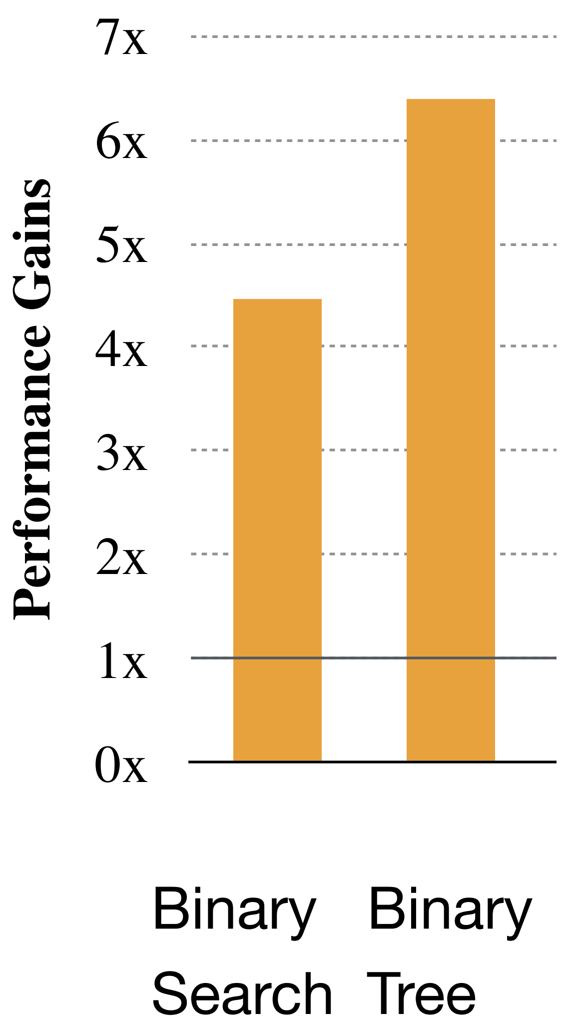
`Cimple`: Instruction and Memory Level Parallelism DSL

Vladimir Kiriansky, Haoran Xu,
Martin Rinard, Saman Amarasinghe

PACT'18

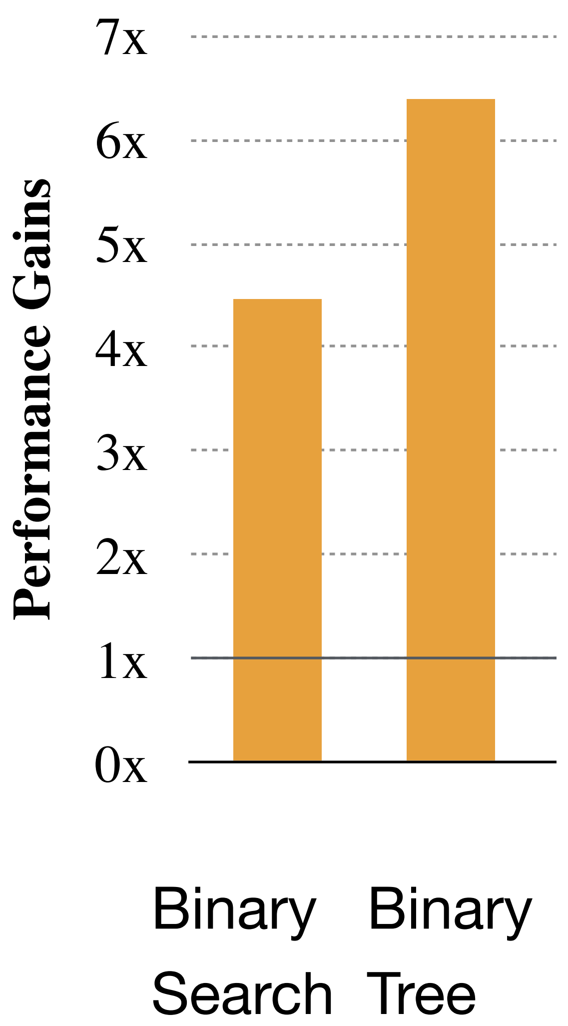
November 3, 2018
Limassol, Cyprus

`Cimple` Performance Gains



[Haswell]

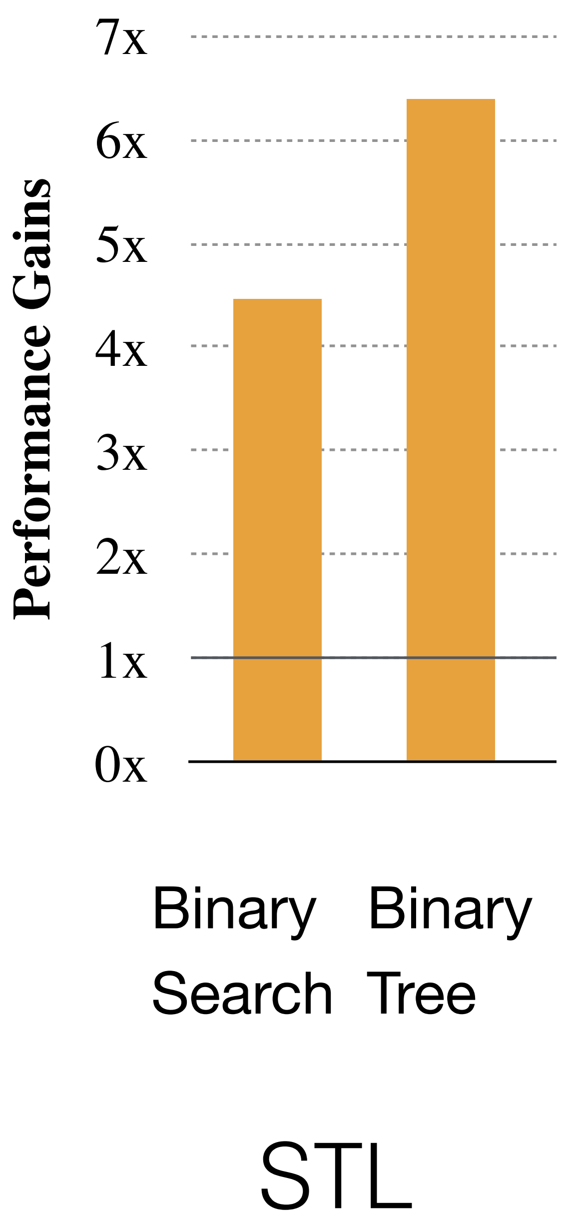
`Cimple` Performance Gains



- Are we teaching the wrong algorithms?

[Haswell]

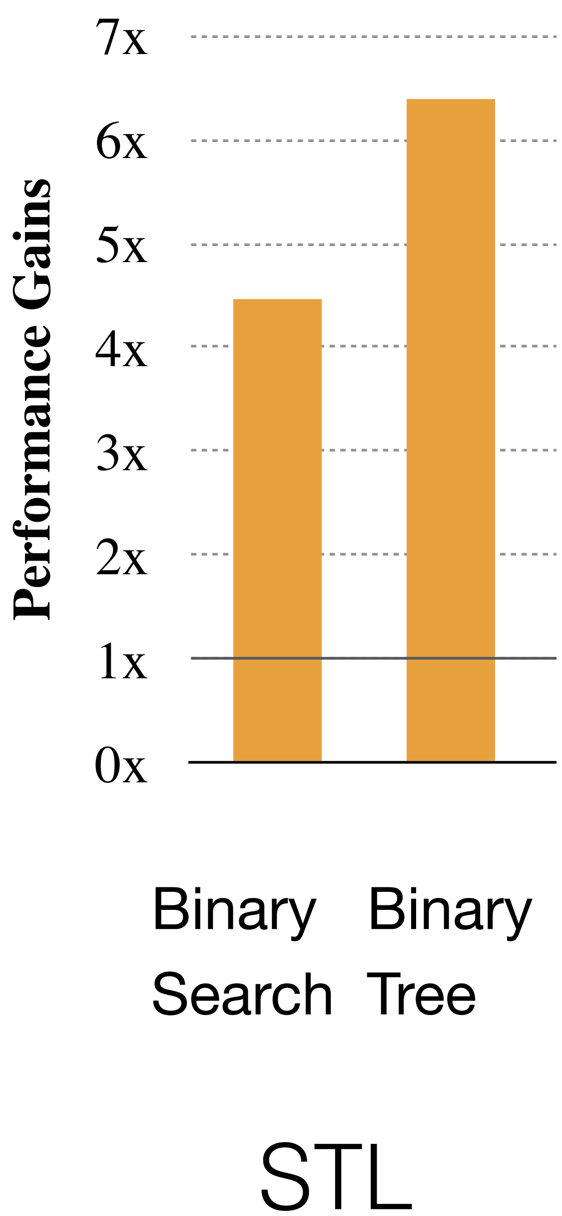
Cimple Performance Gains



- Are we teaching the wrong algorithms?

[Haswell]

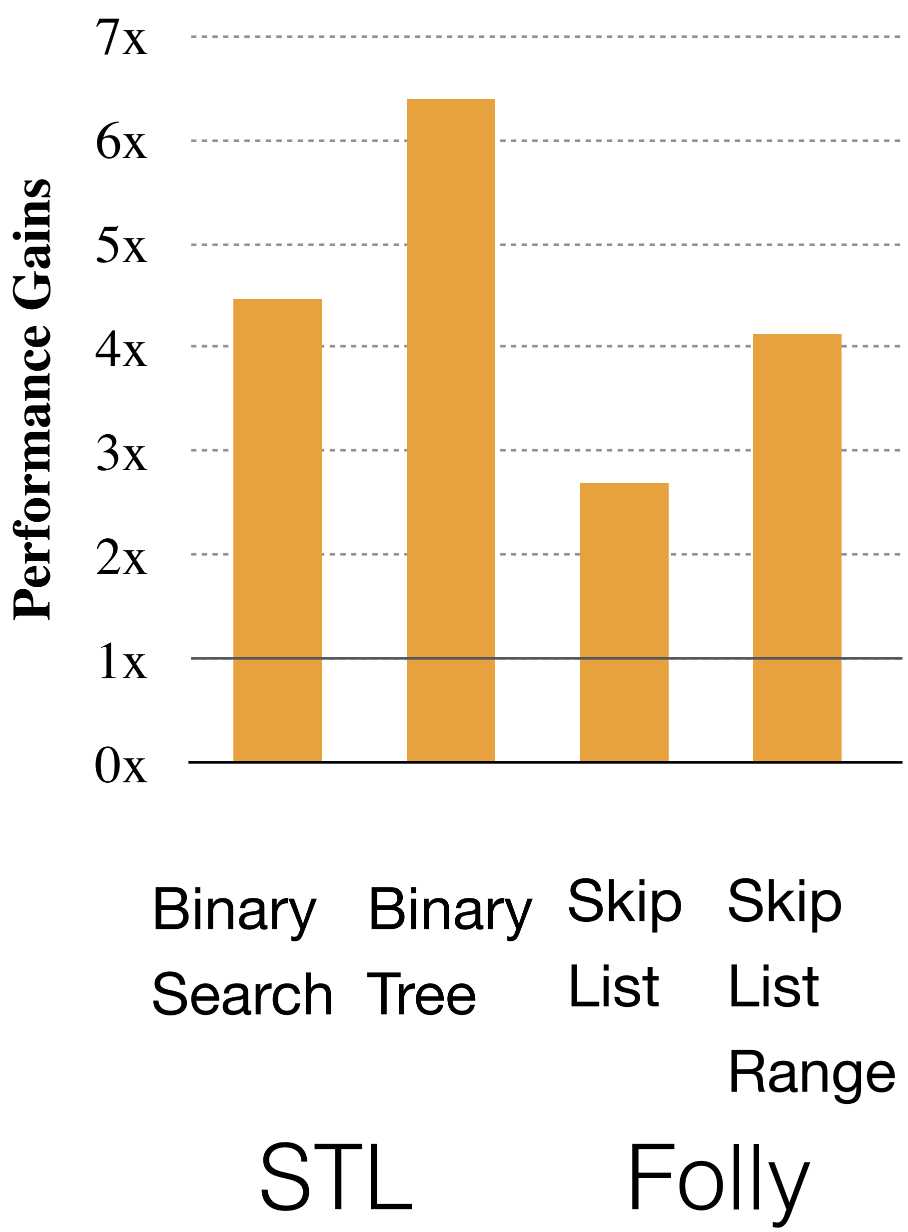
`Cimple` Performance Gains



- Are we teaching the wrong algorithms?
- Is STL using bad implementations?

[Haswell]

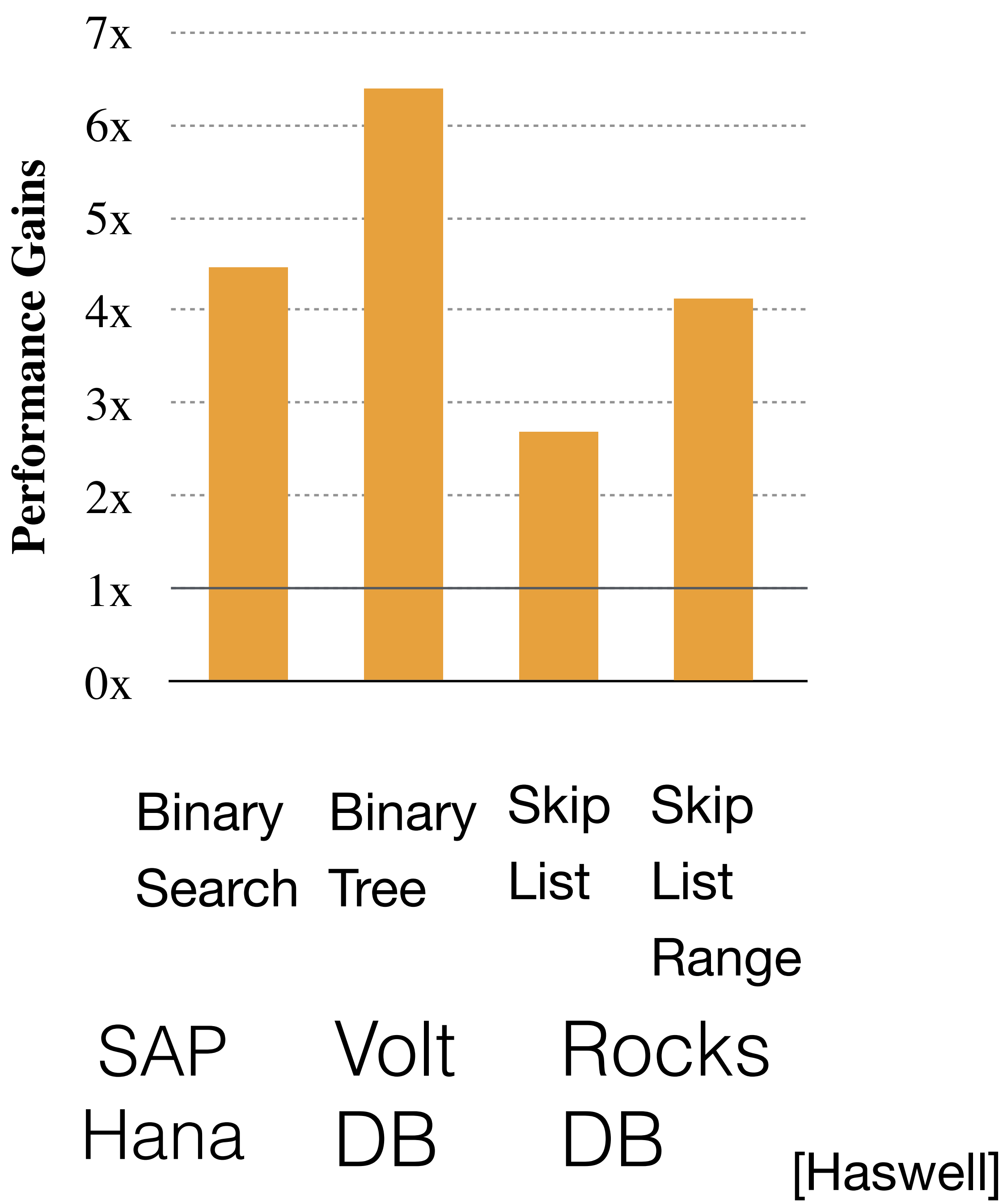
`Cimple` Performance Gains



[Haswell]

- Are we teaching the wrong algorithms?
- Is STL using bad implementations?

Cimple Performance Gains

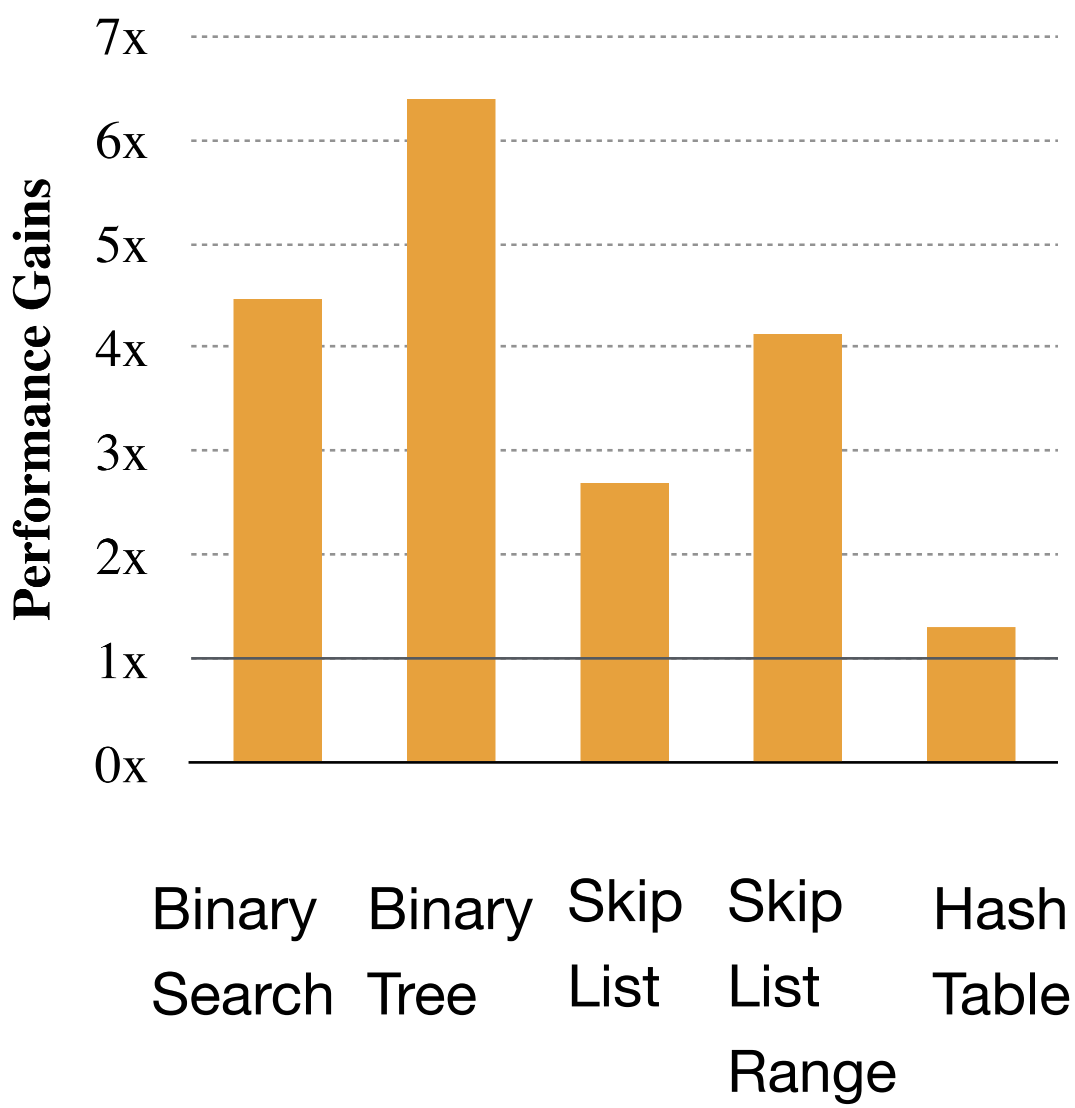


- Are we teaching the wrong algorithms?
- Is STL using bad implementations?

