Update

- There is a rich history of using integer linear programming as a compiler optimization
  - Most work restricted to single basic blocks
  - Work that extends across basic blocks is constrained by control flow and machine resource complexity − common to reduce problem to *regions* of intrinsic instructions
- Our problem is unique:
  - No control flow to consider
  - No need to consider register allocation and other machine resources
  - Narrowing the problem to the interaction between parallelism and code size enables deep optimization

Algorithm Example

- **Main Procedure**:
- For each starting point $\mathbf{x_k} \in [0, |\alpha|]$:
  - For each call site α_i:
    - For each inst I $\in [\mathbf{x_k}, |\alpha|]$:
      - **search** for available slots
- **Search**(instruction I):
  - In dependency DAG of program, find latest scheduled parent instruction **J** in sequential program
  - For each instruction **p** $\in$ **[J, I]**:
    - If **I** can be parallelized with **p**, return true, mark location
  - Return false

Example - Setup

P

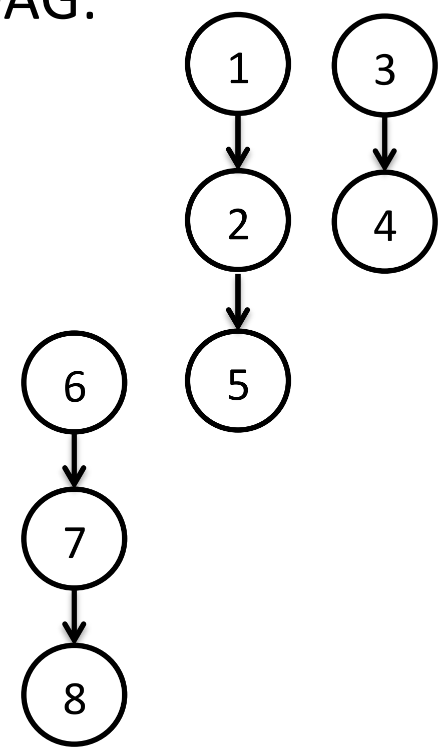
```
1 PrepZ q0, 0
2 H q0
3 PrepZ q1, 0
4 H q1
5 foo q0
…
```

foo (q)

```
6 H q
7 T q
8 H q
```

Running algorithm on function *foo*, consider call site shown

Sequential program dependency DAG:



Example – Call Site Expansion

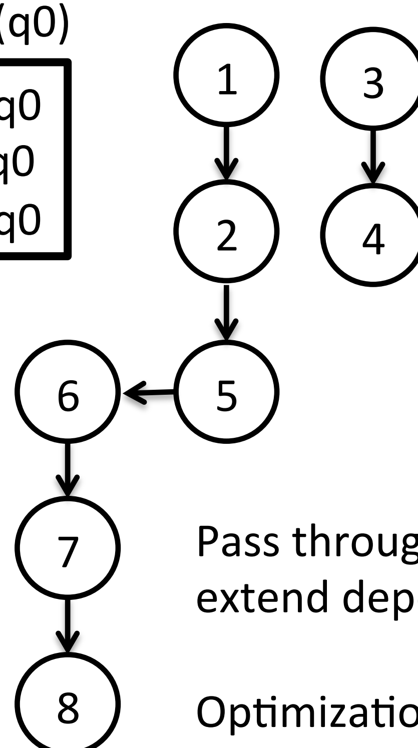
P

```
1 PrepZ q0, 0
2 H q0
3 PrepZ q1, 0
4 H q1
5 foo q0
…
```

foo (q0)

```
6 H q0
7 T q0
8 H q0
```

1: Call site expansion:



Pass through function arguments to extend dependency analysis

Optimization works across basic blocks: each function call needs to be expanded to a basic block for complete optimization

Example – Dependency Analysis

P

```
1 PrepZ q0, 0
2 H q0
3 PrepZ q1, 0
4 H q1
5 foo q0
…
```

foo (q0)

```
6 H q0
7 T q0
8 H q0
```

2. Find "latest" parent instruction for each instruction in *foo*

   lateness defined by sequential program order

foo		latest parent
6 H q0	→	2 H q0
7 T q0	→	6 H q0
8 H q0	→	7 T q0

Example – Parallelism Profiling

P

```
1 PrepZ q0, 0
2 H q0
3 PrepZ q1, 0
4 H q1
5 foo q0
…
```

foo (q0)

```
6 H q0
7 T q0
8 H q0
```

foo		latest parent
6 H q0	→	2 H q0
7 T q0	→	6 H q0
8 H q0	→	7 T q0

3. Picking $\mathbf{x_k = 1}$, starting with first instruction in *foo*:

Select instruction **6: H q0**
Begin search at latest parent instruction + 1:
- **3: H q0** is compatible with **6: H q0**
- Mark this instruction as parallelizable with instruction 3

Select instruction **7: T q0**
Begin search at latest parent instruction + 1:
- Parent is parallelizable, go to new timestep
- **4: H q1** is compatible with **7: T q0**
- Mark this instruction as parallelizable with instruction 4

Select instruction **8: H q0**
Begin search at latest parent instruction + 1:
- Parent is parallelizable, go to new timestep
- No compatible instructions, return 0

Add appropriate edges to graph **G**

First Pass

- Inline all parallelizable lines from each call site
  - Wrap remaining instructions in new function call
  - Does not gather information about re-use of function calls
  - Produces optimal parallelism
  - Analogous to module reconstruction
    - Heavy handed
- Nesting:
  - 46% of SHA-1 parallelism available at depth - 2
  - Complexity does not grow as quickly as previously indicated

