The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs

Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, Jason Anderson

IEEE FCCM 2013

Seattle, Washington, April 29, 2013

Dept. of Electrical and Computer Engineering
University of Toronto

Motivation

- Compilers execute optimization passes:
  - dead code elim, constant propagation, loop rotation, inlining, exlining, etc.
- LLVM incorporates >50 different passes
- The familiar optimization levels –O2, -O3 correspond to recipes of such passes
- The levels are designed to optimize SW not HW
- **Question:** Which passes are good for HLS?

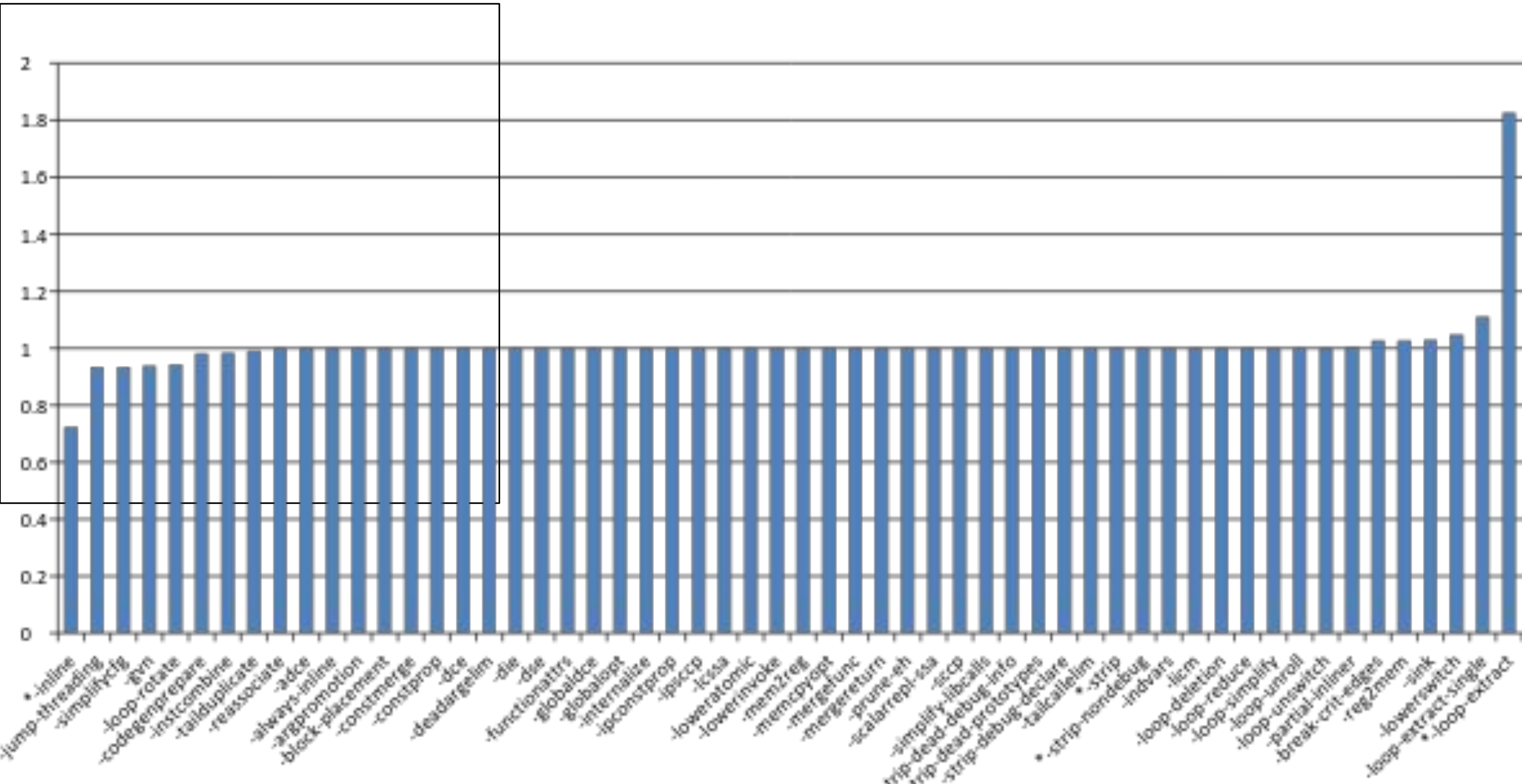
Analysis of Compiler Passes

- We applied each pass (of 56) in isolation
- Compare with –O0 (no optimization)
- LegUp: Open-source high-level synthesis
  - http://legup.eecg.toronto.edu
- Target Cyclone II FPGA
- 11 CHStone benchmarks
  - Standard high-level synthesis suite
  - http://www.ertl.jp/chstone

Analysis of Compiler Passes (2)

- Consider 4 hardware metrics:
  - Cycle latency (# of clock cycles)
  - FMax
  - Wall-clock time (= Cycle latency / FMax)
  - # of LEs (area)
- Large # of ModelSim + Quartus II runs
  - 56 x 11 = 616 synthesis, place, route runs
  - Used SciNet (Canadian cloud computing system)