In-Memory Databases

- Terabyte Working Sets
  - AWS 12 TB VM
- Binary Search
  SAP Hana
- Binary Tree (partitioned)
  VoltDB
- Skip List (shared)
  RocksDB, MemSQL

`Cimple` Performance Gains



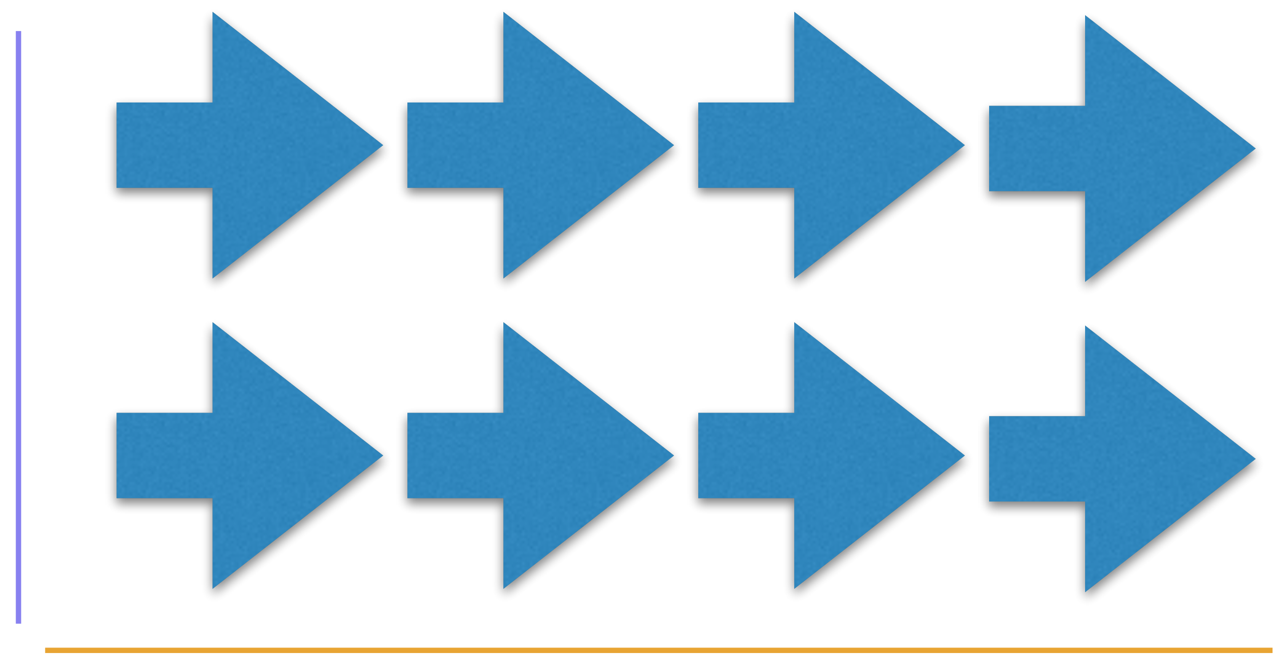
- Are we teaching the wrong algorithms?
- Is STL using bad implementations?
- Does our programming model match hardware?

[Haswell]

Little's Law

$L = \lambda W$

Arrival Rate
λ

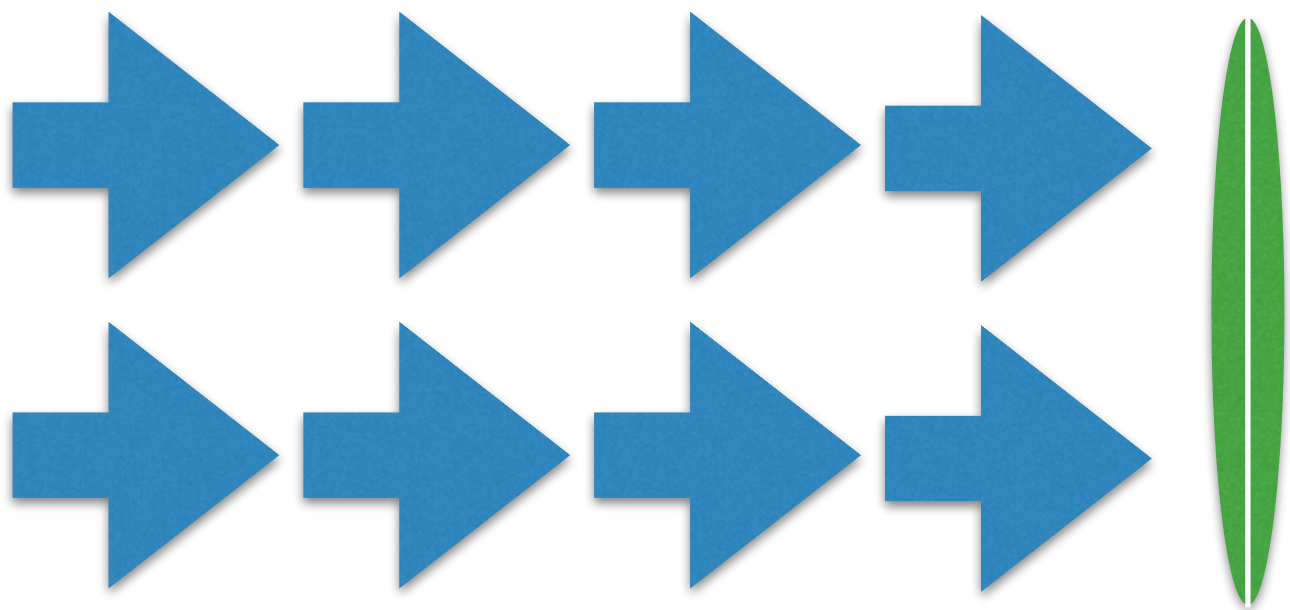


Concurrency
L

Latency
W

Little's Law

$$L = \lambda W$$



Arrival = Departure

$\lambda = X = 2$

Bandwidth

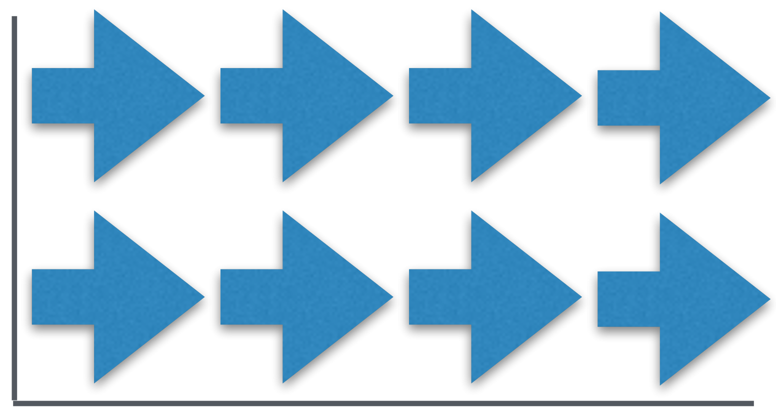
Concurrency

$L = 8$

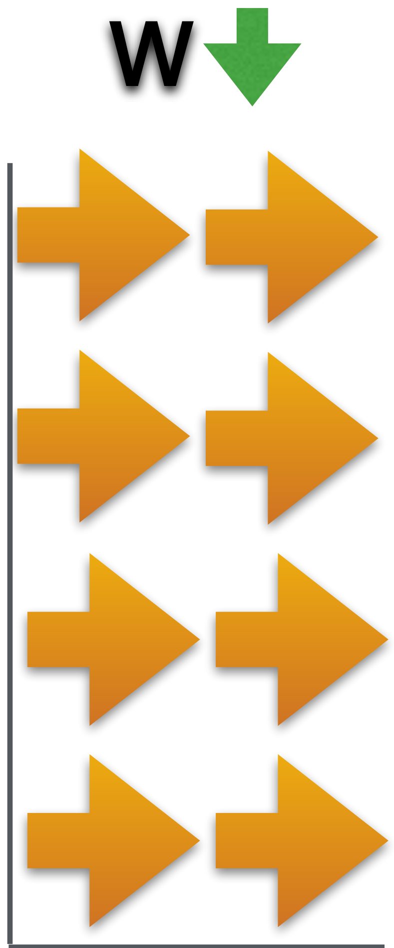
Latency

$W = 4$

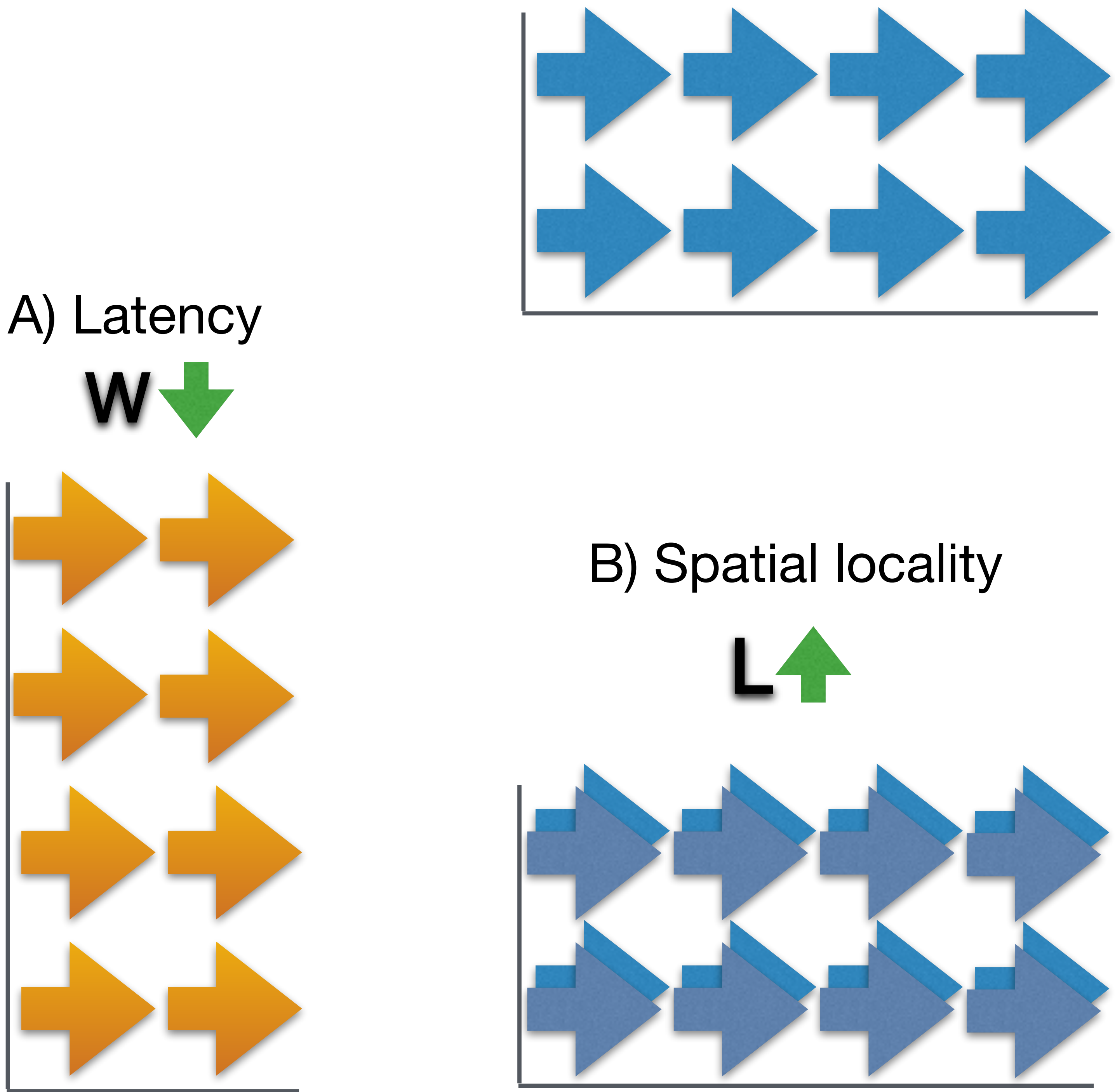
Three Improvement Paths



A) Latency

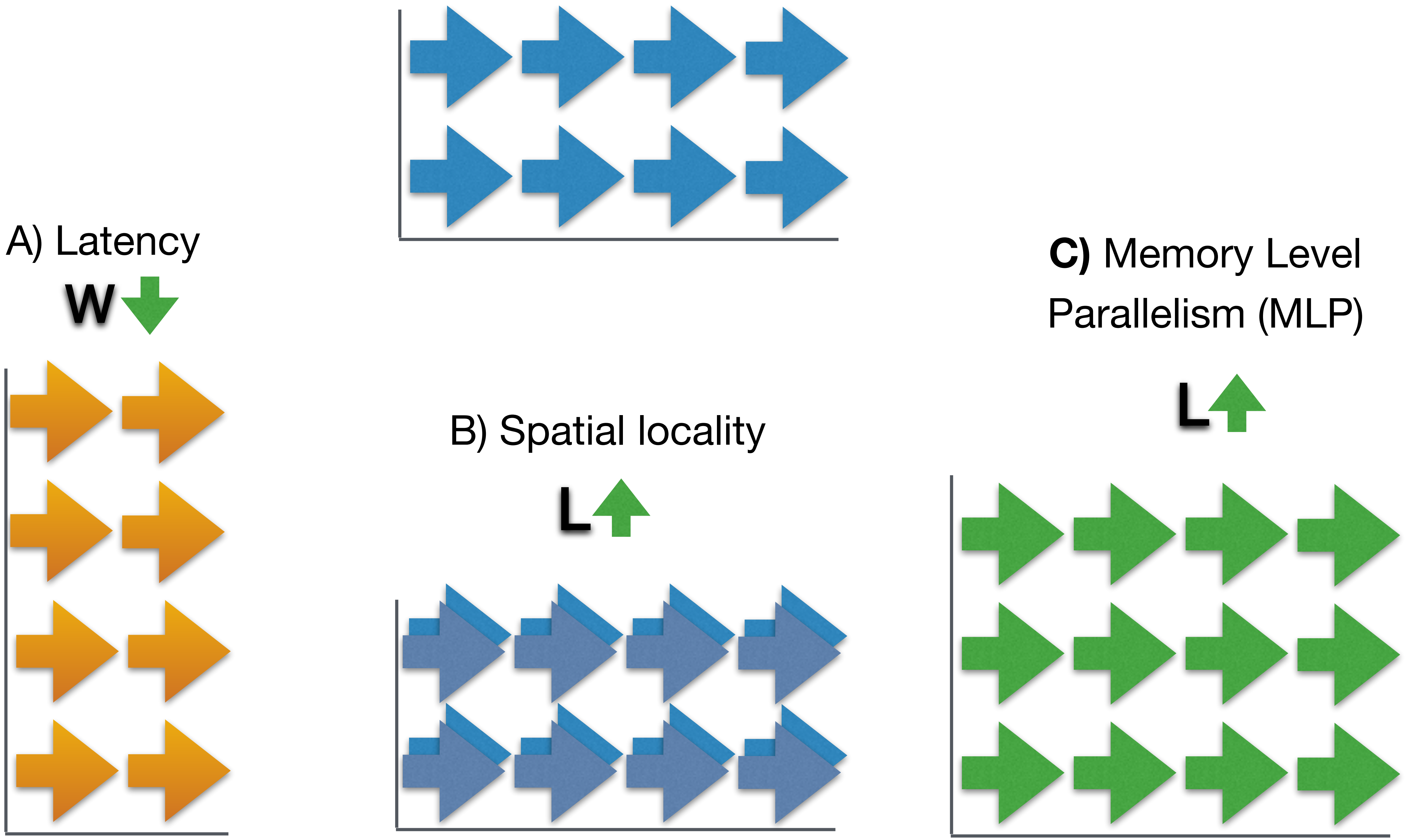


Three Improvement Paths



milk [Kiriansky et al, PACT'16]

Three Improvement Paths



Memory Wall

- Speculative out-of-order processors: automatically discover MLP

Memory Wall

- Speculative out-of-order processors: automatically discover MLP
- Non-blocking caches -
Miss Status Handling Registers (MSHRs)
- Large Instruction Windows

Memory Wall Conquered?

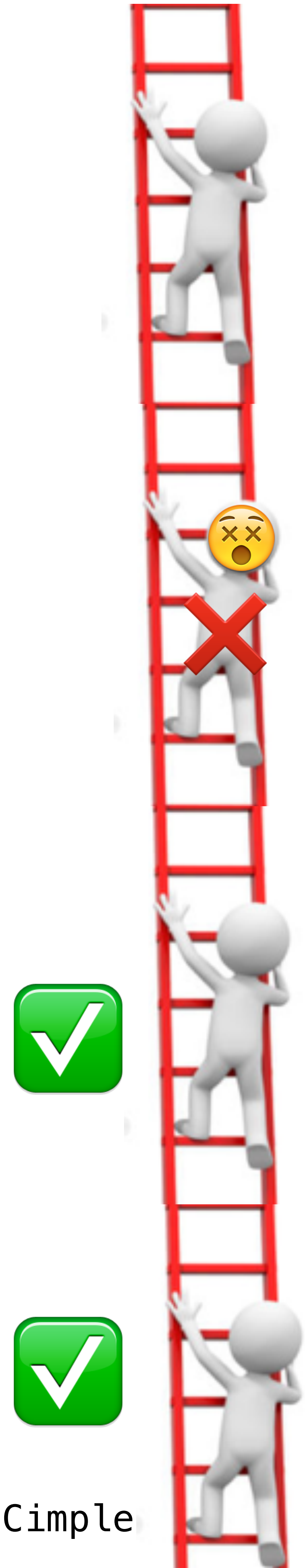
- Speculative out-of-order processors: automatically discover MLP
- Non-blocking caches - Miss Status Handling Registers (MSHRs) = 10 misses
- Large Instruction Windows ~200 instructions

Branch Mispredictions

http://www.dreamstime.com/abluecup

Memory Wall Conquered?

❌ Branch Misprediction



Memory Wall Conquered?

