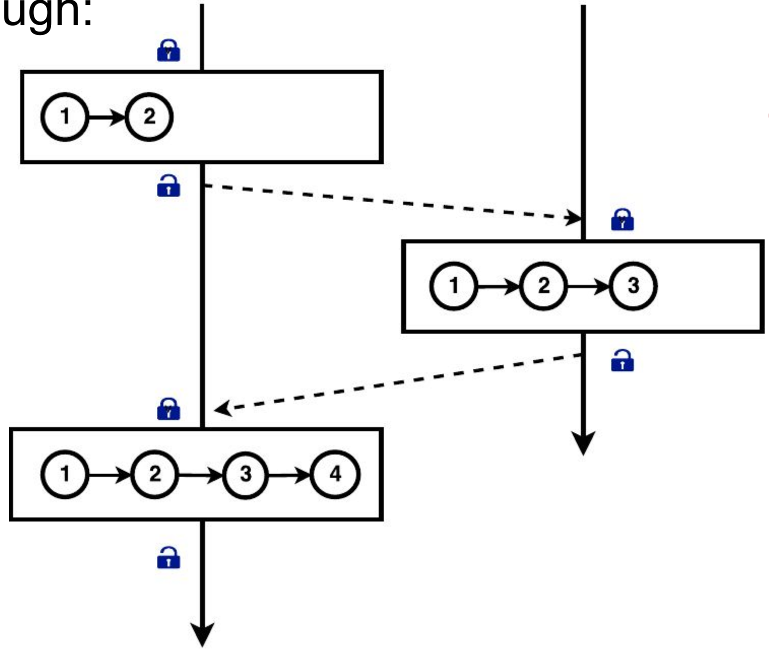
Approximating Data Movement via Compiler Support

Junhan Zhou
Vignesh Balaji

The Problem

Data movement is a major bottleneck for performance scaling of parallel programs

Cores communicate through:



Synchronization

The blue locks ensures consistent updates to data values by multiple threads

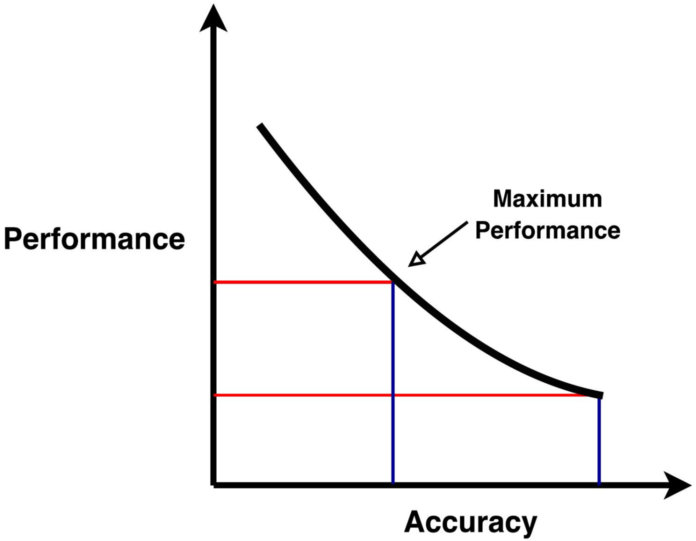
Cache coherence

The arrows shows data movement between cores via cache coherence

Potential Solution

Data movement can be reduced by writing better code (smaller critical regions, better synchronization strategies, etc.)

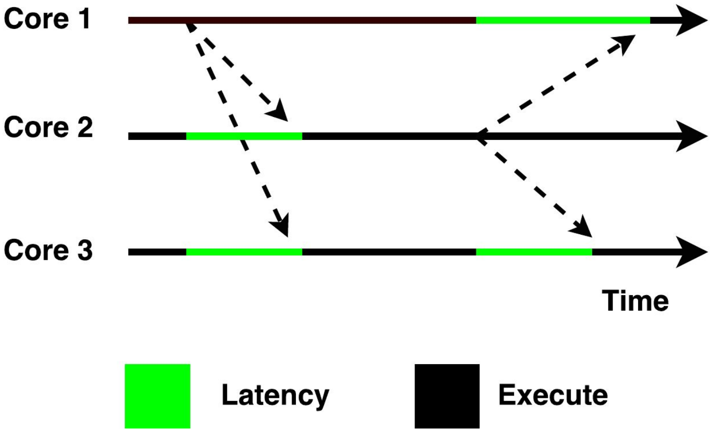
OR we could use Approximate Computing!



