Leveraging Caches to Accelerate Hash Tables and Memoization

Guowei Zhang and Daniel Sanchez

MICRO 2019

Massachusetts Institute of Technology

CSAIL

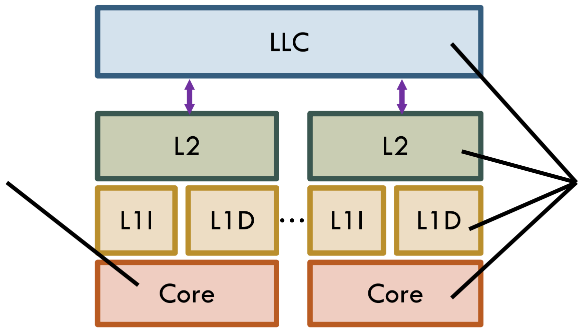
Executive summary

- Hash tables suffer from poor core utilization & poor spatial locality

- **HTA** accelerates hash tables with simple ISA & HW changes
  - Adopts *HTA table format* that leverages cache characteristics
  - Leaves rare cases to software

Flat-HTA

Reduces runtime overheads



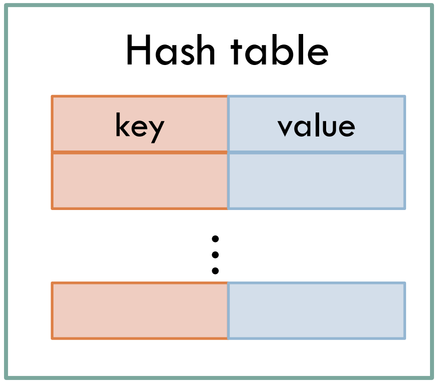
Hierarchical-HTA

Improves spatial locality

- HTA accelerates hash-table-intensive applications by up to 2x

- HTA-based **memoization** improves performance significantly

Hash table performance is critical




```
found, value = hashtable.lookup(key);

hashtable.insert(key, value);

hashtable.delete(key);
```


Database

Key-value store


Networking

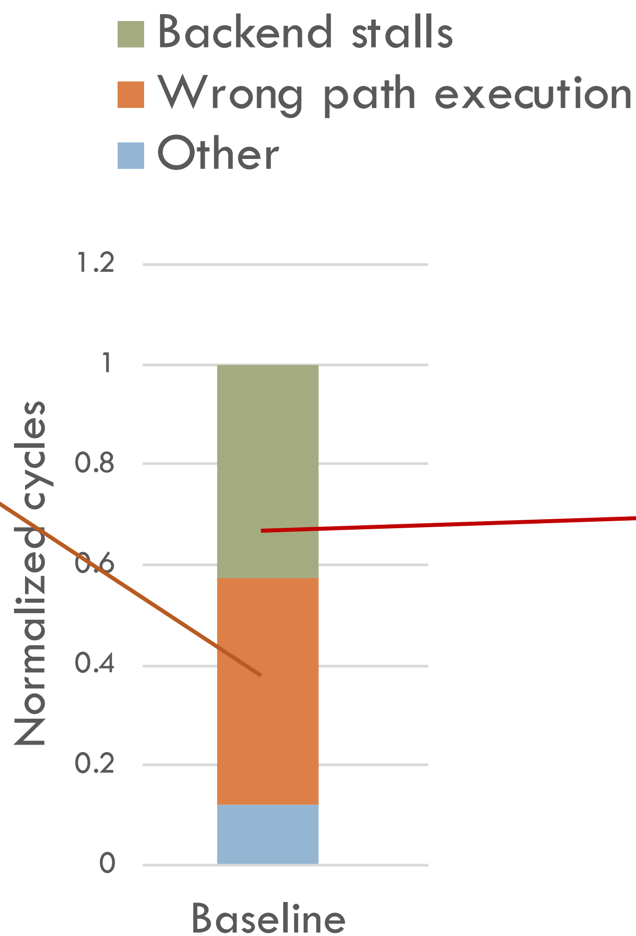

Genomics

Memoization

- Hash table performance is critical for **memoization**
  - Uses **hash tables** to skip repetitive computation
  - Beneficial only if hash table lookups are cheaper than memoized code

Issue 1: Poor core utilization

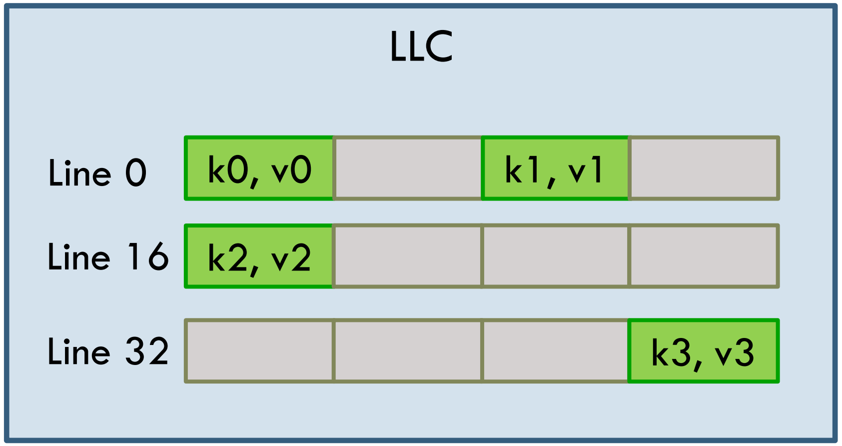
Data-dependent branches
- High misprediction rate
- High penalty



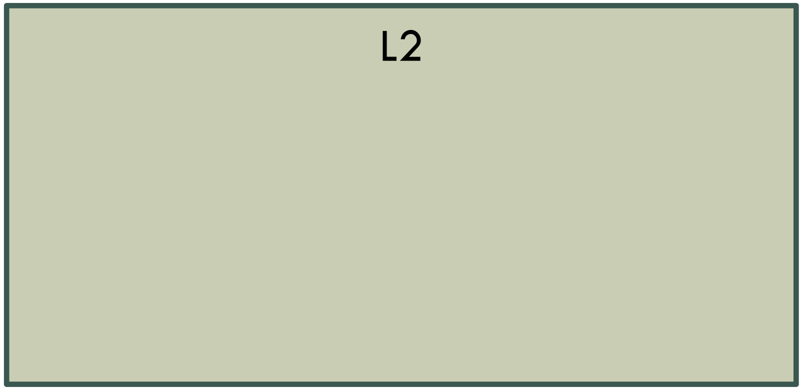
Poor use of core backend
- Frequent misses
- Hard-to-overlap due to too many µops

Flat-HTA reduces runtime overheads!