Representative Results: Cycle Latency



Overall Results Summary

Across 56 passes

	Clock cycles	FMax	Wall-clock time	LEs
Min.	0.72	0.76	0.92	0.93
Max.	1.83	1.05	2.24	1.34
St. Dev.	0.12	0.04	0.17	0.05

- Widest swings observed in **clock cycles, wall-clock time.**

Custom Recipes

- Create a recipe of passes to the "taste" of each CHStone benchmark
- For each benchmark (11), for each HW metric (4), create a recipe containing **just** the beneficial passes (in alphabetical order)

Custom Recipes

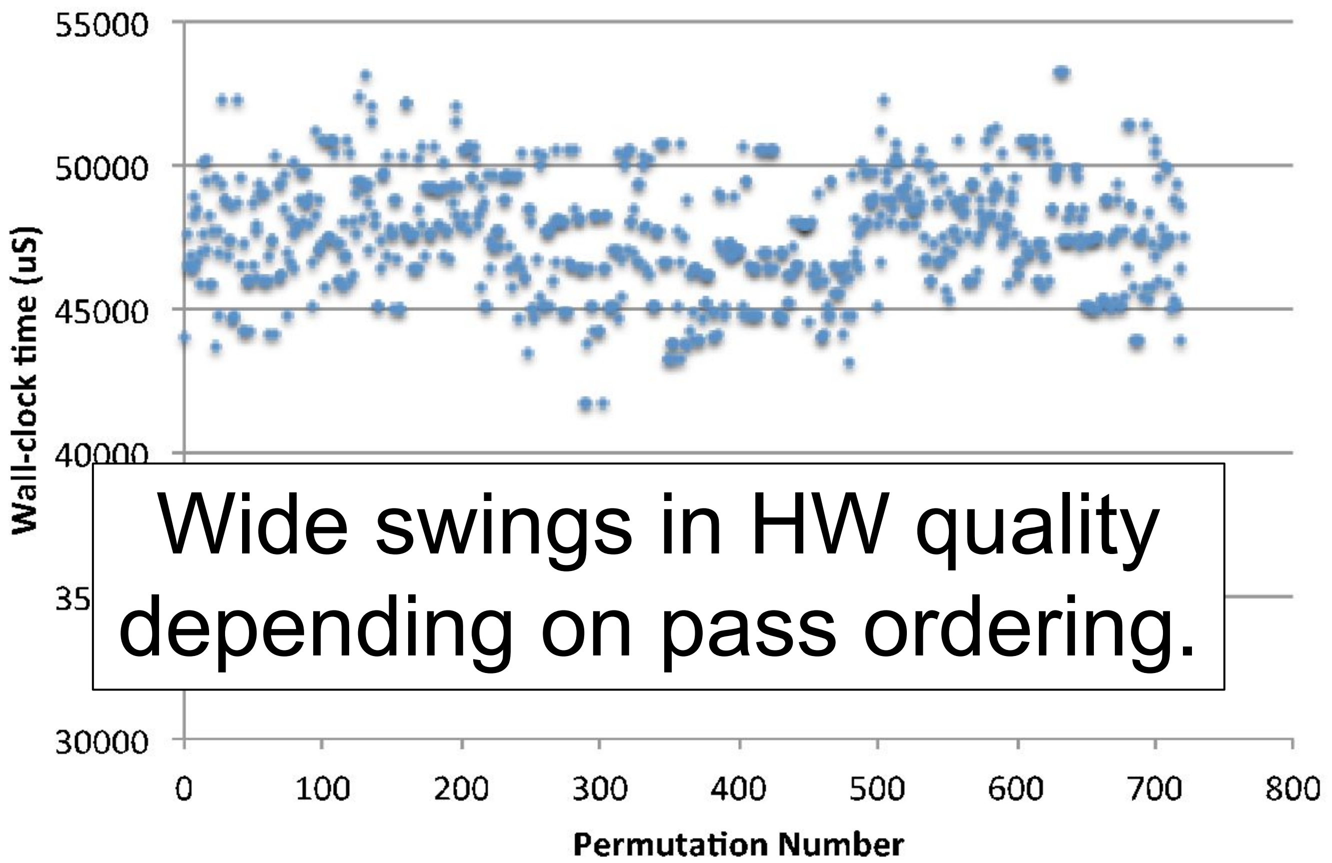
- Sample results for `dfmul` benchmark:

Recipe	Clock cycles	FMax	Wall-clock time	LEs
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Clo	[illegible]	[illegible]	[illegible]	2
FMax	1.42	**1.02**	1.39	1.29
Wall-clock time	0.92	1.01	**0.91**	0.93
LEs	1.02	0.99	1.02	**0.91**

Significant potential to improve upon –O3 optimization!

Order Dependence

- Consider **all** orderings of 6 passes shown beneficial to wall-clock time for `jpeg` benchmark (6! = 720 runs)



Wide swings in HW quality depending on pass ordering.

Dynamic Pass Recipes

- “Good” passes to apply:
  highly benchmark dependent
- Impact of passes highly order dependent
- **We need a dynamic feedback-driven approach**

Early Prediction of Cycle Latency

- Basic block (BB): a section of code with a single entry/exit point.