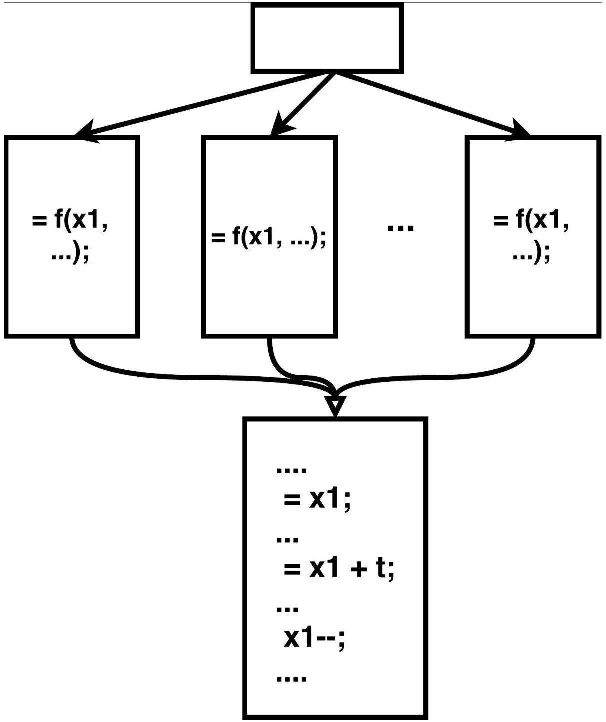
People approximated synchronization before...

But data movement still exists for coherence...

So we try approximating cache coherence! (Incoherence cache)



Heuristic 1: Estimating importance of Program variables

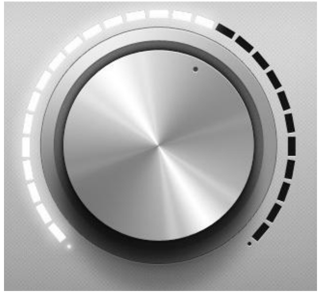


- Count the number of uses of variables that are *shared* among different threads

- Variables that are used less might be better approximation targets (lead to lesser quality degradation)

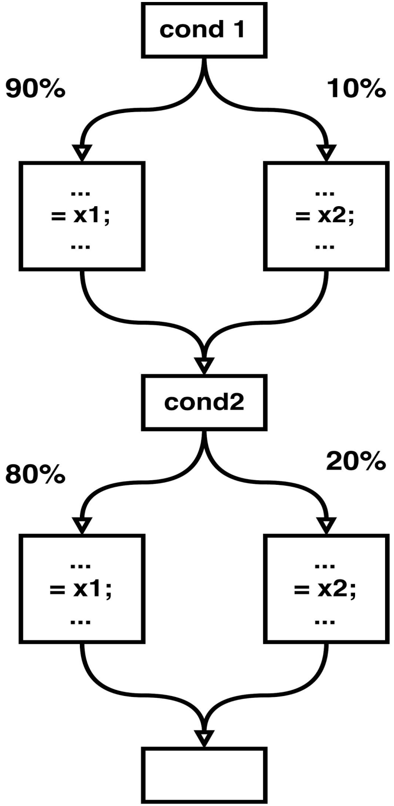
- Knob:

