ISP RAS
Institute for System Programming
of the Russian Academy of Sciences

Improving a selective scheduling approach for GCC: performance and compile time issues

Andrey Belevantsev <abel@ispras.ru>

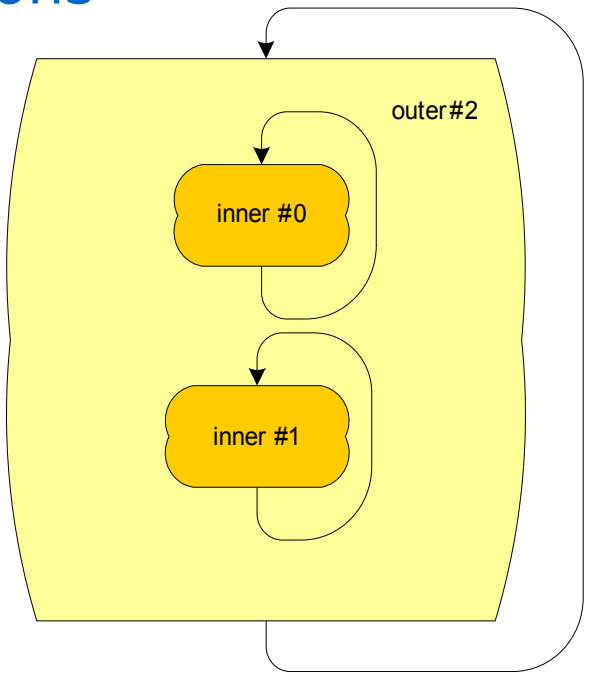
GREPS Workshop
September 16th, 2007

Selective scheduling approach

- Provides a scheduling framework
  - Supports scheduling along all paths in a DAG
    - by handling multiple scheduling points (fences)
  - Supports a number of instruction transformations
    - *local* – speculation/substitution, they happen when one insn is being moved through another
    - *global* – instruction cloning/register renaming, these require the knowledge of code motion paths
- Provides software pipelining implementation
  - supports control-flow intensive and non-countable loops
  - can pipeline loop nests starting from the innermost loop to the outermost

High-level view of the scheduler

- Initialize global data – alias analysis, df, ...
- Form scheduling regions
  - Find acyclic regions of control flow which are no bigger than 10 blocks and/or 100 instructions
  - For pipelining: find all loop nests, form loop regions starting from innermost loops, form acyclic regions from the rest of blocks
  - Pipelining will be enabled for any loop region which is not too large
- Schedule every region
- Finalize the data



Scheduling the region

- Gather available instructions/RHSes to each available fence
  - Local transformations are done on the way
  - Intermediate av sets are saved at each basic block
- Choose the best instruction from available ones
  - By calling DFA lookahead routines and target hooks
  - Check that we do not cross any live ranges with a given code motion
  - Choose the destination register if renaming
- Fixup the program for the selected code motion
  - Traverse code motion paths and insert bookkeeping at join points of control flow
  - Update saved av sets and liveness info
- When no insns are ready, advance the fences

Subtleties of the approach

- Distinguishing between instruction types
- Choosing the best available instruction
- Improving profitability of pipelining
- Handling “ambiguous” transformations
- Avoiding changes in control flow

- Speeding up dependence analysis
- Speeding up the search for original insns
- Limiting register renaming
- Optimizing memory allocation scheme

Handling different instruction types

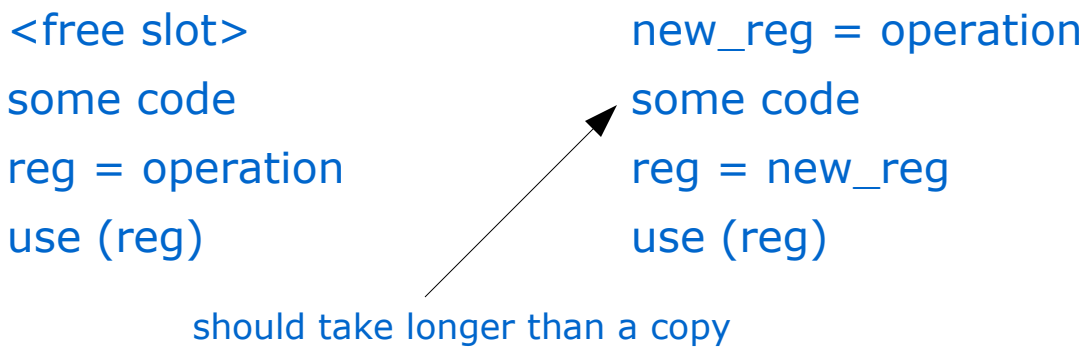
- It is assumed that bookkeeping/renaming are applicable to *every* instruction
  - But: calls and asms cannot be copied
  - But: insns with side-effects in RHS cannot be renamed
- Clonable/unique instructions:
  - Unique insns can be only moved up to the dominator block
  - AND when they are available along ALL paths to it
- Separable/non-separable instructions:
  - Only the former can be renamed
  - Dependence analysis should be able to determine insn part (LHS/RHS) from which the dependence comes