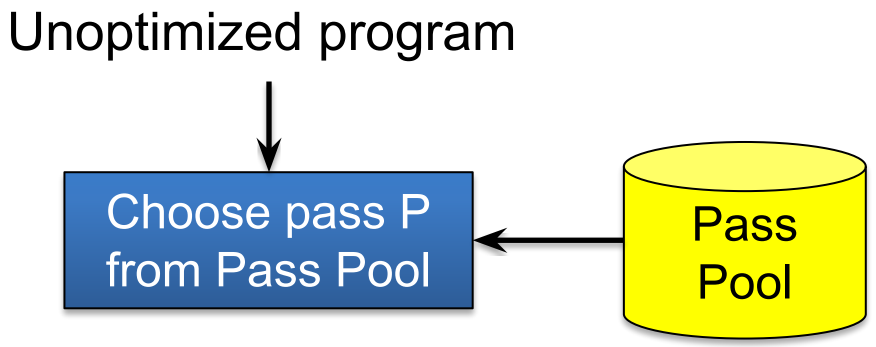
We do not need to execute synthesis, place, route to determine cycle latency!

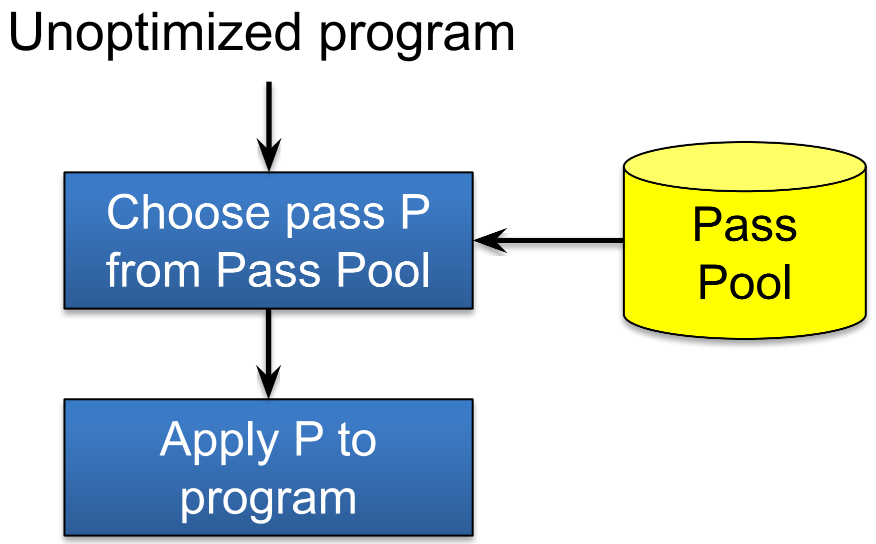
- Through partial HLS, we can determine the schedule length for each BB.

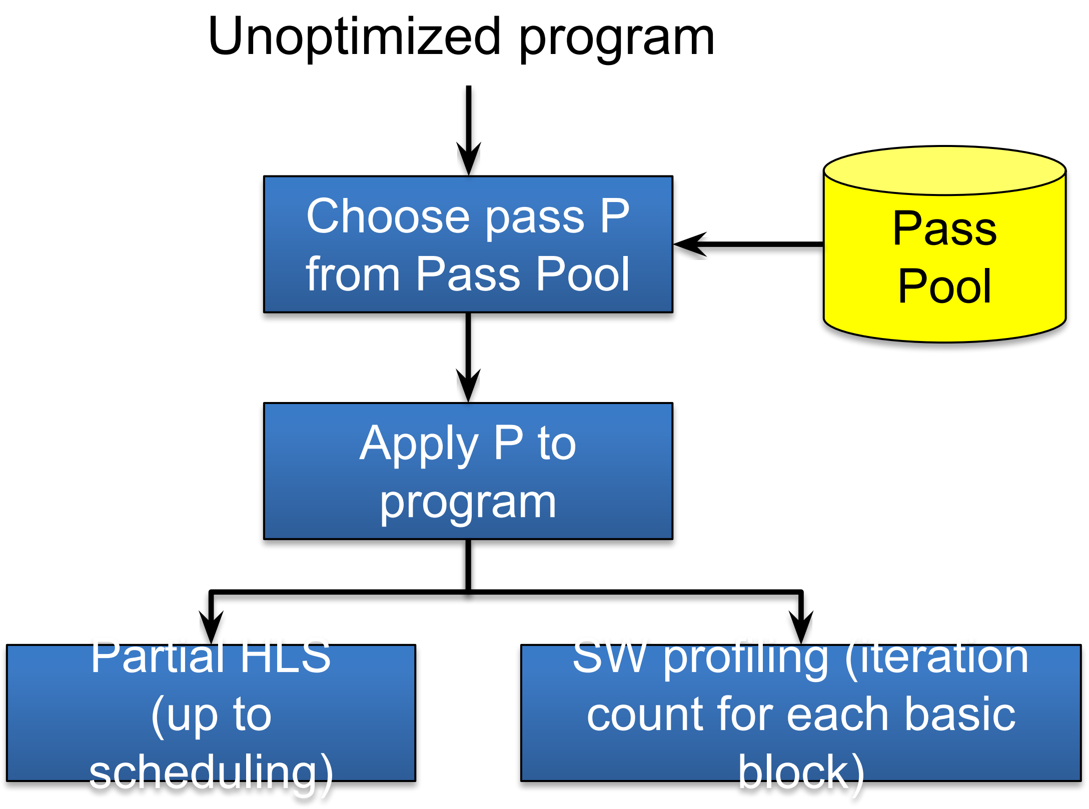
$$CycleLatency = \sum_{b \in BasicBlocks} \#Execs(b) \times SchedLength(b)$$

