Clairvoyance: Look- Ahead Compile Time Scheduling

Parth Oak
Jiaqing Ni
Joseph Sorenson

Nov. 28, 2018

Agenda

- What Clairvoyance is
- Why it is used
- Utilization challenges and solutions
- When to not use it
- Experimental results
- Conclusion

Introduction/Problem

- Innovation of hardware comes at a cost
- Fast processors
  - Power hungry
- Efficient energy usage
  - Slow processing
- More memory-bound application
  - Requires more aggressive engine
- Clairvoyance
  - Uses simple out-of-order (OoO) core

Clairvoyance: Why Use It?

- Balances performance and energy efficiency
- Hides memory latency
  - Masks memory operation dependency

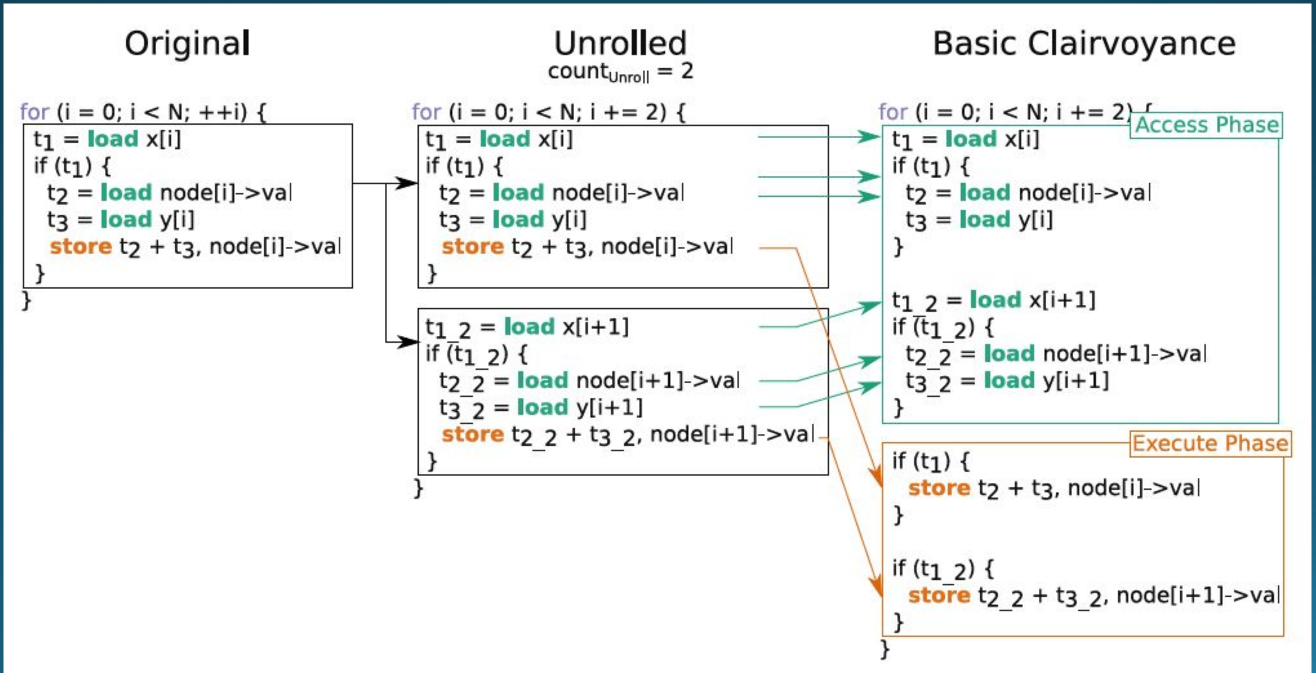
Major issue: Slow loads

```
1.  R4 = Load(R2)
2.  R5 = Load(R3)
3.  R6 = R5 + 7
4.  R7 = R6 * 2
5.  R8 = R4 - 3
```

- Some loads have high latency (i.e., cache misses)
- All instructions that depend on load are blocked
- Example:
    - Instruction 3 is blocked until Instruction 2 finishes
    - Instruction 5 is blocked until Instruction 1 finishes
    - Latter case is more ideal (more time for load to finish)