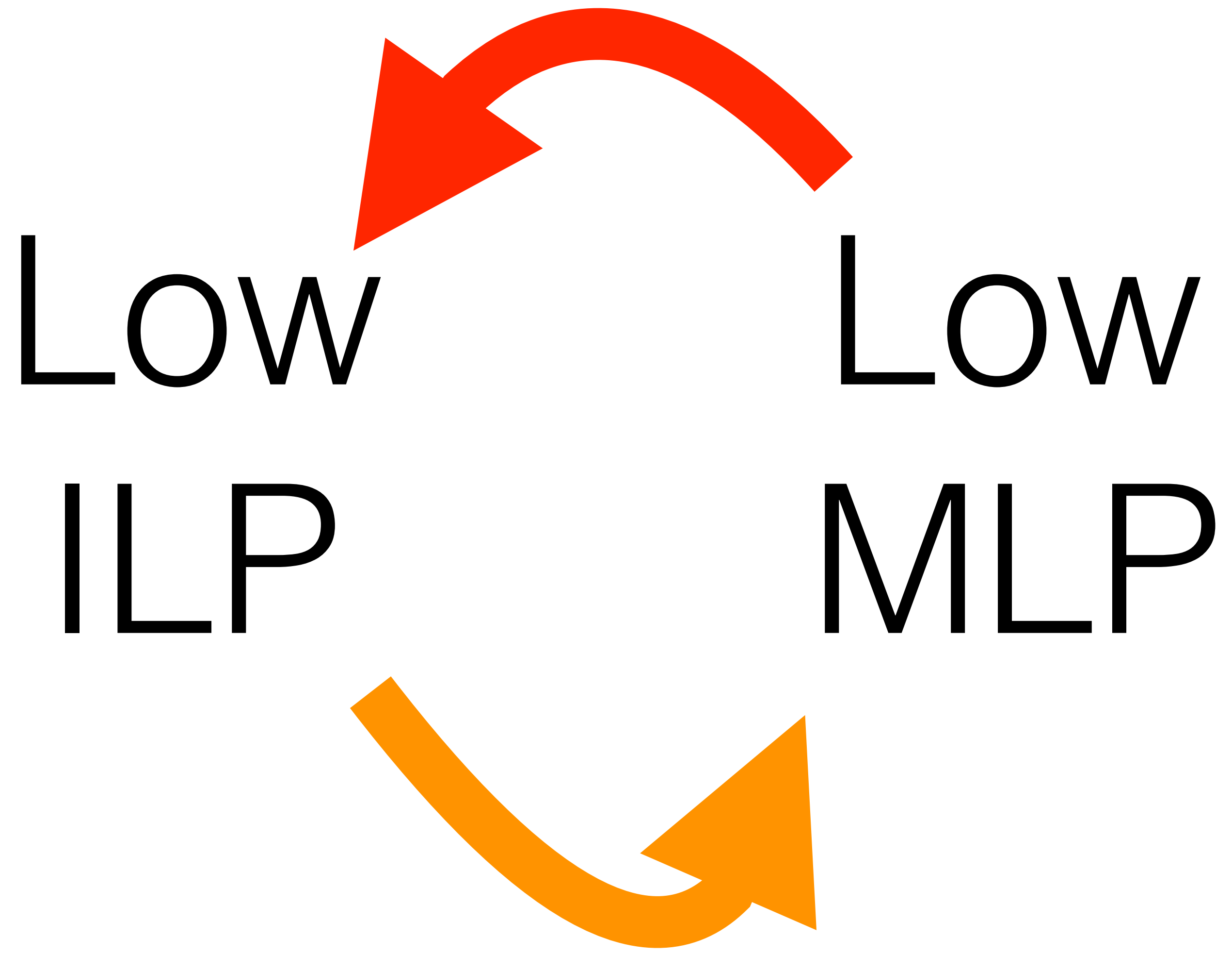
❌ Branch Misprediction

Cimple

Memory Wall

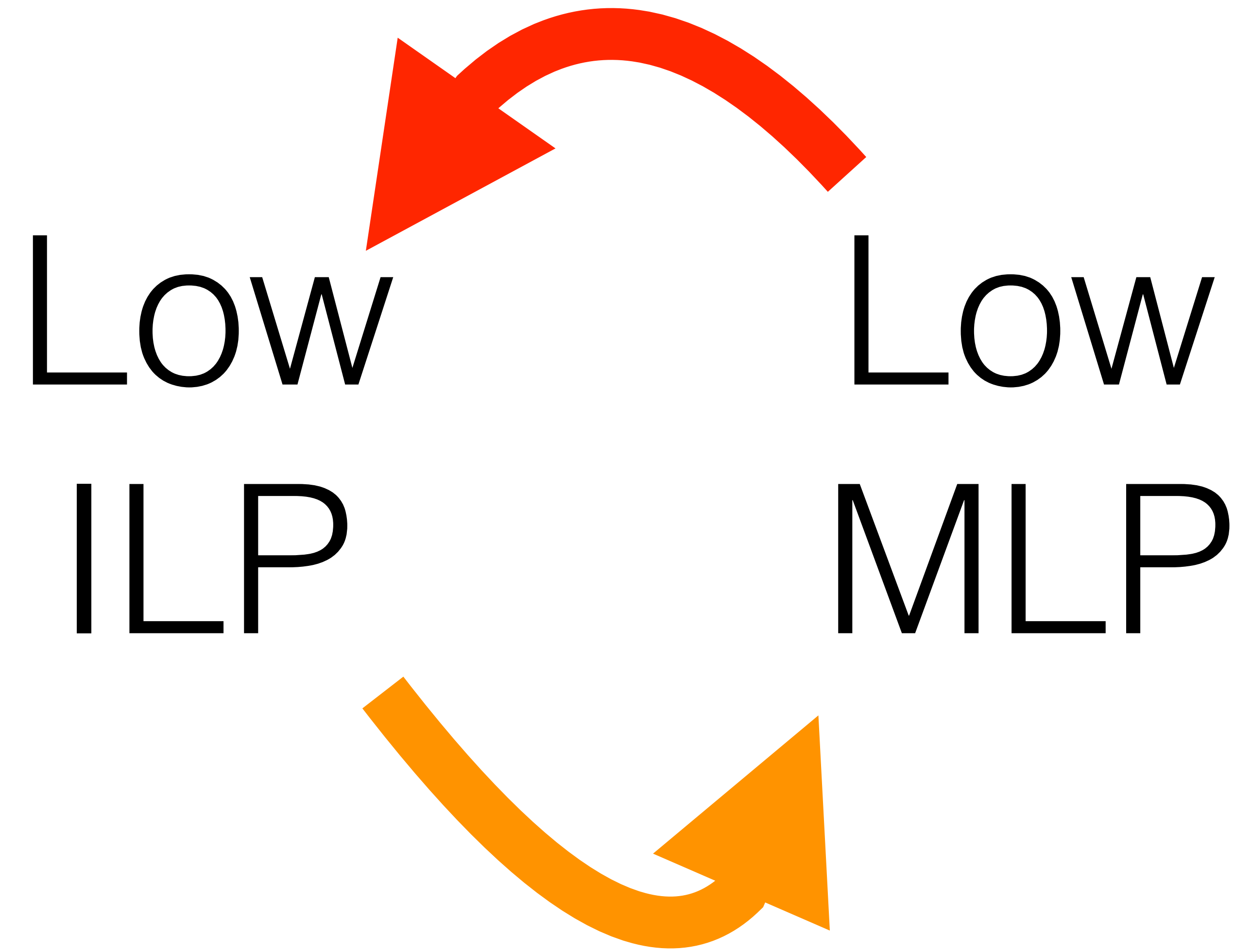
- ✗ Branch Misprediction
- ✗ Re-execution of correct path of
- ✗ independent tasks

ILP / MLP Vicious cycle

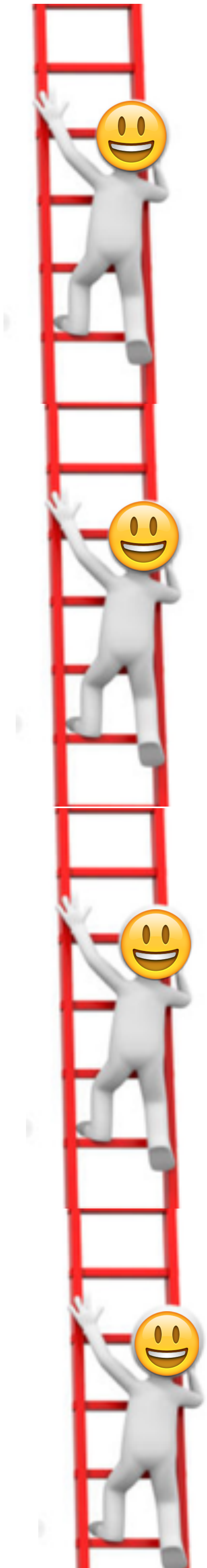


ILP / MLP Vicious cycle

Branch Misprediction



High Cache Miss Penalty



`Cimple`
Alternative

- Avoid speculation for MLP - harness Request Level Parallelism (RLP)
- Tasks pipelined on one thread
- Cooperatively context switch on *likely* cache miss

Outline

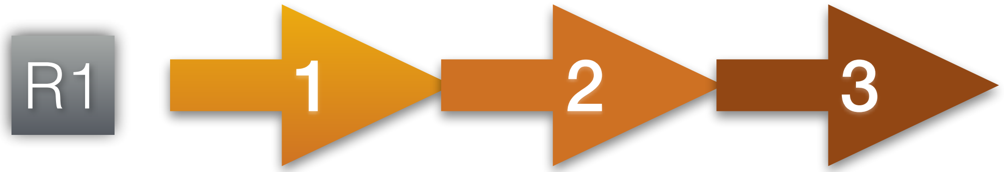
- `Cimple` Co-Routines Overview
- Static and Dynamic Schedulers
- Related Work
- `Cimple` DSL and Code Generation
- Performance Evaluation
- Conclusion & Q/A

`Cimple`

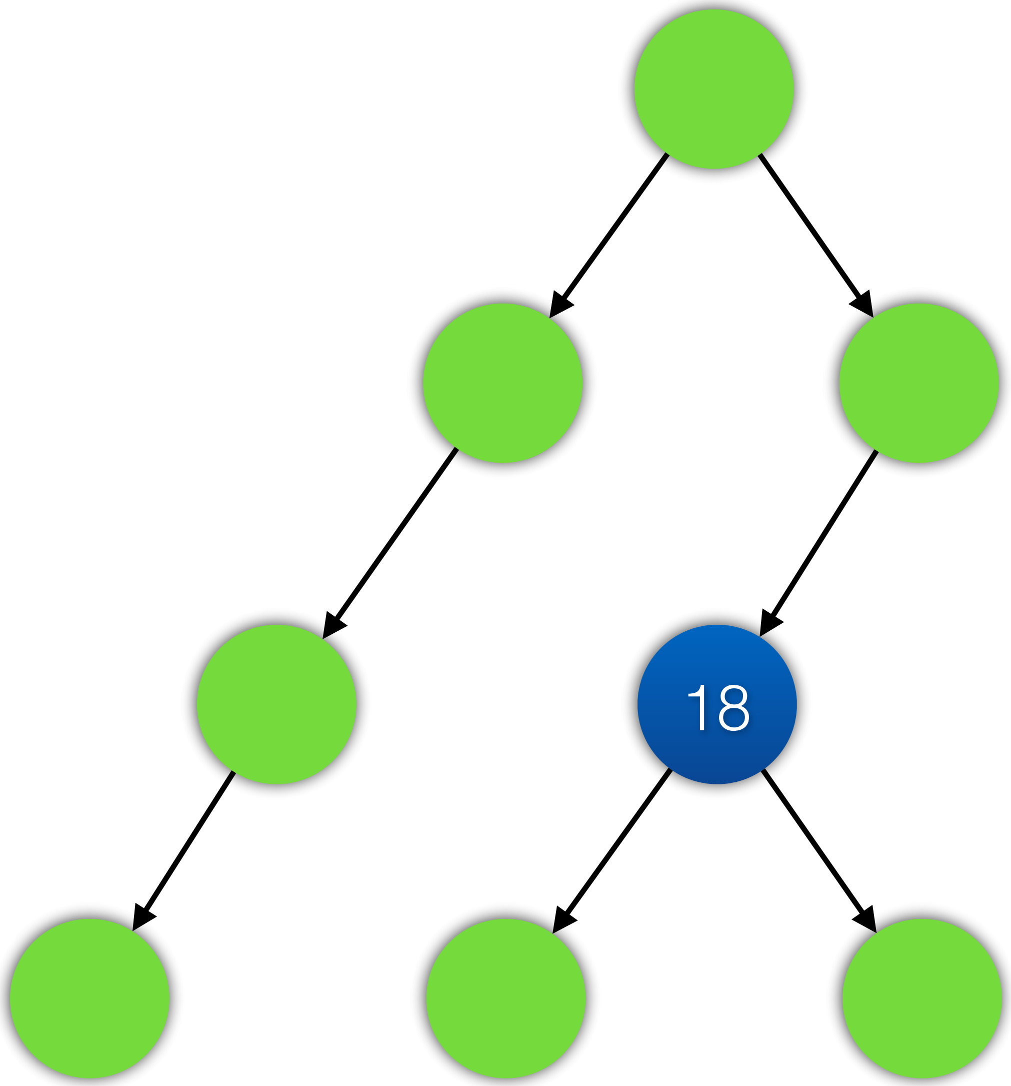
Overview

Traditional
One task per thread

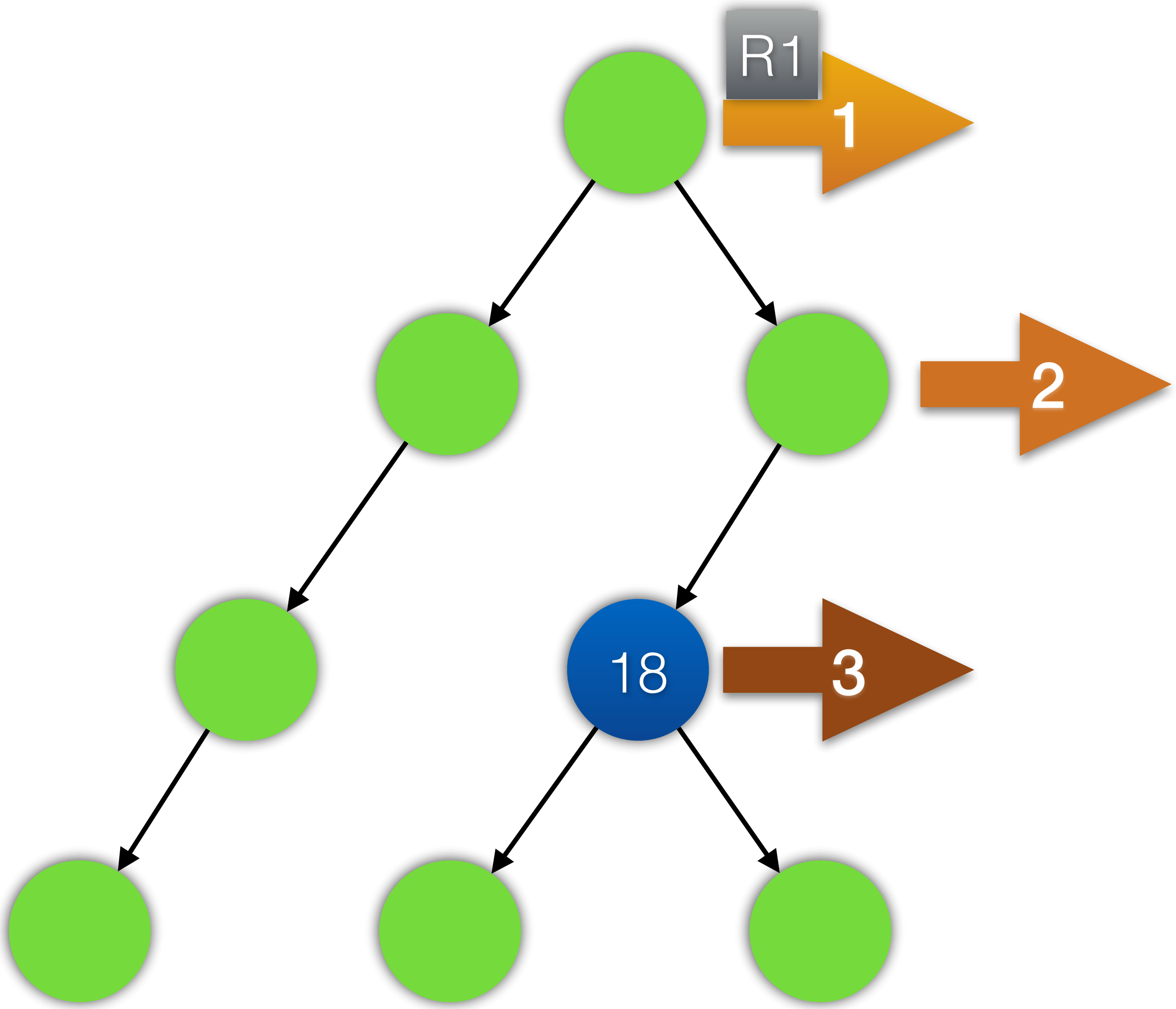
- Traditional dependence chain
- Request 1 executed to completion