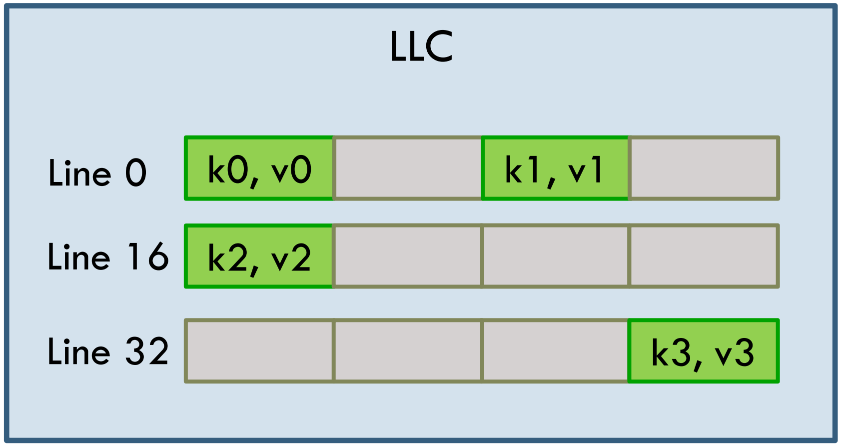
Issue 2: Poor spatial locality



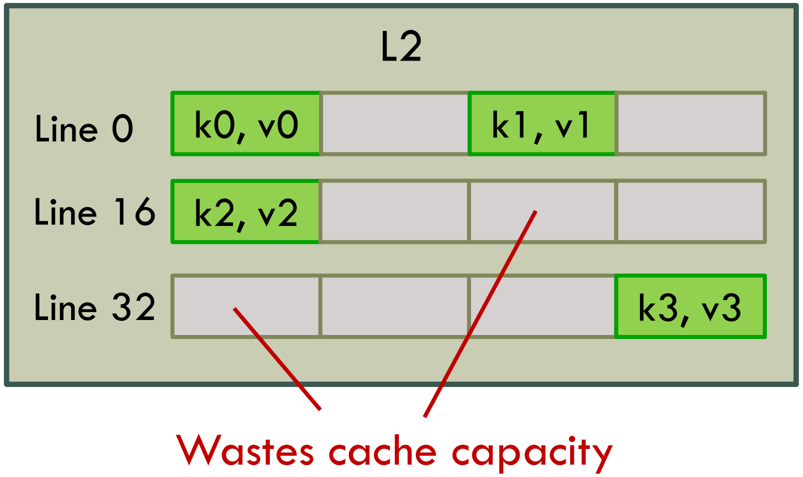
Conventional system



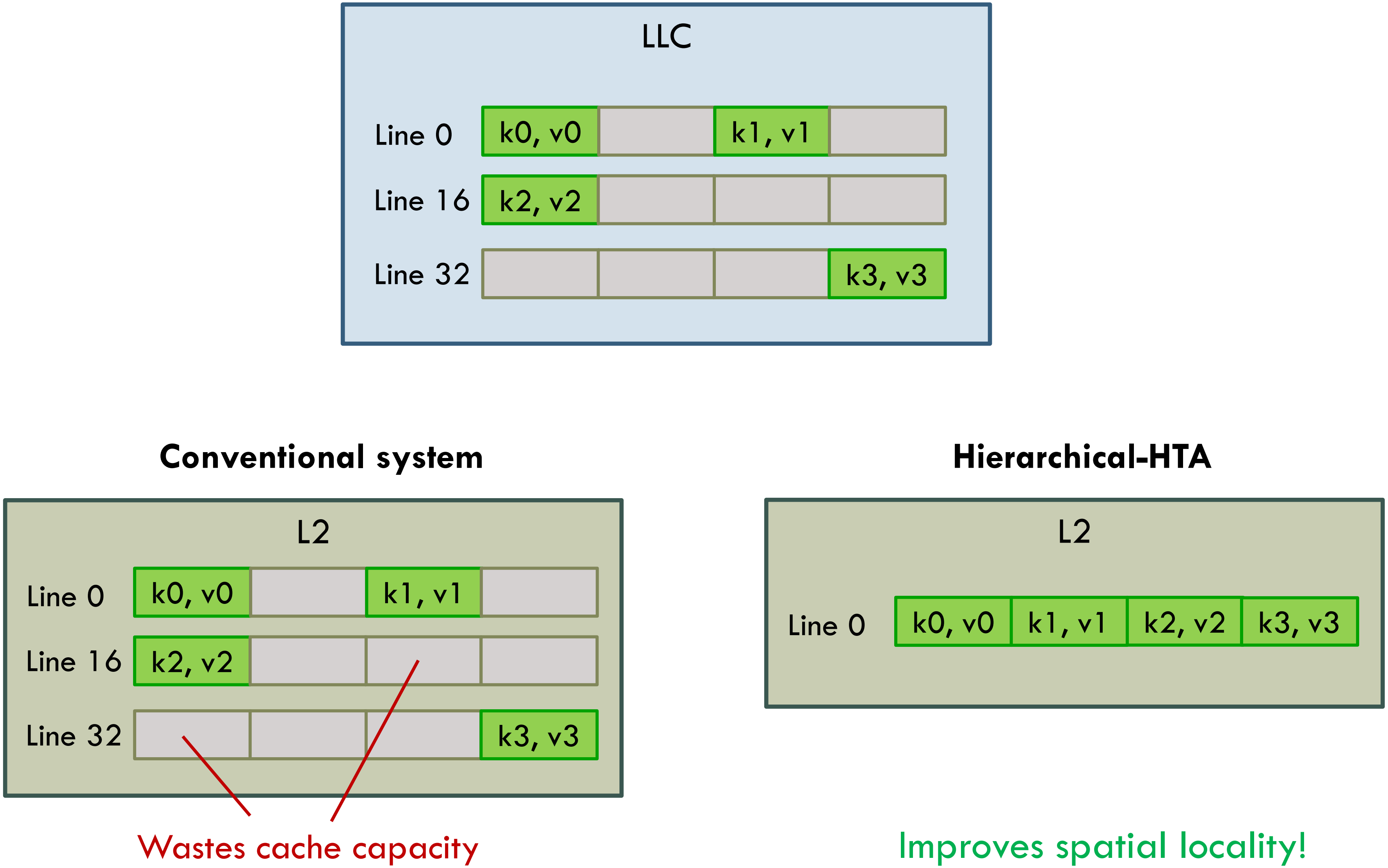
Issue 2: Poor spatial locality



Conventional system



Issue 2: Poor spatial locality



Prior hardware acceleration underused caches

- Domain-specific management [Costa 2000, Choi 2008, Chalamalasetti 2013, Lim 2013, Gope 2017…]
  - E.g., PHP processing, distributed key-value store, memoization
  - Requires dedicated on-chip storage (e.g., 98KB [Costa et al 2000])
  - Or bypasses memory hierarchy [Lloyd 2017, Tanaka 2014, Xu 2016…]