Access and Execute Phases

- Unroll main loop 2^n times
- Split all instructions into 2 phases:
  - Access phase has all of the important loads
  - Execute phase has everything else
- Dependent instructions are far away from their loads



Challenges

- Simply hoisting loads doesn't always work
- Challenges:
  1. Critical loads
  2. Unknown aliasing
  3. Load chains
  4. Instruction count overhead

Challenge #1: Critical Loads

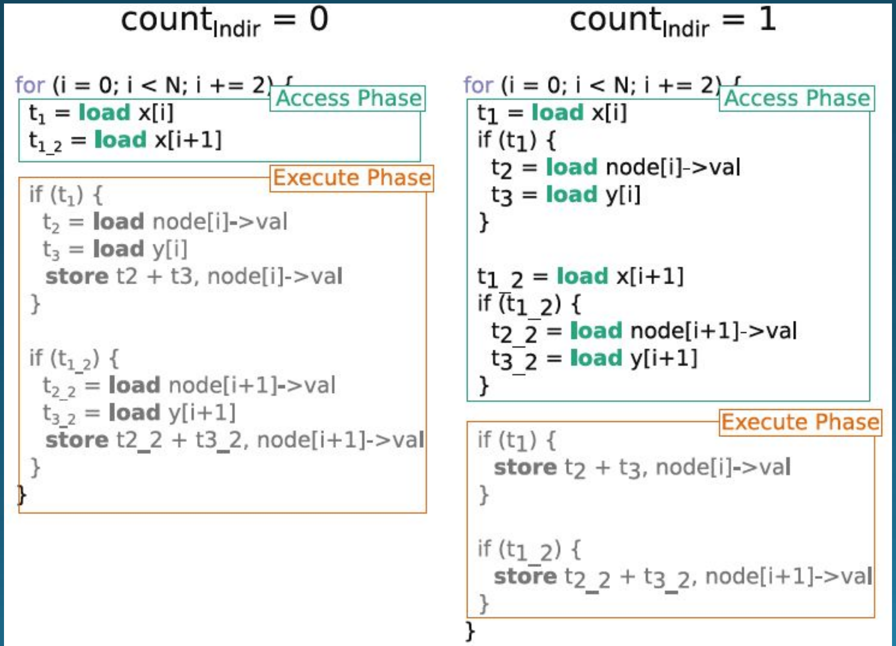
- Some loads if hoisted, are not as beneficial as others
- Hoisting every load bloats the code
- The longer the dependency chain, the more work needs to be done to hoist

```
1.  R4 = Load(R2)
2.  R5 = Load(R3)
3.  If R5 == 0, jump to 8
4.  R6 = Load(R5)
```

```
Hoisting instruction 4
introduces more code, and
it's not always executed.
```

Challenge #1 Solution

- Indirection Count: # memory accesses to reach a load
  - Example: x[y[z[i]]] has indirection count = 2
  - Higher indirection counts are harder to prefetch
  - Prefetch only loads with small indirection values