Binary Tree

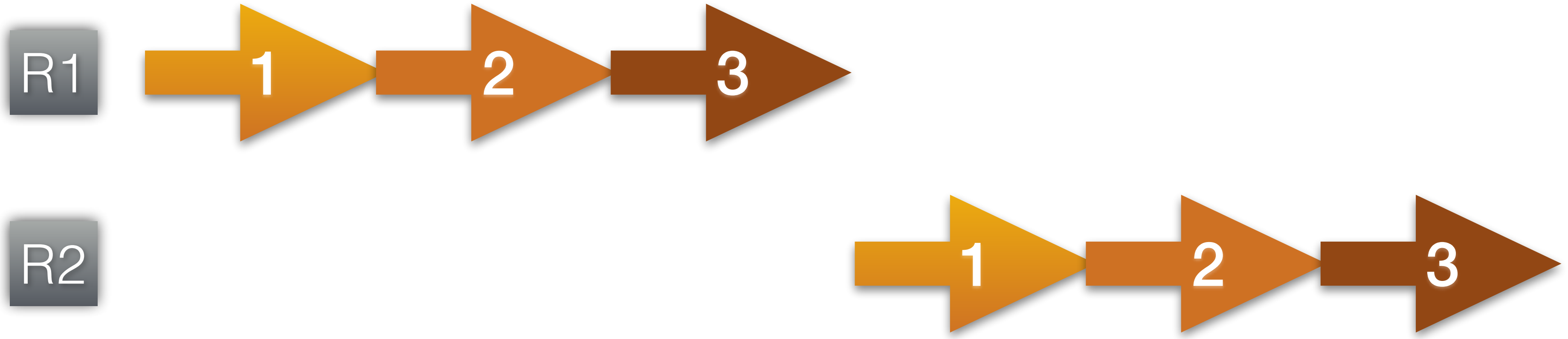


Binary Tree



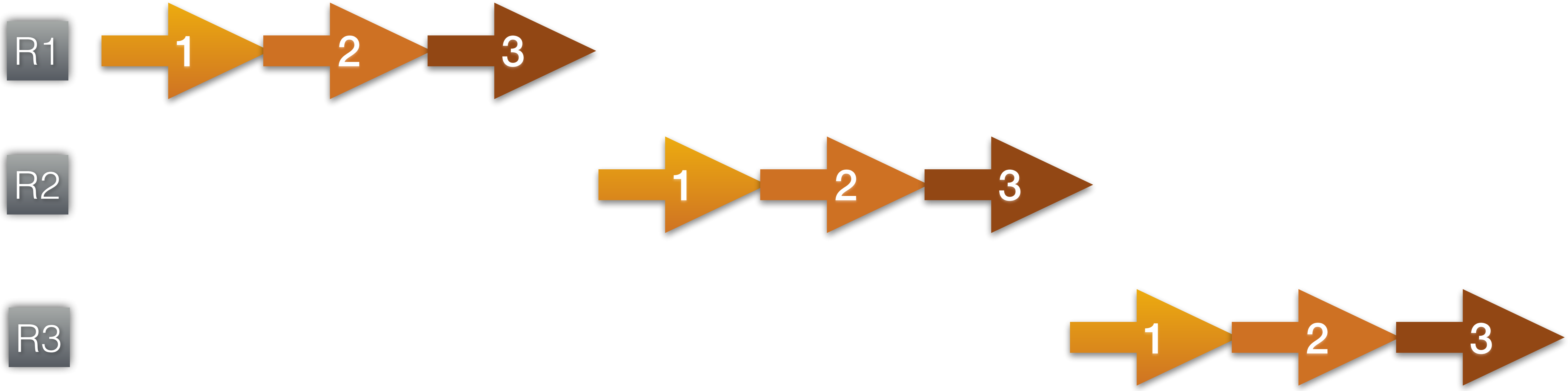
Traditional
One task per thread

- Traditional dependence chain
- When Request 1 is complete, start Request 2



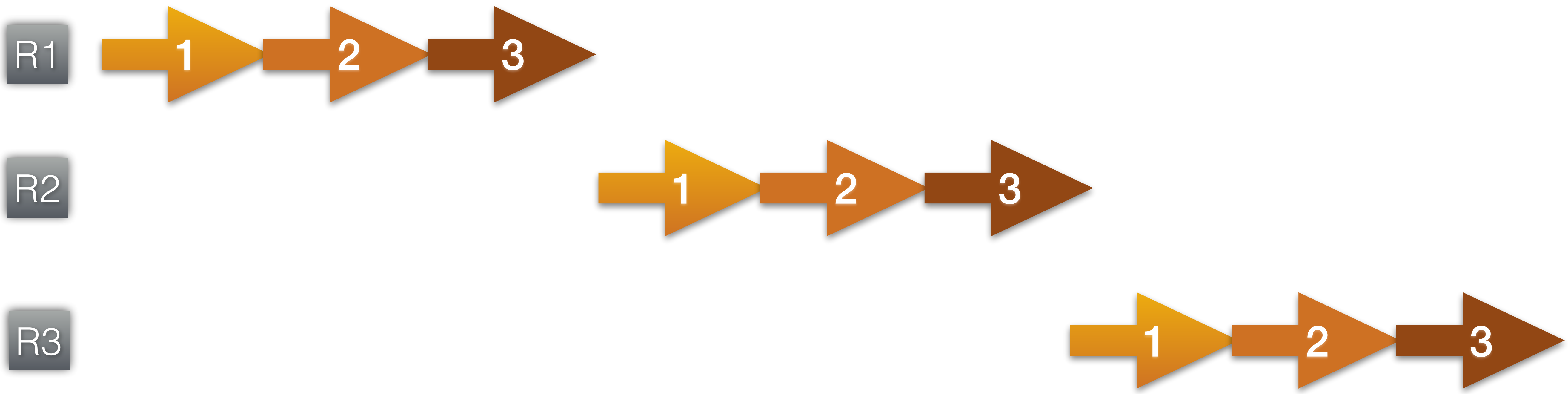
Traditional
One task per thread

- Traditional dependence chain
- When Request 2 is complete, start Request 3



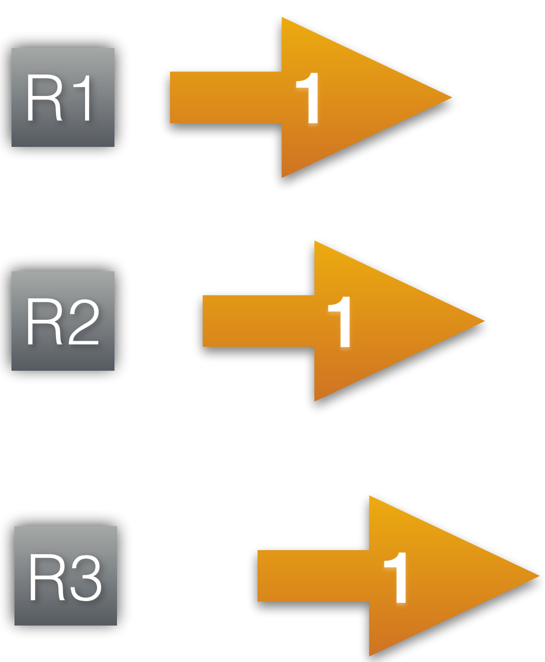
Traditional
Limited HW Reordering

- Traditional dependence chain
- HW out-of-order execution only if predictable&short



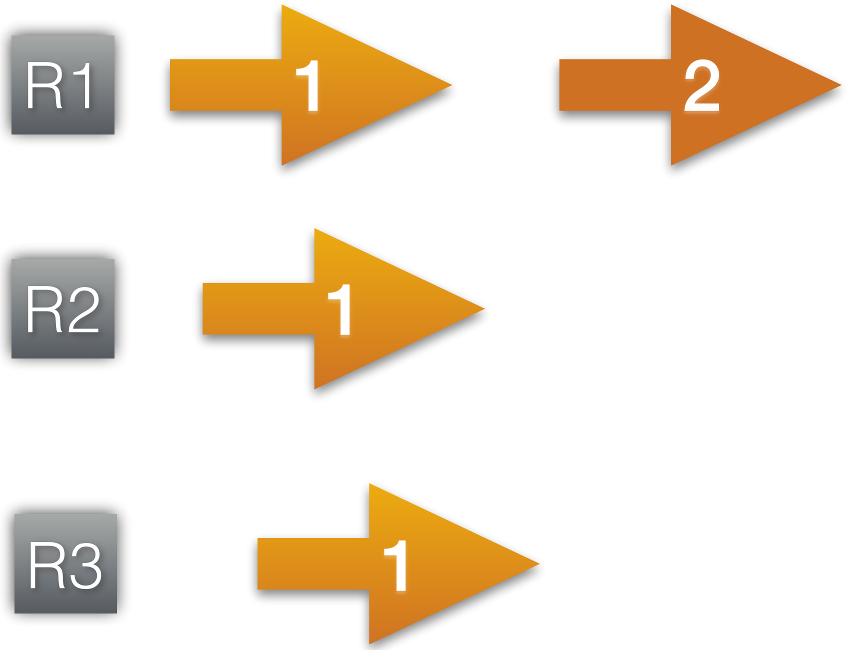
`Cimple` Co-routines Co-operative Scheduling

- Voluntary context switches after memory access



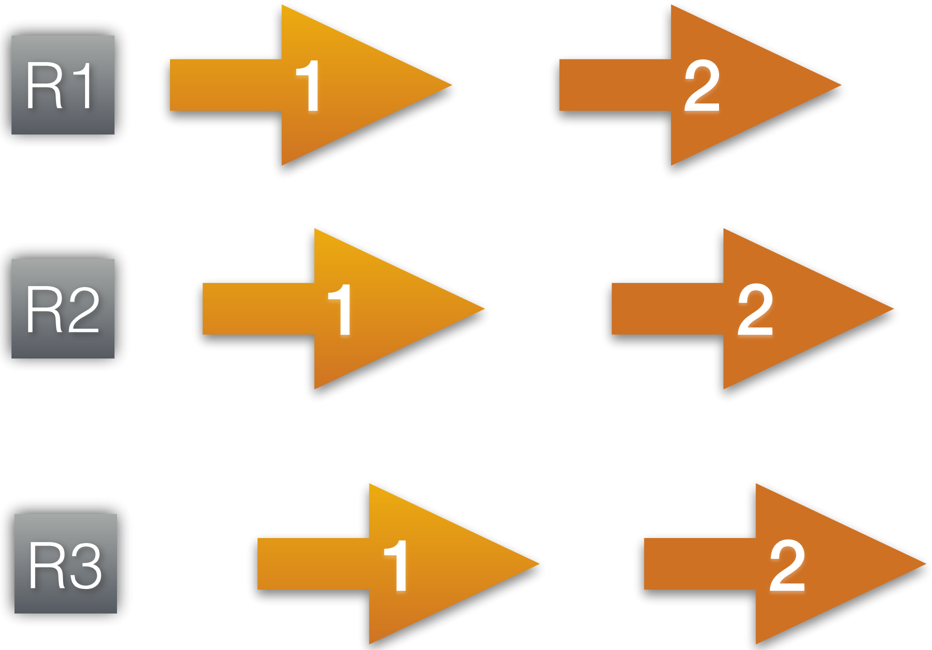
Cimple Co-routines Co-operative Scheduling

- Voluntary context switches after memory access
- No wait for completion - assume latency is hidden



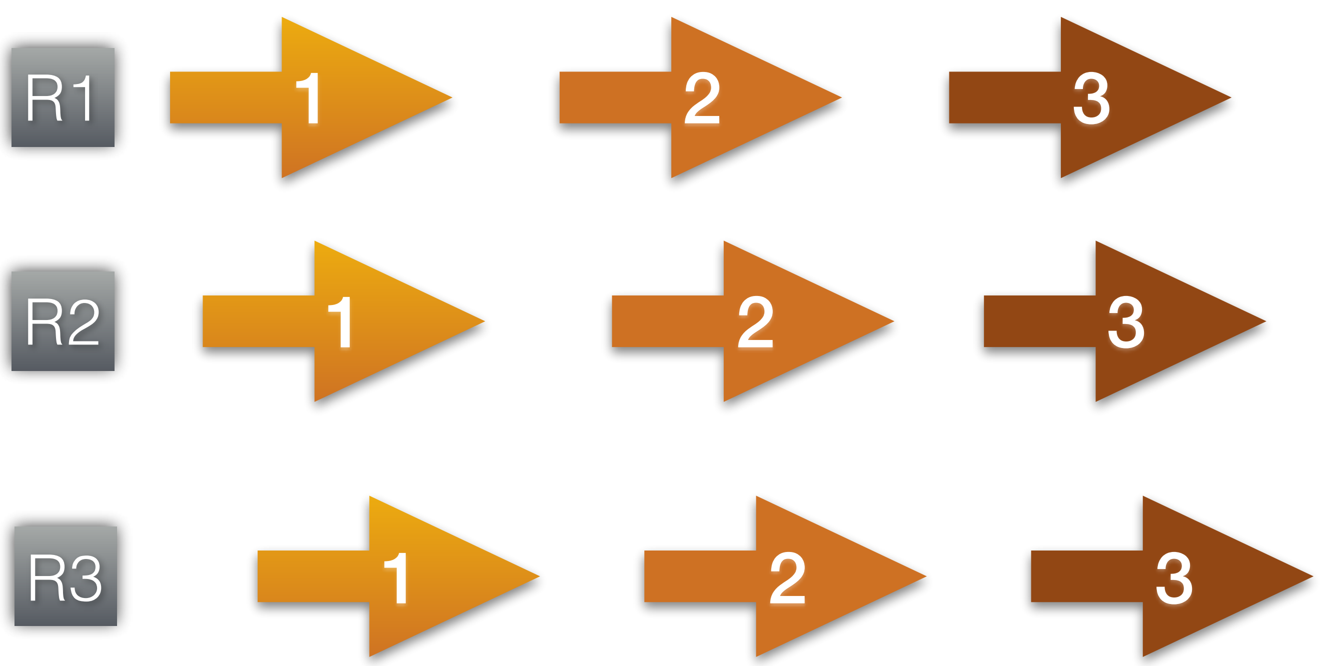
`Cimple` Co-routines Co-operative Scheduling

- Voluntary context switches after memory access
- No wait for completion - assume latency is hidden



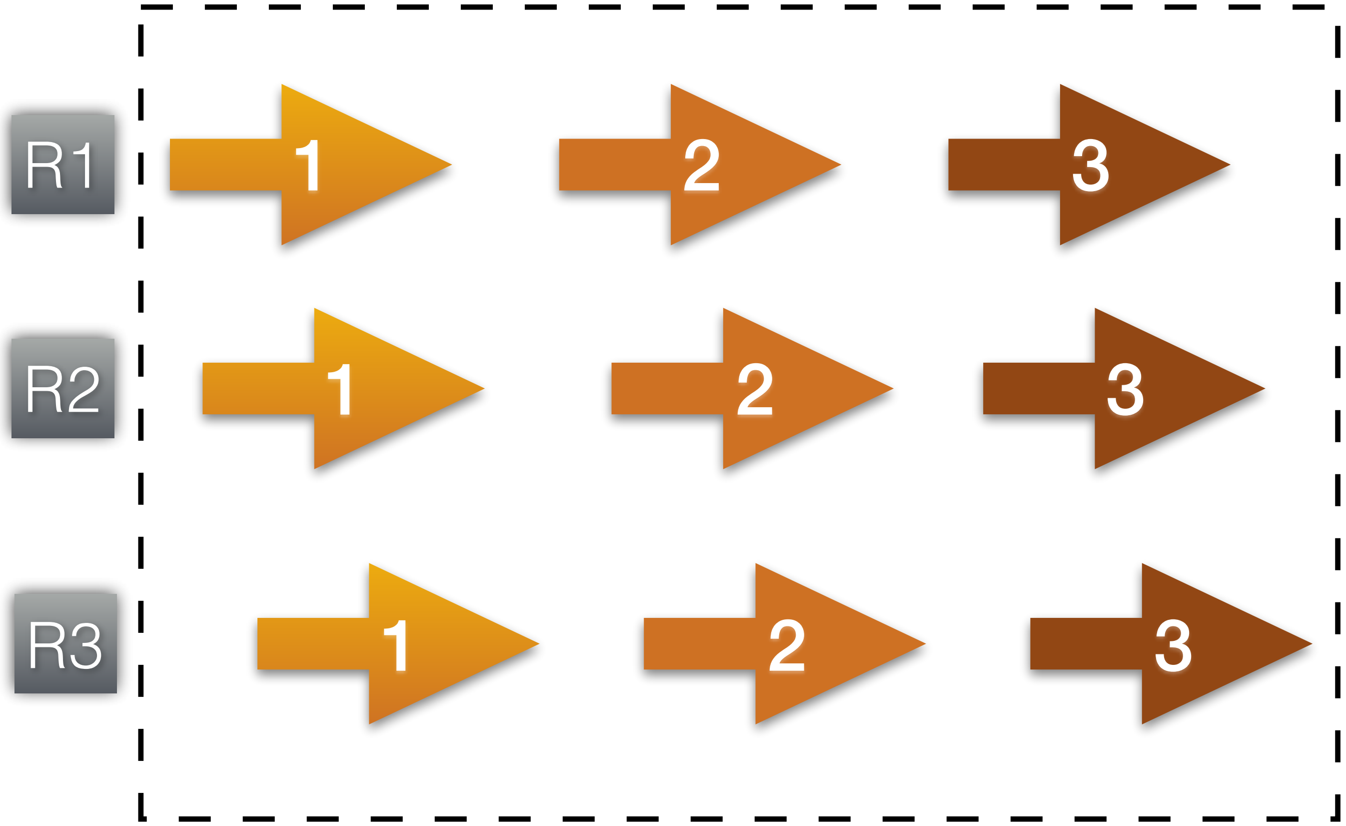
`Cimple` Co-routines Co-operative Scheduling

- Voluntary context switches after memory access
- Mark explicitly with **`Yield`**



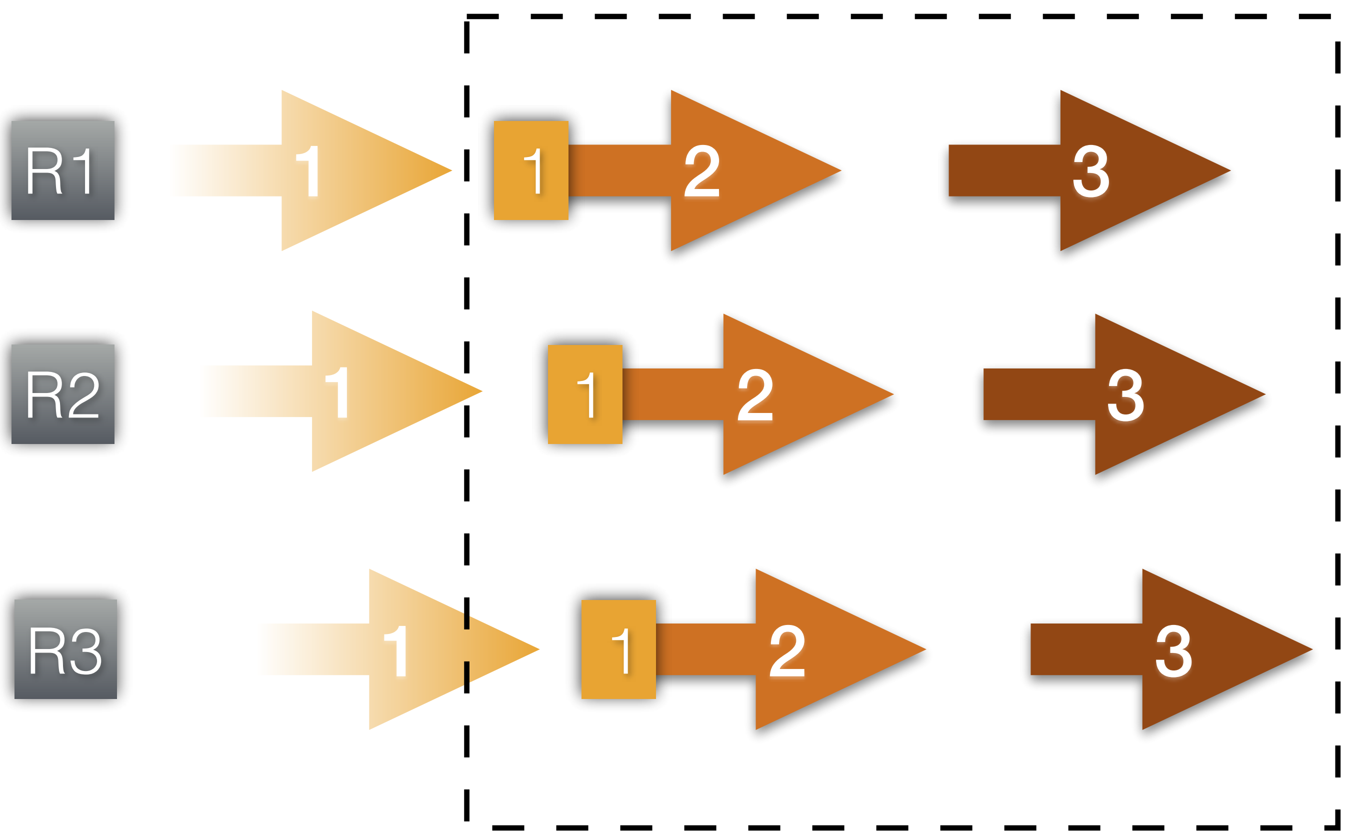
Co-routine
Yield

- Mark voluntary context switches with **`Yield`**
- Must fit all in instruction window



