Instruction Scheduling Beyond Basic Blocks

Extended Basic Blocks, Superblock Cloning, & Traces, with a quick introduction to Dominators

Comp 412
Local Instruction Scheduling

Greedy heuristic technique that operates over a single basic block
1. Rename registers to eliminate artificial constraints *(anti-dependences)*
2. Build a dependence graph
3. Compute one or more priority functions on the graph
4. Schedule all the operations with respect to the dependences & priorities

As long as we stay within a single block
• List scheduling does well
• Underlying ideas remain easy to understand (& to implement)
Scheduling Larger Regions

To achieve further improvement, the compiler can schedule over larger regions

- An Extended Basic Block is a maximal set of blocks such that:
  - The set has a single entry node, $B_i$
  - Each block other than $B_i$ has exactly one predecessor $B_j$, and $B_j$ is in the set

- Example CFG to the right has three EBBs *
  - Big EBB has two paths
    - $\{B_1, B_2, B_4\}$ & $\{B_1, B_3\}$

- Other two EBBs are degenerate
  - Each contains a single block
    - $\{B_3\}$ & $\{B_6\}$

COMP 412, Fall 2018
Scheduling Larger Regions

Superlocal Scheduling

• Schedule entire paths through an EBB together

• In example, schedule \{B_1,B_2,B_4\}, \{B_1,B_3\}, \{B_3\}, & \{B_6\}

For example, first schedule \{B_1,B_2,B_4\}, then \{B_1,B_3\}, then \{B_5\}, & \{B_6\}

• Having \(B_1\) in both \{B_1,B_2,B_4\} and \{B_1,B_3\} causes conflicts
  − Moving an op out of \(B_1\) causes problems on the other path
  − Must insert compensation code in \(B_3\) unless \(c\) is dead in \(B_3\)
  − Increases code space, may not help \{B_1,B_3\}

C was moved for correctness not for speed!
Scheduling Larger Regions

Superlocal Scheduling

• Schedule entire paths through an EBB together
• In example, schedule \{B_1, B_2, B_4\}, \{B_1, B_3\}, \{B_3\}, & \{B_6\}

For example, first schedule \{B_1, B_2, B_4\}, then \{B_1, B_3\}, then \{B_5\}, & \{B_6\}

• Having \(B_1\) in both \{B_1, B_2, B_4\} and \{B_1, B_3\} causes conflicts
  - Moving an op into \(B_1\) causes problems on the other path
  - May need compensation code in \(B_3\), although renaming may avoid it
  - Lengthens \{B_1, B_3\}, even without compensation code

Compensation code makes the path \(B_1B_3\) even longer!
Scheduling Larger Regions

Superlocal Scheduling

• How much improvement can we get?
  – Schielke showed 11 to 12% speed ups
  – Constrained away compensation code
  – So, it *is* worth doing ...

Cooper & Schielke, “Non-local instruction scheduling with limited code growth,” LCTES Workshop, June 1998

COMP 412, Fall 2018
Scheduling Larger Regions

Can We Be More Aggressive?

• Create even more context for local scheduler to use
• Join points in the CFG create blocks that must work in multiple contexts
  – Maybe we can eliminate join points

• Superblock cloning is a transformation that eliminates some of the join points in the CFG
  – Has application to other optimizations
Scheduling Larger Regions

More Aggressive Superlocal Scheduling

• Clone blocks at each join point that does not involve a loop-closing branch

Superblock Cloning

• Enabling transformation
• Start at the root and clone up to a backward branch
• Creates better conditions for other transformations—e.g., superlocal scheduling or superlocal value numbering
• Increases code size (replication)
• Other enabling transformations include: loop unrolling (§8.5.2) and inline substitution (§8.7.1)
Scheduling Larger Regions

More Aggressive Superlocal Scheduling

- Clone blocks at each join point that does not involve a loop-closing branch
- Some of the resulting blocks can combine
  - Single successor, single predecessor

