Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

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

Register Rematerialization In GCC

Mukta Punjani
Cadence Design Systems India Pvt Limited
muktap@cadence.com

Abstract

In most modern processor architectures, difference in data access time for values kept in registers as compared to those in memory is quite high. Thus, compilers should implement efficient register allocation strategies to improve the runtime performance of the code. Register rematerialization is a technique to improve register allocation effectively by improving spill code generation. It is often desirable to compute expressions at a "use" rather than use an earlier spilled value. Normally, "rematerializable" values are derived from registers that are live throughout the function. On register-starved architectures with with addressing modes supporting limited displacement, spilling values, which can be rematerialized, incurs an additional loss in performance due to instructions generated to fetch data from the frame. Hence, rematerialization aids in good usage of registers to give a good gain in execution performance of the code generated. Experimental results indicate a gain of 1-6% in code size and 1-4% improvement in execution performance.

1 Introduction

Let us see the register rematerialization (remat) concept in GCC in more detail. We discuss the proposed improved remat implementation in GCC as it occurs as a part of graph coloring register allocator (in the new regalloc branch). This optimization is supported by target architecture hooks and is currently implemented and tested for SH4 architecture.

1.1 What is remat?

Certain values in a function can be recomputed at any point, as the required source operands will always be available for the computation. Such values are called never killed values. During global register allocation pass, if such never killed values cannot be kept in registers and need to be spilled, the register allocator should recognize when it is cheaper to recompute the value i.e. to rematerialize it [REMAT], rather than to store and reload it from stack. This often happens with frame pointer (FP) relative address computations as well as address computations of large struct or arrays in local scope, where unnecessary spills are seen. A prime example of this can be cited:

1. GCC calculates FP + offset and stores into r3 (say).
2. GCC spills and restores r3 to (from) stack even though it would be cheaper to clobber the register and recompute the value of FP + offset.

Register remat occurs as part of a larger problem of improved spill code generation during global register allocation. The description below co-relates the two facets—Register Allocation Problem and remat as a method of improved spill code generation.

1.2 Register Allocation as Graph coloring Problem

The new register allocator in GCC models register allocation as a graph-coloring problem. It first constructs an interference graph G where nodes in G represent live ranges and edges represent *interferences*. So there is an edge from node i to node j if and only if live range l_i interferes with live range l_j i.e. they are simultaneously live at some point and hence cannot occupy the same register. Live ranges that interfere with l_i are its neighbors in the graph, the degree of l_i is the number of neighbors it has in the graph.

To find an allocation from G, the compiler looks for a k-coloring of G, i.e. an assignment such that neighboring nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring for G into a feasible register assignment for the underlying code. Because finding a k-coloring of an arbitrary graph is NP-complete, the compiler uses a heuristic method to search for a coloring, it is not guaranteed to find a k-coloring for all k-colorable graphs. If a k-coloring is not discovered, some live ranges are spilled, i.e. the values are kept in memory rather than in registers [GCRA].

Spilling one or more live ranges changes both the intermediate code and the interference graphs; hence register allocation. The compiler proceeds by iteratively spilling live ranges and attempting to color the resulting new graph. This process is guaranteed to terminate.

The new register allocator framework takes two approaches while spilling a live range

- A simple *Spill Everywhere* approach involves spilling the entire live range in case it needs. This would involve spilling all the defs of value live range is representing on stack and inserting corresponding reloads before all uses these defs flow into. This spilling technique is fast but not optimal as it would generate lot of spill code.

- An improved but slower *Interference Region Spilling* approach, which involves spilling a live range partly. An interference region for two live ranges can be defined as the portion of the data flow where they are live simultaneously. By spilling interference region for one of the live ranges, they will no longer be live simultaneously, thus will no longer interfere. This effectively removes an edge between the two nodes in the interference graph, making the graph more easily colored. Any spill code addition due to interference region spilling would insert spill code say after a particular use point (only those uses which lie in the interference region). This use point may or may not be the first one in the live range.

