Instruction scheduling for Superscalar and VLIW platforms Temporal perspective

Andy Trick, Apple
Sergei Larin, QuIC
Nov 07 2012

Temporal Perspective

- The temporal perspective of workflow specifications is through a series of temporal constructs that may occur when defining a process model
- This characterization is independent of any specific modeling formalism or approach

What are we trying to achieve?

- Better understanding on instruction scheduling in LLVM today
  - Including timeline perspective
- Plan for the future
  - Desired features and implementation details

Vertical Divide (the somewhat Great Schism)

- There are really two separate sets of requirements are present in LLVM
  - Scheduling for Superscalar targets
    - Less strict, more forgiving
    - Less mem dep ordering sensitive
    - Fine with BB scope
  - Scheduling for VLIW (like) targets
    - Very strict to ordering and mem disambiguation
    - Creates bundles
    - Needs some form of global scheduling

Pre-RA Instruction Scheduling milestones

- SDNode scheduler
  - Simple list scheduler (Sethi-Ullman)
  - Operates on mixture of lowered instructions and remains of SSA representation
  - Simple DAG mem deps pruning
  - Use DFA model for VLIW targets
- MIScheduler
  - Converging List Scheduler
  - Operates on Mis and mixture of virtual and physical registers
  - Simple DAG mem deps pruning
  - Use DFA model for VLIW targets
- RegPressure tracking
- TargetSchedModel/MCSchedModel

Scheduling Infrastructure Topics

- Infrastructure design goals
- Superscalar scheduling design goals
- Machine model
- Pass order
- Driver
- Register pressure infrastructure
- Scheduling strategy and heuristics
  - Register pressure heuristics
  - Resource balancing

Infrastructure Design Goals

- Remove all dependence on SelectionDAG scheduling
- Make instruction scheduling completely optional
- Support out-of-order targets
- Support VLIW targets (bundling)
- Provide a place for target-specific optimizations

Superscalar scheduling design goals

- Do no harm and preserve source order
- Model out-of-order processor resources

New machine model

- Three levels
  - Coarse grain properties
  - Per-opcode resources and latency
  - Pipeline itineraries (primarily for VLIW)
- Free form description
  - Highly customizable
  - Allows incremental development
  - Allows the description format to closely match the microarchitecture spec

Machine model example
Target Description

```
See TargetSchedule.td

// Define target specific SchedReadWrite types.


def WriteI;  // ALU
def WriteIS; // Shift
def WriteF;  // Float
def WriteL;  // Load
def ReadAdr; // Memory Address
def ReadFAcc; // Accumulator


def LoadPostinc<...> : Instruction, Sched<[WriteL, WriteI, ReadAdr]>
{...}


def FMA<...> : Instruction, Sched<[WriteF, ReadFAcc]> {...}
```

Machine model example
Processor description

```
// Define kinds of processor resources and quantities.

def YourProcUnitI  : ProcResource<2>;
def YourProcUnitIS : ProcResource<1> { let Super = YourProcUnitI; }
def YourProcUnitLS : ProcResource<1>;
def YourProcUnitFP : ProcResource<1> { let Buffered = 0; }

// Define processor specific operand latencies and resource requirements

let SchedModel = YourProcModel in {

def : WriteRes<WriteI, [YourProcUnitI]>;
def : WriteRes<WriteF, [YourProcUnitFP]> { let Latency = 4; };

// If the result is produced by a load, we can read it one cycle before it is ready.
def : ReadAdvance<ReadA, 1, [WriteL]>

}
```

Machine model example
Processor opcode override

```
// Override operations with more interesting processor specific behavior

def YourProcWriteFMovI : SchedWriteRes<[YourProcUnitF, YourProcUnitLS]> {
  let Latency = 6;
}

let SchedModel = YourProcModel in {

def : InstRW<[YourProcWriteFMovI], FMovIOpc>;

}
```

Machine model example Variants

```
// Define C++ functions to distinguish between multiple models for a single
// opcode based on immediate operand values or other modifiers.

def YourProcWriteIS : SchedWriteRes<[YourProcUnitIS]>;
def YourProcWriteExtract : SchedWriteRes<[YourProcUnitIS]> {
  let Latency = 2;
  let ResourceCycles = [2];
}

def ExtractPred : SchedPredicate<[{TII->isExtractOrDeposit(MI)}]>

def YourProcWriteIS: SchedWriteVariant<[
  SchedVar<ExtractPred, [YourProcWriteExtract],
  SchedVar<NoSchedPred, [YourProcWriteIS]>]>;

let SchedModel = YourProcModel in {

SchedAlias<WriteIS, YourProcWriteIS>

}
```

Machine model example Sequences

```
// Combine existing definitions use sequences for additive resources and latency.

def YourProcWriteShlAdd : WriteSequence<[WriteIS, WriteI]>;

let SchedModel = YourProcModel in {
def : InstRW<[YourProcWriteShlAdd], [ShlAddOpc]>;
}
```

Pass order

SSA Opts

-> Register Coalescing

-> Machine Scheduling

-> Register Allocation

- Subregister copies get in the way of the scheduler.
- The scheduler can easily recover from register coalescing within a block.
- Some targets require scheduling/ bundling after coalescing (VLIW).

