Clairvoyance: Look-Ahead Compile-Time Scheduling

Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos,
Magnus Själander, Vasileios Spiliopoulos,
Stefanos Kaxiras, Alexandra Jimborean

Presenters: Lee Moore, Nicholas Wilson, Jiong Zhu, Do June Min

Memory latency in memory-bound applications

```
for (i=0; i<N; i++) {
   t1 = load x[i]
   if(t1) {
      t2 = load node[i]->val
      t3 = load y[i]
      store t2 + t3,
            node[i]->val
   }
}
```

- Chains of dependent loads and stores
- Last-level cache misses (LLC) can cost hundreds of cycles

How can we hide memory latency?

- Hardware-level Out-of-Order (OoO) Execution Engine
  - Aggressive OoO Engine needs aggressive energy

Fast and power-hungry OoO

Simple but energy-efficient OoO

Could compile-time scheduling help in hiding memory latency?

How can we hide memory latency?

- Compile-time scheduling

Challenges to handle:

- Insufficient independent instructions
- Chains of dependent loads
- Statically unknown dependencies
- Register pressure

```
for (i=0; i<N; i++) {
   t1 = load x[i]
   if(t1) {
      t2 = load node[i]->val
      t3 = load y[i]
      store t2 + t3,
            node[i]->val
   }
}
```

Introducing Clairvoyance

- Goal: **hide memory latency** **for simple OoO engine** in compile time.
- How to approach this goal?