HTA is general

HTA avoids dedicated on-chip storage

HTA exploits memory hierarchy for spatial locality

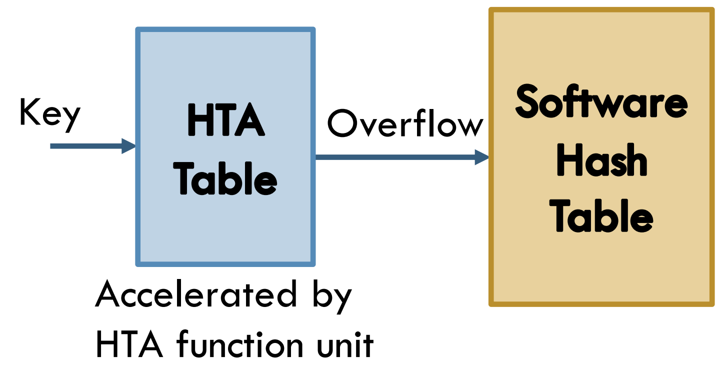
HTA: Hash Table Acceleration

HTA overview

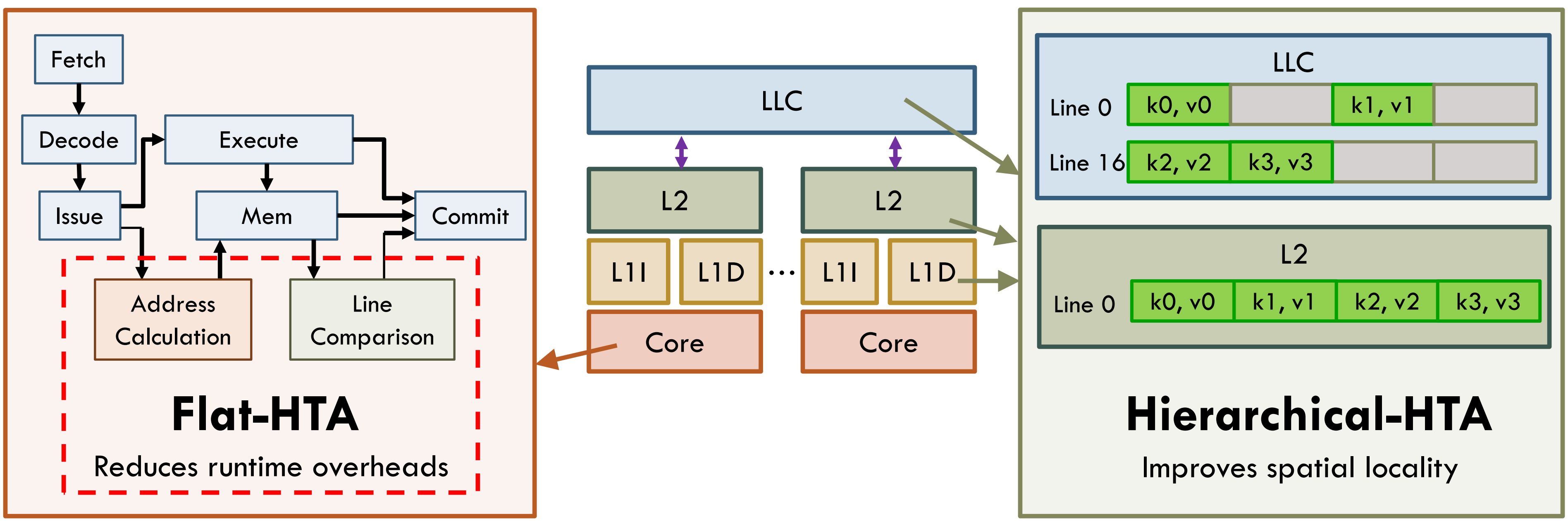
1. **Table format**

2. **ISA extensions**

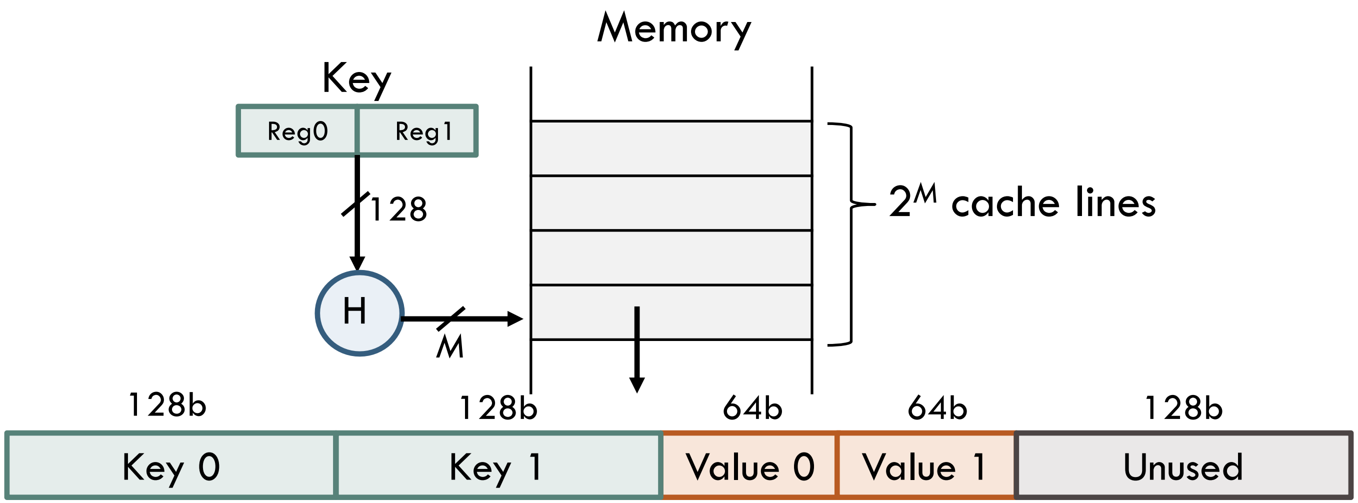
Make the common case fast!



3. **Hardware implementation**



HTA Table format



Conventional table

- Variable number of probes
- Introduces hard-to-predict branches
- Minimizes work

```
while (key != curSlot.key) {
     // Probe next slot
}
```

HTA table

- Small, fixed number of probes
- Overflows are handled by software path
- Avoids hard-to-predict branches
- Enables hardware acceleration

HTA ISA extensions

Single-threaded lookup

```
lookup:   hta_lookup <table_id>, <key_reg>, <value_reg>, done
          call swLookup # Accesses software hash table
done:     …
```

Branch semantics

- Easy to predict
- Exploits existing predictors

```
if (key is found) or (line is not full):
    taken # done
else:
    not taken # call swLookup
```

Single-threaded insert

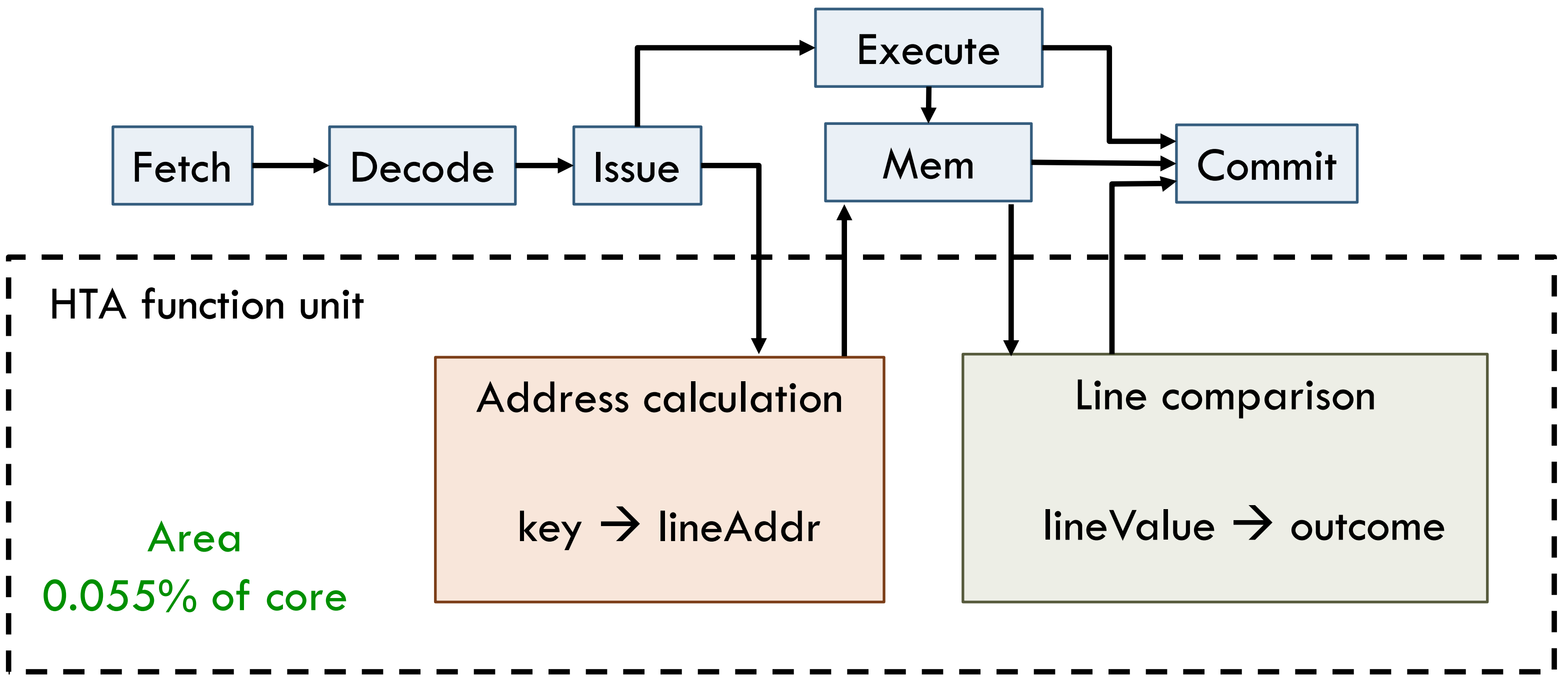
```
insert:   hta_swap <table_id>, <key_reg>, <value_reg>, done
          call swHandleInsert # Accesses software hash table
done:     …
```

