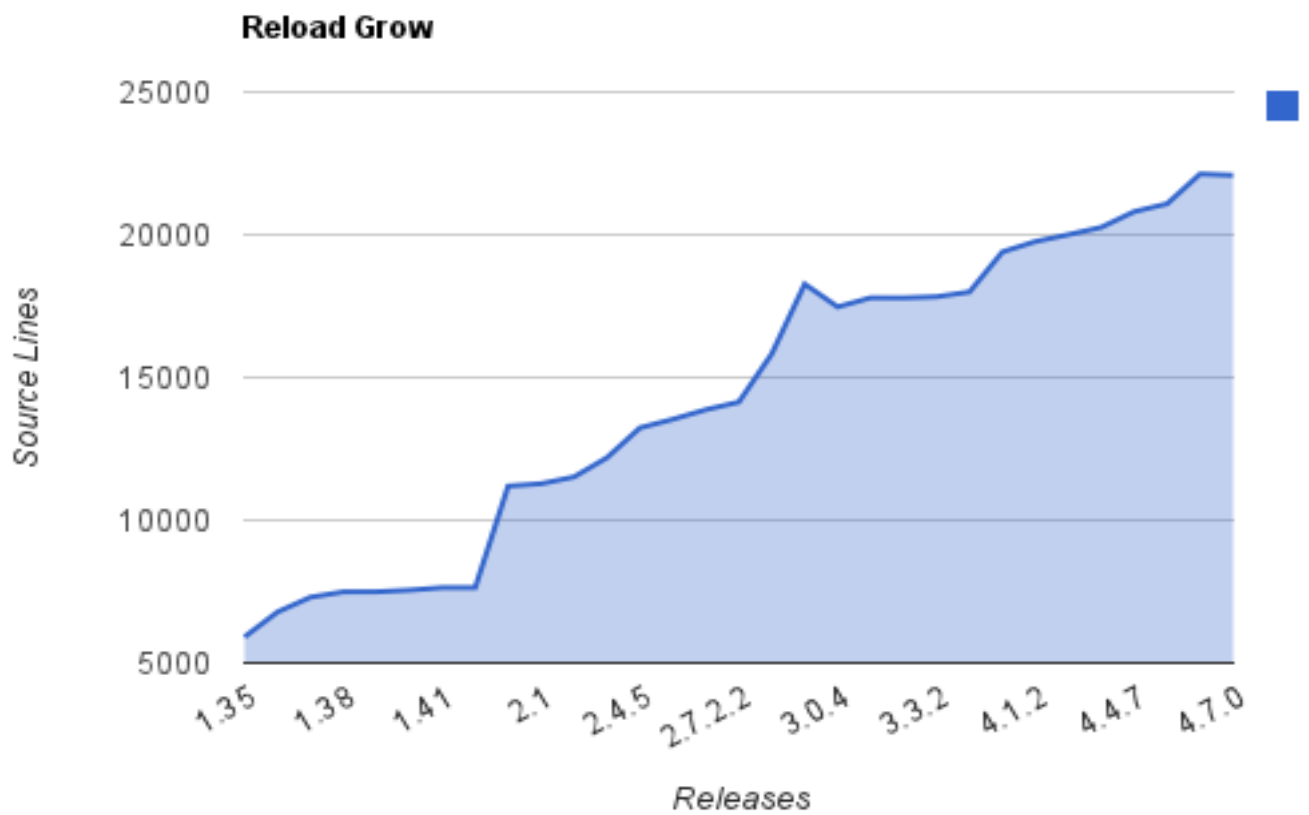
Local Register Allocator Project

Vladimir Makarov
vmakarov@redhat.com
July 2012

Reload and Local Register Allocator (LRA)

- LRA is yet another approach to reload replacement.
- Reload is most machine-dependent and essential part of GCC RA.
- Reload exists since GCC day 1 (> 25 years).
- Reload became a bloated, buggy, and very complicated GCC part.



LRA -- Motivation

- The project motivation:
  - Simplification for easier maintance and bug fixing.
    - As reload is a complicated part of compiler, it is hard to maintain (e.g. 9 year old bug has been recently fixed). Some bugs probably will never be fixed:

      "can not find a register in class..."