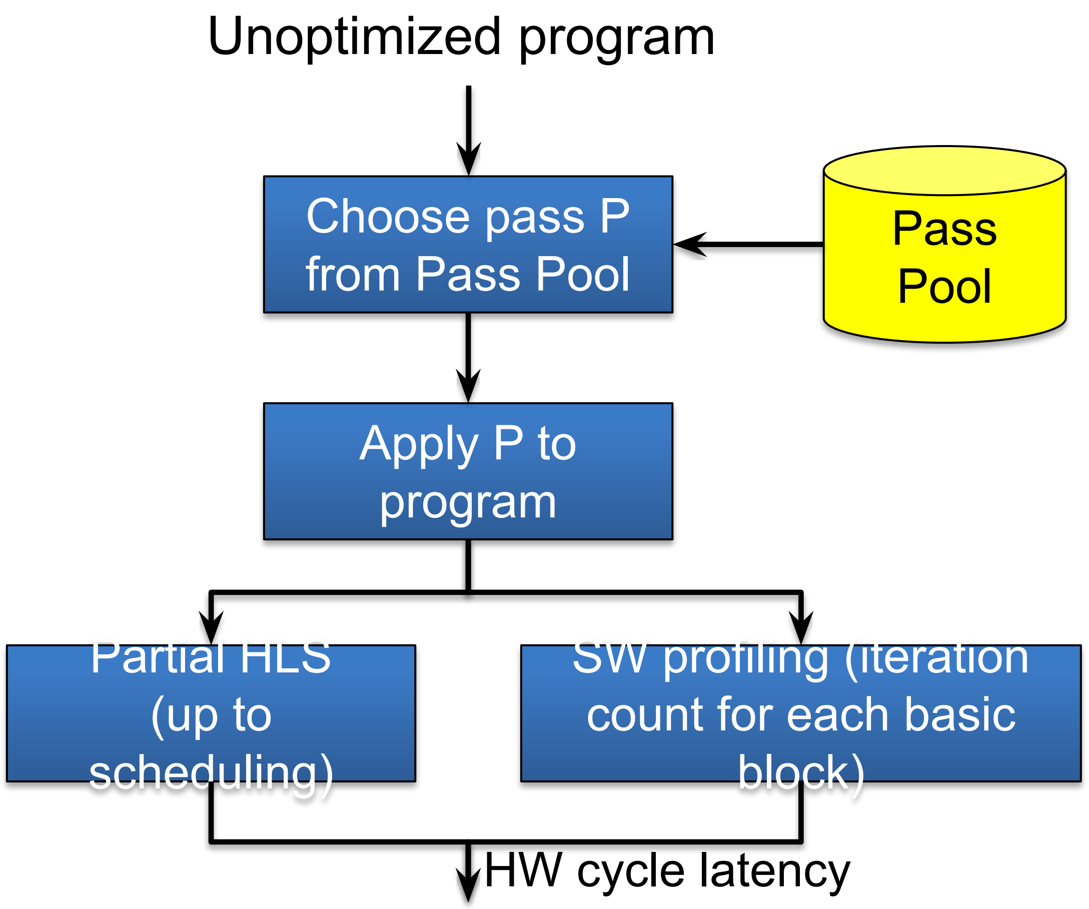
Dynamic Optimization Flow

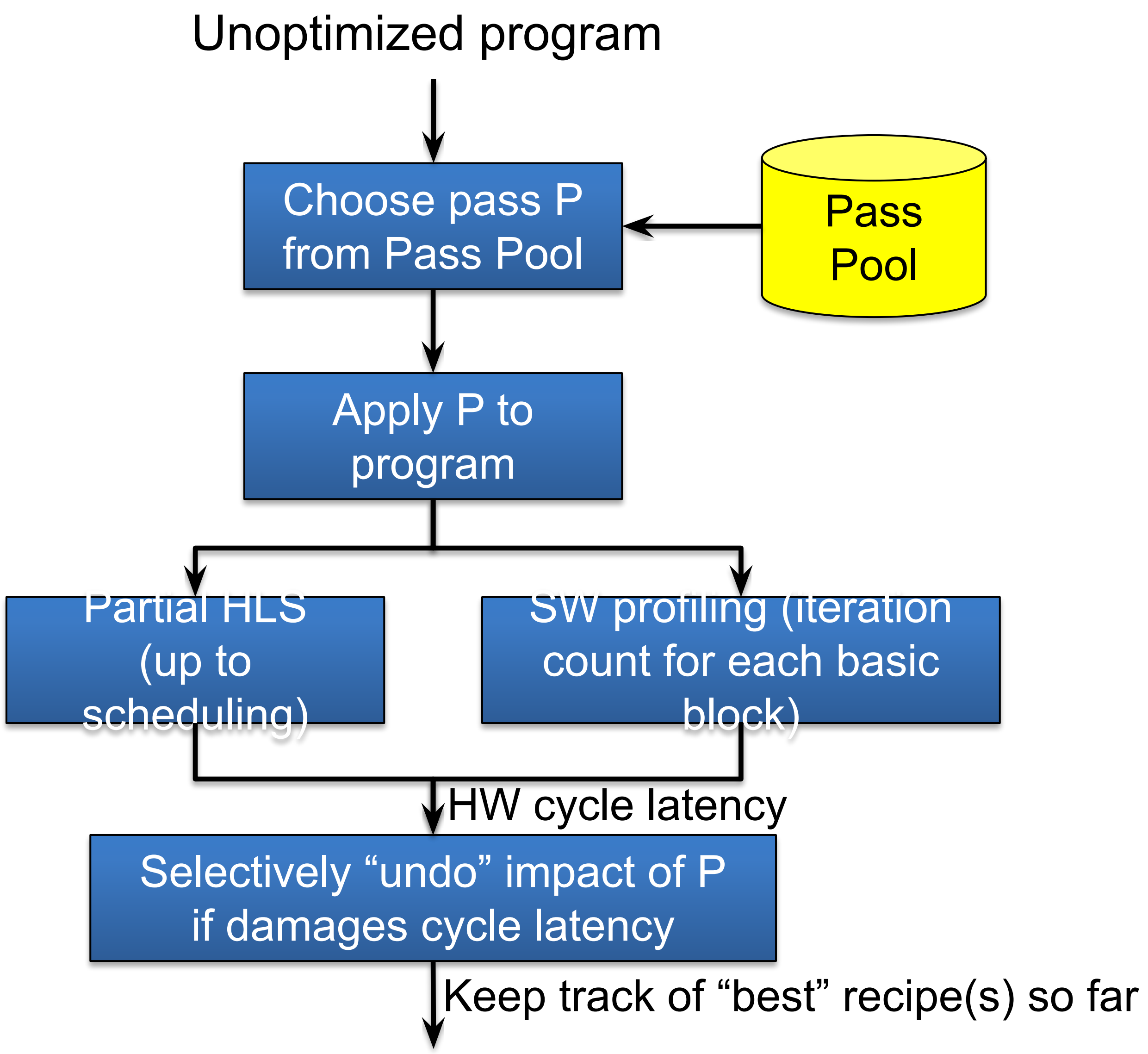
Unoptimized program

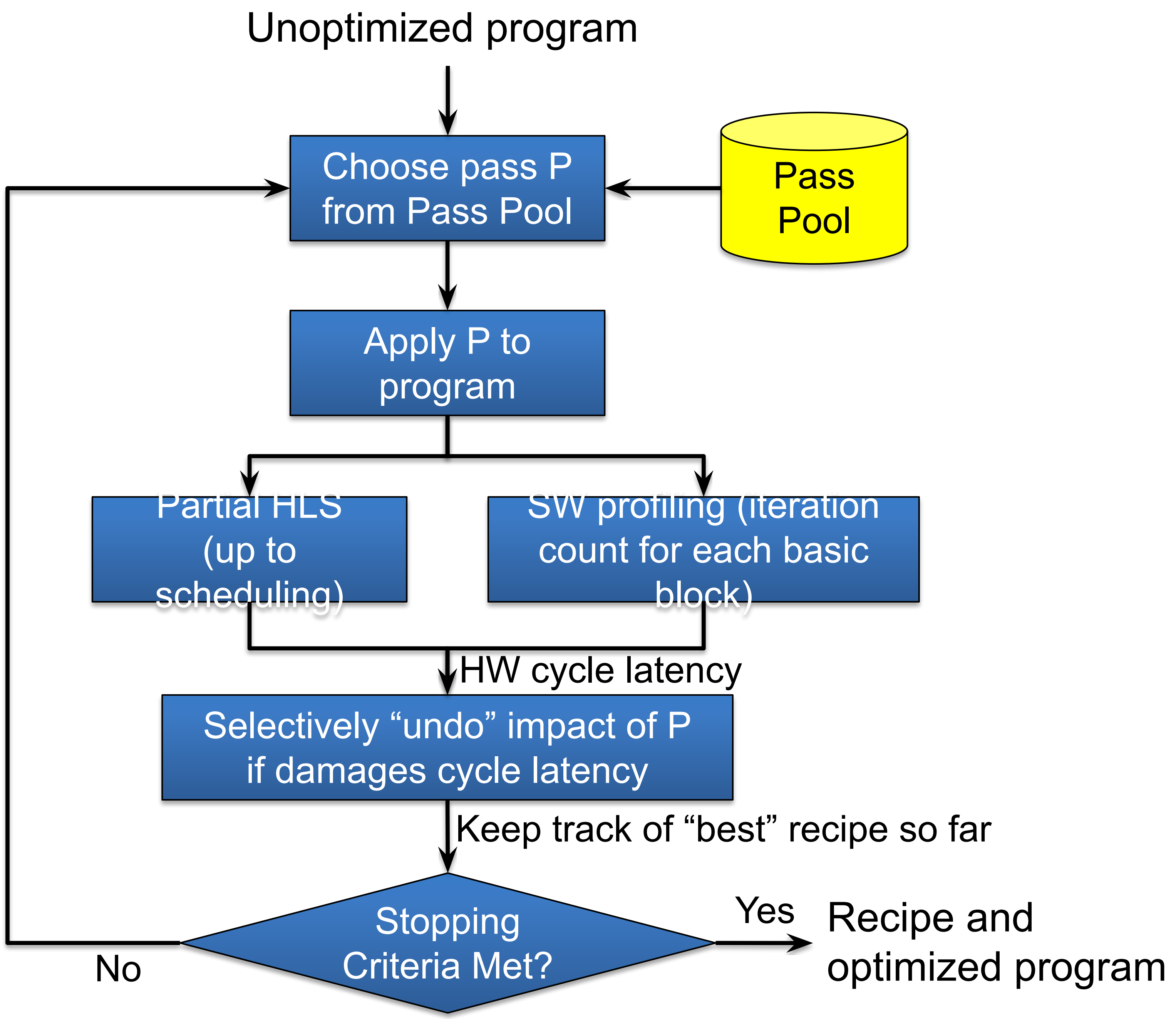

Unoptimized program
Choose pass P from Pass Pool
Pass Pool


Unoptimized program
Choose pass P from Pass Pool
Pass Pool
Apply P to program


Unoptimized program
Choose pass P from Pass Pool
Pass Pool
Apply P to program
Partial HLS (up to scheduling)
SW profiling (iteration count for each basic block)


Unoptimized program
Choose pass P from Pass Pool
Pass Pool
Apply P to program
Partial HLS (up to scheduling)
SW profiling (iteration count for each basic block)
HW cycle latency


Unoptimized program
Choose pass P from Pass Pool
Pass Pool
Apply P to program
Partial HLS (up to scheduling)
SW profiling (iteration count for each basic block)
HW cycle latency
Selectively “undo” impact of P if damages cycle latency
Keep track of “best” recipe(s) so far


Unoptimized program
Choose pass P from Pass Pool
Pass Pool
Apply P to program
Partial HLS (up to scheduling)
SW profiling (iteration count for each basic block)
HW cycle latency
Selectively "undo" impact of P if damages cycle latency
Keep track of "best" recipe so far
Stopping Criteria Met?
Yes
Recipe and optimized program
No

Pass Pool

- Contains 41 of 56 passes as follows:
  - Any pass that showed impact on any benchmark vs. no optimization (-O0)
  - Any pass in –O3 that showed impact on any benchmark when removed from –O3
  - Passes not in –O3 that showed no impact in isolation (they may show impact when combined)
- Passes ordered based on “pairs” analysis
  - Analyzed all pairs of passes to determine “preferred” order