Challenge #2: Unknown aliasing

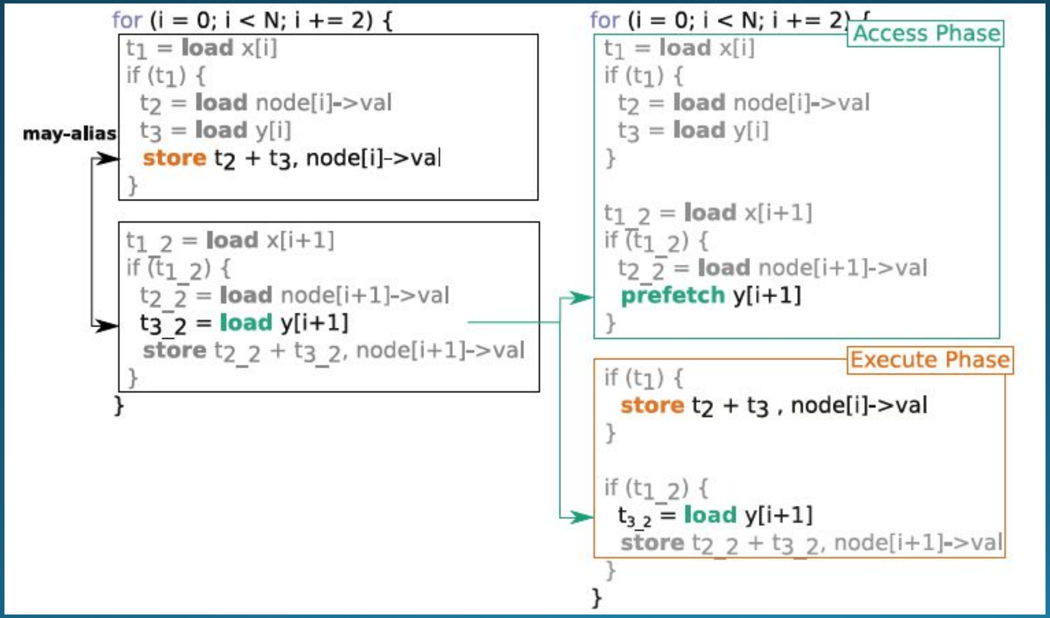
- Some memory operations alias to the same location
- If hoisted, changes program behavior
- Traditional RAW handling:
  - Read-after-write dependencies

    1. Store(R1, R2) # R1: addr
    2. R4 = Load(R3)

    (Possible alias: R3 = R1; DON'T HOIST)
  - Not too much can be done

Challenge #2 Solution

- Prefetch in Access phase anyway
  - Assume no aliasing
- If no alias, then use prefetched value
- If aliasing occurs, re-load to get correct value

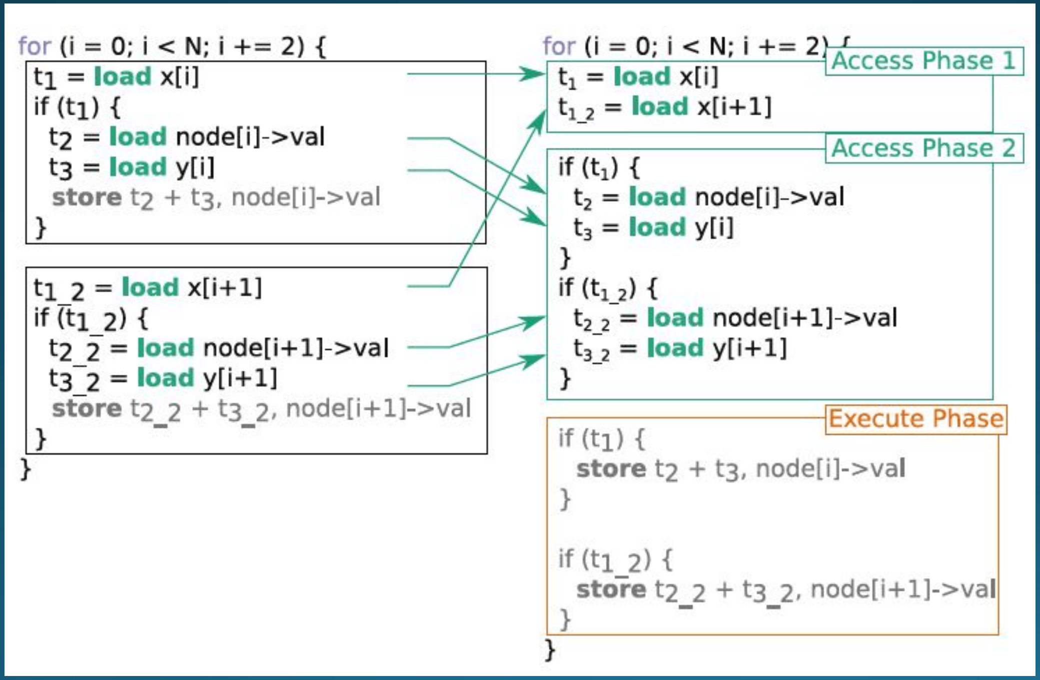


Challenge #3: Load Chains

- Load Chain (similar to challenge #1):
  1. R2 = Load(R1)
  2. R3 = Load(R2)
- If both loads are hoisted to access, then the second load still is blocked by the first.
- Need to separate dependent loads