  - New optimizations, e.g. spilling not one by one case, live-range splitting, inheritance beyond BB, assigning to improve inheritance, spilling into vector registers, bitwidth-wise RA etc., needs a better foundation. Their implementation in reload would make reload even more buggy.

LRA – reload approach

- What is wrong with reload approach which makes it complicated? My point of view:
  - Trying to do everything at once.
  - A lot of additional structures on RTL sides.
  - Using a lot of hooks, although MD insn constraints should be primary source of the info.

LRA – proposed approach

- To solve reload approach disadvantages:
  - Division on small manageable, separated sub-tasks.
  - All transformations and decisions are reflected in RTL as more as possible.
  - Insn constraints as a primary source of the info (minimizing number of target-depended macros/hooks).

LRA – code transformation

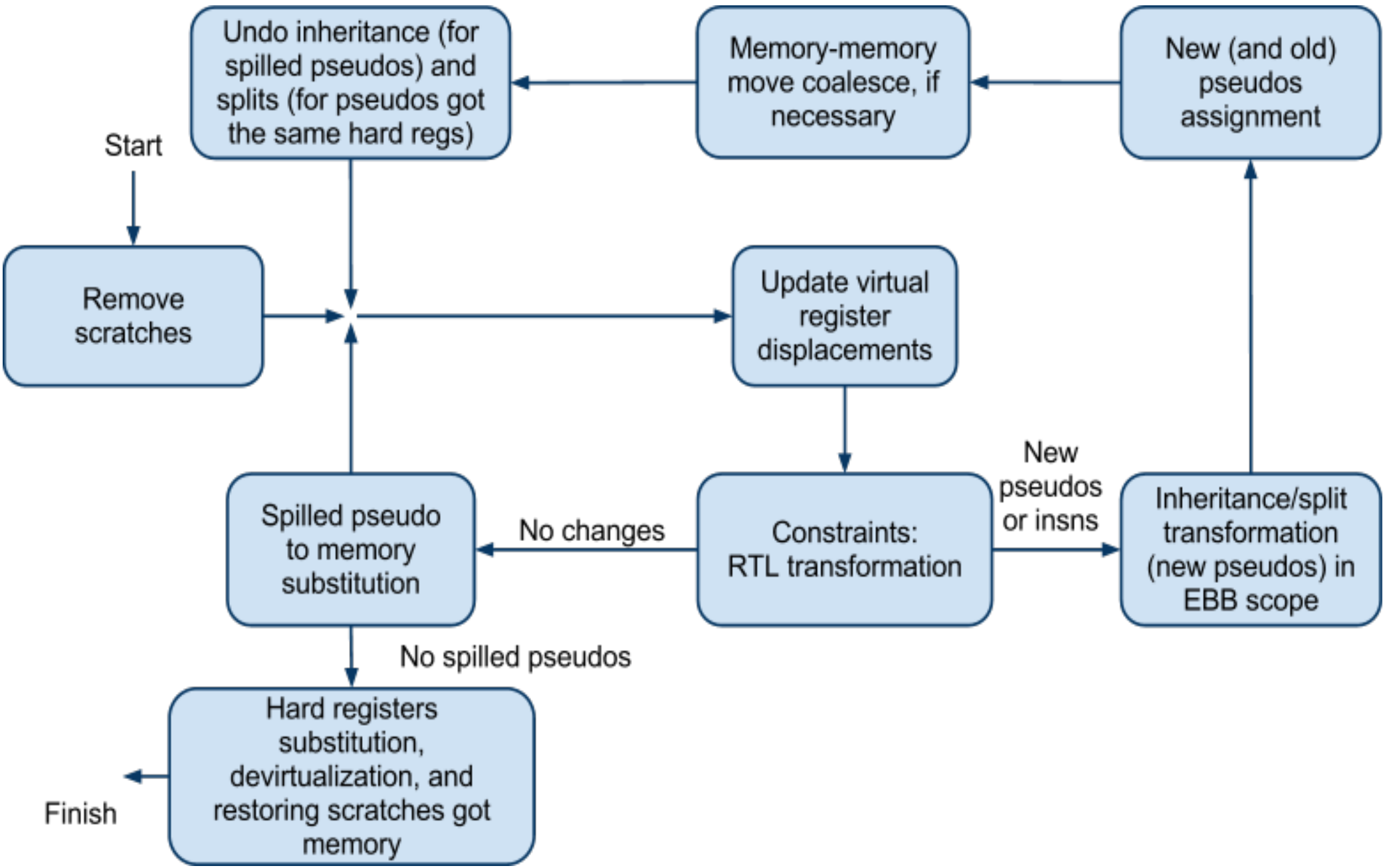
- Major LRA task is to transform code to satisfy insn constraints.
- Split-one pass approach.
  - Morgan's Book, Andrew MacLeod's RA in IBM compiler:
    - One pass through BB insns following GRA allocation, splitting pseudo live-ranges and spilling them (e.g. using latest reference heuristics).
- Spill iterative passes.
  - Classical approach with Chaitin-Briggs colouring (current reload pass approach also).
    - Iteratively pass through BB insns: spill a pseudo when needed a register until all constraints are satisfied and there is no spilling any more (danger of cycling).
- I like the both. Therefore I use iterative passes with spills and splits because IMHO
  - Iterative approach is simpler and better fit for multi-target compiler.
  - Spills tends to generate smaller code.
  - Splitting can be profitable and simplify caller-saves implementation