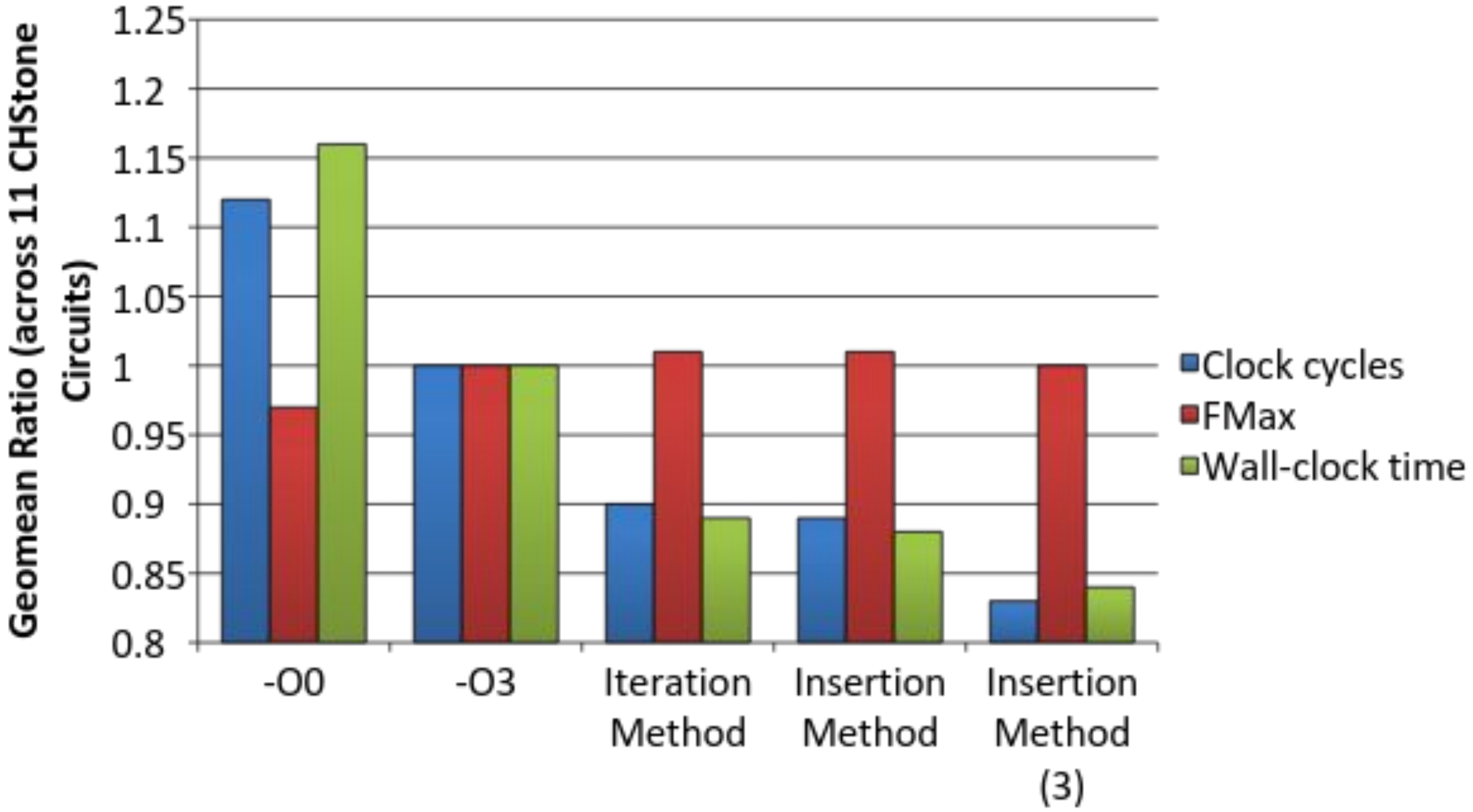
Applying a Pass: 3 Ways

1. **Iteration method:**
   i) Passes in pass pool are sorted
   (based on pairs analysis).
   ii) Selected pass is added to end of “best” recipe
2. **Insertion method:** all possible insertion points in “best” recipe are considered for each selected pass