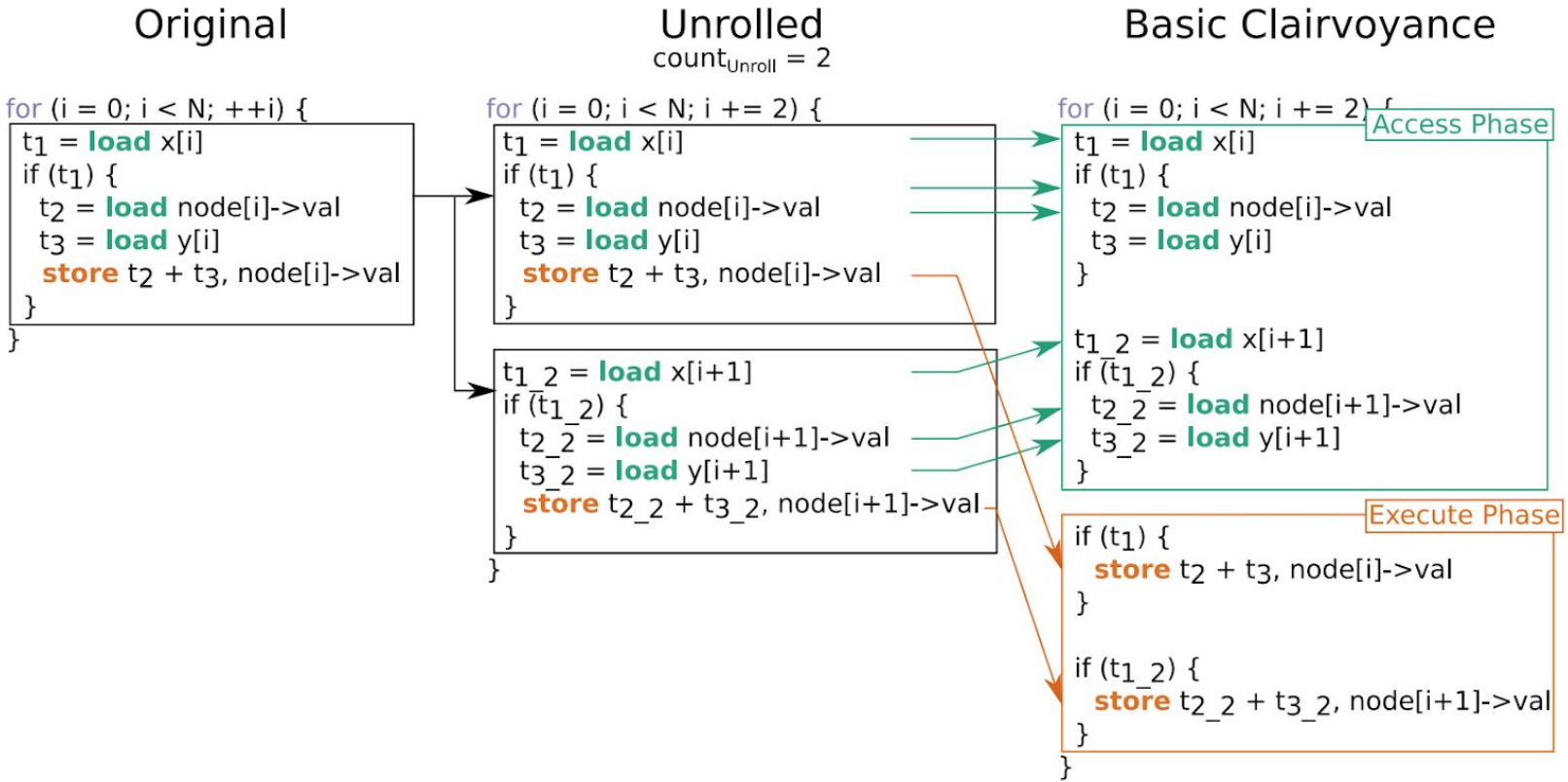
Basic Clairvoyance Algorithm

- **(2) -** Unroll Loop
- **(3) -** Initialize Access
- **(4) -** Identify load instructions
- **(5-9) -** Identify instructions required by load instructions
- **(10-11) -** Create access and execute phases
- **(12-13) -** Join access and execute phases

Input: Loop L, Unroll Count $\text{count}_{\text{unroll}}$
Output: Clairvoyance Loop $L_{Clairvoyance}$

```
1  begin
2      L_unrolled ← Unroll(L, count_unroll)
3      L_access ← Copy(L_unrolled)
4      hoist_list ← FindLoads(L_access)
5      to_keep ← ∅
6      for load in hoist_list do
7          requirements ← FindRequirements(load)
8          to_keep ← Union(to_keep, requirements)
9      end
10     L_access ← RemoveUnlisted(L_access, to_keep)
11     L_execute ← ReplaceListed(L_access, L_unrolled)
12     L_Clairvoyance ← Combine(L_access, L_unrolled)
13     return L_Clairvoyance
14 end
```

Example Transformation



Improvement to Basic Clairvoyance

- **(2) -** Unroll Loop
- **(3) -** Initialize Access
- **(4) -** Identify load instructions
- **(5-9) -** Identify instructions required by load instructions
- **(10-11) -** Create access and execute phases
- **(12-13) -** Join access and execute phases

Input: Loop L, Unroll Count $\text{count}_{\text{unroll}}$
Output: Clairvoyance Loop $L_{Clairvoyance}$

```
1  begin
2      L_unrolled ← Unroll(L, count_unroll)
3      L_access ← Copy(L_unrolled)
4      hoist_list ← FindLoads(L_access)
5      to_keep ← ∅
6      for load in hoist_list do
7          requirements ← FindRequirements(load)
8          to_keep ← Union(to_keep, requirements)
9      end
10     L_access ← RemoveUnlisted(L_access, to_keep)
11     L_execute ← ReplaceListed(L_access, L_unrolled)
12     L_Clairvoyance ← Combine(L_access, L_unrolled)
13     return L_Clairvoyance
14 end
```

Identifying Critical Loads

- “critical loads” are load instructions selected for the access phase
- **Indirection Count:**
  - number of loads a load instruction is dependent on
  - ex: x[y[z[i]]] = 2
- Load instructions with indirection count <= $count_{indir}$ are selected as critical loads