LRA -- brief overview

- Iterative insn process with the final goal is to satisfy all insn and address constraints:
  - New reload insns and reload pseudos might be generated.
  - Some pseudos might be spilled to assign hard registers to new reload pseudos.
  - Some pseudos are **bound**. It means they always got the same memory or hard register.
  - Changing spilled pseudos to stack memory or their equivalences.
  - Allocation stack memory changes the address displacement and new iteration is needed.

- To speed up the process:
  - We process only insns affected by changes on previous iterations.
  - We use a special insn representation for quick access to insn info which is always synchronized with the current RTL.
    - We don't use DFA-infrastructure because it results in about 10% overall compiler speed degradation.
    - We use own specialized internal insn representation which is divided by static info (from machine description) and minimized dynamic info (from RTL). Static info is kept in one exemplar for all insn with the same insn code.

LRA - passes/sub-tasks



LRA passes: Scratches

- RTL code can contain scratches which should be changed on hard registers for some insn alternatives. An x86 example:

```
(define_insn "extendsidi2_1"
[(set (match_operand:DI 0 "nonimmediate_operand" "=*A,r,?r,?*o")
      (sign_extend:DI (match_operand:SI 1 "register_operand" "0,0,r,r")))
 (clobber (reg:CC FLAGS_REG))
 (clobber (match_scratch:SI 2 "=X,X,X,&r"))]
```

- We change scratches into pseudos at the beginning of LRA to simplify dealing with them (conflicts, hard register assignments).
- If the pseudo denoting scratch was spilled it means that we do need a hard register for it. Such pseudos are transformed back to scratches at the end of LRA.

LRA passes: Updating virtual register displacements

- Virtual registers (like argument and frame pointer) are widely used in RTL.
- Virtual registers should be changed by real hard registers (like stack pointer or hard frame pointer) plus some offset.
- The offsets are changed usually every time when stack is expanded.
- We know the final offsets only at the very end of LRA.
- We update virtual registers to the same virtual registers + corresponding offsets before every LRA constraint pass because it affects constraint satisfaction (e.g. an address displacement became too big for some target).
- The final change of virtual registers to the corresponding hard registers are done at the very end of LRA when there were no change in offsets anymore:

```
fp + 12      =>      sp + 12
```

- Such approach requires a few changes in GCC code because virtual registers are not recognized as real ones in some constraints and predicates.