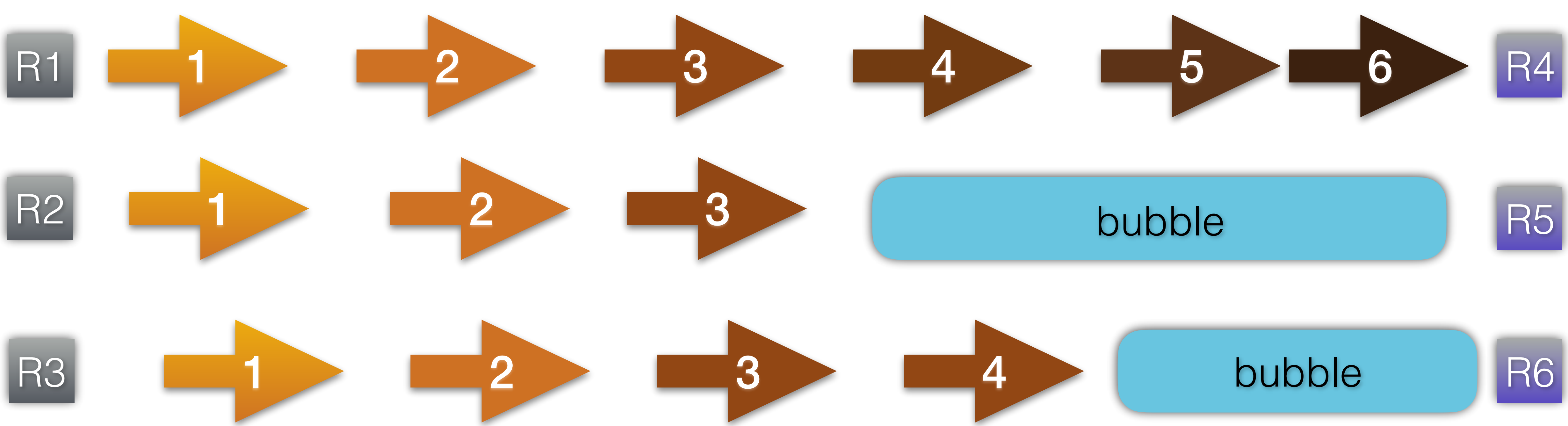
Co-routine + Prefetch

- **`Prefetch`** - Overlaps loads and computation
- More requests fit the instruction window



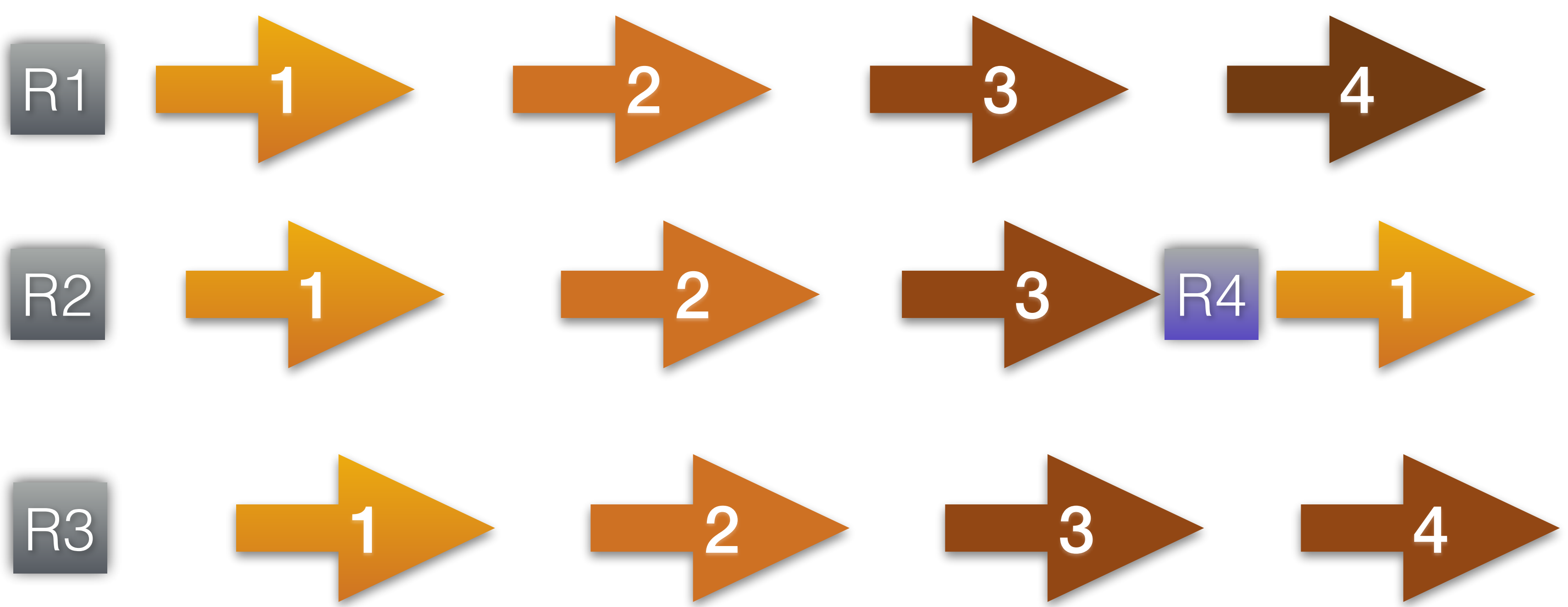
Co-routine
Static Scheduling

- Execute a group at a time
- Wait until all tasks complete



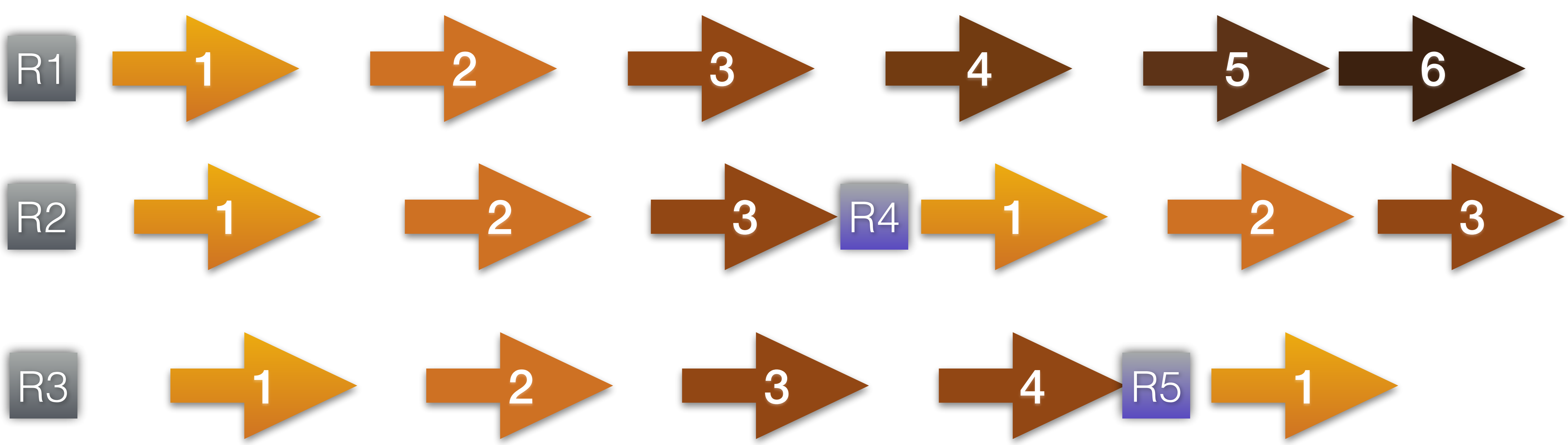
Co-routine Dynamic Scheduling

- Refill one task at a time
- Refill R4 as soon as R2 completes



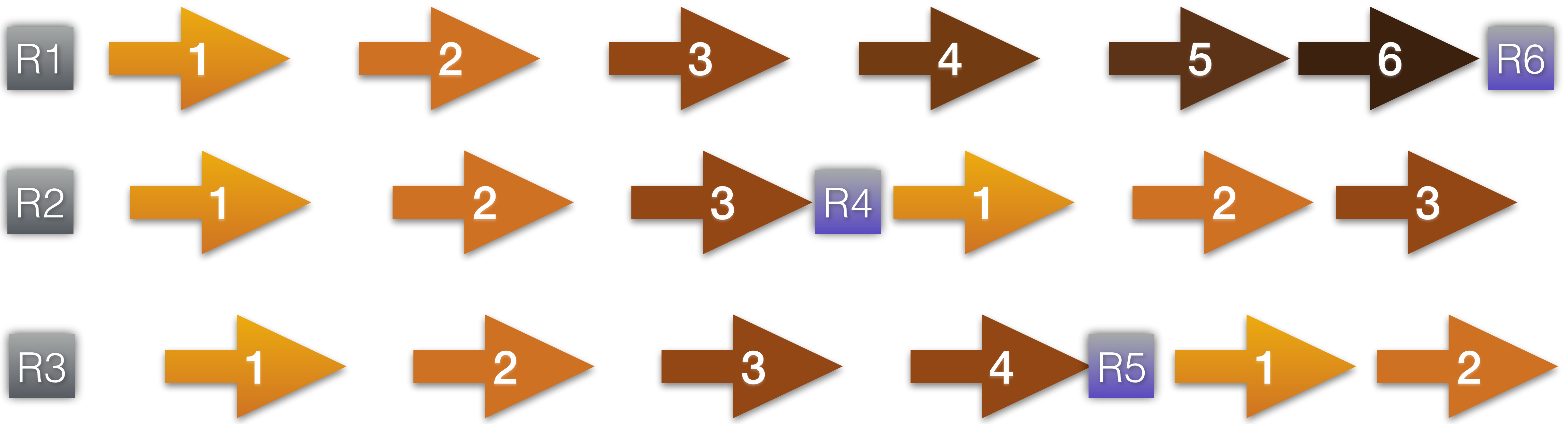
Co-routine Dynamic Scheduling

- Refill one task at a time
- Refill R5 as soon as R3 completes



Co-routine Dynamic Scheduling

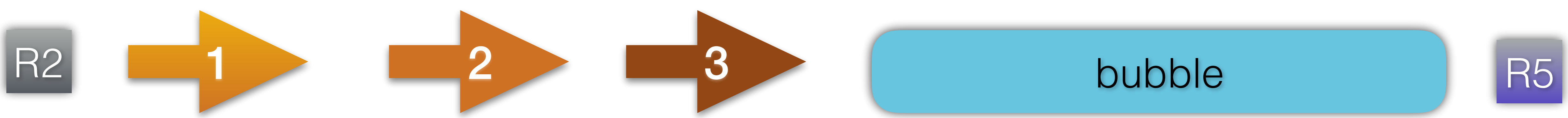
- Refill one task at a time
- Refill R6 as soon as R1 completes



Static vs Dynamic

- Is Dynamic always better?

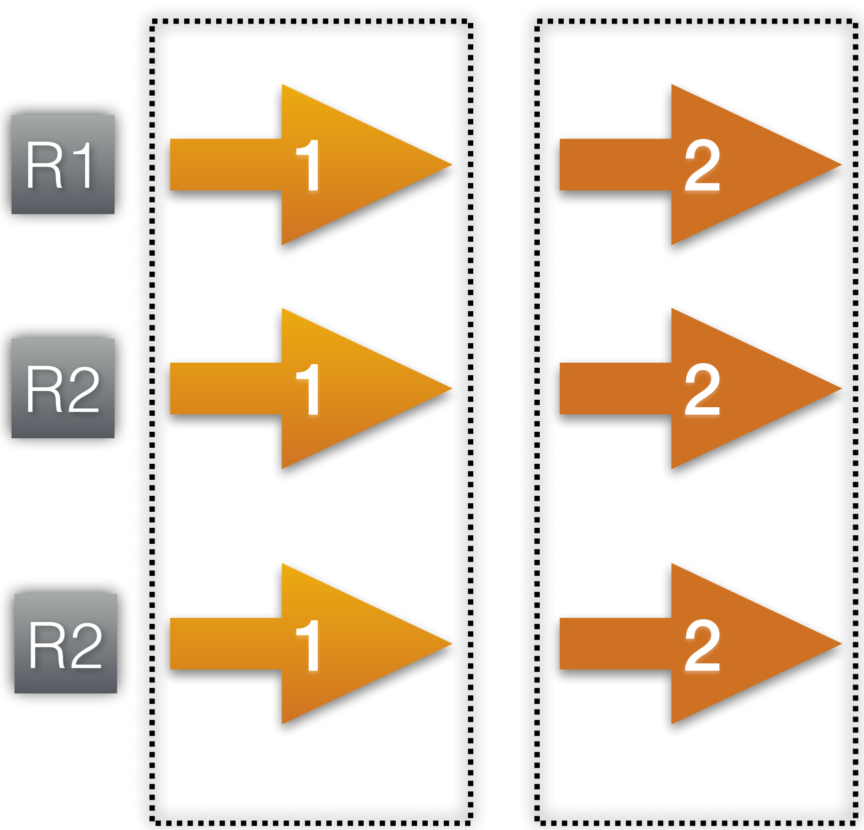
Static



Dynamic

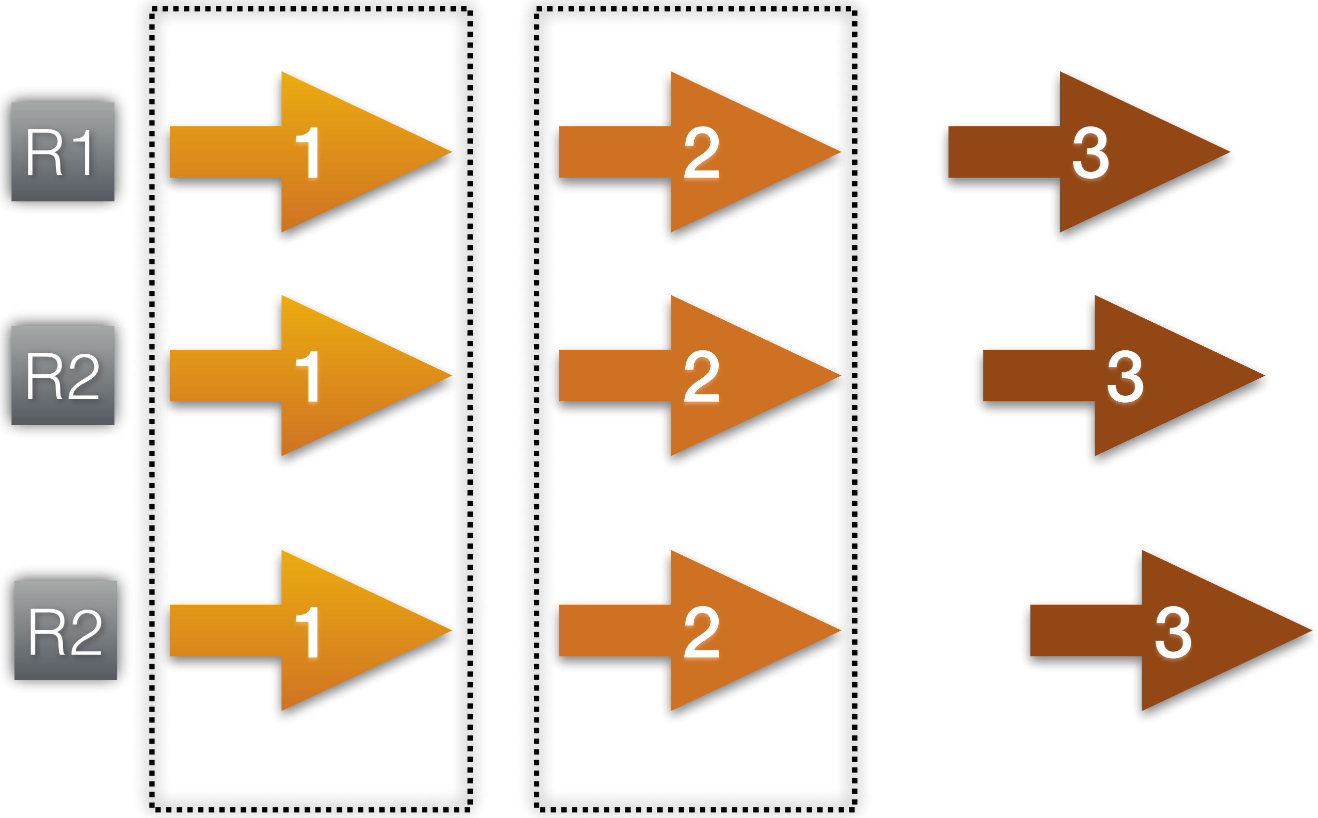
Co-routine Vectorization

- Static Scheduling



Co-routine Vectorization

- Hybrid Static+Dynamic Scheduling



Three Keys to High MLP

- Cooperative scheduling of co-routines
  `Yield` at memory requests

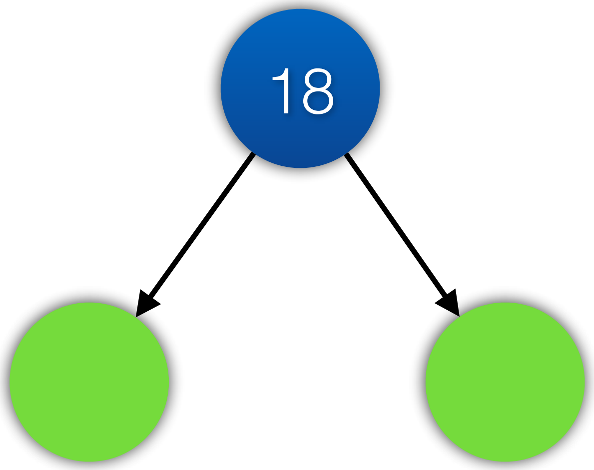
- Non-blocking loads overlap with computation
  `Prefetch` avoids instruction window overflow

- Branch misprediction penalty minimized
  `If`/**`Switch`** grouping, and *branchless* code

Binary Tree Lookup

```
node* BinaryTree::find(node* n, Key key){
    while (n) {
        if ( n->key == key )
            return n;

        if ( n->key < key )
            n = n->right;
        else
            n = n->left;
    }
    return n;
}
```



Binary Tree Hotspots

```
node* BinaryTree::find(node* n, Key key){
    while (n) {
        if ( n->key == key ) // 1. cache miss
            return n;

        if ( n->key < key )
            n = n->right;
        else
            n = n->left;
    }
    return n;
}
```

Binary Tree Hotspots

```
node* BinaryTree::find(node* n, Key key){
    while (n) {
        if ( n->key == key ) // 1. cache miss
            return n;

        if ( n->key < key )  // 2. branch
            n = n->right;    // misprediction
        else
            n = n->left;
    }
    return n;
}
```

Binary Tree Branchless

```
node* BinaryTree::find(node* n, Key key){
    while (n) {
        if ( n->key == key )
            return n;


        n = n->child[n->key < key];


    }
    return n;
}
```

Today: `Cimple` DSL for Experts

If it is fast and ugly, they will use it and curse you;
if it is slow, they will not use it.
- *David Cheriton*

[Jain, *The Art of Computer Systems Performance Analysis*]

Today: `Cimple` DSL for Experts

If it is fast and ugly, they will use it and curse you;
if it is slow, they will not use it.
- *David Cheriton*

- Performance critical database indices:
  Replace LLVM IR builders in JIT query engines
- C++ Standard Template Library replacement