$count_{Indir} = 0$

```
for (i = 0; i < N; i += 2) {
  Access Phase
  t1 = load x[i]
  t1_2 = load x[i+1]
  Execute Phase
  if (t1) {
    t2 = load node[i]->val
    t3 = load y[i]
    store t2 + t3, node[i]->val
  }

  if (t1_2) {
    t2_2 = load node[i+1]->val
    t3_2 = load y[i+1]
    store t2_2 + t3_2, node[i+1]->val
  }
}
```

$count_{Indir} = 1$

```
for (i = 0; i < N; i += 2) {
  Access Phase
  t1 = load x[i]
  if (t1) {
    t2 = load node[i]->val
    t3 = load y[i]
  }

  t1_2 = load x[i+1]
  if (t1_2) {
    t2_2 = load node[i+1]->val
    t3_2 = load y[i+1]
  }
  Execute Phase
  if (t1) {
    store t2 + t3, node[i]->val
  }

  if (t1_2) {
    store t2_2 + t3_2, node[i+1]->val
  }
}
```

Improvement to Basic Clairvoyance

- **(2) -** Unroll Loop
- **(3) -** Initialize Access
- **(4) -** Identify load instructions
- **(5-9) -** Identify instructions required by load instructions
- **(10-11) -** Create access and execute phases
- **(12-13) -** Join access and execute phases

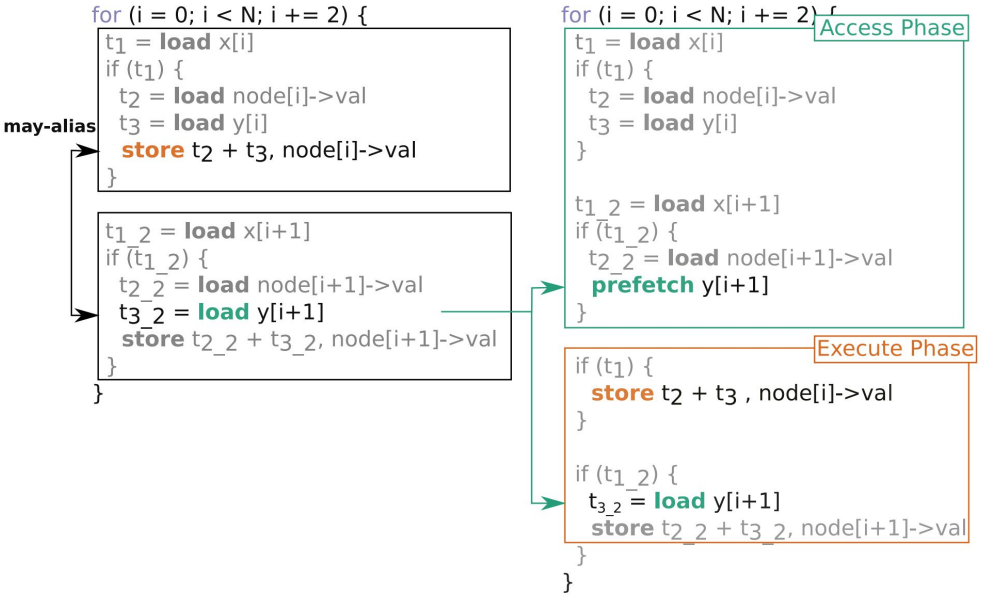
Input: Loop L, Unroll Count $\text{count}_{\text{unroll}}$
Output: Clairvoyance Loop $L_{Clairvoyance}$

```
1  begin
2      L_unrolled ← Unroll(L, count_unroll)
3      L_access ← Copy(L_unrolled)
4      hoist_list ← FindLoads(L_access)
5      to_keep ← ∅
6      for load in hoist_list do
7          requirements ← FindRequirements(load)
8          to_keep ← Union(to_keep, requirements)
9      end
10     L_access ← RemoveUnlisted(L_access, to_keep)
11     L_execute ← ReplaceListed(L_access, L_unrolled)
12     L_Clairvoyance ← Combine(L_access, L_unrolled)
13     return L_Clairvoyance
14 end
```

Handling Unknown Dependencies

- Load should be hoisted only if data dependency is respected.