LRA passes: Constraints

- The major pass goal is to transform RTL to satisfy insn and address constraints by:
  - choosing insn alternatives;
  - generating reload insns (or ***reloads*** in brief) and reload pseudos which will got necessary hard registers later;
  - substituting pseudo equivalences (if it is done once, is done everywhere) and removes insns initializing used equivalent substitution.
- This biggest and most complicated code in LRA
  - A lot of important details:
    - Reuse of input reload pseudos to simplify reload pseudo allocations.
    - ***Bound*** reload pseudos (when different modes are needed).
    - Some heuristics to choose insn alternative to improve the inheritance.
    - Early clobbers etc.
  - It is mimicking former reload pass in alternative choosing because the reload pass is oriented to current machine description model. It might be changed if the machine description model is changed.
  - Special code for preventing all LRA and this pass cycling.
  - To speed LRA up: processing only necessary insns (first time all insns) and reusage of already chosen alternatives in some cases.

LRA passes: Pseudo assignments

- There is no any RTL code transformation on this pass.
- ***Reload*** pseudos get what they need (usually) hard registers in ***anyway*** possibly by spilling non-reload pseudos and by assignment reload pseudos with smallest number of available hard registers first.
- If reload pseudos can get hard registers only through spilling other pseudos, we choose what pseudos to spill taking into account how given reload pseudo benefits *and* also how other reload pseudos not assigned yet benefit too.
- ***Non-reload*** pseudos can get hard registers too if it is possible and improves the code. It might be possible because of spilling no-reload pseudos on given***.***
- ***Bound*** pseudos always get the same hard register if any.
- If two hard registers are equally good for assigning the pseudo, we prefer a hard register in smaller ***register bank.***
  - By default, there is only one register bank. A target can define register banks by a hook.
  - x86-64 has a few register banks, e.g. for hard regs with and without REX.
- Only insns with changed allocation pseudos are processed on the next constraint pass.
- Pseudo live-ranges is used to find conflicting pseudos:

LRA passes: Memory-memory move coalescing

- Spilling pseudos in LRA can create memory-memory moves.
- We should remove potential memory-memory moves before the next constraint pass because the constraint pass will generate additional insns for such moves and all these insns will be hard to remove afterwards.
- We coalesce only spilled pseudos. Coalescing non-spilled pseudos (with different hard regs) might result in spilling additional pseudos because of possible conflicts with other non-spilled pseudos and, as a consequence, in more constraint passes and even LRA ***infinite cycling***. Trivial the same hard register moves will be removed by subsequent compiler passes.
- We don't coalesce ***bound*** pseudos. It complicates LRA code a lot without visible generated code improvement.
- The pseudo live-ranges are used to find conflicting pseudos during coalescing.
- Most frequently executed moves is tried to be coalesced first.

LRA passes: Inheritance

- The inheritance optimization goal is to reuse values in hard registers. There is analogous optimization in reload.
- There are two passes before and after the assignment pass.
- The first pass does transformations:

`reload_p1 <- p`
`...`
`...`
`reload_p2 <- p`
	`reload_p1 <- p`
	`new_p <- reload_p1`
	`...`
	`reload_p2 <-new_p`

where ***p*** is **spilled** and not changed between the insns.

- The subsequent assignment pass will try to assign the same (or another if it is not possible) hard register to ***new_p*** as to ***reload_p1*** or ***reload_p2***.
- If it fails to assign a hard register, the opposite transformation will restore the original code on the second pass (called undoing inheritance) because with spilled ***new_p*** the code would be much worse.
- The inheritance is done in EBB scope.
- This is just a simplified description to get an idea of the inheritance. Inheritance is also done for **non-reload** insns.

LRA passes: Pseudo/hard reg splitting

- Splitting is done in EBB scope on the same pass as the inheritance:

r <-... or ...<- r ... ...<- r	r <-... or ...<- r s <- r (new insn - save) ... r <- s (new insn - restore) ...<- r

- The split pseudo ***s*** is assigned to the hard register of the original pseudo ***p***.
- Splitting is done:
  - for *global* pseudos and for hard registers when there are more one reloads needing the hard registers
  - for pseudos needing save/restore code around calls
- If the split pseudo still has the same hard register as the original pseudo after the assignment pass, the opposite transformation is done on the same pass for undoing inheritance.

