Instruction Scheduling Beyond Basic Blocks
*Extended Basic Blocks, Superblock Cloning, & Traces,*
*with a quick mention of Dominators*

Comp 412

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 11 (& 8) in EaC2e
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)
How Well Does List Scheduling Do?

List scheduling does well

• The literature suggests that it finds optimal schedules most of the time
• Some authors report, incorrectly, that it always finds optimal schedules

Schielske did an extensive study using his RBF algorithm

The RBF Algorithm

• Randomize all tie breaking decisions
• Run 5 passes of Backward list scheduling
• Run 5 passes of Forward list scheduling
• Keep the best schedule \( \text{(shortest time to completion)} \)

To measure the effectiveness of RBF, Schielke compared the results of RBF against a long-running iterative repair scheduler that, in practice.
How Well Does List Scheduling Do?

Non-optimal list schedules (%) versus available parallelism †

- 1 functional unit, 85,000 randomly generated blocks of 10, 20, 50 ops
- RBF found optimal schedule over 80% of the time
- We expect better on human-written code

† Think of available parallelism as the average ready queue length, taken over each cycle of the schedule.
From Phil Schielke’s thesis

How Well Does List Scheduling Do?

Non-optimal list schedules (%) versus available parallelism

- 1 functional unit, 85,000 randomly generated blocks of 10, 20, 50 ops
- RBF found optimal schedule over 80% of the time
- We expect better on human-written code

At the peak, the compiler should apply other techniques

- Measure parallelism in list scheduler (average ready queue length)
- Invoke stronger techniques when high-probability of payoff

If the compiler transforms the code, it should avoid this area!

COMP 412, Fall 2019
How Well Does List Scheduling Do?

Does this result make sense?

- At AP = 1, the scheduler makes no bad choices
- At AP = 2, it has a 50% chance of making the wrong choice
- At AP = 3, it has a 66.6% chance of making the wrong choice
  - But, as AP rises, the dependence chains get shorter; there are more of them; and, therefore, the block is easier to schedule

AP between 2 and 3 is the sweet spot for bad decisions!

Am I making a bad assumption?

- I assumed that one of the choices is better than the others.
- In practice, that might not be the case, which drives the worst-case AP a little higher.
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
    – $\{B_3\}$ & $\{B_6\}$
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 $B_1 B_3$ 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
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 2019
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_{5q}\}, \{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?

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

The computation of DOM sets is covered in § 9.2.1 in EaC2e. It is straightforward.
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**
- \[ \{ B_1, B_2, B_4, B_6 \} \text{ then } \{ B_3, B_5 \} \]
- 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

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

COMP 412, Fall 2019
Trace Construction

Building Traces — First Trace

• Initial edge is $\langle B_3, B_6 \rangle$
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 \( \{ B_1, B_3, B_6, B_8, B_{10} \} \).
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 \(<B_6, B_9>, \quad <B_{10}, B_3>\) 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 $\langle B_5, B_7 \rangle$
Trace Construction

Building Traces — Third Trace

• Initial edge is \(<B_5, B_7>\)
• \(<B_2, B_5>\) and \(<B_3, B_5>\) are eligible
Building Traces — Third Trace

- Initial edge is $<B_5, B_7>$
- $<B_2, B_5>$ and $<B_3, B_5>$ are eligible
- Adds $<B_3, B_5>$
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 \(<B_2, B_5>\)
• 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 $<B_1, B_2>$
Building Traces — Fourth Trace

- Initial edge is $\langle B_1, B_2 \rangle$
- $\langle B_2, B_4 \rangle$ and $\langle B_2, B_5 \rangle$ 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 \(B_2\) and \(B_4\) with context from \(B_1\)
Trace Scheduling

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*

• Power of the technique comes from strong local list scheduling

Trace scheduling has been used successfully in commercial practice

• Particularly useful in compilers for VLIW architectures
• Larger contexts create more *instruction-level parallelism*