- Aliases have 3 cases
  a. No-Alias ⇒ Hoist!
  b. Must-Alias ⇒ Don't hoist!
  c. May-Alias ⇒ Prefetch!



Improvement to Basic Clairvoyance

- **(2) -** Unroll Loop
- **(3) -** Initialize Access
- **(4) -** Identify load instructions
- **(5-9) -** Identify instructions required by load instructions
- **(10-11) -** Create access and execute phases
- **(12-13) -** Join access and execute phases

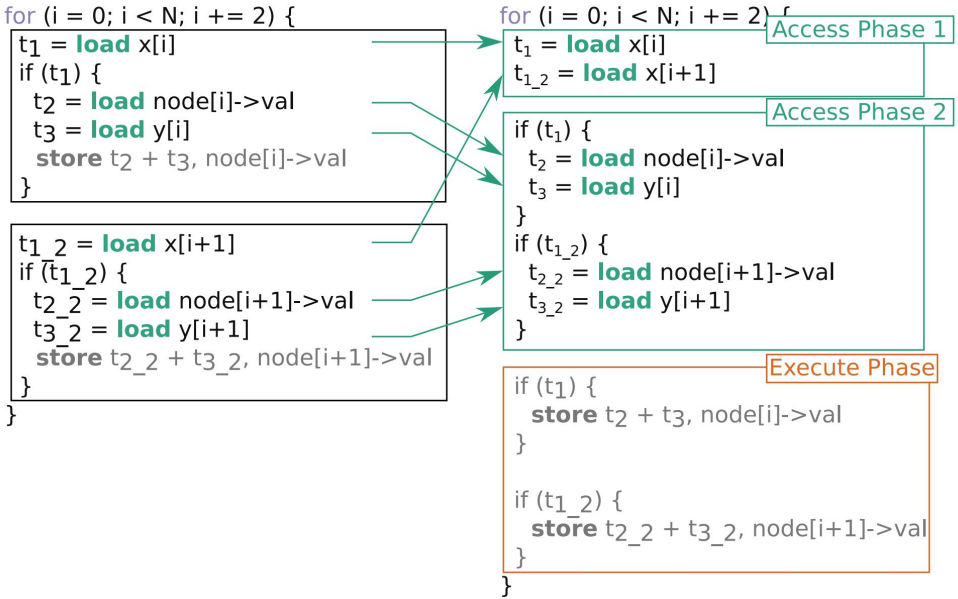
Input: Loop L, Unroll Count $\text{count}_{\text{unroll}}$
Output: Clairvoyance Loop $L_{Clairvoyance}$

```
1  begin
2      L_unrolled ← Unroll(L, count_unroll)
3      L_access ← Copy(L_unrolled)
4      hoist_list ← FindLoads(L_access)
5      to_keep ← ∅
6      for load in hoist_list do
7          requirements ← FindRequirements(load)
8          to_keep ← Union(to_keep, requirements)
9      end
10     L_access ← RemoveUnlisted(L_access, to_keep)
11     L_execute ← ReplaceListed(L_access, L_unrolled)
12     L_Clairvoyance ← Combine(L_access, L_unrolled)
13     return L_Clairvoyance
14 end
```

Handling Chains of Dependent Loads

Input: Set of ***loads***
Output: List of sets *phase_loads*

```
1  begin
2      remaining_loads ← loads
3      phase_loads ← []
4      while remaining_loads ≠ ∅ do
5          phase ← ∅
6          for ld in remaining_loads do
7              reqs ← ∅
8              FindCFGRequirements (ld, reqs)
9              FindDataRequirements (ld, reqs)
10             is_independent ← Intersection(reqs,
               remaining_loads) == ∅
11             if is_independent then
12                 phase ← phase + ld
13             end
14         end
15         remaining_loads ← remaining_loads \ phase
16         phase_loads ← phase_loads + phase
17     end
18     return phase_loads
19 end
```



