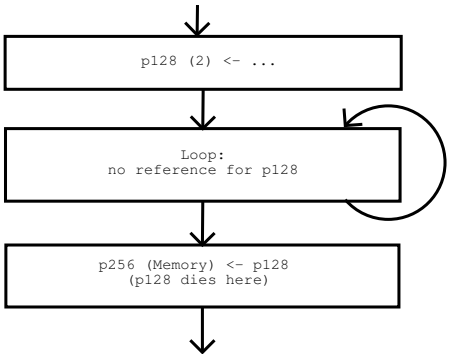

Figure 6: Coalescing memory and register.

Even if there is no move instruction between two pseudo-registers which are placed in memory (usually on the program stack), we can coalesce them. What is the sense of such an optimization? Although the optimization does not remove instructions, it decreases the size of the used stack (it is very important for the Linux kernel which usually has strict constraints for the size of the program stack). For example, the average decrease of function stack frames is about 4% with this optimization for Linpack x86 code. The optimization also improves data locality and code locality for some architectures like x86 because in many cases smaller displacements in instruction are used (we are using the first found stack slot approach). Table 8 shows the text segment's size decrease for the SPECfp2000 benchmarks for Pentium 4. The improved code and data locality considerably improves the code. Table 9 shows the SPECfp2000 performance results for code without and with the optimization for Pentium 4.

Benchmarks	Base	Peak	Change
168.wupwise	25128	24648	-1.910%
171.swim	7078	7014	-0.904%
173.applu	58741	58453	-0.490%
177.mesa	443993	439369	-1.041%
179.art	12011	12011	0.000%
183.equake	17026	17026	0.000%
200.sixtrack	844452	815060	-3.481%
301.apsi	106317	103341	-2.799%
Average			-1.33%

Table 8: SPECfp2000 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with coalescing the program stack slots.

The patch improves code and data locality, therefore GCC becomes a bit faster. User time for x86 bootstrapping decreased from 14m0.150s to 13m58.890s. The better code and data locality improves SPECFP2000 benchmark results too (about 2.4%).

4.2 Register migration

When the reload pass needs a hard register for a reload, it expels a living pseudo-register from the hard register assigned to it by the local or global register allocator. Then it tries to reas-

Benchmarks	Base ratio	Peak ratio	Change
168.wupwise	890	887	-0.34%
171.swim	604	609	+0.83%
173.applu	624	627	+0.48%
177.mesa	629	639	+1.59%
179.art	244	248	+1.64%
183.equake	964	963	-0.01%
200.sixtrack	337	385	+14.24%
301.apsi	401	407	+1.97%
SPECint2000	388	399	+2.83%

Table 9: SPECfp2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with coalescing the program stack slots.

sign a free hard register to the pseudo-register (function `retry_global_alloc`). Usually it fails especially when the processor has few registers or there is a high register pressure in the function. So finally the pseudo-register is placed in memory. Figure 7 shows an example of such a situation (the pseudo-register *p128* is expelled from hard register *A* because it is needed for an instruction which is in the live range of *p128*).

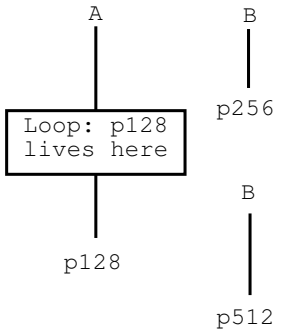

Figure 7: Case for the register migration.

Sometimes it is more profitable to use another hard register (*B* in the example) instead of memory for the pseudo-register. It might be possible by expelling another rarely used pseudo-register (*p256* and *p512* in the example) from their hard registers. In their own turn the expelled pseudo-registers can also migrate.

The optimization works well with processors with irregular register files (which means generation of more reloads because of strict instruction constraints for input/output registers).

Tables 10 and 11 contain SPEC2000 results for Pentium 4 for benchmarks whose codes are different when the optimization is used. We see that the code is smaller and the results are better. Practically the single important degradation is perlbmk (but it can be fixed by the register rematerialization and live range splitting mentioned above). Significant improvement for GCC is more important than perlbmk degradation because it is more difficult to improve GCC than perlbmk; 50% of all time of perlbmk is spent in one very specific function. It is regular expression matching. The SPEC95 perlbmk was a more fare benchmark because it tested the interpreter itself, not regular expression matching.