Machine scheduling driver

- Target configuration can replace the misched pass with a target scheduler
- The target may reuse the misched pass and misched driver, but register a new scheduler implementation (VLIW approach)
- The target may reuse the misched implementation but plugin its own MachineSchedStrategy for heuristics

Register pressure

- RegClass -> (Pressure Sets, Unit Weight)
- PressureSet -> Limit
- x86 example

AH in {GR8, GR8_NOREX, GR8+GR64...}, weight = 1
AX in {GR64}, weight = 2
GR8_NOREX limit=8
GR64 limit=34 (17 including RIP)

- ARM example

S0 in {DPR, SPR}, weight=1
D0 in {DPR}, weight=2
SPR limit=32
DPR limit=64

Scheduling Direction

- Some targets may want top-down scheduling (VLIW). So it's useful to have an infrastructure support both.
- It's also handy to experiment with new scheduler heuristics.

Bidirectional heuristics for superscalar

- Bidirectional support is mainly intended to handle medium-size blocks with odd scheduling problems. The solution is to proceed in the direction in which choices are more limited, or clearly beneficial.
- For large blocks, bidirectional scheduling is intended to result in more symmetrical schedules, without jamming like-resource consumers at one end, or oddly shuffling code.

Current register pressure heuristics

- Greedy pressure backoff
  - 3 levels: excess pressure, critical pressure, max pressure
- Adjacent def-uses fallback

Potential register pressure heuristics

- Pressure avoidance using subtree detection
- Pressure avoidance using sethi-ullman numbers
- Greedy pressure reduction using intervals
- Pressure avoidance using precomputed lineages
- Example: matmul

Resource balancing heuristics

- Reduce/demand resources
- ShouldIncreaseILP
- Example: blowfish

Nearly there

- Macro-fusion support (cmp+jmp)
- Load/store clustering
- Incremental improvement to the register coalescer algorithm.
- Performance analysis an the necessary platforms to make it the default

Around the corner

- Better API for live intervals
- Move local target-specific peephole opts into the sched pass
- Local live range splitting in the coalescer
- Expression height reduction (not actually in the scheduler)
- Target-specific folding/unfolding (e.g. postinc load formation)

Looking ahead

- Finalize general scheduling framework
- Back-tracking scheduling?
- Early bundle formation
- Global scheduling
  - Any change to IR is needed?
  - Any relationship to sophisticated predication support?
- DAG construction determinism
- Post-RA Scheduler?

Common MI Sched heuristics (backup slide)

- The current (experimental) MI scheduler implements a 3-level "back-off":
  - 1) Respect the target's register limits at all times.
  - 2) Indentify critical register classes (pressure sets) before scheduling
    - Track pressure within the currently scheduled region
    - Avoid increasing scheduled pressure for critical registers
  - 3) Avoid exceeding the max pressure of the region prior to scheduling (don't make things locally worse)
- All of the heuristics that I have planned are greedy
  - some require precomputing register lineages (dependence chains that reuse a single register)
  - MI scheduler can alternate between top-up and bottom-down, which doesn't fundamentally change the problem, but avoids the common cases in which greedy schedulers "get stuck"
- Plan for the near future
  - SpillCost: Map register units onto a spill cost that is more meaningful for heuristics
  - Pressure Query: (compile time) Redesign the pressure tracker to summarize information at the instruction level for fast queries during scheduling
  - Pressure Range: Before scheduling, compute the high pressure region as a range of instructions
    - If the scheduler is not currently under pressure, prioritize instructions from within the range
  - Register Lineages: Before scheduling, use a heuristic to select desirable lineages
    - Select the longest lineage from the queue
    - After scheduling an instruction, look at the next instruction in the lineage. If it has an unscheduled operand, mark that operand's lineage as pending, and prioritize the head of that lineage
    - This solves some interesting cases where a greedy scheduler is normally unable to choose among a set of identical looking instructions by knowing how their dependence chain relates to any already scheduled instructions