   – Order independent
3. **Insertion method (3):** same as #2, but store top 3 “best” recipes encountered

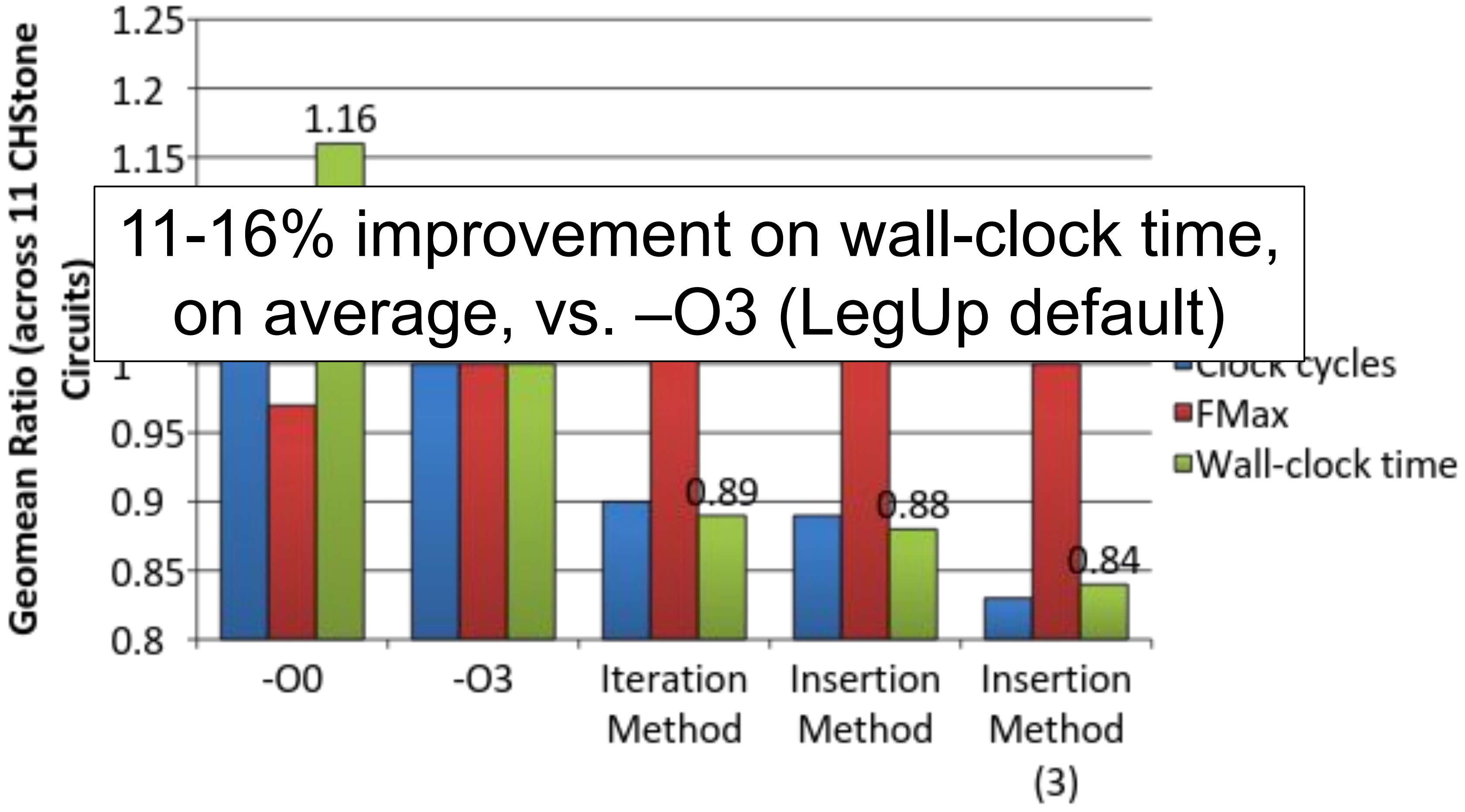
Stopping Criteria

- Stop if no “walk” through pass pool improves results
- At most, 3 “walks” through the pass pool