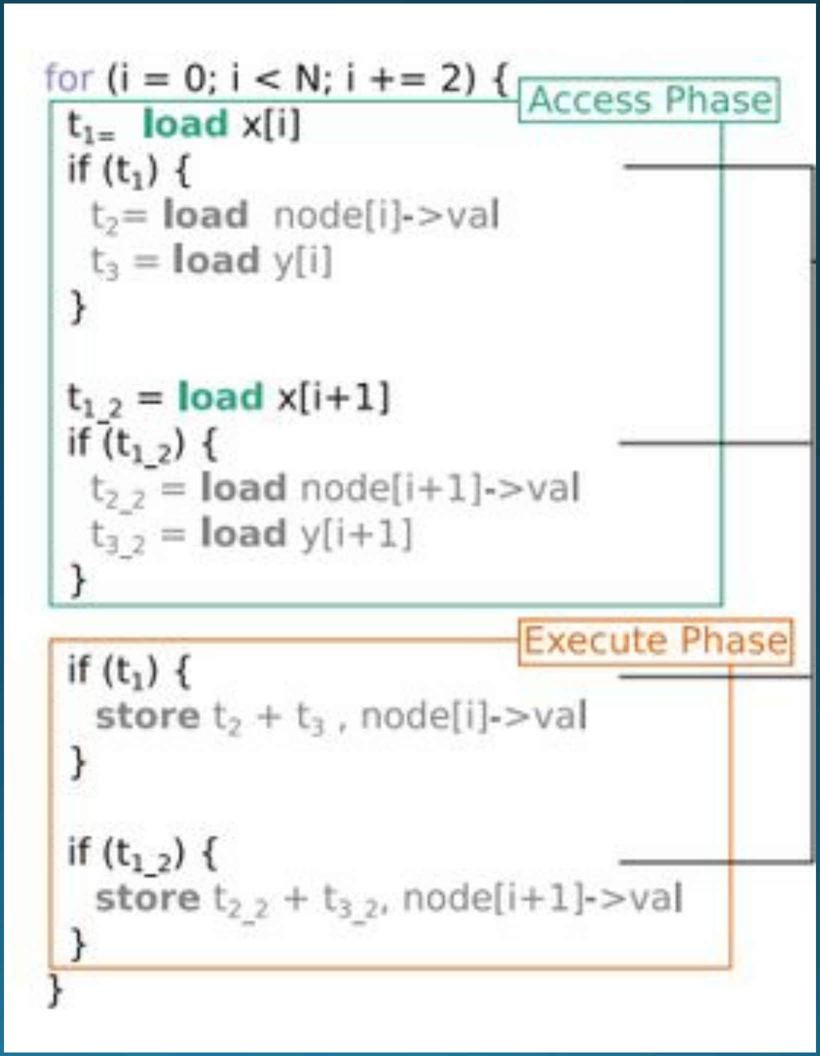
Challenge #3 Solution

- Multiple access phases
  - Each dependent load goes in a different access phase
  - Separate dependencies as much as possible