COMP 412, Fall 2018
Scheduling Larger Regions

More Aggressive Superlocal Scheduling

• Clone blocks at each join point that does not involve a loop-closing branch
• Some of the resulting blocks can combine
  – Single successor, single predecessor
• Now, schedule the EBB paths
  – \{B_1, B_2, B_4\}, \{B_1, B_2, B_{5a}\}, \{B_1, B_3\}
  – Pay attention to compensation code

• Works well for forward motion
• Backward motion still need compensation code
A Pointed Digression

How does the compiler identify a loop-closing branch?
• It uses the notion of dominance in a flow graph†
• We can compute dominance using data flow analysis (§ 9.2.1 in EaC2e)

Definition

In a flow graph, **y dominates x** iff every path from the graph’s entry node to x includes y.

We often denote **y dominates y** as **y DOM x**.

• ∀ node **x ∈ G**, its set of dominators is called **DOM(x)**
• ∀ node **x ∈ G**, x ∈ **DOM(x)**
• |**DOM(x)**| ≥ 1

A Pointed Digression

How does the compiler identify a loop-closing branch?

By inspection, \((B_9, B_2)\) is a loop-closing branch.

A loop-closing branch runs from a node \(x\) to some node \(y\) in \(\text{DOM}(x)\)

The computation of \(\text{DOM}\) sets is covered in § 9.2.1 in EaC2e. It is straightforward.
Another Pointed Digression

To improve scheduling, we turned to extended basic blocks and superblock cloning to give the scheduler more context

• We can **value number** over extended basic blocks, as well
  – Treat each path in the **EBB** as a single block and apply **LVN**
    → Creates the **Superlocal Value Numbering algorithm (SVN)**
  – Finds more opportunities for improvement
  – Simple approach reanalyzes blocks
  – With **SSA** name space, can make this more efficient  
    (See § 8.5.1 in EaC2e)

• **Superblock cloning** can be applied before **EBB** construction in **SVN**
  – Eliminates more branches and jumps
  – Introduces some code growth

These same tricks have been applied to several local optimizations
Scheduling Larger Regions

Trace Scheduling
• Start with execution counts for edges

Two Options:
1. Profile data
   → instrument the code and run it on “representative data”
   → interrupt periodically & sample
   → Infer edge counts from perf counters

2. Static estimates
   → use some simple heuristic and guess
   → jumps x 1, branches x 0.5, backward branches x 10

See the digression on page 452 of EaC2e for more on gathering profile data

Block counts can mislead us — see $B_5$
Scheduling Larger Regions

Trace Scheduling

- Start with execution counts for edges
  - Obtained from profiling runs
- Pick the “hot” path
  - A “trace” is a maximal length, acyclic path through the CFG
  - The “hot” path is the trace that has, at each point, the highest count

Block counts can mislead us — see $B_5$
Scheduling Larger Regions

**Trace Scheduling**

- Start with execution counts for edges
  - Obtained from profiling runs
- Pick the “hot” path
  - A “trace” is a maximal length, *acyclic* path through the *CFG*
  - The “hot” path is the trace that has, at each point, the highest count
  - \( \{B_1, B_2, B_4, B_6\} \)
- Schedule the hot path
  - Compensation code in \( B_3 \) & \( B_5 \) if needed
  - Get the hot path right!
- If we picked the right hot path, the other blocks do not matter as much
  - Places a premium on quality profiles
Scheduling Larger Regions

Trace Scheduling the Entire CFG
• Pick and schedule the hot path
• Insert compensation code, as needed
• Remove hot path from CFG
Repeat the process until CFG is empty

Example
  – \{ B_1, B_2, B_4, B_6 \} then \{ B_3, B_5 \}
  – All other edges run between scheduled blocks

Idea
• Hot paths matter most
• The farther we go off the hot path, the less it matters
Scheduling Larger Regions