1.3 Improving the quality of spill code—Remat Opportunities

In the existing implementation of new register allocator, remat is performed only for those values whose definition consists of moving a immediate value to a pseudo. The proposed approach can enhance the scope of remat by taking into account more potential remat candidates and hence improvement in spill code generation.

The opportunities identified for remat can be

- Immediate loads of integer/float constants
- Loads from literal pools
- Computing a constant offset from the stack pointer

- Computing a constant offset from the static data area pointer
- Any Branch Related target Labels
- Address computations for access to non-local names in case of nested functions
- Address computations from pointer to global offset table in case of position independent code
- Address computations resulting from parameter registers (e.g. r4...r7 in case of SH4) or parameter register copied to callee save registers which dont have defs elsewhere in the function
- Address computations resulting from return register (e.g. r0 in case of SH4) or return register copied to other callee save registers which don't have defs elsewhere in the function

2 Remat Strategy

The new register allocator is based on the analysis of definitions and uses of all the pseudos in the instructions stream to form webs (nodes of the interference graph), which are the basic entity for allocation to a register. Hence, each web corresponds to the live range of a variable, which can be allocated a distinct symbolic register number. The interference graph for webs consists of a number of such intersecting webs, the intersection between any two webs occurs when they have a use in common. If a web can't be assigned a register then a decision is made to spill it.

The proposed improved remat optimization consists of identifying remat defs during dataflow analysis and propagating this information to the web spilling phase. The spilling phase can choose between spilling or rematerialization of a web based on relative cost analysis. Hence the remat strategy can be segregated into the following sub-problems

- Gather and propagate remat/live-range information
- Criterion for spilling/remat decision
- Performing remat

2.1 Remat Information

To identify rematerializable candidates, remat information needs to be built during the dataflow analysis. All the defs can be analyzed to see whether they come from any of the remat sources. Such defs can be tagged with their corresponding remat efinitions. Any remat definition resulting from an operation on two or more rematerializable definitions (say a def consisting of adding a constant value to the label) can be tagged likewise.

2.2 Remat Criterion

The obvious criterion for remat/spilling decision would be to compare the relative cost of both the decisions in terms of aggregate cost of instructions each would generate, and choose the one with lower cost. This criterion is described in Section 2.3. But due to some issues mentioned further, it might not be possible to calculate the exact spill costs. Another method to choose remat in the absence of spill cost criterion has been described in Section 2.4.

2.3 Spill v/s Remat Cost

Whenever a decision is taken to mark a web for spilling, check if the definition in the web is rematerializable. Calculation of the remat and spill cost will be implemented in a target dependent hook. Remat cost will be calculated in terms of the aggregate of all the insns' costs,

which would be required to recompute a remat definition before every use of such definition. Similarly spill cost can be computed as instruction cost aggregate of all the insn's cost generated to spill (and restore) the value. Choose the one with lower cost. However in the existing new register allocator, it may be difficult to calculate the exact accurate remat (spill) costs of any value because

- During web spilling, actual stack offsets of spill locations are not defined, instead only stack pseudos are assigned
- The `reload_cse` pass may use address inheritance information and may merge some instructions for loading the offset, by reusing any value close to or equal to the offset loaded in some other pseudo. Hence cost decisions may get invalidated later

e.g. In case of SH4, the number of instructions to spill (restore) can be a minimum of 1, if spilling takes place at an offset less than 64 bytes relative to a base register. In case the spill offset is more than 64 bytes, spilling (restoring) may take two or more instructions, one instruction for loading the stack offset to spill (restore) and another consisting of storing (restoring) the value. In order to calculate spill (restore) offset (and the number of instructions required for spill), stack slot information to which the pseudo is likely to be spilled to, needs to be built and tracked for all such pseudos, depending on which the number of instructions needed to spill (restore) can be calculated.