Remaining Questions

1. Commutability of algorithm instances on distinct function calls can affect results
   - Consider local combinations of algorithm instances, heuristically choose between them
     - May not produce optimal global schedule
   - Inline directly into other function calls via expansion
     - Could increase code size
     - Most related work in global scope performs scheduling across all basic blocks − expanded function calls
2. Nesting complexity
   - Related work restricts regions to $O(10^3)$ intrinsic instructions

Nesting Complexity

- Restricting to ***contiguously parallelizable subsequences*** → total number of such sequences bounded by the number of intrinsic instructions within a procedure
- Overall complexity of building algorithm graph on a function with nested calls:
  - $O((\textit{length of flattened function})^2) = O(n^2)$
- Full complexity:
  - Still $O(n^3)$ − upper bound still holds
- Assumption: no consideration of changes to nested function calls

Related Work

- Chia-Ming Chang, Chien-Ming Chen, Chung-Ta King, *Using integer linear programming for instruction scheduling and register allocation in multi-issue processors*, Computers & Mathematics with Applications, Volume 34, Issue 9, 1997
  - Early work, operated on single basic blocks
  - Only incorporated interaction between schedules and register spilling
- Wilken, Kent, Jack Liu, and Mark Hefferman. *Optimal instruction scheduling using integer programming."* ACM SIGPLAN Notices. Vol. 35. No. 5. ACM, 2000
  - Highly cited work, again focuses on single basic blocks (no nesting, no combining of basic blocks)
- Winkel, Sebastian. *Optimal versus heuristic global code scheduling*. Proceedings of the 40th Annual IEEE/ACM MICRO. IEEE Computer Society, 2007.
  - One of the only attempts to use ILP's to schedule code globally across basic blocks
  - Only considers *regions* of < 1000 instructions (including nesting)

Overview of Modeled Optimizations

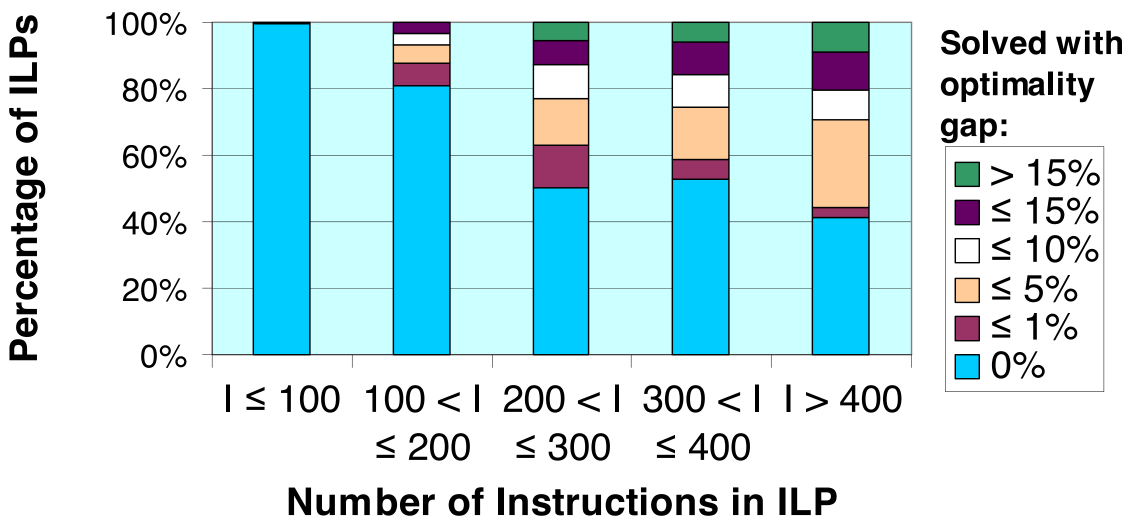
- Global code motion:
  - Directions: upward, downward
  - Control conditions: predicated, speculative
  - Boundaries: across, into, and out of loops (cyclic)
  - Enablers: renaming, compensation copies
  - Global propagation of non-unit latencies
- Supported speculation features:
  - Control- and data-speculative loads
  - Partial-ready code motion, compare speculation
- Block model:
  - Block emptying and collapsing
  - Resulting multiway branch generation
  - Choose fall-through edges and block order

(Highlighted: new or significantly improved parts vs. previous work)

ILP scheduler can resolve all interdependences between these optimizations and deliver a global optimum

ILP Solvability

- 625 (first pass) ILPs solved on a 1.6 GHz Itanium® 2
- Standard scheduling region size limit of 500 instructions
  - For hard-to-solve routines, decremented in steps of 50 until the ILPs can be solved within 4 hours



Closeness to optimality that can be proven by the solver

382 ILPs, average sol. time 6s

88 ILPs, average sol. time 20 minutes, average ILP size 8257 constraints x 5225 variables

Possible Heuristics

- $\text{Temperature}_{\alpha i}$:
  - *cycle_ratio ÷ size_ratio*
  - $cycle\_ratio = freq_{\alpha i} \div freq_{\alpha} \times cycle\_count_{\alpha} \div total\_cycles$
  - Estimates "contribution of call graph edge to whole application", weighted by size ratio
  - Small, "hot" functions that are called often in a procedure prioritized
  - Can use *cycle_density* to identify looped func calls

- $\text{Cost}_{\alpha i}$:
  - *parallelizable_lines ÷ number_of_lines*
- Combine temperature and cost analysis on each function call instance → inlining decision