LRA passes: Spills

- Create necessary stack slots and assign spilled pseudos to the stack slots:

```
for all spilled pseudos P most frequently used first  do
  for all stack slots S  do
     if P doesn't conflict with pseudos assigned to S  then
       assign S to P and goto to the next pseudo process
  create new stack slot S and assign P to S
```

  - The algorithm is bit more complicated because different pseudo sizes.
- Change spilled pseudos (except ones created from scratches) by corresponding stack slot memory in RTL.
- If a stack slot was created,we need to run *more passes* because address displacements might change and address constraints (*or* insn memory constraints) might be not satisfied any more.

LRA IR

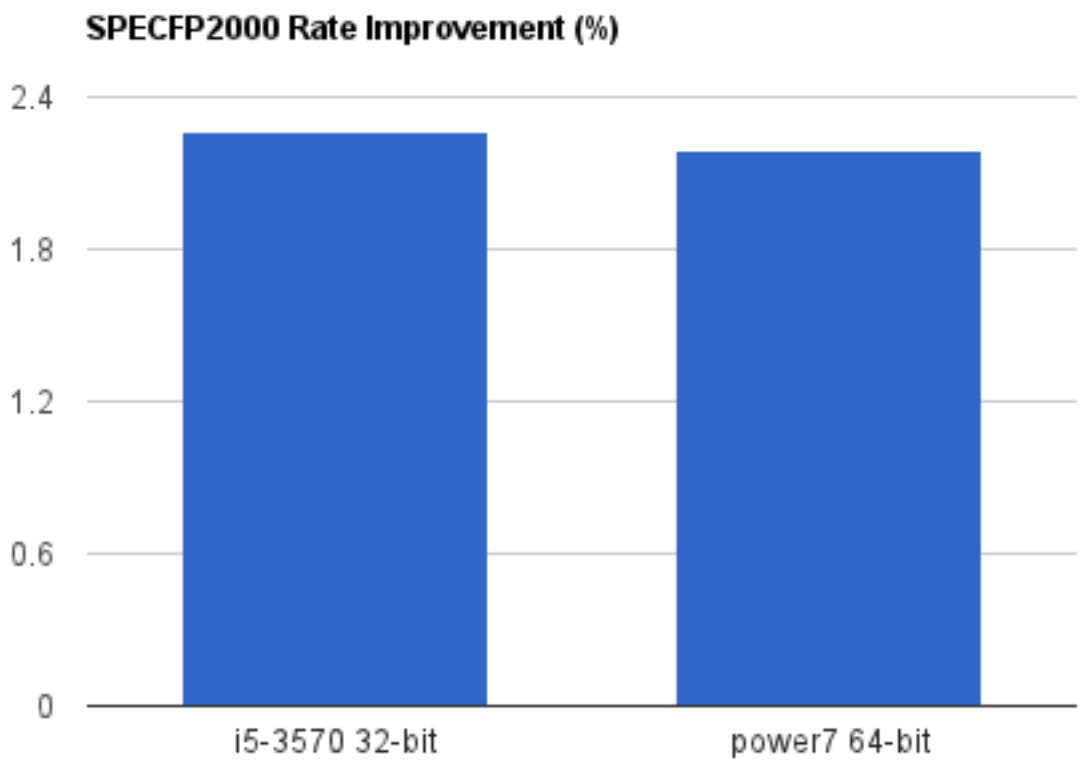
- Major LRA IR is RTL: all transformations are done on RTL.
- Very frequent accessing to RTL (e.g. insn operands through `insn_extract`) is very slow.
- Using DF-infrastructure speeds LRA up but still makes whole compiler 10% slower.
- LRA uses specialized IR which is always synchronized with RTL:
  - Insn IR is minimized by memory. It is divided on two parts:
    - one specific for each insn in RTL (only operand locations)
    - one common for all insns in RTL with the same insn code (different operand attributes from machine descriptions)
    - one oriented for maintenance of live info (list of pseudos)
  - Pseudo data:
    - all insns where the pseudo is referenced
    - live info (conflicting hard regs, live ranges, # of references etc)
    - data used for assigning (preferred hard regs, costs etc)
- This specialized IR permitted LRA to achieve the same speed as reload or even better.