The `reload_cse` pass may further merge instructions for loading the offset, by reusing any value close to or equal to the offset loaded in some other pseudo. In this case, even if the spill location is at an offset greater than 64 bytes, it may require 1 instruction.

The first problem related to spill cost computation can be partially resolved by tracking the size of frame data and number of pseudos being spilled so as to have an estimate of spill offsets. This can predict the number of instructions that would actually be required to spill (restore). However in the existing framework of new register allocator, spill cost cannot be computed correctly in some cases due to `reload_cse` issue mentioned.

2.4 Remat Without Spill Cost Calculation

In the absence of a definite cost available for spilling, spill cases can be segregated according to the reason for their occurrence and also cases, which will definitely be cheaper to rematerialize than to spill

Definitive Remat Some defs can be identified as remat cases based on the fact that they require 1/2 insns to compute and re-computing them will always be less than or equal to minimum cost of spilling a def for the given target. Examples for such definitive remat would include

- Constant loads
- Label Loads

Defs having up to 1 insn in their remat insn chain (spill (restore) together would require a minimum of two insns).

For defs having two or more insns in their recomputation sequence, insn merging can be attempted on that sequence based on a target dependent hook. If such sequence merging generates a valid single insn for the target, then it fits as a candidate for definitive remat.

For remat in such cases, all the insns leading to definition of rematerializable value being spilled can be moved immediately before its use.

Moving Defs Near Uses This approach may be used for remat cases not covered in previous section. Spilling generally happens as a result of spacing between the actual def and use of values in case of high register pressure. This can happen in case of complex computations involved in some program statements, where LHS computations and some other intermediate computations have to be spilled. Such cases of spilling require generation of 2–4 spill and restore instructions (e.g. 64-byte stack offset criterion in SH4). In case the computation being spilled is rematerializable, spilling becomes unnecessary. The remat cost criterion need not be taken into account here because spilling is definitely not required.

In both the approaches for spilling described above, unnecessary spill cases can be identified as those having their spill point just after def (before 1st use).

The following assembly illustrates this case:

```
chanserv.i/load_cs_dbase (in
stress 1.17) compiled with -O2
-ml -m4 -fnew -ra
-fno-argument-alias
-fno-schedule -insns
-fno-schedule-insns2 -g -S
-fpic generates

.L1404:
  .loc 1 2844 0
  mov.l   .L1036,r0   <-- 1
  mov.l   @(r0,r12),r0<-- 2
  mov.w   @r0,r1      <-- 3 (remat
                    insn chain 1,2)
  mov.w   r1,@r14     <-- Spill
                 before 1st use
.L612:
  .loc 1 2845 0
  mov.w   .L1037,r0
  mov.w   @r14,r1     <-- Reload
  mov.w   r1,@(r0,r11)
  .loc 1 2848 0
  mov     r14,r4
  mov.l   .L1038,r1
  bsrf    r1
```

For remat in such cases, the all the insns leading to definition of rematerializable value being spilled can be moved immediately before its use.

There are certain issues regarding the movement of insns in this case. The placement of a def before use requires addition of insns before the use point which leads to increased register usage there. Hence, a good heuristic needs to be devised to make sure that such insn insertion keeps the register pressure in check and does not actually end up increasing it.

Interference Region Spills As discussed earlier, in case of interference region spilling, spill point may or may not be before the first use point. Hence in such a case for spilling, we have to choose between spill/rematerialize (in the absence of cost of spilling). This case of spilling generally occurs in spilling calculated offsets for arrays/structs, and may not require a lesser remat cost in most of the cases. Here rematerialization decision is not taken, as spilling might actually be cheaper. Nevertheless, the decision may not be correct for all the cases.

2.5 Performing Remat

Remat of a value involves inserting recomputation sequence for a definition before use points by moving the insns forming the definition before use points.

3 Implementation in GCC

The patch at the link

`http://gcc.gnu.org/ml/gcc-patches/2003-12/msg01985.html`

is the implementation of the ideas presented in the paper in GCC. This implementation scope and its constraints are discussed as follows.