Past:
Related Work

- GP: Group prefetching - [Chen et al'04]
  manual *static scheduling* for hash-join

- AMAC: Asynchronous Memory Access Chaining
  [Kocberber et al, VLDB'15]
  manual *dynamic scheduling*

Concurrent: C++20 co_routines

- SAP Hana [Psaropoulos et al, VLDB'18]
- Microsoft SQLServer [Jonathan et al, VLDB'18]
  automated *dynamic scheduling*
- Slower than manual GP!
  Pretty front-end, high-overhead backend
- Dynamic schedule only, no vectorization

Binary Tree Lookup

```
node* BinaryTree::find(node* n, Key key){
    while (n) {
        if ( n->key == key )
            return n;


        n = n->child[n->key < key];


    }
    return n;
}
```

Cimple DSL: Binary Tree

```
While(n).Do(

    If( n->key == key ).
    Then( Return(n) ).
    Stmt( n = n->child[n->key < key]; )
).
Return(n);
```

Cimple DSL: Binary Tree

```
While(n).Do(
  Prefetch(n).Yield().
  If( n->key == key ).
  Then( Return(n) ).
  Stmt( n = n->child[n->key < key]; )
).
Return(n);
```

Cimple DSL: Binary Tree

```
auto c = Coroutine(BST_find);
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
Body().
   While(n).Do(
      Prefetch(n).Yield().
      If( n->key == key ).
      Then( Return(n) ).
      Stmt( n = n->child[n->key < key]; )
   ).
   Return(n);
```

Co-routine State

```
Arg(node*, n).
Arg(KeyType, key).
```

- **Arg**uments, **Variable**s

```
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
```

Co-routine State

```
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
```

- **Result**

```
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
  node* _result;
```

Co-routine State

```
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
```

- Dynamic Schedule Finite State Machine `_state`

```
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
  node* _result;
  int _state = 0;
```

Dynamic Schedule: Co-routine with **switch**

```
bool Step() {
  switch(_state) {
  case 0:
        return false;
  case 1:

    return true;
  case _Finished:
    return true;
```

Dynamic Schedule: Co-routine with **switch**

```
bool Step() {
  switch(_state) {
  case 0:
      return false;
  case 1:
    return true;
  case _Finished:
    return true;
```

Duff's device co-routine

Scheduler Width

- **Width** high to hide latency, low to fit state in L1

```
template<int Width = 48>


    using Next = CoroutineState_SkipList_next_limit;
    SimplestScheduler<Width, Next>(len,
        [&](Next* cs, size_t i) {
                *cs = Next(&answers[i], IterateLimit,
                           iter[i]);
    });
```

Static Schedule: **for**

Vectorization friendly Struct-of-Arrays

```
bool SuperStep() {
  for(int _i = 0; _i < _Width ; _i++) {
    KeyType& k = _soa_k[_i];
    HashType& hash = _soa_hash[_i];


  }
```

Static Schedule: **for**

Vectorization friendly Struct-of-Arrays

```
bool SuperStep() {
  for(int _i = 0; _i < _Width ; _i++) {
    KeyType& k = _soa_k[_i];
    HashType& hash = _soa_hash[_i];


  }
  for(int _i = 0; _i < _Width ; _i++) {
    KeyType& k = _soa_k[_i];
    HashType& hash = _soa_hash[_i];

  }
```

Applications

Binary Search

```
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While( l != r ).Do(
  Stmts(R""( {
    int mid = (l+r)/2;
    bool less = (a[mid] < k);
    l = less ? (mid+1) : l;
    r = less ? r : mid;
   } )"").
   Prefetch(&a[(l+r)/2]).Yield()
).
Stmt( *result = l; );
```

Binary Search

```
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While( l != r ).Do(
  Stmts(R""( {
    int mid = (l+r)/2;
    bool less = (a[mid] < k);
    l = less ? (mid+1) : l;
    r = less ? r : mid;
     } )"").
   Prefetch(&a[(l+r)/2]).Yield()
).
Stmt( *result = l; );
```

Binary Search

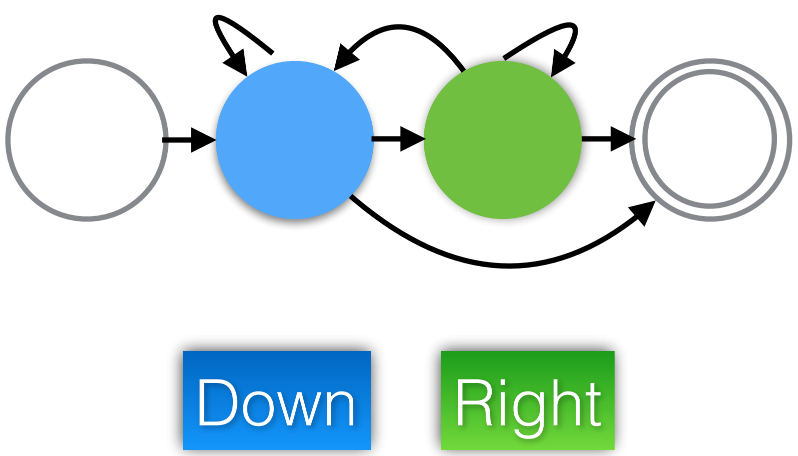
```
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While( l != r ).Do(
  Stmts(R""( {
    int mid = (l+r)/2;
    bool less = (a[mid] < k);
    l = less ? (mid+1) : l;
    r = less ? r : mid;
     } )"").
   Prefetch(&a[(l+r)/2]).Yield()
).
Stmt( *result = l; );
```

Skip List Lookup

```
VariableInit(SkipListNode*, n, {}).
VariableInit(uint8, ht, {pred->height}).
While(true).Do(
     While(ht > 0).Do( // down
       Stmt( n = pred->skip[ht - 1]; ).
       Prefetch(n).Yield().
       If(!less(k, n->key)).Then(Break()).
       Stmt( --ht; )
     ).
     If (ht == 0).Then( Return( nullptr )).
     Stmt( --ht; ).
     While (greater(k, n->key)).Do(
       Stmt( pred = n; n = n->skip[ht]; ).
       Prefetch(n).Yield().
     ).
     If(!less(k, n->key)).Then(
        Return( n )));
```

Skip List Lookup

```
VariableInit(SkipListNode*, n, {}).
VariableInit(uint8, ht, {pred->height}).
While(true).Do(
    While(ht > 0).Do( // down
      Stmt( n = pred->skip[ht - 1]; ).
      Prefetch(n).Yield().
      If(!less(k, n->key)).Then(Break()).
      Stmt( --ht; )
    ).
    If (ht == 0).Then( Return( nullptr )).
    Stmt( --ht; ).
    While (greater(k, n->key)).Do(
      Stmt( pred = n; n = n->skip[ht]; ).
      Prefetch(n).Yield().
    ).
    If(!less(k, n->key)).Then(
       Return( n )));
```



Skip List Iteration

```
While( limit-- ).Do(
    Prefetch(n).Yield().
    Stmt( n = n->skip[0]; )
).
Prefetch(n).Yield().
Return( n->key );
```

- Pointer chasing

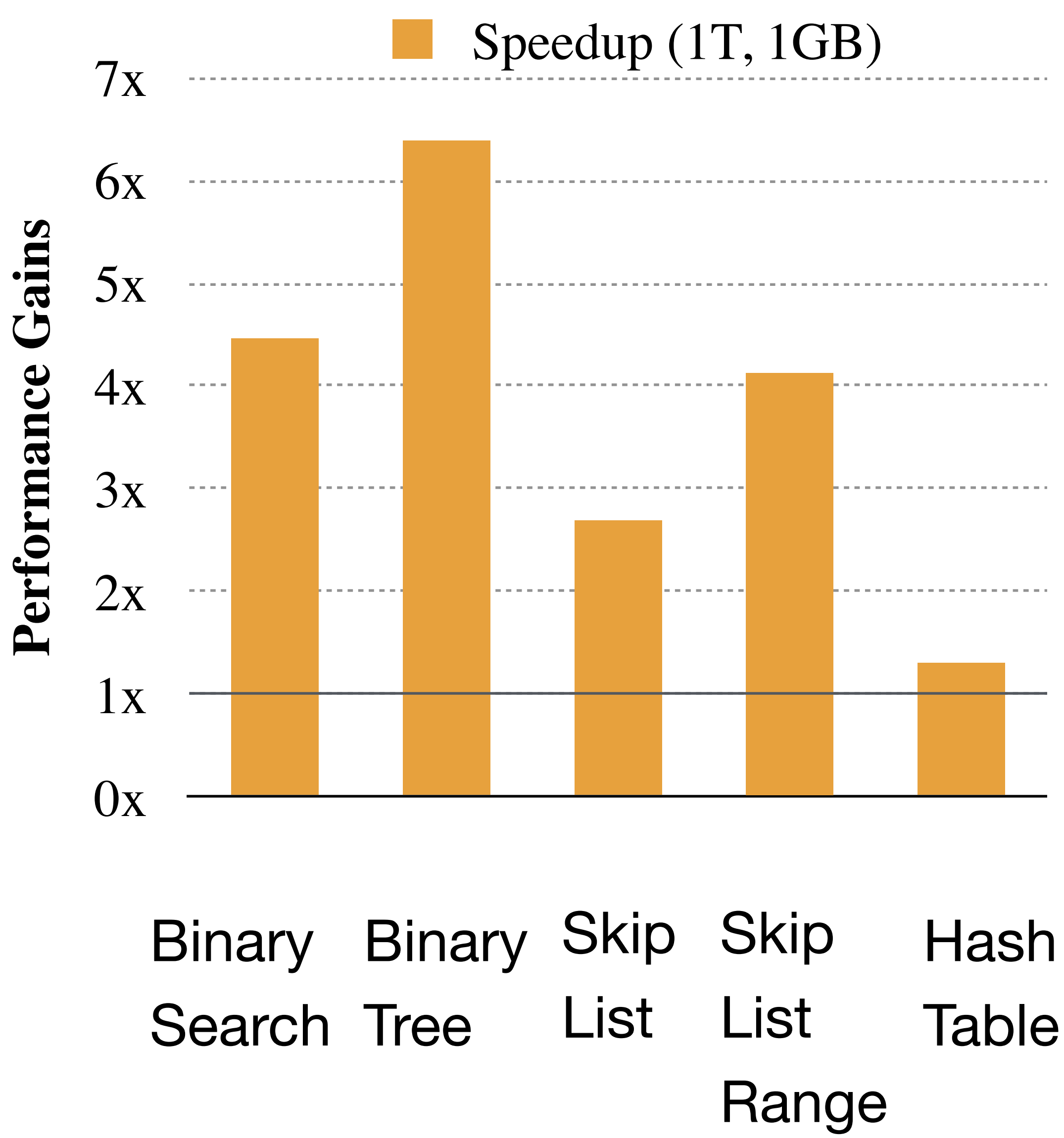
Hash Table Lookup (Linear Probing)

```
Result(KeyValue*).
Arg(KeyType, k).
Variable(HashType, hash).
Body().
  Stmt ( hash = Murmur3::fmix(k); ).
  Stmt ( hash &= this->size_1; ).Yield().
  Prefetch( &ht[hash] ).Yield()
  << R""(
  while (ht[hash].key != k &&
         ht[hash].key != 0) {
      hash++;
      if (hash == size) hash = 0;
  } )"" <<
  Return( &ht[hash] );
```

- SIMD
- One cache line

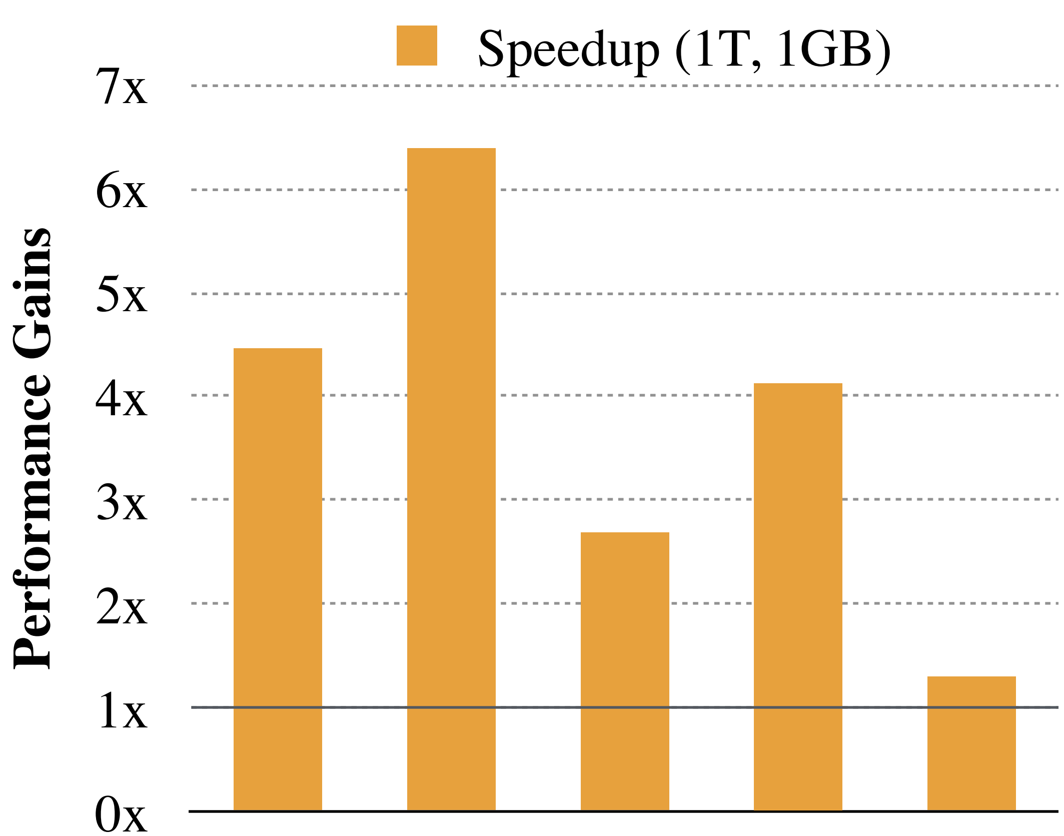
Performance Evaluation

Performance



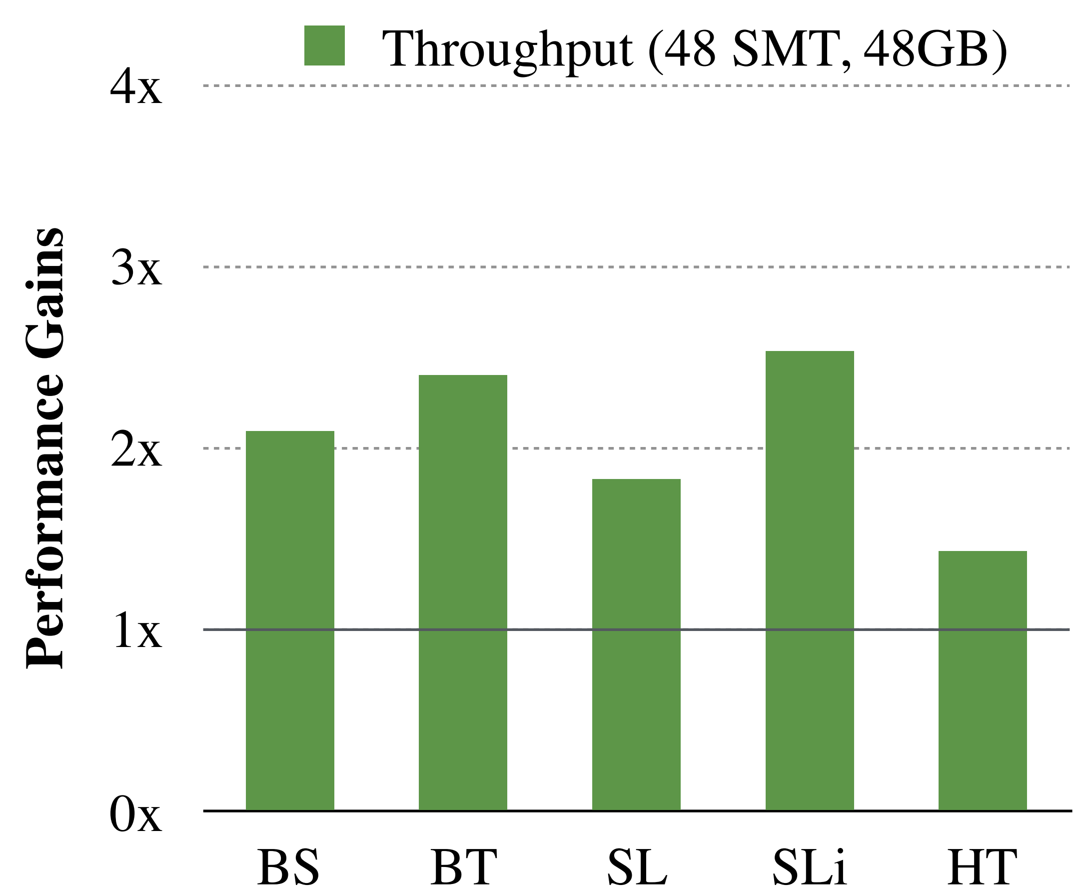
[Default indices of VoltDB/RocksDB]

Performance



Binary Search | Binary Tree | Skip List | Skip List Range | Hash Table

[Default indices of VoltDB/RocksDB]



Haswell EP (24 cores, 2-way SMT)

Thread Level Parallelism

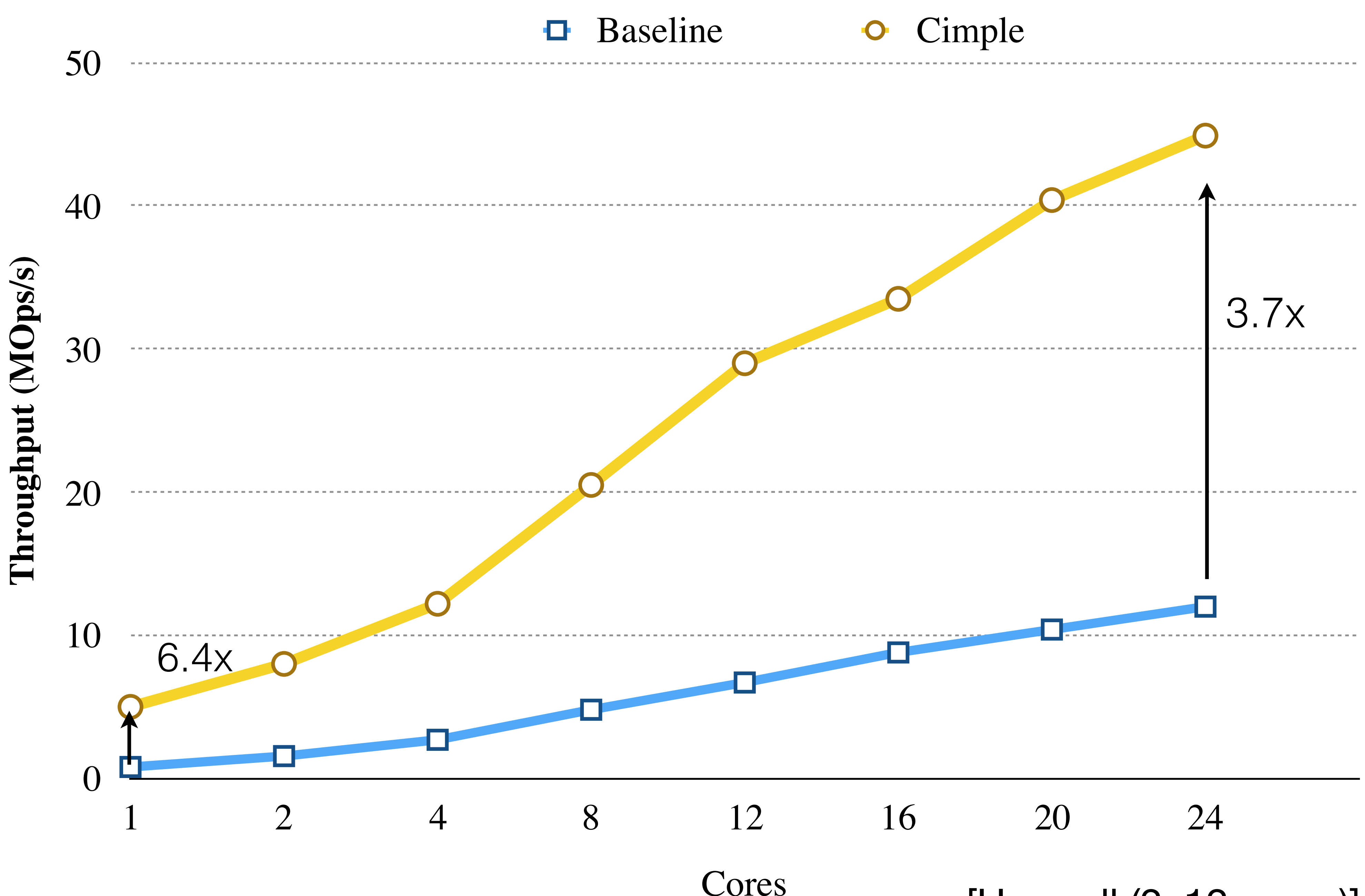
✅ Multi-core

✅ SMT hardware thread

❌ Single-thread OS context switching?