Multi-threaded insert

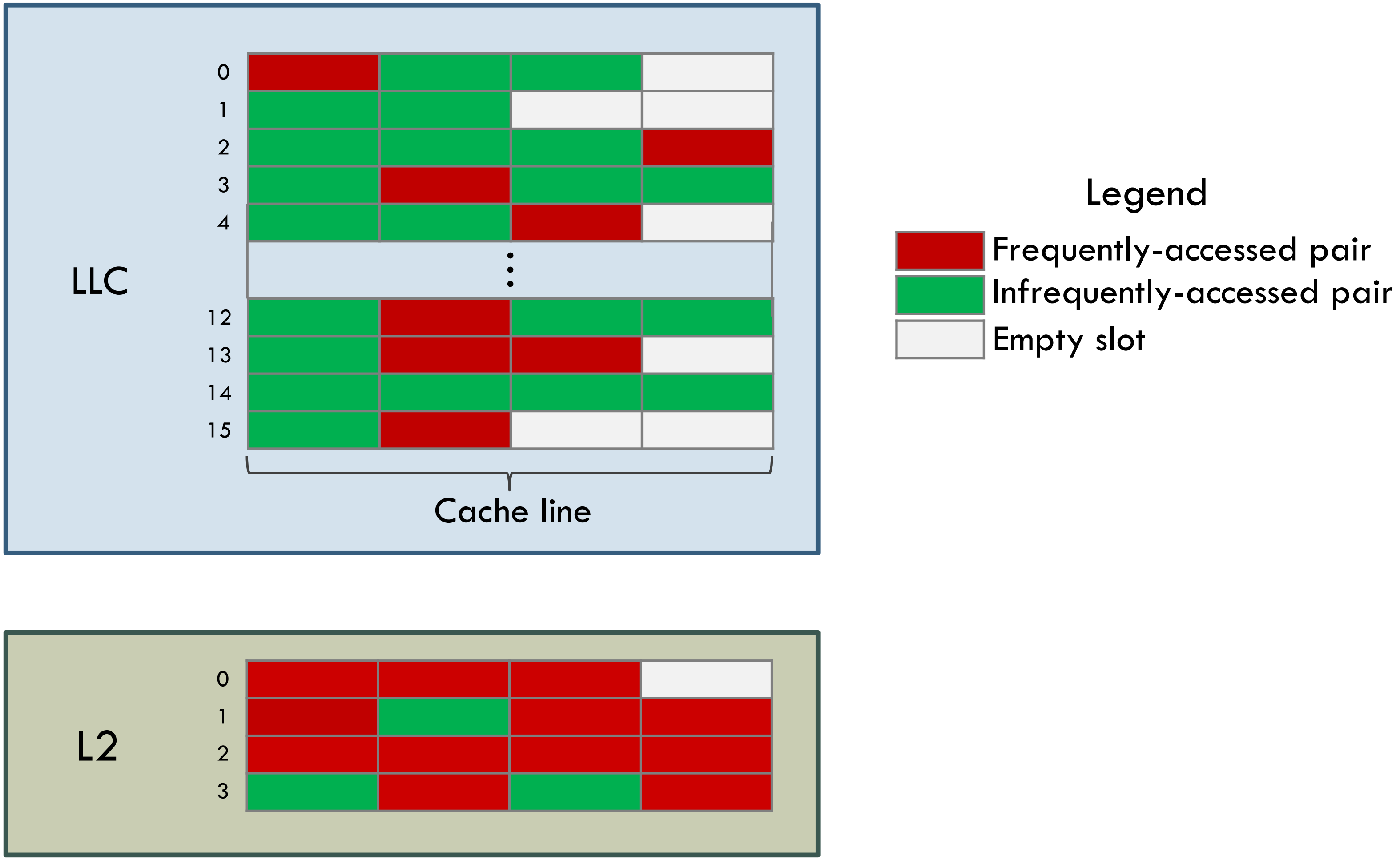
```
insert:   hta_update <table_id>, <key_reg>, <value_reg>, done
          call swLockLine
          hta_swap <table_id>, <key_reg>, <value_reg>, release
          call swHandleInsert
release:  call swUnlockLine
done:     …
```