3.1 Current Implementation Approach

Data Flow Phase The improved remat handler detemines the defs, which have been generated from never-killed sources and creates a remat pattern sequence to recompute those defs. This code is implemented along with the data flow routines in df.c

```
Eg. r159 <- const1      (1)
r160 <- r14 +  r159     (2)
r161 <- mem (r160)      (3)
```

Here defs for r159, r160, r161 are all remat, as all have been derived from never-killed sources. The data flow routines determine all such defs and store the respective patterns that would be required for recomputing each potential remat definition.

So in the above example,
def(r159)->remat_sequence = (1)
def(r160)->remat_sequence = (1), (2)
and def(r161)->remat_sequence = (1), (2), (3)

Web Construction Phase The cost of webs in `ra_build.c` is modified to accommodate the cost of those webs which have rematerializable defs. The cost for all the defs is added up whether for remat defs or non-remat ones.

Web Colorize Phase The webs, which have potentially, remat defs and the cost consequently is lower than other defs are promoted for spilling. This advocates their spilling, in turn relieving the conflict edges. Remat handler later picks up such webs and appropriate processing is done there. This happens in `ra_colorize.c`

Web Spilling Phase In this file i.e. `ra_rewrite.c`, the allocator is let to spill as it was doing earlier. After the first level spilling is done, the spilled webs whose defs can be remat are picked up and the remat patterns for those defs can replace the restore insns for the defs spilled. At the point of inserting computations, it needs to be ensured that the regs that form the remat sequence

- Do not increase the register pressure at that point.
- Live range of those regs is not exceeded.

The remat handler only concentrates on the first pass of the allocator and spills generated for the first time are handled only. Later rounds do not call the remat sequence building routines.

Problems Encountered With Full Scope

- In the absence of a good register pressure estimation heuristic, insertion of defs with multiple insns in the remat sequence poses problems. Also, the register allocator has strong asssumpions about the web structure. Hence after inserting recomputation patterns of length > 1 in place of the restore insn, the allocator got stuck up in a lot of in tight consistency checks of `ra_build.c` especially in `parts_to_webs_1`.
- Due to the same problem, function pointer and return register are not being considered for remat.

Current Implementation Scope Due to the problems cited above, the current implementation implements the following remat handling:

- The data flow phase constructs remat sequences in full and then tries to collapse

them (if allowed by the target) in `df_remat_validate`. Rest of the chains are discarded at this point.

- The build phase involving modifying costs for remat webs functions as before.
- The colorize phase functions as before.
- The rewrite pass replaces the source of restore insns before the spilled uses of the webs and replaces it with the remat source (a single pattern).

The restricted implementation is scalable enough to support the original concept of remat envisaged and should stand any changes in the register allocator passes.

4 Performance Data

The performance improvement due to register rematerialization depends on the following factors.

1. Register Pressure and consequently number of values spilled.
2. The values spilled from rematerializable sources and found to be obeying the constraints for performing remat

During performance analyis, select benchmarks were compiled using GCC-2.3 20021119 (new-regalloc branch) for SH4 target using options `-O2 -ml -m4 -static`. A new option namely `-fimproved-remat` is introduced to enable improved remat. The benchmarks were executed on SH4 evaluation boards with QNX 6.1. It is observed that best performance improvement for execution performance is 4% and that for code size is 6.46%. Table 1 gives code size comparisons of stress1.17 files with and without improved remat. The execution results for some benchmarks are shown in Table 2.