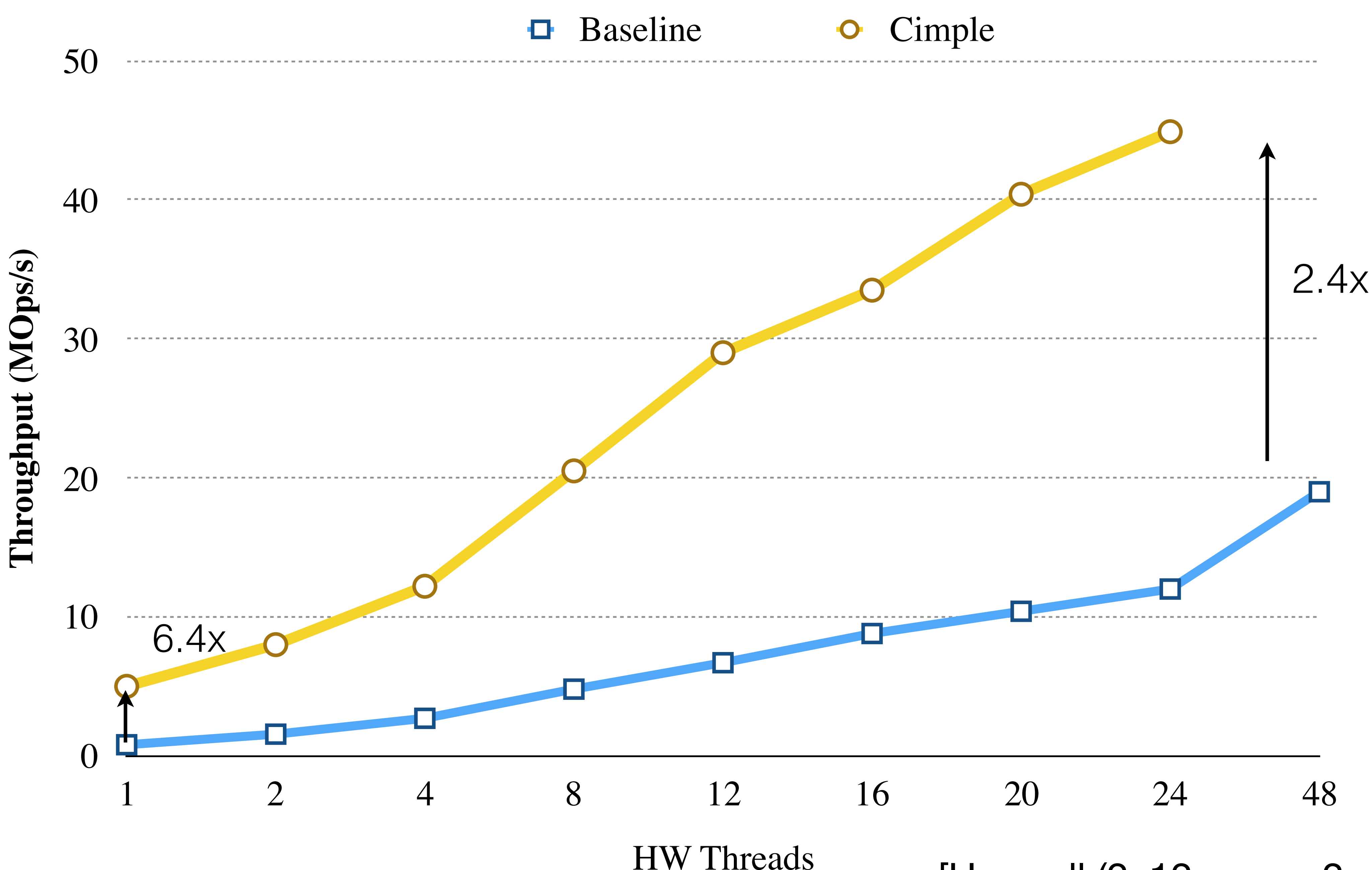
50x slower

Cimple Throughput Gains on Multicore



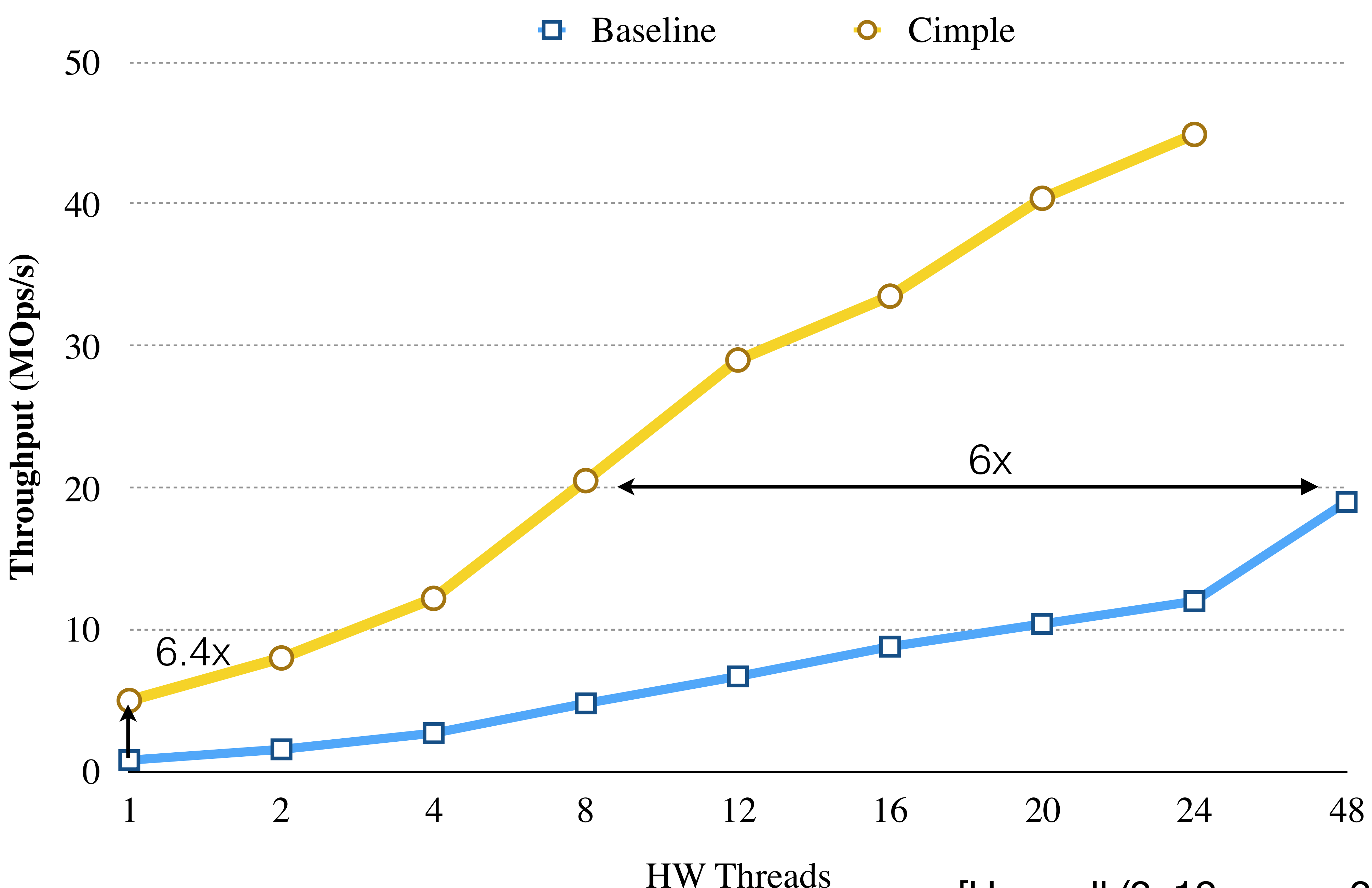
[Haswell (2x12 cores)]

Cimple Throughput Gains vs Hyper-threading



[Haswell (2x12 cores, 2-way SMT)]

Cimple Throughput Gains vs Hyper-threading



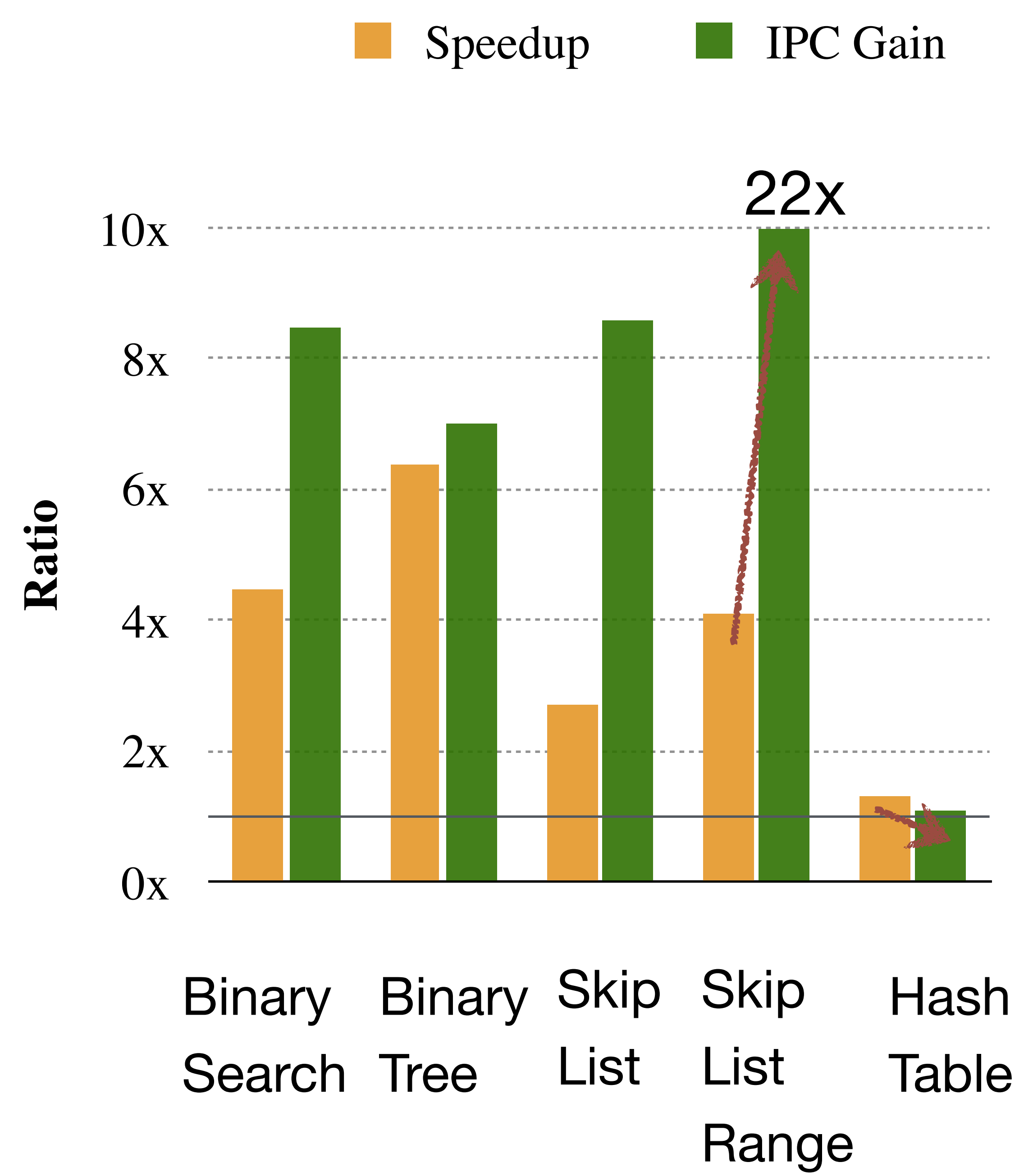
[Haswell (2x12 cores, 2-way SMT)]