- We prototype a CISC (x86) implementation
- RISC is also possible

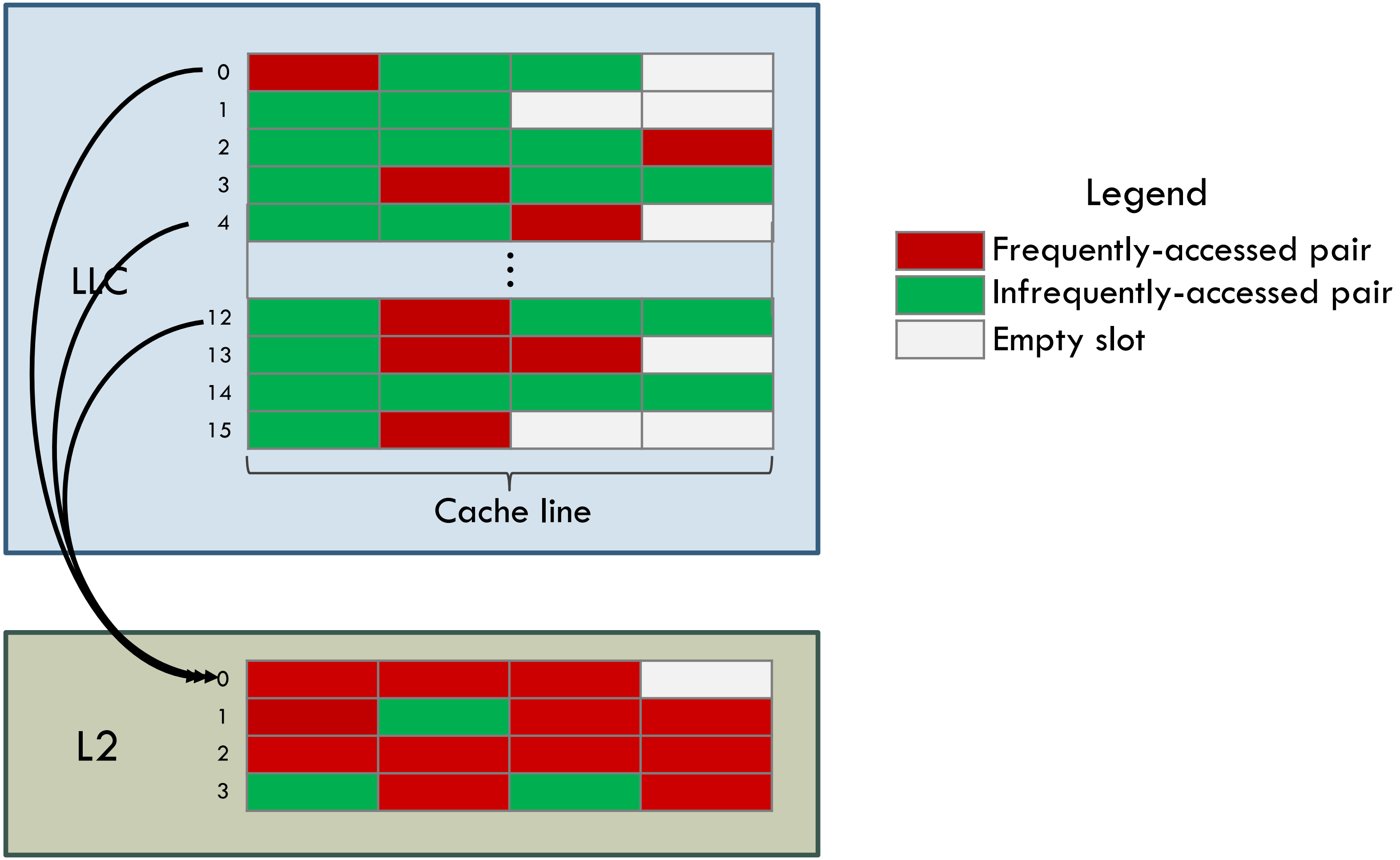
Flat-HTA implementation



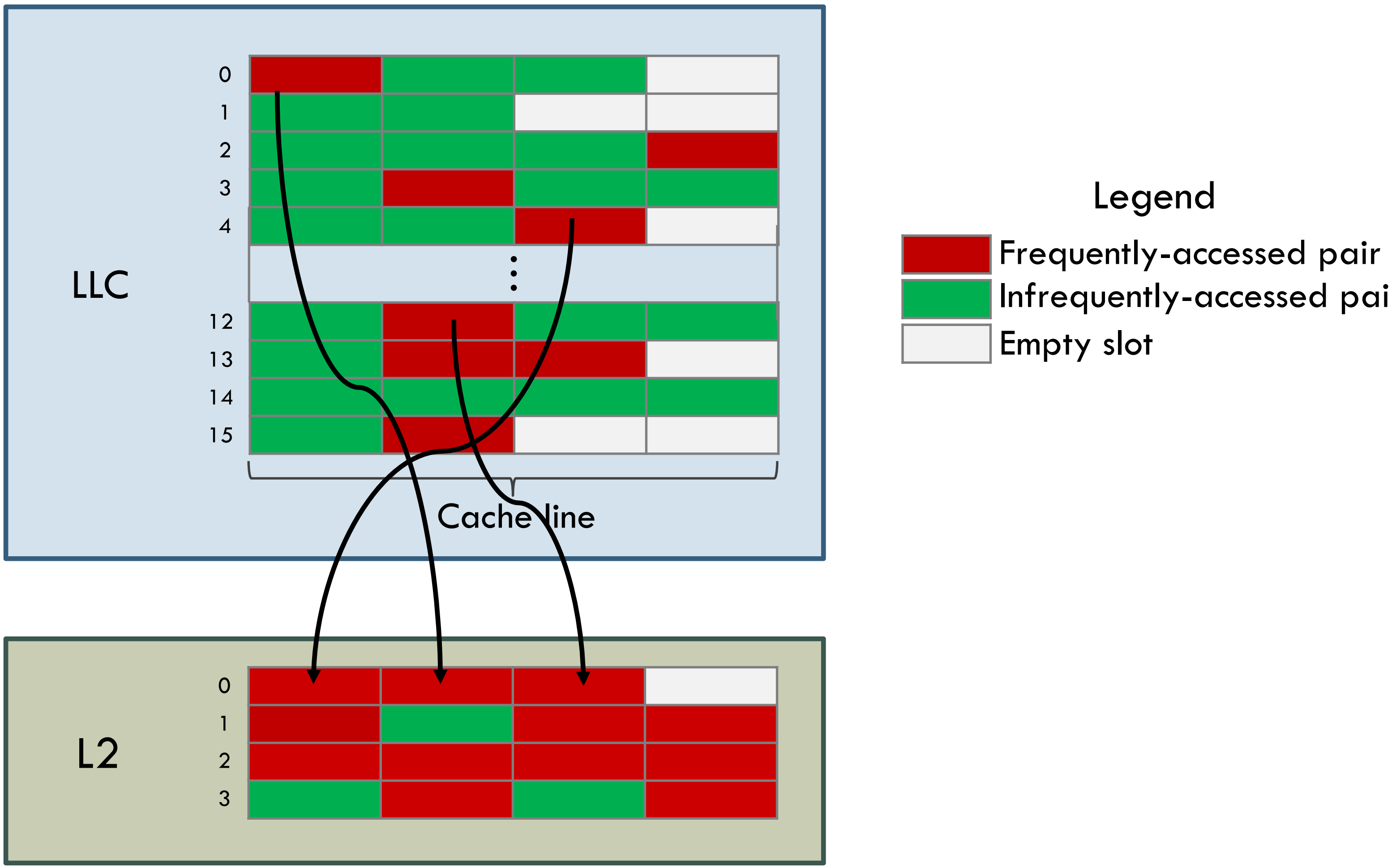
Hierarchical-HTA overview



Hierarchical-HTA overview



Hierarchical-HTA overview

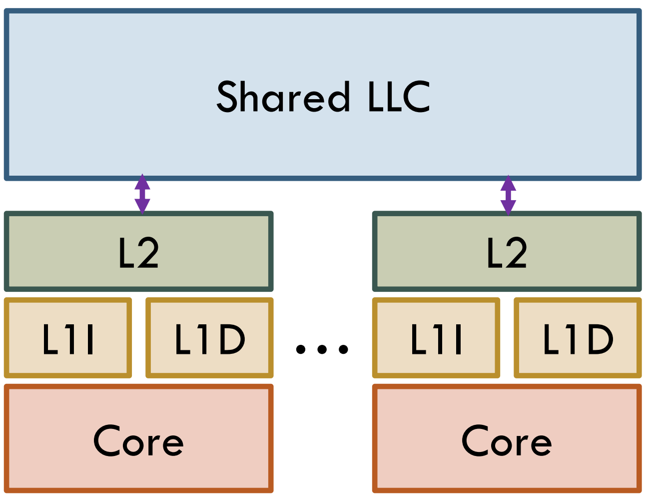


Check out paper for more

- Hierarchical-HTA implementation
  - Maintains coherence conservatively
  - Handles overflows conservatively
- Details on ISA and Flat-HTA implementation