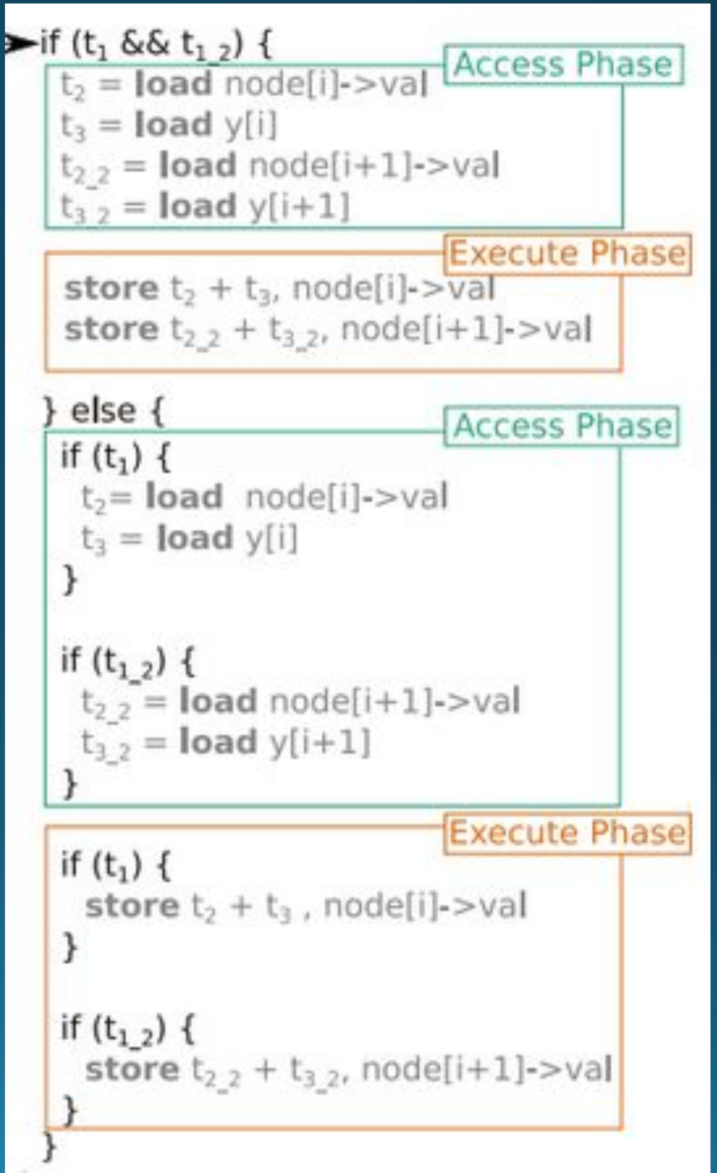
Challenge #4: Instruction Count Overhead

- Branching instructions are duplicated in Access and Execute phases.
- Increases complexity of control flow graph.



Challenge #4 Solution

- Check for situation where all predicates are true.
- Create special Access and Execute phases for this scenario.
  - Merge control flow
- Default to normal Access and Execute otherwise.