Results (Speed)

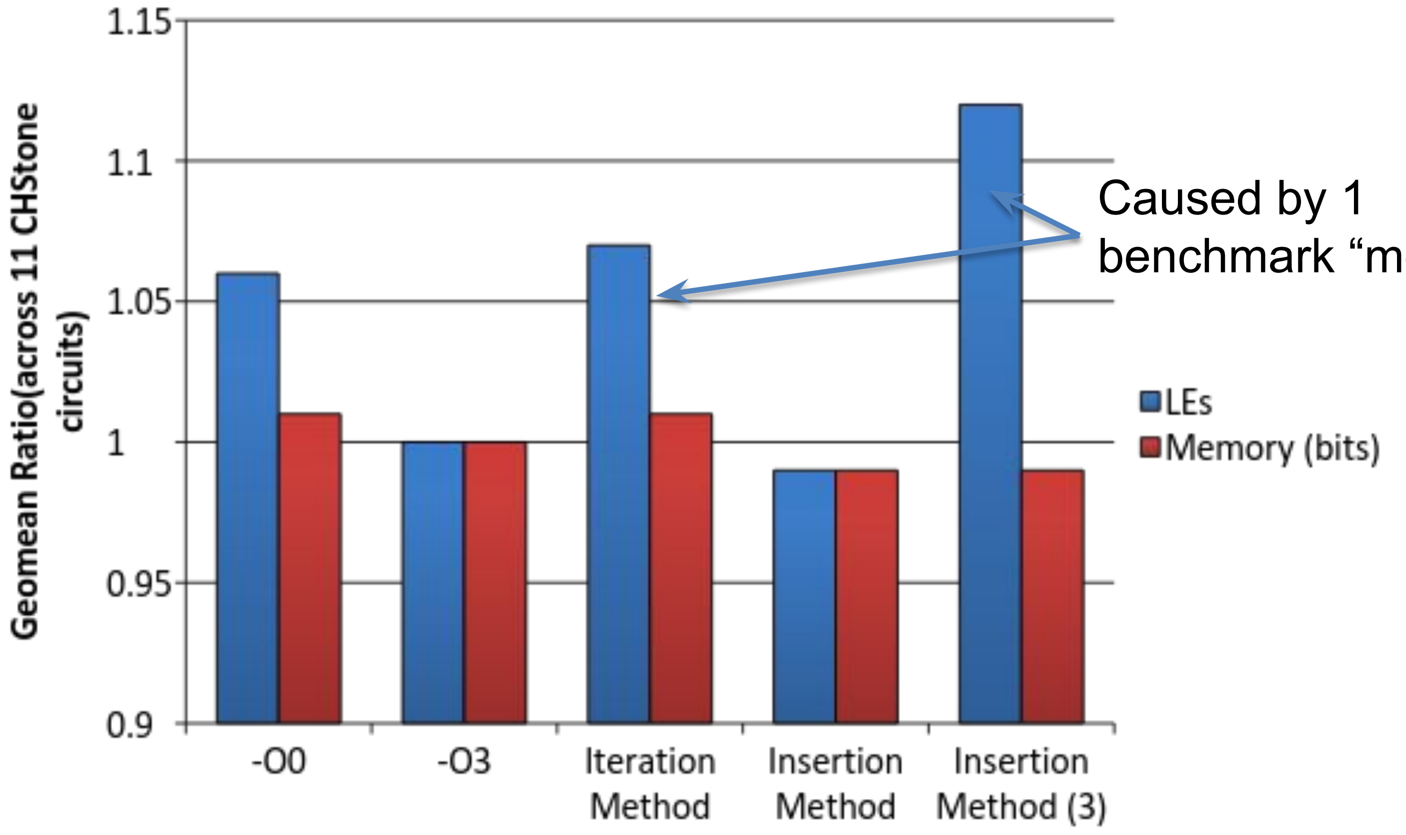


Results (Speed)



11-16% improvement on wall-clock time, on average, vs. –O3 (LegUp default)

Results (Area)



Caused by 1 benchmark “motion”

Summary

- Compiler passes have strong influence on quality of HLS-generated HW
- Impact of passes highly order dependent
- Significant potential to raise HW quality via dynamic feedback-driven approach:
  - Circuit wall-clock time can be reduced by 11-16%, on average, depending on run-time allowed (vs. –O3 recipe)

Future Work

- Our approach is somewhat “brute force”
  - Heuristics to “predict” which passes benefit certain code styles
- Apply different pass recipes to different parts of code
- New passes specifically targeted to HLS hardware generation

Questions?

legup.eecg.toronto.edu