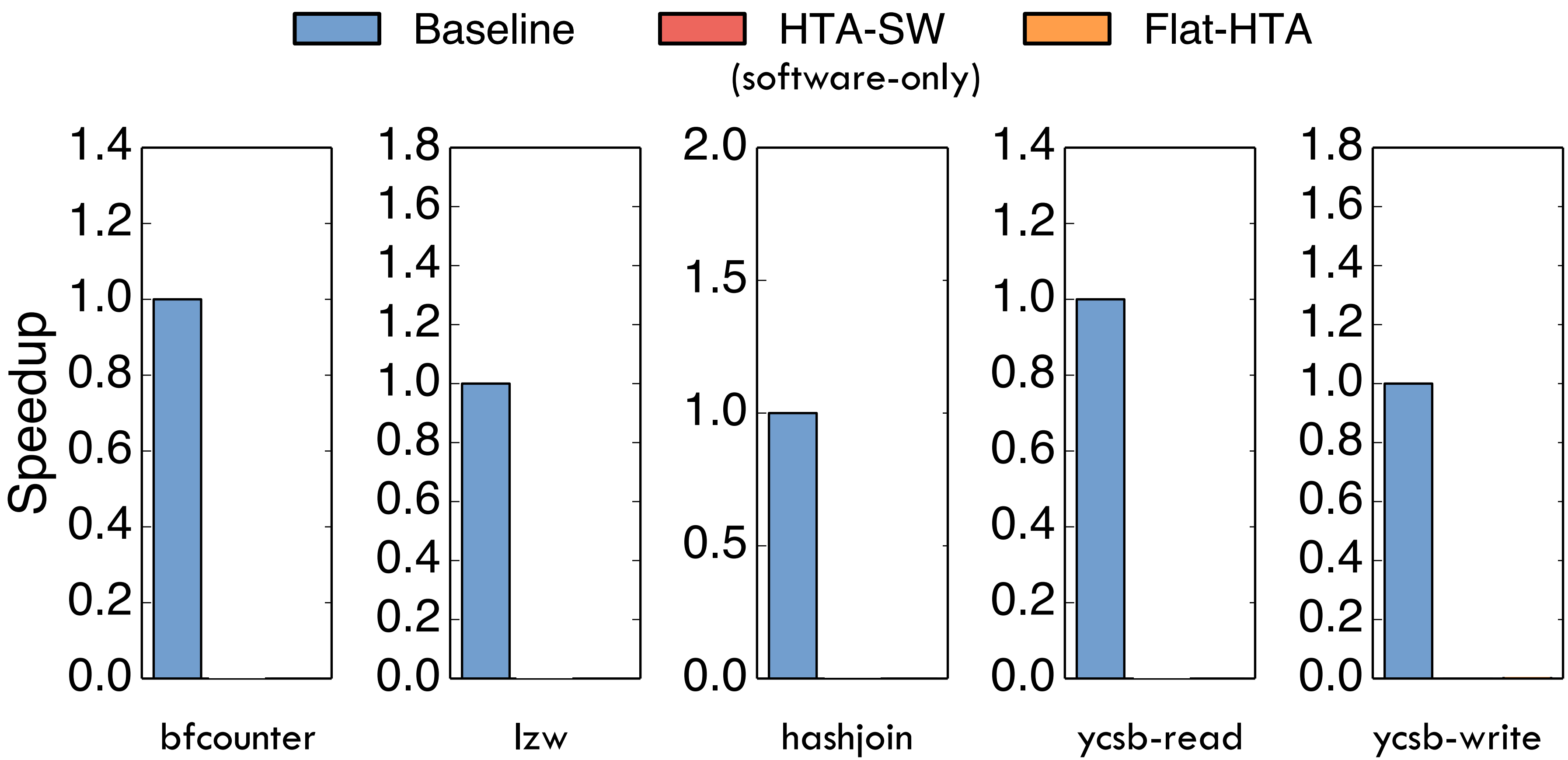
Methodology

- Simulation with zsim
- System
  - 1 to16 cores
  - 2MB LLC per core

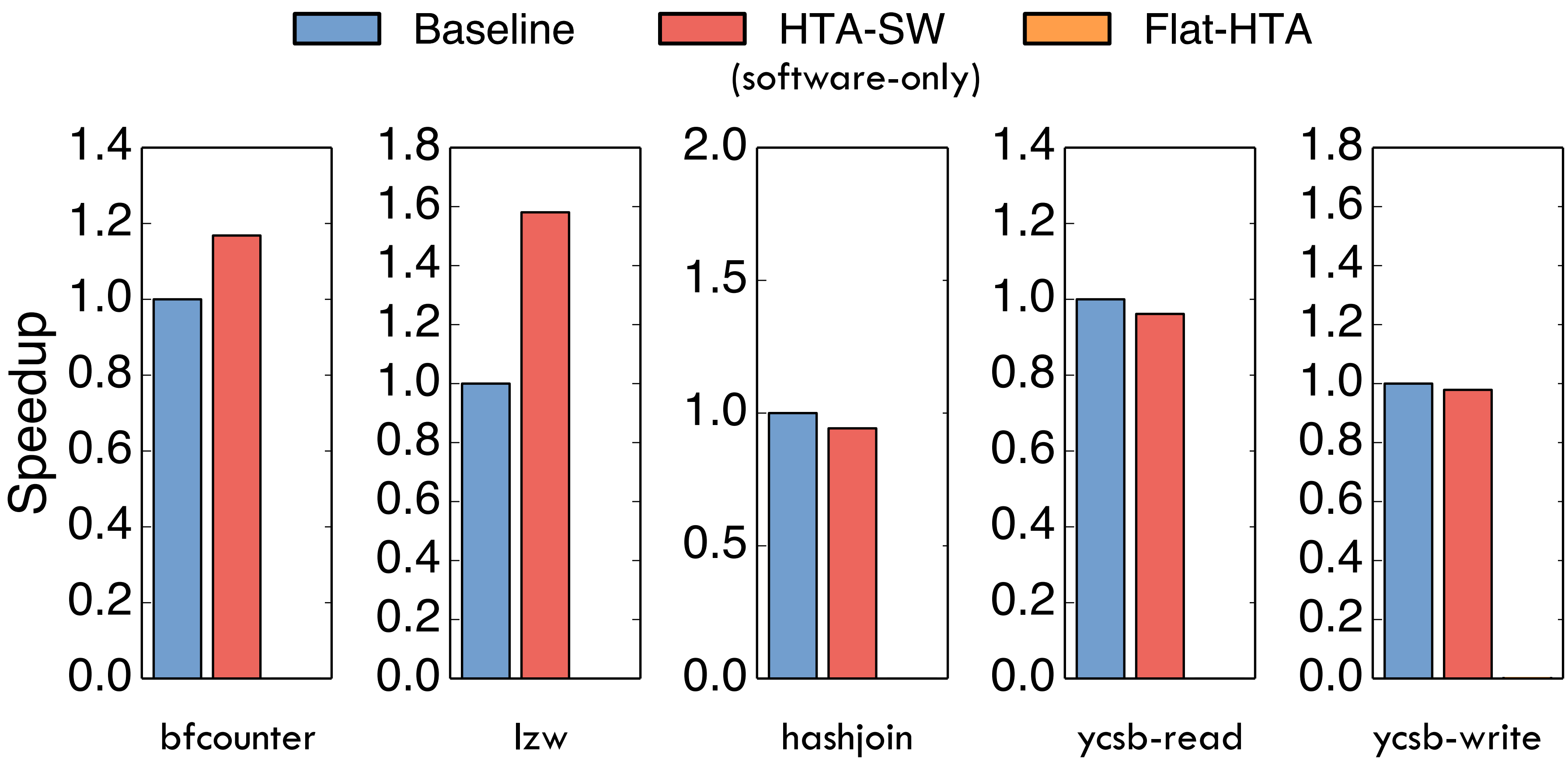


- Schemes
  - Baseline: best of
    - Google dense_hash_map
    - C++11 unordered_map
  - HTA-SW
    - w/ HTA table format
    - w/o HTA function unit
  - Flat-HTA
  - Hierarchical-HTA
- Applications
  - bfcounter (bioinformatics)
  - lzw (data compression)
  - Hashjoin (database)
  - ycsb-read (key-value store)
  - ycsb-write (key-value store)