LRA - the current state

- There is already a good prototype.
  - The code is on the branch called **`lra`**.
  - Successful bootstrap for x86/x86_64, PowerPC, SPARC64, S390, ARM, IA-64, MIPS64, and PA-RISC.
  - LRA source code is 70% of reload one. Many reload specific macros are not used. There are a big potential to remove a lot of unnecessary machine-dependent code.

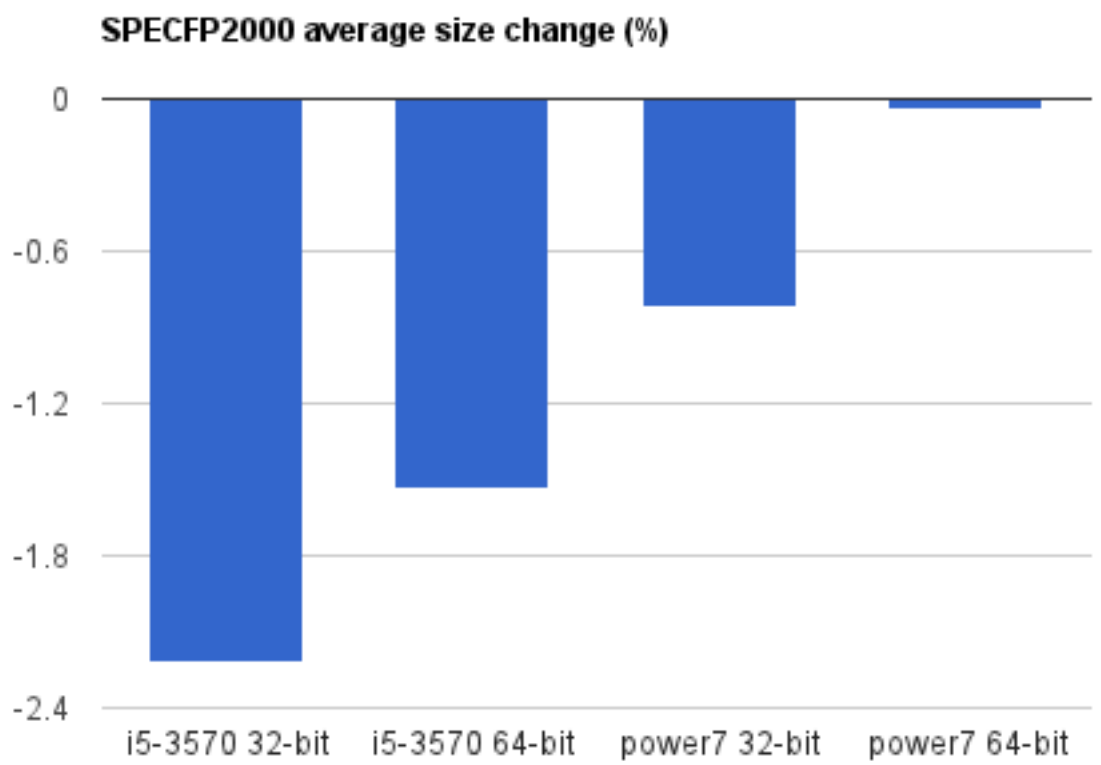
LRA - performance change

- -O3 -march=corei7 -mtune=corei7 -mfpmath=sse for x86
- -O3 -mtune=power7 for Power