Benchmarks	Base ratio	Peak ratio	Change
175.vpr	594	596	+0.34%
176.gcc	1123	1133	+0.89%
186.crafty	869	877	+0.92%
197.parser	730	729	-0.14%
252.eon	765	764	-0.13%
253.perlbmk	1159	1133	-2.24%
254.gap	943	944	+0.11%
255.vortex	1052	1056	+0.38%
256.bzip2	737	735	-0.27%
300.twolf	753	763	+1.33%
173.applu	771	772	+0.13%
177.mesa	720	726	+0.83%
200.sixtrack	394	392	-0.51%
301.apsi	486	489	+0.62%

Table 10: SPEC2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the register migration.

This optimization makes GCC a bit faster too (the compiler bootstrap test on Pentium 4 is 0.13% faster with the optimization). As for architectures with more regular register files, I found that three SPECfp95 test codes for

Benchmarks	Base	Peak	Change
175.vpr	128917	128949	0.025%
176.gcc	1241720	1241440	-0.022%
186.crafty	204846	204878	0.016%
197.parser	85436	85420	-0.019%
252.eon	480338	480354	0.003%
253.perlbmk	473971	473667	-0.064%
254.gap	421816	421592	-0.053%
255.vortex	568904	569128	0.039%
256.bzip2	28133	28117	-0.057%
300.twolf	181055	181055	0.000%
Average			-0.013%
173.applu	58741	58741	0.000%
177.mesa	443993	443049	-0.213%
200.sixtrack	844452	843892	-0.066%
301.apsi	106317	106317	0.000%
Average			-0.070%

Table 11: SPEC2000 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the register migration.

PowerPC were different (applu, turb3d, and wave5). Test applu was sped up about 1% (two others had the same result).

4.3 More accurate information about register conflicts

The original register allocator used standard live information to build a conflict graph. This live information is based on the most widely used definition of pseudo-register liveness: Register *R* lives at point *p* if there is a path from *p* to some use of *R* along which *R* is not redefined. The live information is described by the following data flow equations:

$$LiveIn_i = (LiveOut_i - Def_i) \bigcup Use_i$$
$$LiveOut_i = \bigcup_{j \in Succ(i)} LiveIn_j$$

$LiveIn_i$ and $LiveOut_i$ are sets of registers correspondingly living at the start and at the end of basic block i. Use_i is the set of registers used in basic block i and not redefined after the usage in the basic block. Def_i is the set of registers defined or clobbered in basic block i.

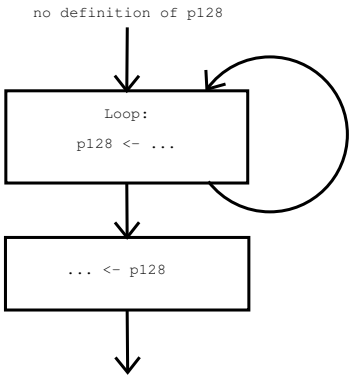

Figure 8: A typical case when accurate life information is different from the standard one.

This information is actually inaccurate because according to it a pseudo-register may live before the first assignment to it. Figure 8 demonstrates such situation. The first assignment to pseudo-register *p128* happens in the loop. According to GCC life analysis, *p128* will live in any basic block where there is a path from the basic block to the loop. Such inaccurate live information results in bigger evaluated register pressure and worse register allocation because *p128* conflicts with all pseudo-registers in the basic blocks preceding the loop.

To make the live information more accurate (RealLive sets) in building conflict graphs we could use the partial availability according to the following equations:

$$PavIn_i = \bigcup_{j \in Pred(i)} PavOut_j$$
$$PavOut_i = (PavIn_i - Kill_i) \bigcup Gen_i$$

$$RealLiveIn_i = LiveIn_i \bigcap PavIn_i$$
$$RealLiveOut_i = LiveOut_i \bigcap PavOut_i$$

$PavIn_i$ and $PavOut_i$ are sets of registers correspondingly partially available at the start and at the end of basic block i. $Kill_i$ is the set of registers killed (clobbered) in basic block

i. Gen_i is the set of registers defined in basic block i and not killed after their definition in the basic block.

It seems that there are few cases where Real-Live and Live sets are different. In reality there are a lot of benchmarks whose code is different when the accurate live information is used. Tables 12 and 13 contains SPEC95 results for tests which have a different code when more accurate information is used.

Benchmarks	Base ratio	Peak ratio	Change
126.gcc	80.8	81.4	+0.74%
130.li	86.4	86.6	+0.23%
132.ijpeg	79.5	80.0	+0.63%
134.perl	86.8	87.9	+1.27%
141.apsi	57.6	58.0	+0.69%
146.wave5	95.6	95.8	+0.21%

Table 12: SPEC95 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the accurate life information.