Sketch of the Trace Selection Algorithm ("pick the hot path")

mark each edge as eligible or ineligible
initialize the trace with the best eligible edge

let x be the best eligible edge entering source(trace)
let y be the best eligible edge leaving sink(trace)

while(one of x or y is non-null) {
    let z be the best of x and y
    add z to the appropriate end of the trace
    let x be the best eligible edge entering source(trace)
    let y be the best eligible edge leaving sink(trace)
}

Some Critical Definitions:
An edge is ineligible if:
(1) Both ends are already in a trace, or
(2) It is a loop-closing branch (i.e., y dominates x)

"Best" means the edge in a set of edges that has the highest execution frequency

The details of finding the edges are complicated but straightforward
⇒ Iterate over inbound edges to find max eligible edge
Trace Construction on the More Complex Graph

- Consider the graph to the right
- Edges annotated with execution frequencies
  - Can rank edges by frequency
  - Sums make sense along paths

Example Control-Flow Graph
Trace Construction

Building Traces — First Trace

- Initial edge is $<B_3, B_6>$
Trace Construction

Building Traces — First Trace

• Initial edge is $<B_3, B_6>$
• Algorithm looks at $<B_1, B_3>$, $<B_6, B_8>$, and $<B_6, B_9>$.
• $<B_{10}, B_3>$ is a loop closing branch and, therefore, ineligible.
Trace Construction

Building Traces — First Trace

• Initial edge is \(<B_3, B_6>\)
• Algorithm looks at \(<B_1, B_3>, <B_6, B_8>, \) and \(<B_6, B_9>\).
• Adds \(<B_6, B_8>\)
Trace Construction

Building Traces — First Trace

- Initial edge is \(<B_3,B_6>\)
- Algorithm looks at \(<B_1,B_3>, <B_6,B_8>,\) and \(<B_6,B_9>\).
- Adds \(<B_6,B_8>\)
- Algorithm looks at \(<B_1,B_3>\) and \(<B_8,B_{10}>\)
Trace Construction

Building Traces — First Trace

- Initial edge is \(<B_3, B_6>\)
- Algorithm looks at \(<B_1, B_3>, <B_6, B_8>,\) and \(<B_6, B_9>\).
- Adds \(<B_6, B_8>\)
- Algorithm looks at \(<B_1, B_3>\) and \(<B_8, B_{10}>\)
- Adds \(<B_8, B_{10}>\)
Trace Construction

Building Traces — First Trace

• Initial edge is \(<B_3,B_6>\)
• Algorithm looks at \(<B_1,B_3>, <B_6,B_8>,\) and \(<B_6,B_9>\).
• Adds \(<B_6,B_8>\)
• Algorithm looks at \(<B_1,B_3>\) and \(<B_8,B_{10}>\)
• Adds \(<B_8,B_{10}>\)
• Algorithm looks at \(<B_1,B_3>\) and adds it

The first trace is \{B_1, B_3, B_6, B_8, B_{10}\}. 

COMP 412, Fall 2018
Trace Construction

Building Traces — Second Trace

- Initial edge is one of \(<B_6,B_9>\) or \(<B_9,B_{10}>\)
Trace Construction

Building Traces — Second Trace

- Initial edge is one of \(<B_6, B_9>\) or \(<B_9, B_{10}>\)
- Use \(<B_9, B_{10}>\) \((arbitrary\ choice)\)
Trace Construction

Building Traces — Second Trace

- Initial edge is one of $<B_6, B_9>$ or $<B_9, B_{10}>$
- Use $<B_9, B_{10}>$
- Algorithm considers $<B_6, B_9>$; $<B_{10}, B_3>$ is ineligible.
Trace Construction

Building Traces — Second Trace

- Initial edge is one of \( <B_6, B_9> \) or \( <B_9, B_{10}> \)
- Use \( <B_9, B_{10}> \)
- Algorithm considers \( <B_6, B_9>, <B_{10}, B_3> \) is ineligible.
- Adds \( <B_6, B_9> \)
Trace Construction