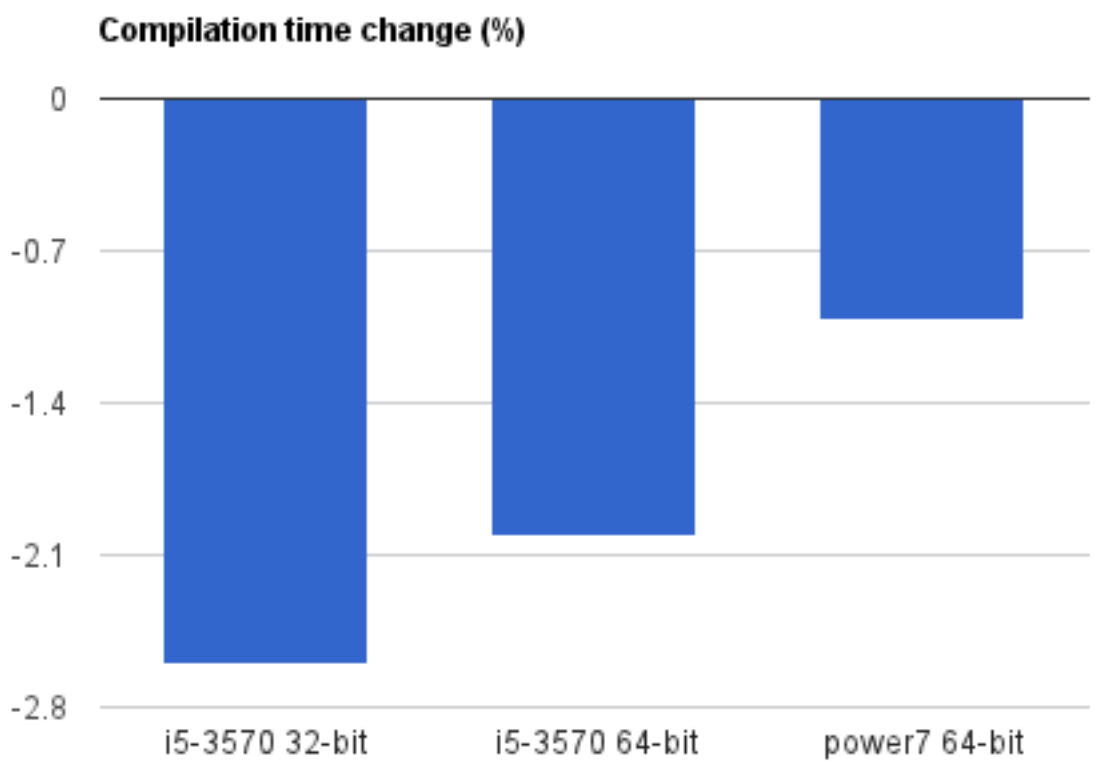
LRA - average code size change

- -O3 -march=corei7 -mtune=corei7 -mfpmath=sse for x86/x86-64
- -O3 -mtune=power7 for Power



LRA - compilation time change

- Test is all_cp2k_gfortran.f90 (420K source lines)
  - -O3 -march=corei7 -mtune=corei7 -mfpmath=sse for x86(-64)
  - -O3 -mtune=power7 for Power



LRA - Todo list

- The merge target is gcc4.9 because
- There are still a lot of thing todo:
  - Fixing gcc testsuite regressions for all targets except for x86/x86-64.
  - More platforms to bootstrap and test.
  - Improving performance.
  - LRA speedup (it is hard because reload uses structures on the side for finding ***final*** RTL transformations instead of explicit RTL transformations on each step).
  - Cleaning a lot of machine-dependent code which implements a lot of reload macros not necessary anymore.

LRA - how to merge?

- The merge should be done at the beginning of stage 1.
- Reload exists with LRA.
  - LRA is default for ported targets which were bootstrapped with LRA and have no more GCC testsuite regressions.
  - Making LRA by default for more targets
  - When all targets are switched to LRA, remove Reload (may be in a release after gcc4.9).
- *or* Reload is removed
  - Long time when many targets will be broken.
- I prefer the 1st approach as more realistic one.
- I am interesting in your opinions.

LRA - New optimizations

- Spilling in global scope, inheritance in EBB, and pseudo and hard register live-range splitting all absent in reload were already implemented in LRA.
- New processor features:
  - a lot of vector registers (x86/x86-64 SSE regs, IBM VSX, Arm neon).
  - a very wide registers (128-256 bits wide vector registers).
- How to utilize the new features:
  - **Spilling into vector registers instead of memory**. Intel optimization guide recommends it. Unfortunately, AMD inter-unit movements are slow. It is ***done***.
  - Holding several pseudo values in one wide register. It is so called **bitwidth-wise RA**. The access overhead can make it unprofitable (Intel processors). Still there are processors where it could be profitable (AMD ones).
- Hard to implement them on the base or reload which is already too much complicated.