Benchmarks	Base	Peak	Change
126.gcc	1102160	1101830	-0.030%
130.li	44047	44031	-0.036%
132.ijpeg	120904	120808	-0.079%
134.perl	233331	233315	-0.007%
141.apsi	103221	103205	-0.016%
146.wave5	96668	96668	0.000%
Average			-0.028%

Table 13: SPEC95 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the accurate life information.

Another way to decrease the number of conflicts and as a consequence improve the register allocation is to consider the values of pseudo-registers. Pseudo-registers may get the same hard-registers if they hold the same value in every point where they live simultaneously. Global value numbering [Simpson96] could be used for this. I have tried a simplified version of GVN where all operators except copies are different. I believed that most cases belong to this category. GVN even in such form is still an expensive optimization and a bit complicated because reaching definitions [Muchnick97] have to be used for this (usually GVN is fulfilled in SSA). There are few tests where GVN results in different code (e.g. *eon* and *perlbmk* SPECint2000 tests for x86. Eon had the same performance, perlbmk was about 0.2% faster). So I think the usage of such optimization in GCC is not reasonable.

4.4 Better utilization of profiling information

The original register allocator mainly utilizes profiling information in its work. But there are some instances where it is not true. One such place is the calculation of profitability of usage of caller-saved hard registers for pseudo-registers crossing function calls. Currently it is based on number of the crossed calls and number of the pseudo-register usages. Usage of the frequencies of the crossed calls and the pseudo-register usages instead of the numbers can improve the generated code especially when the execution profile is used. Tables 14 and 15 contain SPECfp2000 results for Pentium 4 when the profile is used.

Benchmarks	Base ratio	Peak ratio	Change
168.wupwise	996	1006	+1.00%
171.swim	921	928	+0.75%
172.mgrid	702	703	+0.14%
173.applu	766	771	+0.65%
177.mesa	734	739	+0.68%
179.art	381	384	+0.78%
183.equake	1217	1226	+0.74%
200.sixtrack	454	456	+0.44%
301.apsi	450	479	+6.44%
SPECfp2000	688	696	+1.16%

Table 14: SPECfp2000 for Pentium 4 GCC with `-O2` without and with caller-saved register profitability based on frequency. The profile information is used.

Benchmarks	Base	Peak	Change
168.wupwise	25384	25320	-0.252%
171.swim	7174	7174	0.000%
172.mgrid	10015	10111	0.959%
173.applu	59405	59509	0.175%
177.mesa	433609	434105	0.114%
183.equake	16386	16418	0.195%
179.art	12123	12235	0.924%
200.sixtrack	835724	838972	0.389%
301.apsi	104573	104837	0.252%
Average			0.31%

Table 15: SPECfp2000 benchmark code sizes for Pentium 4 GCC with `-O2` without and with caller-saved register profitability based on frequency. The profile information is used.

The results could be better even without the profile information. Tables 16 and 17 contain analogous results without the profile for Athlon.

Benchmarks	Base ratio	Peak ratio	Change
168.wupwise	533	551	+3.38%
171.swim	428	441	+3.03%
172.mgrid	404	404	0.0%
173.applu	344	341	-0.87%
177.mesa	623	632	+1.44%
179.art	165	163	-1.21%
183.equake	404	403	-0.25%
200.sixtrack	369	368	-0.27%
301.apsi	282	287	+1.77%
SPECint2000	372	375	+0.81%

Table 16: SPECfp2000 for Athlon GCC with `-O2 -mtune=athlon` without and with caller-saved register profitability based on frequency. Profile information is not used.

4.5 Global common subexpression elimination

As I wrote, the post-reload pass of the original register allocator removes redundant instructions (mostly loads and stores) generated by the reload pass. It uses the CSE (common subexpression elimination) library for this. This

Benchmarks	Base	Peak	Change
168.wupwise	24872	24792	-0.322%
171.swim	7142	7142	0.000%
172.mgrid	9791	9807	0.163%
173.applu	58197	58317	0.206%
177.mesa	456005	458773	0.607%
179.art	13254	13494	1.811%
183.equake	16724	16788	0.383%
200.sixtrack	830268	831468	0.145%
301.apsi	103981	103773	-0.200%
Average			0.31%

Table 17: SPECfp2000 benchmark code sizes for Athlon GCC with `-O2 -mtune=athlon` without and with caller-saved register profitability based on frequency. Profile information is not used.

permits to remove redundancy only in basic blocks.