Building Traces — Second Trace

• Initial edge is one of \( <B_6, B_9> \) or \( <B_9, B_{10}> \)
• Use \( <B_9, B_{10}> \)
• Algorithm considers \( <B_6, B_9>, <B_{10}, B_3> \) is ineligible.
• Adds \( <B_6, B_9> \)

The second trace is \( \{ B_9 \} \), a degenerate (single block) trace.
• Scheduler might get some context from the prior schedule of \( B_6 \)
Trace Construction

Building Traces — Third Trace

- Initial edge is \(<B_5, B_7>\)
Trace Construction

Building Traces — Third Trace

• Initial edge is $<B_5, B_7>$
• $<B_2, B_5>$ and $<B_3, B_5>$ are eligible
Trace Construction

Building Traces — Third Trace

• Initial edge is $<B_5, B_7>$
• $<B_2, B_5>$ and $<B_3, B_5>$ are eligible
• Adds $<B_3, B_5>$
Building Traces — Third Trace

- Initial edge is \( <B_5, B_7> \)
- \( <B_2, B_5> \) and \( <B_3, B_5> \) are eligible
- Adds \( <B_2, B_5> \)
- No more edges are eligible

The third trace is \( \{B_3, B_5, B_7\} \)

Scheduler can change \( B_5 \) and \( B_7 \), but \( B_3 \) has already been scheduled.

Scheduler can use the previously scheduled \( B_3 \) for context.
Trace Construction

Building Traces — Fourth Trace

- Initial edge is \(<B_1, B_2>\)
Trace Construction

Building Traces — Fourth Trace

• Initial edge is \(<B_1, B_2>\)
• \(<B_2, B_4>\) and \(<B_2, B_5>\) are eligible
Trace Construction

Building Traces — Fourth Trace

- Initial edge is $<B_1, B_2>$
- $<B_2, B_4>$ and $<B_2, B_5>$ are eligible
- Algorithm chooses $<B_2, B_4>$
Trace Construction

**Building Traces — Fourth Trace**

- Initial edge is \(<B_1, B_2>\)
- \(<B_2, B_4>\) and \(<B_2, B_5>\) are eligible
- Algorithm chooses \(<B_2, B_4>\)
- Only \(<B_4, B_7>\) is eligible
Trace Construction

Building Traces — Fourth Trace

• Initial edge is $<B_1, B_2>$
• $<B_2, B_4>$ and $<B_2, B_5>$ are eligible
• Algorithm chooses $<B_2, B_4>$
• Only $<B_4, B_7>$ is eligible
• Algorithm chooses $<B_4, B_7>$
Trace Construction

Building Traces — Fourth Trace

- Initial edge is $<B_1, B_2>$
- $<B_2, B_4>$ and $<B_2, B_5>$ are eligible
- Algorithm chooses $<B_2, B_4>$
- Only $<B_4, B_7>$ is eligible
- Algorithm chooses $<B_4, B_7>$

Fourth Trace is \{ $B_1, B_2, B_4, B_7$ \}

*Scheduler cannot change $B_1$, but it schedules others with context from $B_1$*
Scheduling the Traces

The compiler then schedules the traces, in order of discovery:

1. \( \{B_1, B_3, B_6, B_8, B_{10}\} \)
2. \( \{B_6, B_9, B_{10}\} \)
3. \( \{B_3, B_5, B_7\} \)
4. \( \{B_1, B_2, B_4, B_7\} \)

And, it has selected and scheduled every block ...
Trace Scheduling

Trace scheduling applies list scheduling in larger contexts

• Power of the technique comes from longer code sequences
  – Typical basic blocks, in real code (rather than lab test blocks), are short
  – 3 to 10 operations

• Power of the technique comes from finding traces that execute often
  – Code that executes infrequently has less impact on performance
  – Cardinal principal of optimization: *improve frequently executed code*

Trace scheduling has been used successfully in commercial practice

• Particularly useful in compilers for VLIW architectures

• Larger contexts create more *instruction-level parallelism*