Uses Threshold = 1



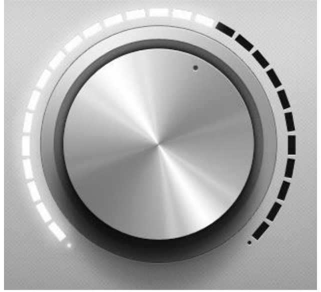
Uses Threshold = n

Heuristic 2: Quality impact of approximating variables



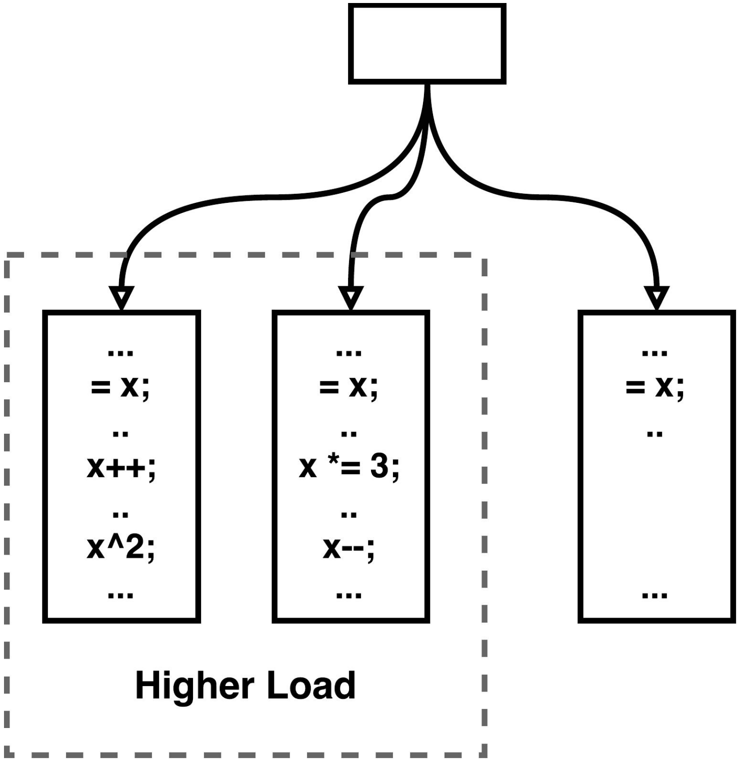
- ➢ Track if a variable is present on all paths at every meet point
- ➢ Variables that appear only on few paths are not likely to cause significant quality degradation
- ➢ Knob:

Path freq. < 0%



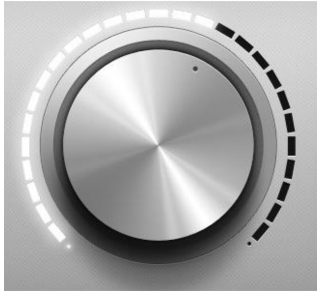
Path freq.<100%

Heuristic 3: Detecting imbalance in computation



- Detect the amount of computation on variables in different threads
- Variables that are heavily operated upon only in a few threads do not need coherence for *all* threads
- Knob:

Uses Threshold = 1



Uses Threshold = n

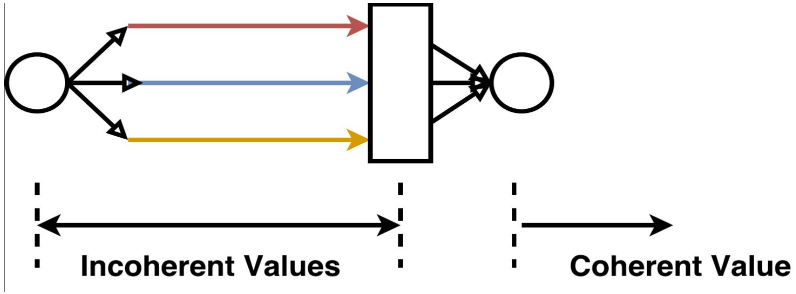
Implementation Details

First identify potential shared variables to approximate (global variables with load-modify-store pattern inside locks that does not determine control flow)

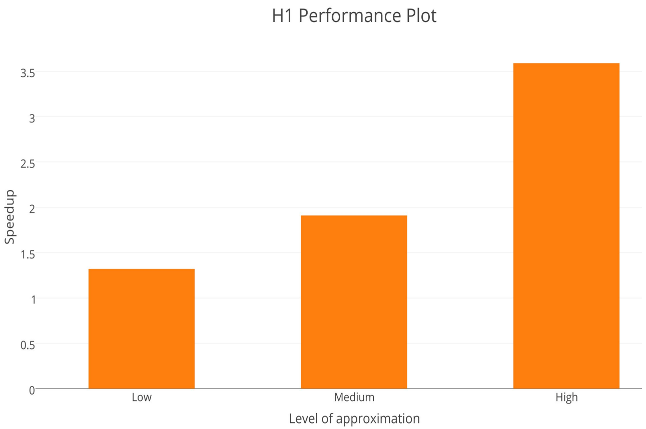
Then use the mentioned heuristics to select the shared variable to approximate.