IPC Analyzed

- SkipList Range: scheduler overhead

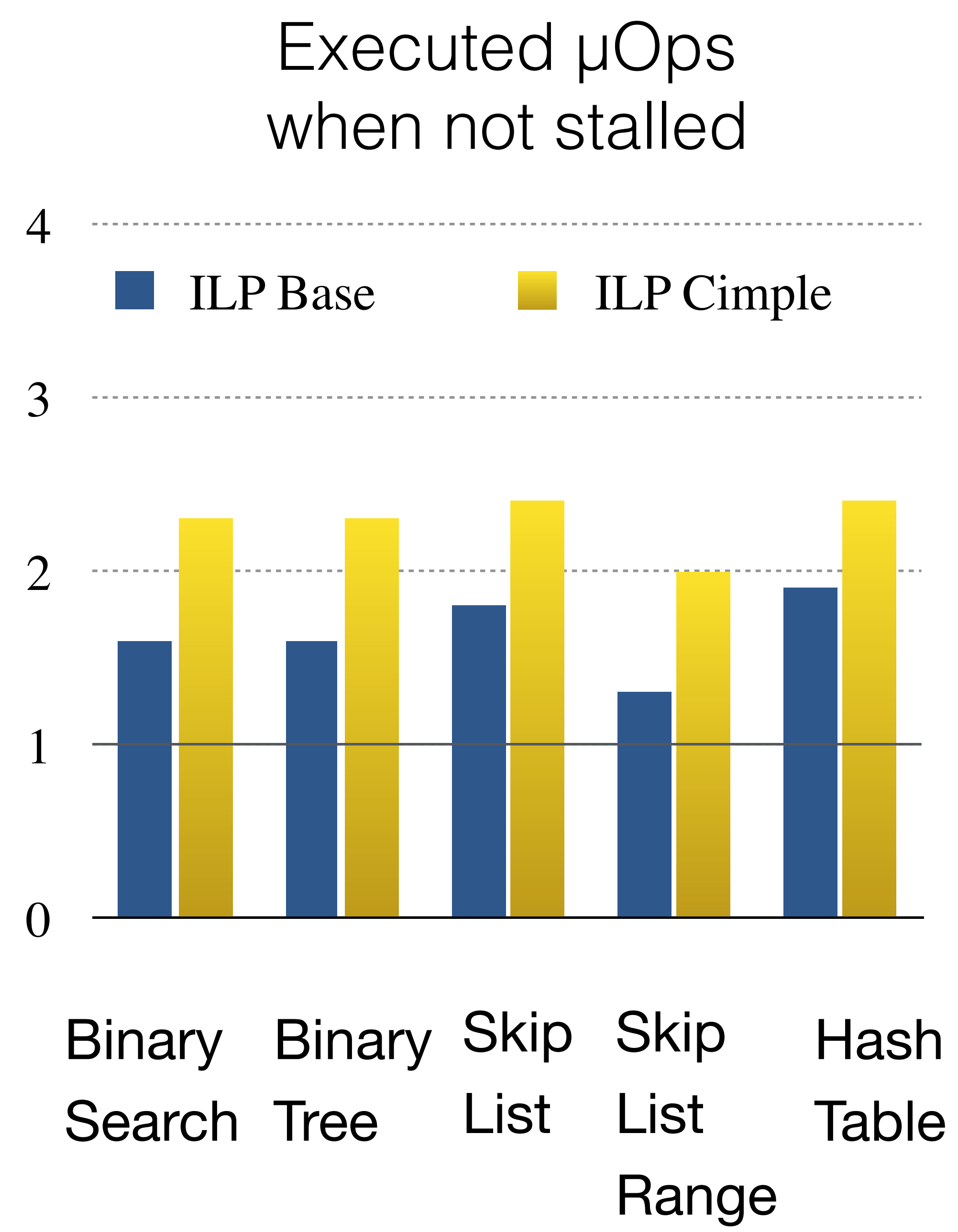
```
p = p->next;
```

- HashTable: SIMD vectorization



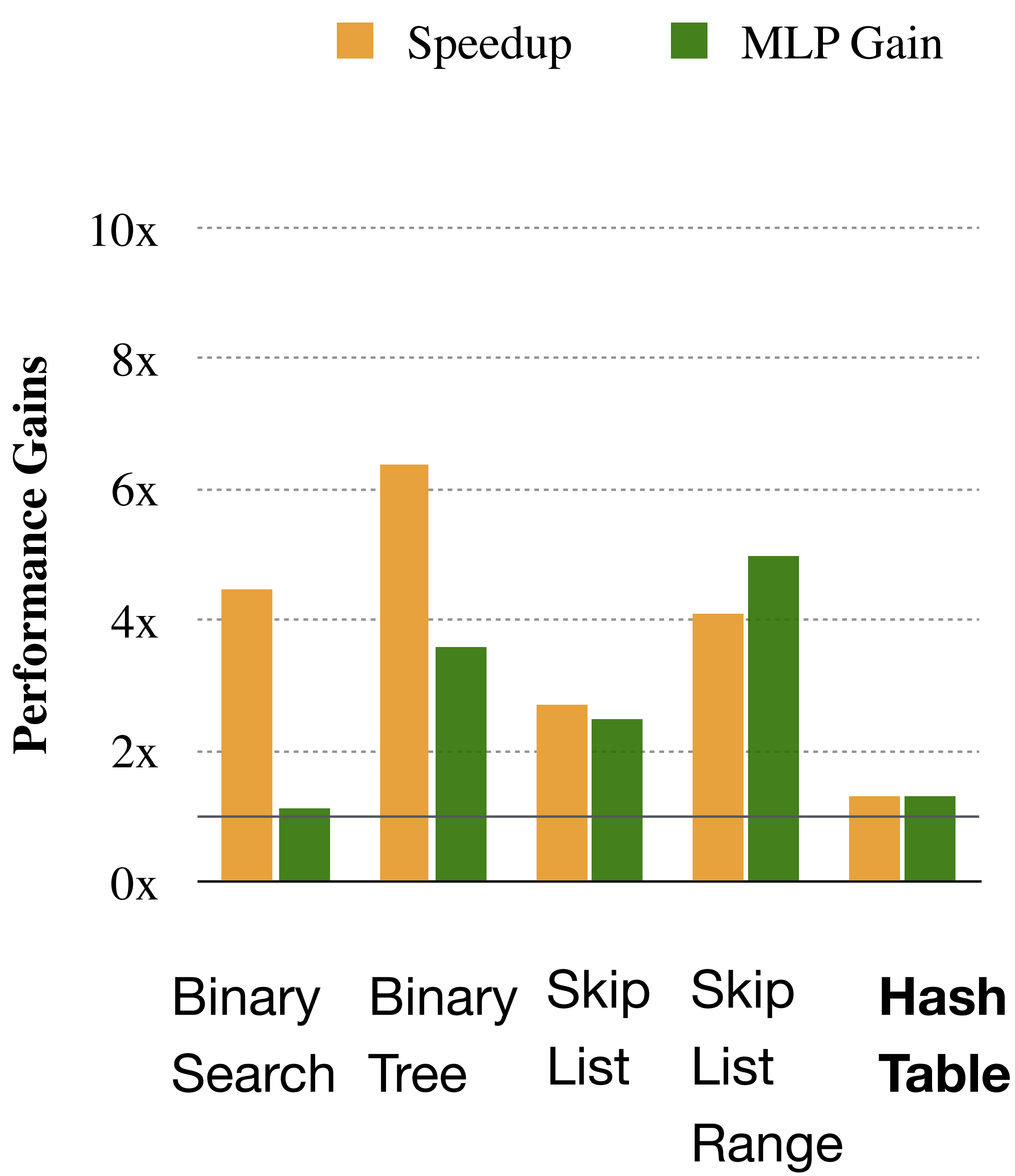
ILP Analyzed

- Uncovered more parallelism
- Absorbed scheduler overhead



MLP Analyzed

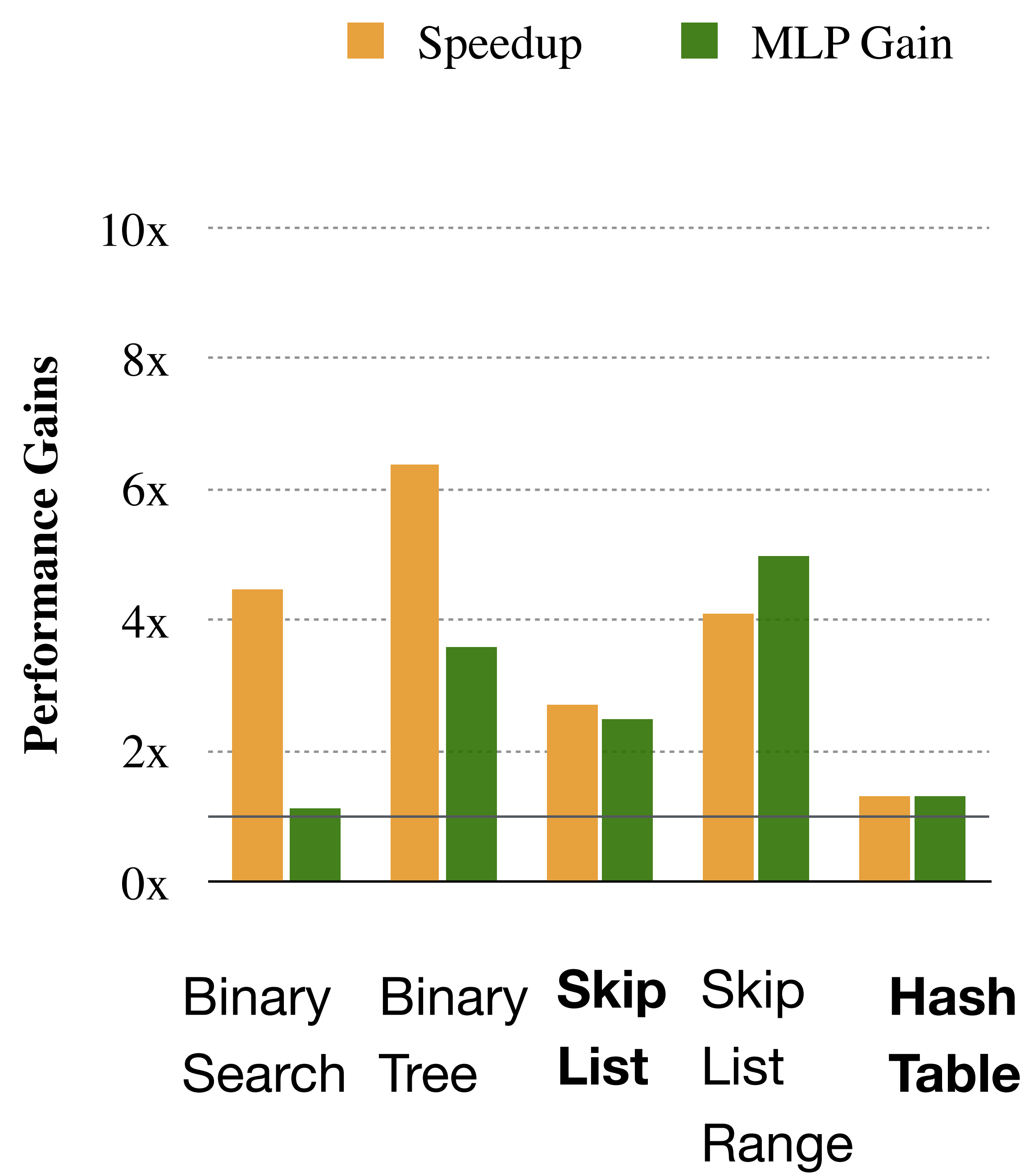
- HashTable -
  OoO HW extracts MLP



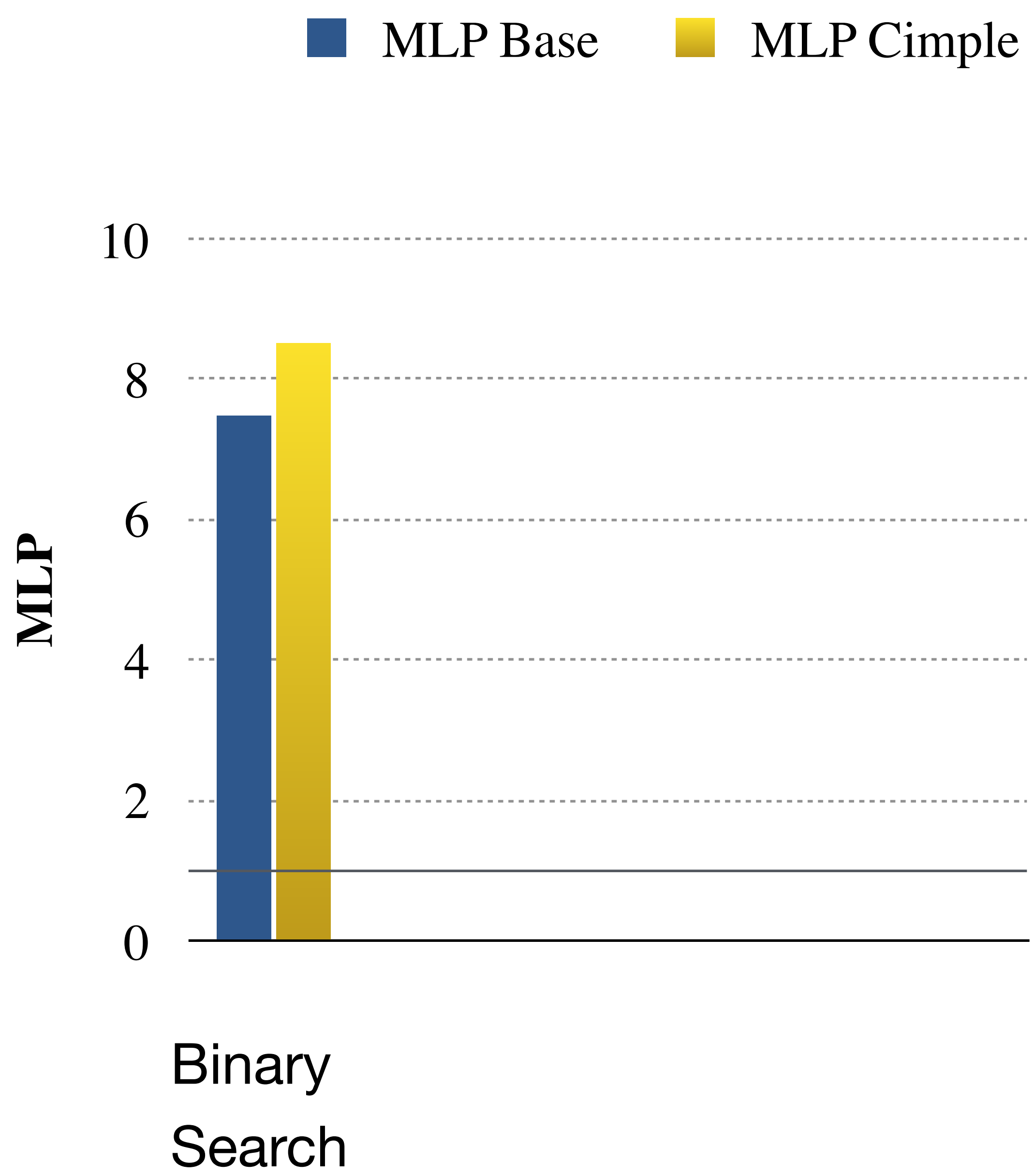
MLP Analyzed

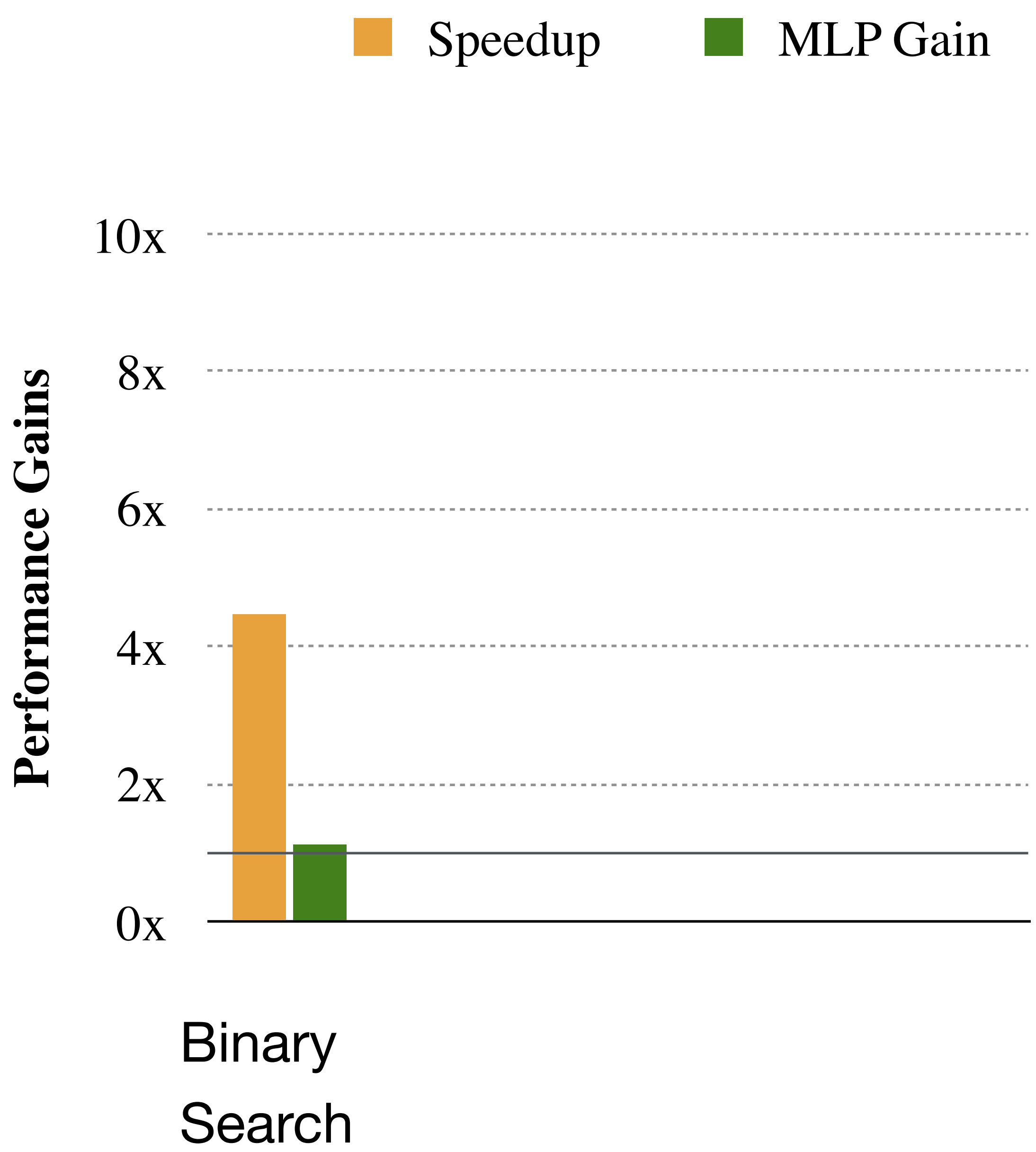
- HashTable -
  OoO HW extracts MLP

- SkipList - speedup
  matching MLP gains

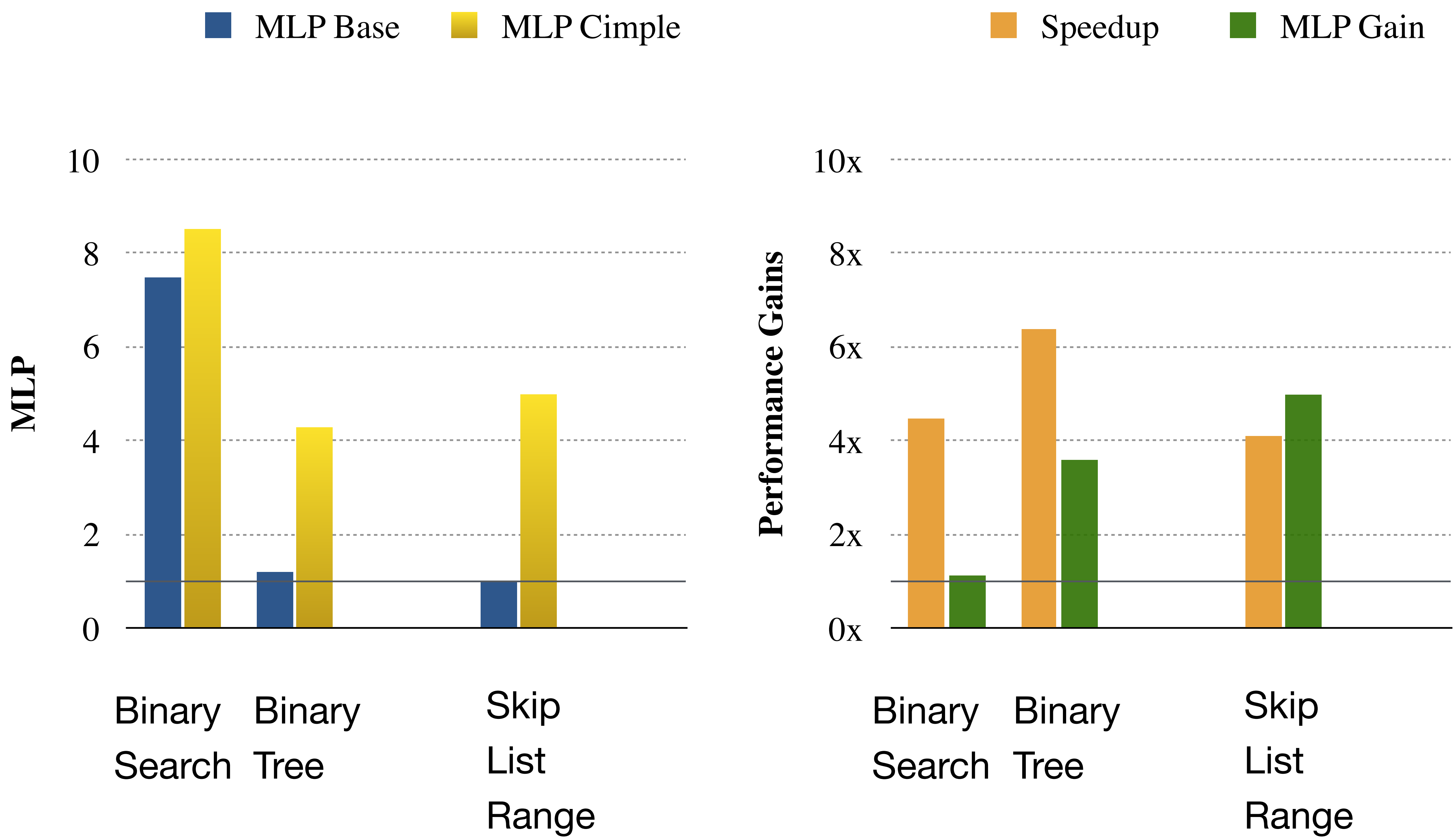


MLP: Ineffective or Low





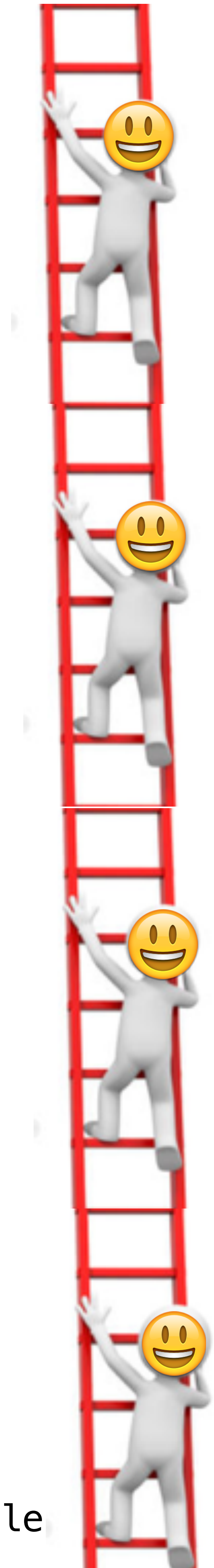
MLP: Ineffective or Low



MLP Improvement Paths

- Increased Efficiency
  - Static scheduling
  - Vectorization

- Increased Effectiveness
  - Dynamic scheduling: no bubbles
  - Branchless code



Conclusion

- Fast
  - up to 6.4× speedup
- Portable DSL *(for Stephanies)*
  - template libraries
  - database query engines
- Next: C++ standards *(for Joes)*

Thanks

Vladimir Kiriansky
vlk@csail.mit.edu

U.S. DEPARTMENT OF ENERGY | Office of Science

TOYOTA RESEARCH INSTITUTE

CSAIL