Choosing the best available instruction

- General problem: there is no easy way to calculate any data for global transformations
  - How many/where bookkeeping copies will be created?
  - Are they expensive or not?
  - Will the given renaming be profitable?
- The number of bookkeeping copies can be precomputed when looking for a new register
- It is hard to find out the cost of bookkeeping
  - This is because we know the simulated processor state only for current fences

Will the given renaming be profitable?

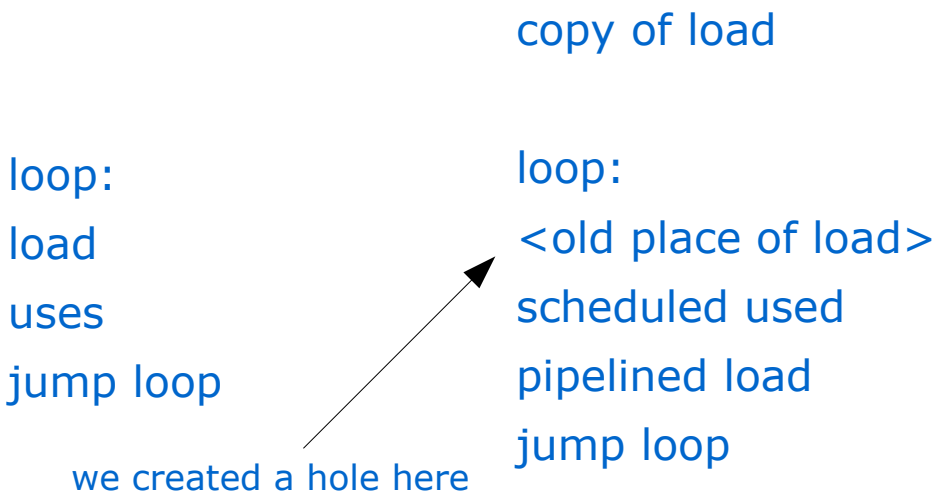
- Simple answer: a register copy should be cheaper than original operation
- Better answer: renamed operation is moved up farther than a latency of a register copy



- But: it is hard to estimate the number of cycles needed to execute the code through which the operation was moved up

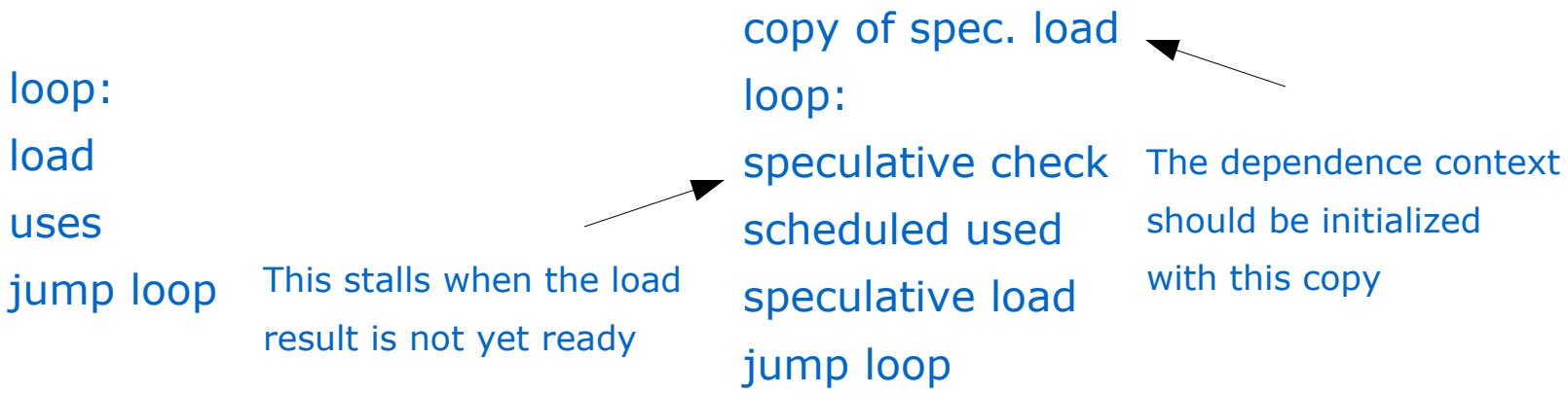
Estimating profitability of pipelining

- Moving instruction along the back edge may result in “holes” in the existing schedule
- To avoid this, we reschedule the pipelined code with disabled pipelining
- But this hurts compile time