I was going to implement global redundancy elimination as the next logical step. Fortunately, it was already done independently by Mostafa Hagog from IBM. For PowerPC G5 he reported 1.4% improvement for SPECint2000 (with stunning 15% improvement for perlbmk) and 0.5% degradation for SPECfp2000 (see table 18).

5 Conclusions

As I wrote, the priority-based colouring register allocator can compete with the Chaitin/Briggs register allocators. Therefore I believe we should work on the original register allocator as much as on the new register allocator. It is good to have two register allocators to choose the better one, depending on architecture used.

There are a lot of ways to improve the original register allocator's code. The most interesting one is live range splitting integrated with the register allocator. This is the single important part which is missed in the original GCC

Benchmarks	Base	Peak	The improvement
164.gzip	775	803	3.6%
175.vpr	513	504	-1.8%
181.mcf	500	500	0.0%
186.crafty	868	872	0.5%
197.parser	679	681	0.3%
252.eon	828	819	-1.1%
253.perlbmk	730	844	15.6%
254.gap	811	790	-2.6%
255.vortex	952	964	1.3%
256.bzip2	619	622	0.5%
300.twolf	605	606	0.2%
Est. SPECint	702.2	712.0	1.4%
168.wupwise	895	895	0.0%
171.swim	249	249	0.0%
172.mgrid	643	643	0.0%
173.applu	647	660	2.0%
177.mesa	904	905	0.1%
178.galgel	696	697	0.1%
179.art	624	590	-5.4%
183.equake	996	994	-0.2%
187.facerec	1142	1143	0.1%
188.ammp	398	398	0.0%
189.lucas	530	530	0.0%
191.fma3d	970	969	-0.1%
200.sixtrack	578	562	-2.8%
301.apsi	554	554	0.0%
Est. SPECfp	656	653	-0.5%

Table 18: SPEC2000 results for PowerPC G5 GCC with `-O3` without and with postreload global redundancy elimination.

register allocator from Chow's algorithm. In comparison with the Chaitin/Briggs approach, the priority-based colouring register allocator has an advantage, which is easier implementation of good live range splitting based on register allocation information. It will probably require closer integration of the reload pass and the global register allocator.

6 Acknowledgments

I would like to thank Richard Stallman and Earl Killian for answering my questions about GCC's history and the Pastel compiler.

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 reach experience in register allocation.

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

References

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

[Balakrishnan01] Saisanthosh Balakrishnan and Vinod Ganapathy, *Splitting and Shrinking Live Ranges*, CS 701, Project 4, Fall 2001. The University of Wisconsin. (`http://www.cs.wisc.edu/~saisanth/papers/liverange.pdf`).

[Berson98] D. Berson, R. Gupta, and M. Soffa, *Integrated Instruction Scheduling and Register Allocation Techniques*, Languages and Compilers for Parallel Computing, pages 247-262, 1998.

[Briggs89] Articles of Preston Briggs in compiler newsgroup, Nov. 1989.

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

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

[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.

[GooHsu88] J. R. Goodman and W. C. Hsu, *Code Scheduling and Register Allocation in Large Basic Blocks*, In Proc. of the 2nd International Conference on Supercomputing, pages 442-452, 1988.

[Huang91] Xiaoqiu Huang and Webb Miller, *A Time-Efficient, Linear-Space Local Similarity Algorithm*, Adv. Appl. Math. 12 (1991), 337–357.

[Killian04] Private communications with Earl Killian, March 2004.

[Ladd03] S. R. Ladd, ACOVEA. `http://www.coyotegulch.com`

[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.

[Norris93] C. Norris and L. Pollock, *A Scheduler-Sensitive Global Register Allocator*, Proceedings of Supercomputing, Portland, Oregon, November 1993.

[Open64] `http://open64.sourceforge.net`.

[ORC] `http://ipf-orc.sourceforge.net`.

[Pro64] `http://oss.sgi.com/projects/Pro64`.

[Simpson96] L. T. Simpson, *Value-driven redundancy elimination*, Ph.D. thesis, Computer Science Department, Rice University.

[Sorkin96] A. Sorkin, *Some Comments on 'The Priority-Based Coloring Approach to Register Allocation'*, ACM SIGPLAN Notices, Vol. 31, No. 7, July 1996.

[Stallman04] Private email from Richard Stallman, March 2004.