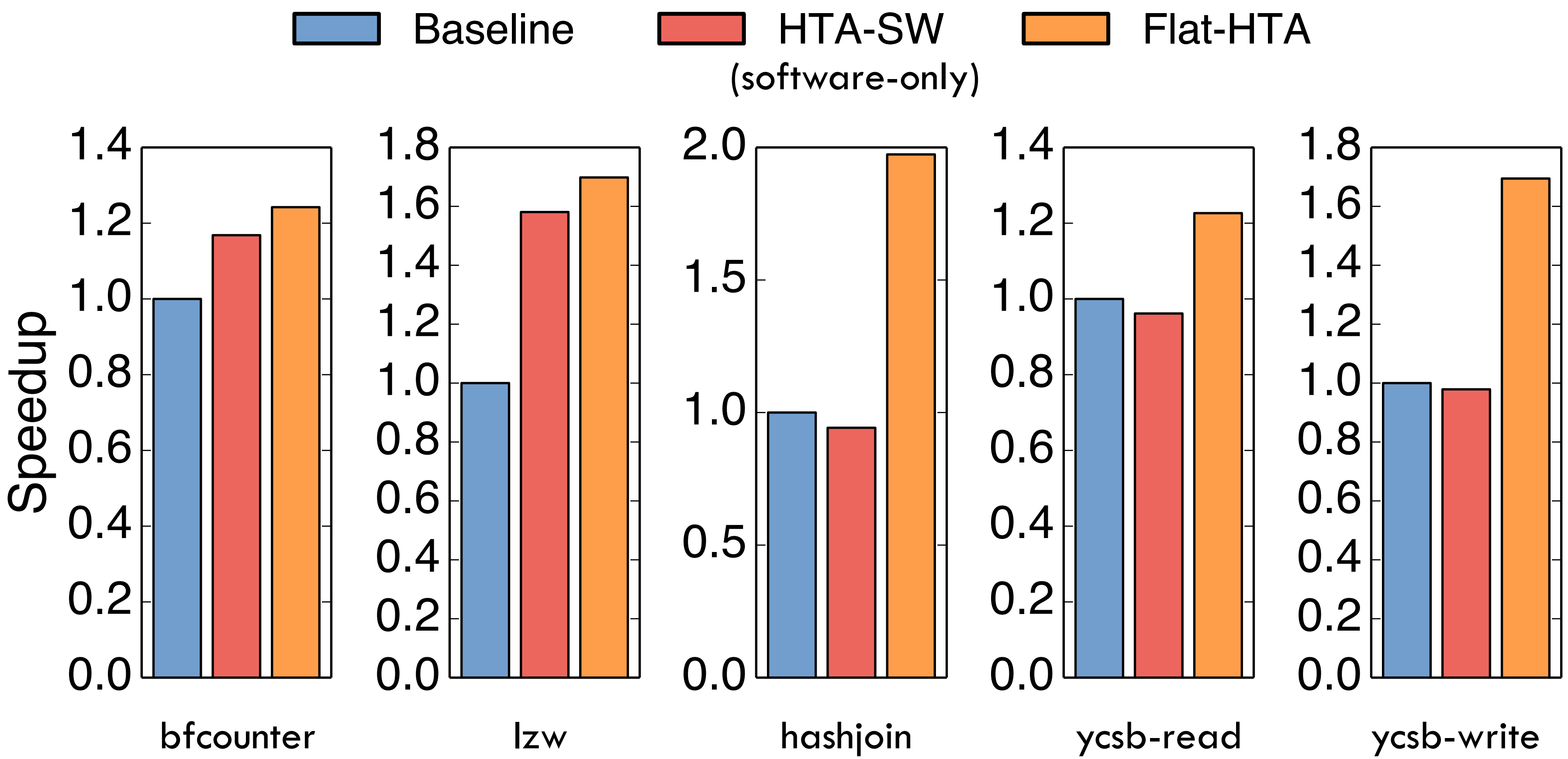
Flat-HTA speedups



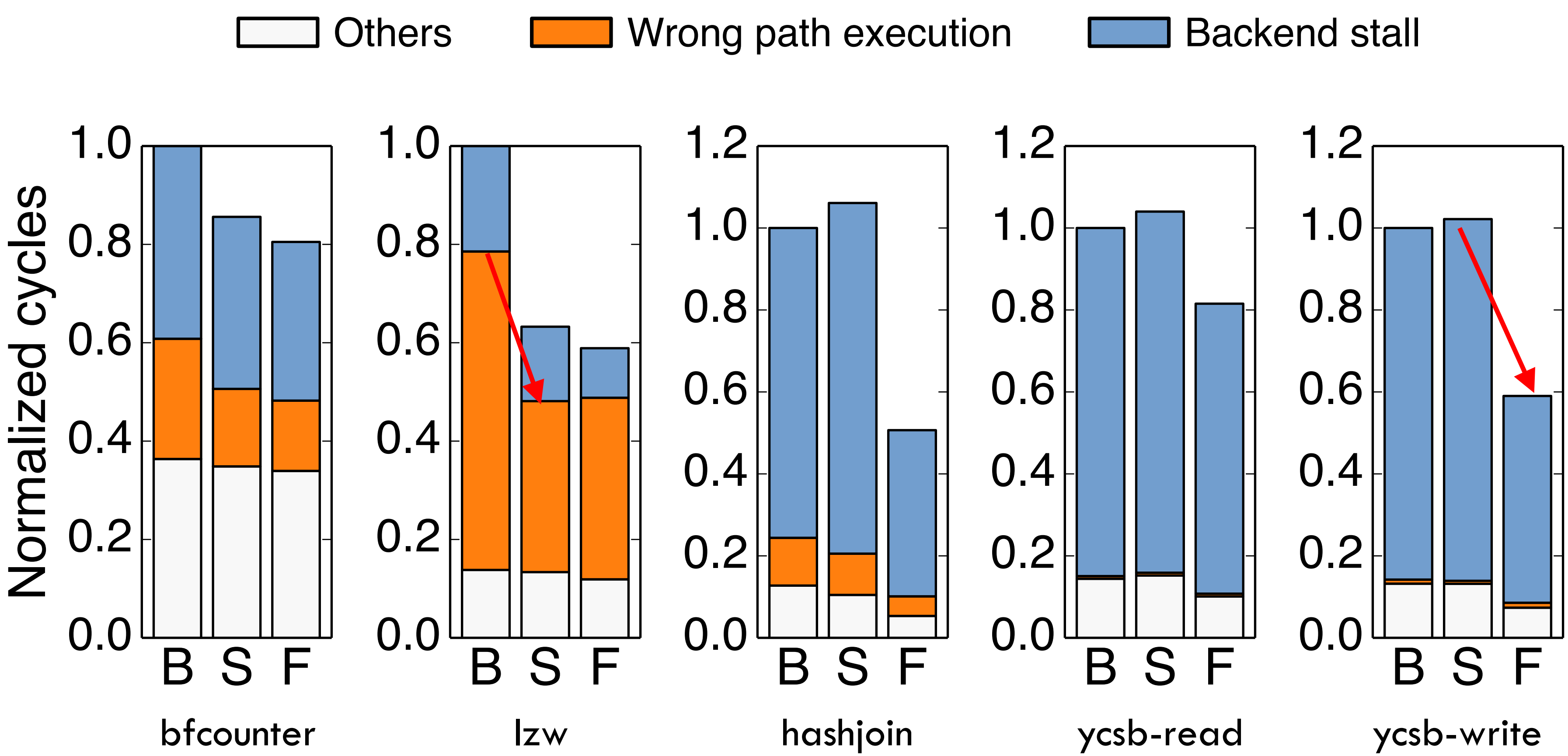
Flat-HTA speedups



Flat-HTA speedups

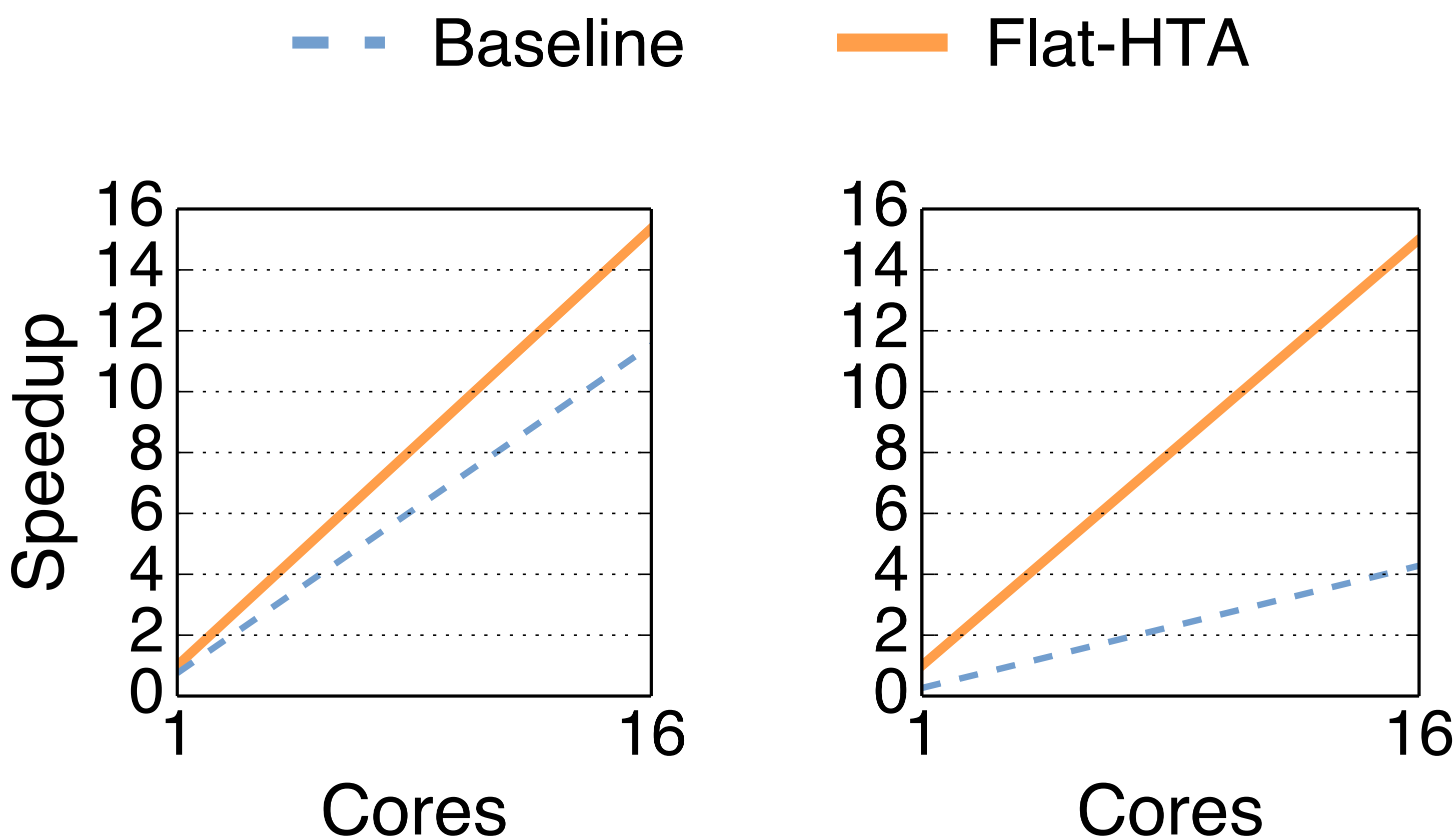


Flat-HTA cycles breakdown