Estimating profitability of pipelining

- With speculation, pipelined load may be too close to the speculative check
- Possible solution: give other insns a chance to increase the gap between the check and the load
  - Start rescheduling with a dependence context of a bookkeeping code
  - Do not treat checks as barriers for trapping instructions (as other jumps are treated)



Handling “ambiguous” transformations

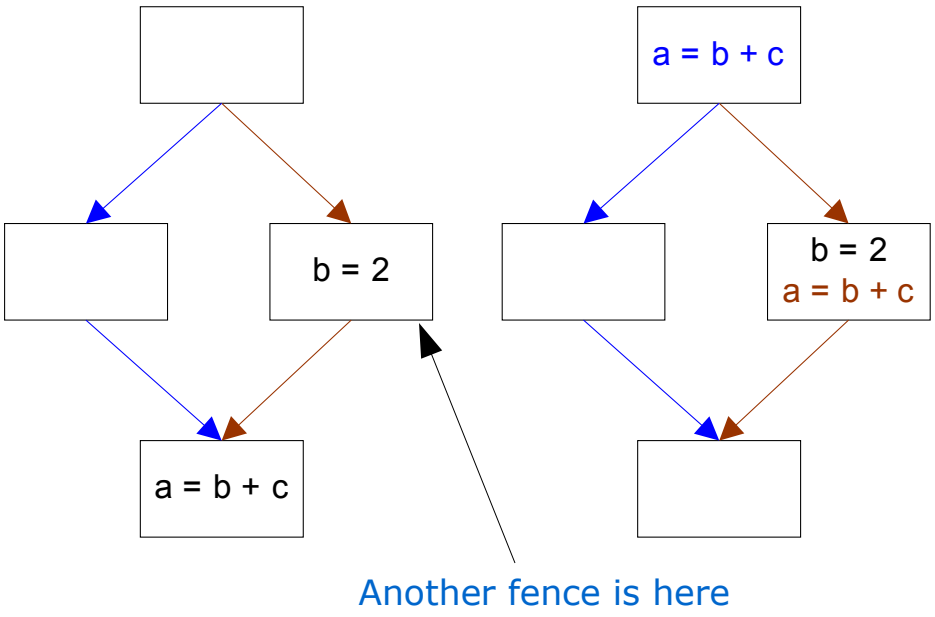
- When transforming insns, we need a way to undo the transformations
  - So we can find them later when fixing up the program (inserting bookkeeping copies, updating sets)
- But: sometimes you can't undo the transformation just by looking at the insn
  - Moving up “r33=sign_extend(r7)” through “r7=0” yields “r33=0”
  - validate_change may simplify the resulting insn
- Solution: simplify when scheduling the insn
- Or: record the history of transformations and avoid undoing altogether

Avoiding changes in control flow

- We want to minimize shuffling of CFG
  - Because this results in extra jumps
- Control flow can be changed when...
  - A bookkeeping copy is created
    - Extra jumps can be created when redirecting a fallthru edge
  - All insns from a block are moved up
    - And the unneeded jump can stay
- Solution: clean up empty blocks on your way
  - Also look up for spurious jumps on the next block

Avoiding changes in control flow

- Do not create new basic blocks for bookkeeping
  - When this can be done
- Watch out for the other fences
  - So we don't miss the bookkeeping code when scheduling



Speeding up dependence analysis

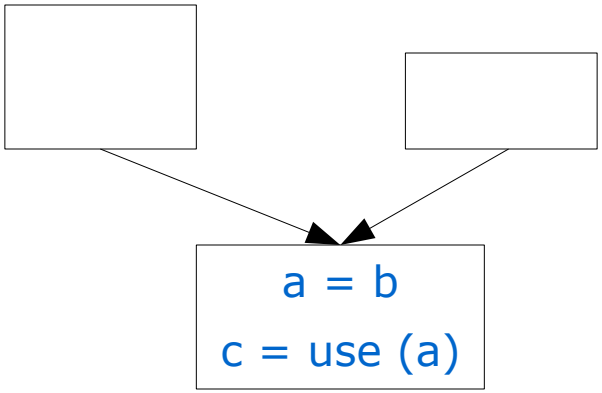
- On-the-fly dependence calculation
  - Hard to deduce which dependencies are valid along which path
  - Control and data dependencies are not distinguished
- Sched_analyze_insn does two things:
  - Find dependencies between the insn and the given dependence context
  - Add insn's side effects to the dependence context
- When insn X is moved through Y:
  - Initialize a dependence context with Y (run the analyzer on Y with an empty context)
  - Run the analyzer on X with Y's context

