Global Instruction Scheduling for LLVM
What are we trying to achieve?

• Restart the discussion on how can we implement global instruction scheduling in LLVM in the most effective way
• How to best utilize the opportunities it presents
  – Global scope usability can go beyond instruction scheduling
  – ...and beyond statically scheduled targets
• See what (if anything) we might want to change in LLVM infrastructure to do it
Conceptual scheduling steps

Statically Scheduled Targets (SST)

1. IR representation
2. Instruction Selection
3. DAG construction/optimization
4. Pre-RA scheduling
5. Register Allocation
6. Post-RA scheduling
7. Bundling/Bundling finalization
8. Machine Instruction emission

Theoretical time accurate domain
Distributed Instruction Scheduling

• Pre-RA Instruction Scheduling
  – Most flexible (least constrained)
  – Has to anticipate register pressure
  – Uses DFA to model timing (which is discarded immediately)
  – Currently done on Basic Block (BB) scope

• Post-RA scheduling is much less effective than Pre-RA scheduling
  – Mainly due to all the new false dependencies introduced due to the constrained physical register set
  – …and mainly helps with scheduling spill/fill code

• Bundling is critical for SST performance and currently done by those back ends that utilize it rather late
  – …leaving plenty of performance unexplored…
  • Anecdotal evidence suggests that about 10-15% improvement is possible with global schedule cleanup pass for some cases
  • Average (geomean) SPEC 2K Int speedup 1.4%
Global Instruction Scheduling

• General premise – allow instruction scheduling beyond Basic Block boundaries
  – Scheduling region == static scheduling scope
    • Benefits all targets and not only for scheduling
  – Trace, Superblock, Hyperblock, TreeRegion
  – Major step is scheduling region formation
    • For best results should span as much scope as possible with as little code duplication as possible
    • In general case requires complex if-conversion and speculation support

• Best performed on the least constrained state of DAG
  – Probably right before RA
  – Needs precise profile information
  – Liveness needs to handle predication properly
Global Instruction Scheduling

• Cyclic/Ayclic cases are handled rather differently
  – Cyclic scheduling is currently performed rather “early”
    • When bundling is not yet available
  – If cyclic scheduling has been performed on a loop (SW pipelining)
    acyclic scheduler might need to
    • Leave it in the “tuned” form
    • Beware of alternative reg allocation strategy for that region
    • Deal with global live ranges around that loop
    • Assign timing (bundling) in accord with the original scheduling intent
• Autovectorization can add additional constrains
  – Outside the scope of this presentation
Global Instruction Scheduling

• Performing global instruction movement on *actual machine instructions* and *allocated physical registers* offers several benefits
  – It is unlikely to make performance worst, but it can make it significantly better
    • Once critical path is established we are free to utilize “holes” around it and shorten it when possible
  – It does not have to worry about increase in register pressure
  – It can perform very target specific optimizations
    • Like custom peephole opt and peculiar code layout or compaction
    • Power consumption related options etc.

• You have nothing to lose but your chains 😊

• Cons include
  – Restricted scheduling region formation
  – Limited flexibility in code motion
  – Complex (if any) register scavenging
  – Multi-way branching
A case study - Hexagon Global Scheduler

• Global Scheduling done after packetization (bundling)
• Major goal is to fill in the gaps in the bundles by “pulling up” instructions from successor basic blocks
  – ...without lengthening the critical path
• Uses the DFA based Hexagon VLIW packetizer
  – To manage resource constraints in bundles
• Utilizes a specialized post-RA predicate-aware liveness analysis pass
• Performs speculative and predicative global scheduling of instructions in a superblock
• 1.4% improvement on SPEC 2K Int
A case study - Hexagon Global Scheduler

After Packetizer (4 wide issue)

Step 1 (pull from BB3 to BB1)
A case study - Hexagon Global Scheduler

Step 2 (pull from BB1 to BB0)

Step 3 (pull from BB1 to BB0)
A case study - Hexagon Global Scheduler

Step 4 (fold BB3)

Step 5 (fold BB2)

Three-way branch!
A case study - Hexagon Global Scheduler

• A robust implementation was achieved only after focusing on a limited set of schedule
• The Machine verifier pass was of great help
  – after tailoring it for the Hexagon backend
• A separate post-RA liveness analysis pass (which understands predication), was required
• We could have simpler implementation and possibly, better results if global scheduling would be done conjointly with bundling
Open questions for Global scheduling in LLVM context

• Proposed region formation implementation in LLVM
  – Possibly provided as analysis pass
    • To be used for extended if conversion, global peephole and global scheduling
    • Provides liveness info at region boundaries
  – Can we have totally autonomous region formation pass?
    • Most likely yes
  – Do we need any IR support for region formation?
    • runOnRegion(Region *R)
    • Might be a compile time issue
    • Can we backtrack a scheduling decision during RA? (or some other pass)
      – Reverse if-conversion

• Do we need full liveness support for predicated data flow?
  – Virtual predicate regs

• Can we unbundle an earlier bundle deterministically?
  – Register swap for instance
  – IR support for ambiguous cases?
Indeterminism in bundle dispersing

- There has been a lot of discussion about serialization indeterminism
  - Order indeterminism
    - \{ a = 0; b = a \}
    - In general case an anti dependency restoration
    - Can possibly be a problem to an exposed pipeline machine
  - Swap
    - \{ a = b; b = a \} \Leftrightarrow t = a; a = b; b = t \Leftrightarrow a = a^b; b = a^b; a = a^b;\n    - Might require a temp storage
    - XOR copy
  - Aliasing indeterminism (memory swap)
    - \{ A[i] = i; i = B[i] \} what if A \neq B
    - Should not be a problem for targets where “left-to-right” semantics are assumed
  - Platform specific side effects
    - \{ p0 = cmp.eq(a,b); p0 = cmp.eq(a,c) \}
- Fortunately it should be a rare event and we should optimize for the common case
Open questions for Global scheduling in LLVM context

• Accurate global region formation requires precise profile info
• Appropriate hooks in the register allocator for handling register allocation on bundles with predicates
• Global scheduling conjoint with register allocation?
Near term solutions

• Predicate aware liveness
• Bundle-aware passes
• Incremental improvement to the register coalescer algorithm
• Load/store speculation support
• Instruction selection functionality can be made timing sensitive
  – Macro-fusion support (compare+jmp)
Mid term

• Better API for live intervals
  – Including target specific hooks
• Access to target-specific peephole opts during scheduling
• Expression height reduction (not actually in the scheduler)
• Target-specific folding/unfolding (e.g. postinc load formation)
Long range perspective

- Finalize general scheduling framework
- Back-tracking scheduling related decisions
- Global DAG construction determinism
- Pre-RA full bundle formation
Thank you

Follow us on: f /twitter/in

For more information on Qualcomm, visit us at:
www.qualcomm.com & www.qualcomm.com/blog

©2013-2014 Qualcomm Incorporated and/or its subsidiaries.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References to “Qualcomm” may mean Qualcomm Incorporated, or subsidiaries or business units within the Qualcomm corporate structure, as applicable.

Qualcomm Incorporated includes Qualcomm’s licensing business, QTL, and the vast majority of its patent portfolio. Qualcomm Technologies, Inc., a wholly-owned subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of Qualcomm’s engineering, research and development functions, and substantially all of its product and services businesses, including its semiconductor business, QCT.