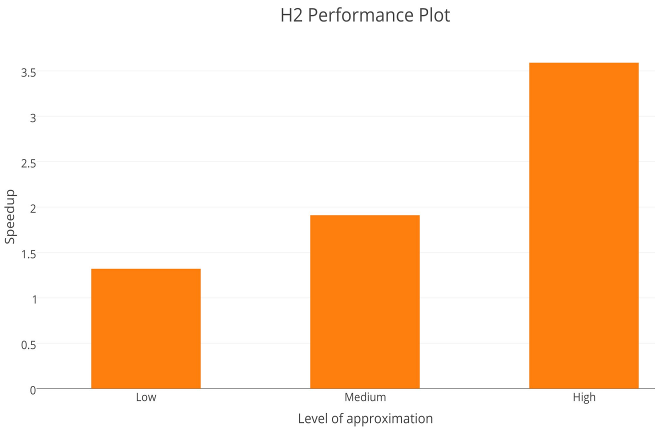
Simulated the performance effect of incoherent caches by not adding cycles for coherence actions (directory lookup, invalidations, etc.)

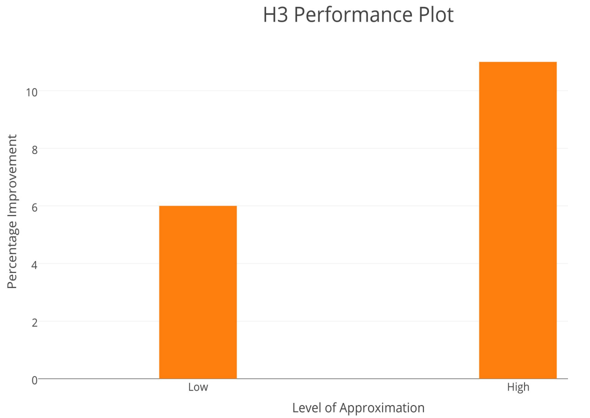
For error numbers, manipulated thread local copies of data followed by an approximate merge of values

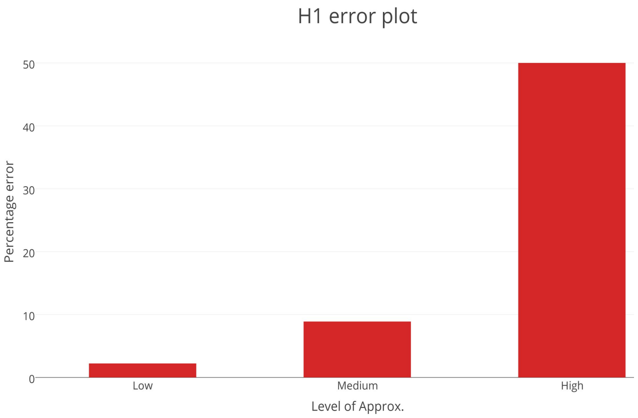


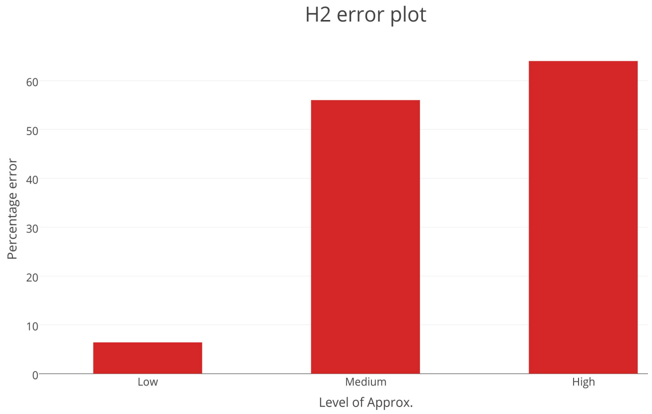
Results


H1 Performance Plot
Speedup
0
0.5
1
1.5
2
2.5
3
3.5
Low
Medium
High
Level of approximation


H2 Performance Plot
Speedup
0
0.5
1
1.5
2
2.5
3
3.5
Low
Medium
High
Level of approximation


H3 Performance Plot
Percentage Improvement
0
2
4
6
8
10
Low
High
Level of Approximation


H1 error plot
Percentage error
0
10
20
30
40
50
Low
Medium
High
Level of Approx.


H2 error plot
Percentage error
0
10
20
30
40
50
60
Low
Medium
High
Level of Approx.


H3 error plot
Percentage error
0
5
10
15
20
25
30
35
Low
High
Level of Approx.

Limitations and Future Work

Our pass can be improved by incorporating:

- ➢ Code annotations (where the programmer identifies approximable parts of code [1][2][3])
- ➢ Feedback from code profiling to get an idea on the dynamic aspects of the code [3]

[1] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11)

[2] Sampson, Adrian, et al. "Accept: A programmer-guided compiler framework for practical approximate computing." *University of Washington Technical Report UW-CSE-15-01* 1 (2015)

[3] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (OOPSLA '14)