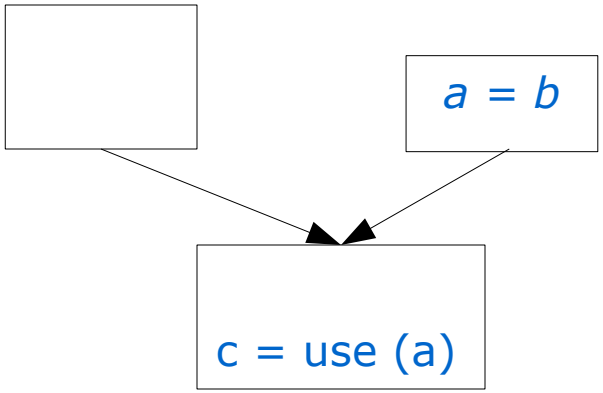
Speeding up dependence analysis

- Caching dependence analysis queries
  - Two bitmaps for yes/no/not analyzed
  - Reanalyze only when insn X has been transformed on Y
  - Even in this case, we can remember the resulting insn
- Supporting read only dependence contexts
  - We need to compute Y's dependence context only once
  - But for this, we need to be able to analyze X and update dependence context separately
  - A *read only* context cannot be modified by the analysis
- In the future: distinguishing between control and data dependencies
  - So data dependencies will be computed just once
  - Control dependencies will be picked up on the fly

Faster search for original instructions

- Use instruction hashes
  - For searching for an operation in a set
  - Hash RHS when only RHS is scheduled
- Hashes are needed to record insns on which an operation was transformed
  - Because the correctness of availability sets is guaranteed with respect to patterns, not UIDs





Speeding up register renaming

- Finding a suitable register for renaming is an expensive operation
  - Need to travel code motion paths to check liveness
  - REGNO_MODE_OK, OK_FOR_RENAME, crosses_call...
- Rename only for the best insn?
  - First choose the insn, then validate its register
  - But this ends up lying to the compiler!
  - DFA lookahead thinks we can issue an insn, but we really can't (and vice versa)
  - Target hooks come up with wrong decisions (on IA-64, we decide a stop bit is needed when it's not)

Speeding up register renaming

- Lazy computation of registers permitted by a target to be renamed from a given regclass
  - Per-regclass X per-mode sets
- A trade-off is needed for renaming
  - Rename the most prioritized insns
  - For others liveness restrictions for the single target register can be easily checked when computing av sets
- Precompute data for the code motion stage when looking for a new register
  - We can record the visited blocks and exact form of insns we're searching for
- A lot of performance tuning should be done

Optimizing memory allocation

- Malloc/free are high up in the profile
- We allocate a lot of short-time lifecycle objects
  - Operations, regsets, NOP patterns
- Regsets and NOPs are taken from the pool
  - The pool of NOP patterns is custom because the patterns are allocated by the garbage collector
- We plan to use alloc_pools for operations

Current status

- Performance and compile-time tuning is a work in progress
- Caching dependence queries and supporting read-only contexts speed up the scheduler by 15-20% each
- Instruction hashes are ~10%
- Still, the compiler (especially with pipelining enabled) can be slower up to 2x (depends on the scheduling window size and region size)

Planned performance tuning

- Kill unneeded jumps
- Use better instruction priorities (speculative yield)
  - Was implemented under a Google SoC project
- Whatever shows up in the assembly code
- **Most important: estimate profitability of a speculation when pipelining**

Planned compile-time tuning

- Implement alloc_pools for operations
- Kill unneeded recursion in traversing routines
- Tune the defaults for scheduling window size and region size
- Use df for calculating used/set/clobbered regsets and for liveness update
  - This will eliminate the dependence analysis run needed to find out these sets
- Whatever shows up in the profiler after fixing up all of the above :)
- **Most important: optimize register renaming**

Acknowledgments

- Vladimir Makarov consults the project
- Mark Davis described icc decisions
- Zdenek Dvorak helped with loop optimizer
- Tehila Meyzels tested on ppc
- Jim Wilson answered on ia64 questions
- People answering questions on the ML

Questions?