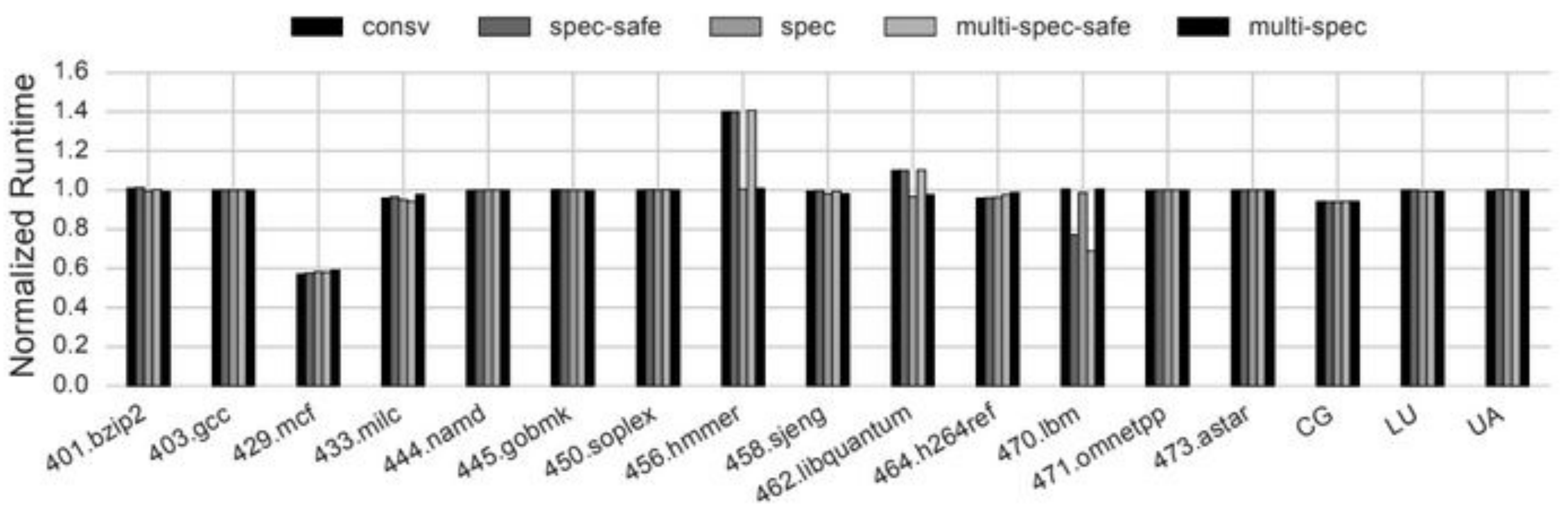


Disabling Clairvoyance

- Even in the best case, may get worse performance
  - Code bloat
  - Branch Complexity
- Determine ahead of of time whether to use Clairvoyance transformations.
- Works better with more loads and less branches
- Heuristic: if #loads / # branches < 0.7, then disable transformations.

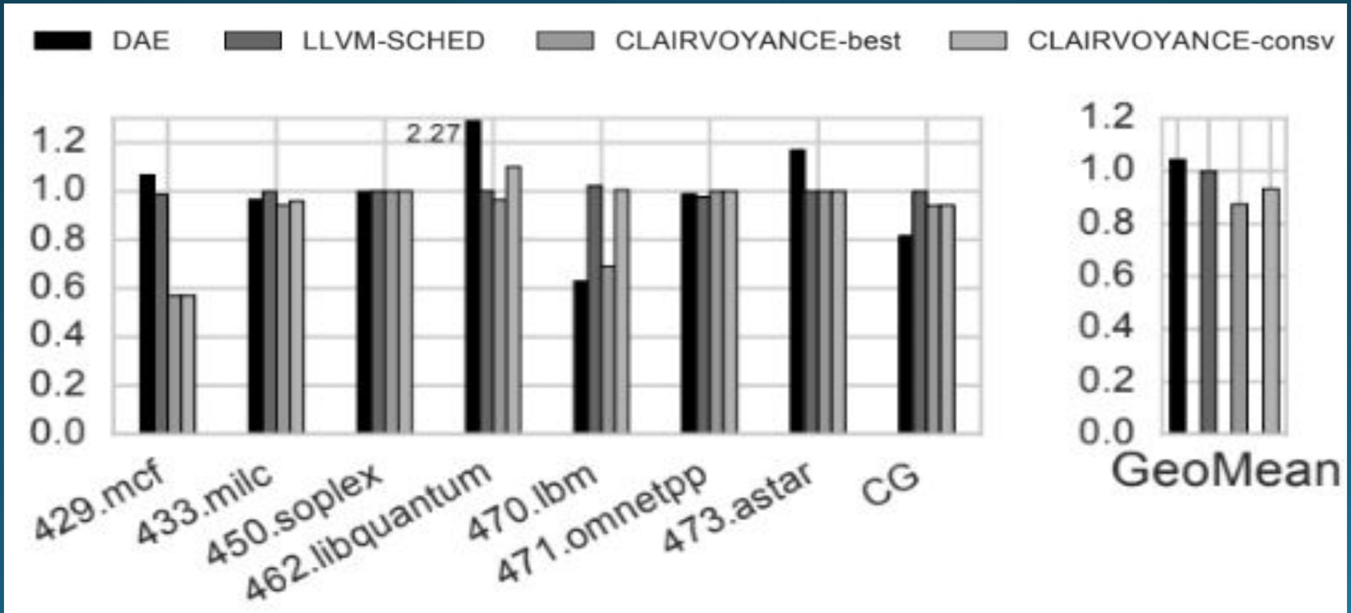
Experimental Results

- Clairvoyance has multiple different settings
  - Experiments compare conservative optimization with more speculative optimization
- Different settings perform better for different workloads
  - No clear best setting



Experimental Results

- Compare Clairvoyance runtime to state of the art systems:
  - Clairvoyance (conservative settings)
  - Clairvoyance (best settings for individual workload)
  - DAE
  - Optimal LLVM scheduler
- Clairvoyance has overall superior performance



Conclusion

- Addresses memory latency
  - Unrolls main loop
  - Lifts loads to separate access and execution phase

- A geomean execution time improvement for memory-bound applications of 7% - 13%.

- Performance improvements of up to 43%

Thank You

Any Questions?