Parameters of the Clairvoyance Pass

- Turn on Clairvoyance Optimization:
  - Heuristics: Disable if loads/branches < 0.7

- Loop Unroll Count, Indirection Count:
  - Choose by using state-of-the-art runtime version selectors

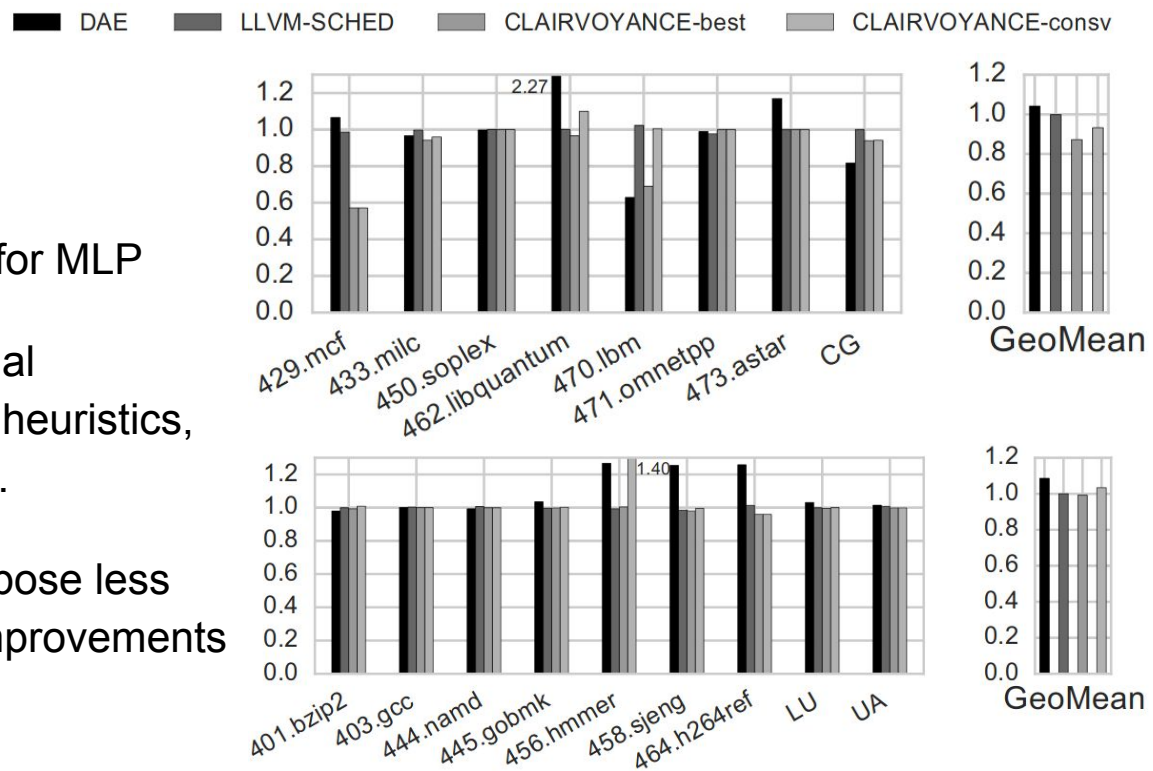
Benchmark Results

Memory-bound benchmarks

- Yield highest performance improvements
- Expose most opportunities for MLP

Clairvoyance-Best shows potential improvements using speculative heuristics, and given better pointer analysis.

Compute-bound benchmarks expose less opportunities for Clairvoyance improvements



Conclusion

Idea is increasing instruction-level, memory-level parallelism to make better use of core resources.

Key technique is reordering dependent-loads across loop iterations, controlling register pressure.

Most improvement in memory-bound scenarios: up to 43%, 7% geomean for conservative, 13% geomean for speculative

Thoughts

+ Implementation software only
+ Effective use of software prefetching
+ Open-source, possible to replicate

- Not enough detail on register pressure
- Effect of indirection count parameter on performance of tests not clear
- Test hardware not aligned with paper motivation

Questions?