File Name	size with new-ra	size with improved-remat	decrease (%)
L3bitstream.o	7424	6944	6.46
aiunit.o	18880	17760	5.93
scanline.o	2400	2272	5.30
advdomestic.o	8280	7960	3.86
tif_fax3.o	10208	10176	3.13
tif_packbits.o	1316	1284	2.43
layer3.o	20752	20272	2.31
melee2.o	27516	26940	2.09
navion_aero.o	1600	1568	2.00
chanserv.o	63296	62112	1.87
s_serv.o	29740	29196	1.82
im_decode.o	290000	285936	1.40
quantize.o	10012	9948	0.64
wizard1.o	24064	23964	0.41
blowfish.o	8808	8776	0.36
map_fog.o	26272	28880	-9.6

Table 1: Code Size Comparisons

Benchmark	Input Data Size	Gain (%)
GZIP Compression	80.5 MB	4
Mpg 123	-	4
GZIP Decompression	16.2 MB	2.6
GSM Compression	1.71 MB	0.05
GSM Decompression	361 KB	0

Table 2: Execution Timings

5 Further Improvements In New RA

5.1 Loop Variable Spilling

Ideally, register allocation should take into account, the variables present inside the loops and in case of high register pressure, try to assign registers to frequently accessed variables in a loop (for example loop indexes) on a priority basis. But such an allocation scheme is not being observed in some cases. The following example illustrates this fact

`chanserv.i/check_modes compiled`

```
with -O2 -ml -m4 -fnew-ra -fno-
argument-alias -fno-schedule-insns
-fno-schedule-insns2 -g -S:

  .loc 1 4743 0
  mov.l   .L2970,r7
  mov.b   @r7,r1
  cmp/pl  r1
  bf/s    .L2942
  and     r0,r3
  mov     #0,r1       <-- (1)
  mov     #64,r0      <-- (2)
  mov.l   r1,@(r0,r14)<-- (3)
  mov     #0,r2
  .loc 1 4745 0
  mov     #64,r0
.L3043:
  mov.l   @(r0,r14),r6   <-- (4)
  add     r7,r6          <-- (5)
  mov.l   r6,@(r0,r14)   <-- (6)
  mov     r6,r0
  mov.l   @(4,r6),r6
  tst     r6,r3
  bt      .L2922
  mov.b   @r0,r1
  .loc 1 4747 0
  mov.b   r1,@r12
  add     #1,r12
  .loc 1 4748 0
  not     r6,r6
  mov.l   @(56,r10),r1
  and     r6,r1
  mov.l   r1,@(56,r10)
  .loc 1 4743 0
.L2922:
  add     #8,r2         <-- (7)
  mov     #64,r0        <-- (8)
  mov.l   r2,@(r0,r14)  <-- (9)
  mov     r2,r0
  mov.b   @(r0,r7),r1
  cmp/pl  r1
  bt/s    .L3043
  mov     #64,r0
```

In the above example, the calculation corresponding to register r1 in (1) is being spilled within a loop instructions (4) through (9). Such an example clearly illustrates inefficient register allocation.

5.2 Loop Invariant Code Spilling

In case of any loop, invariant part of the code is moved outside the loop. In some cases such address computations might be spilled onto a stack locations outside the loop. Inside the loop these values are reloaded from stack. Such cases are NOT direct candidates of remat (as generally remat cost will be higher than the spill cost). However in case the computation requires single instruction within the loop (e.g. loads within 64 byte window to a base register i.e. r14, r12, r11 etc.) then it should well be computed inside the loop instead of being moved out as invariant code. This would result in saving one instruction per loop iteration.

However, changes required in this case would involve GCC passes, which move loop invariant code.

6 Acknowledgments

I would like to thank the GCC community, especially Michael Matz, Denis Chertykov, and Vladimir Makarov, for their ideas. I would also like to thank Sandeep Khurana and Naveen Sharma from HCL Technologies and Toshiyasu Morita from Renesas for their valuable ideas and suggestions.

References

[SRC] Mainline Sources of GNU GCC 3.3

[GCC] GCC Internals Manual. `http://gcc.gnu.org`

[GCRA] Briggs, P., Cooper, K. D., and Torczon, L. 1994. *Improvements to Graph Coloring Register Allocation*

[REMAT] Preston Briggs, Keith D. Cooper, Linda Torczon *Rematerialization*