B: <u>B</u>aseline S: HTA-<u>S</u>W (software-only) F: <u>F</u>lat-HTA

Flat-HTA on multithreaded applications



ycsb-read ycsb-write

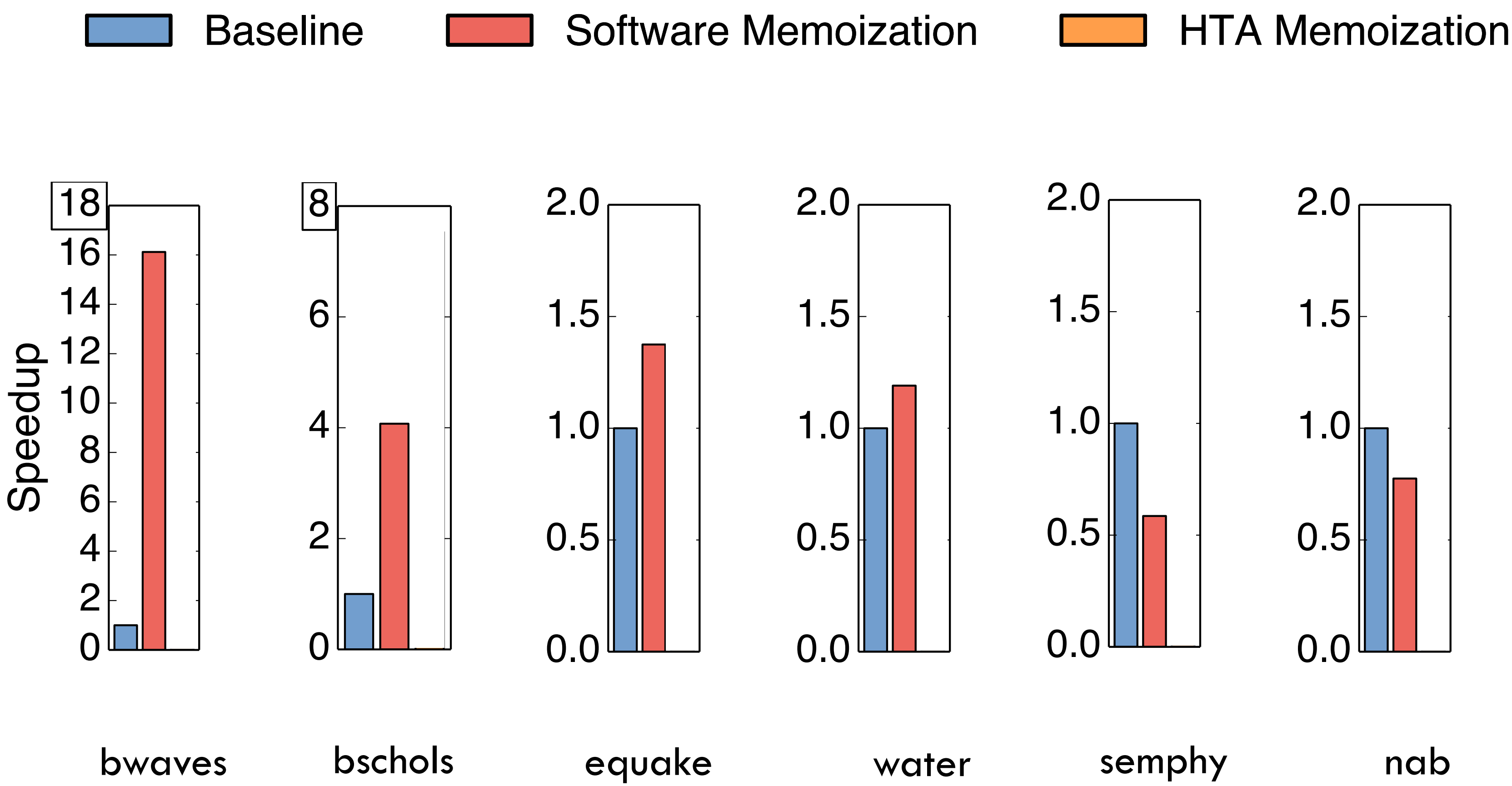
HTA on memoization

- Example

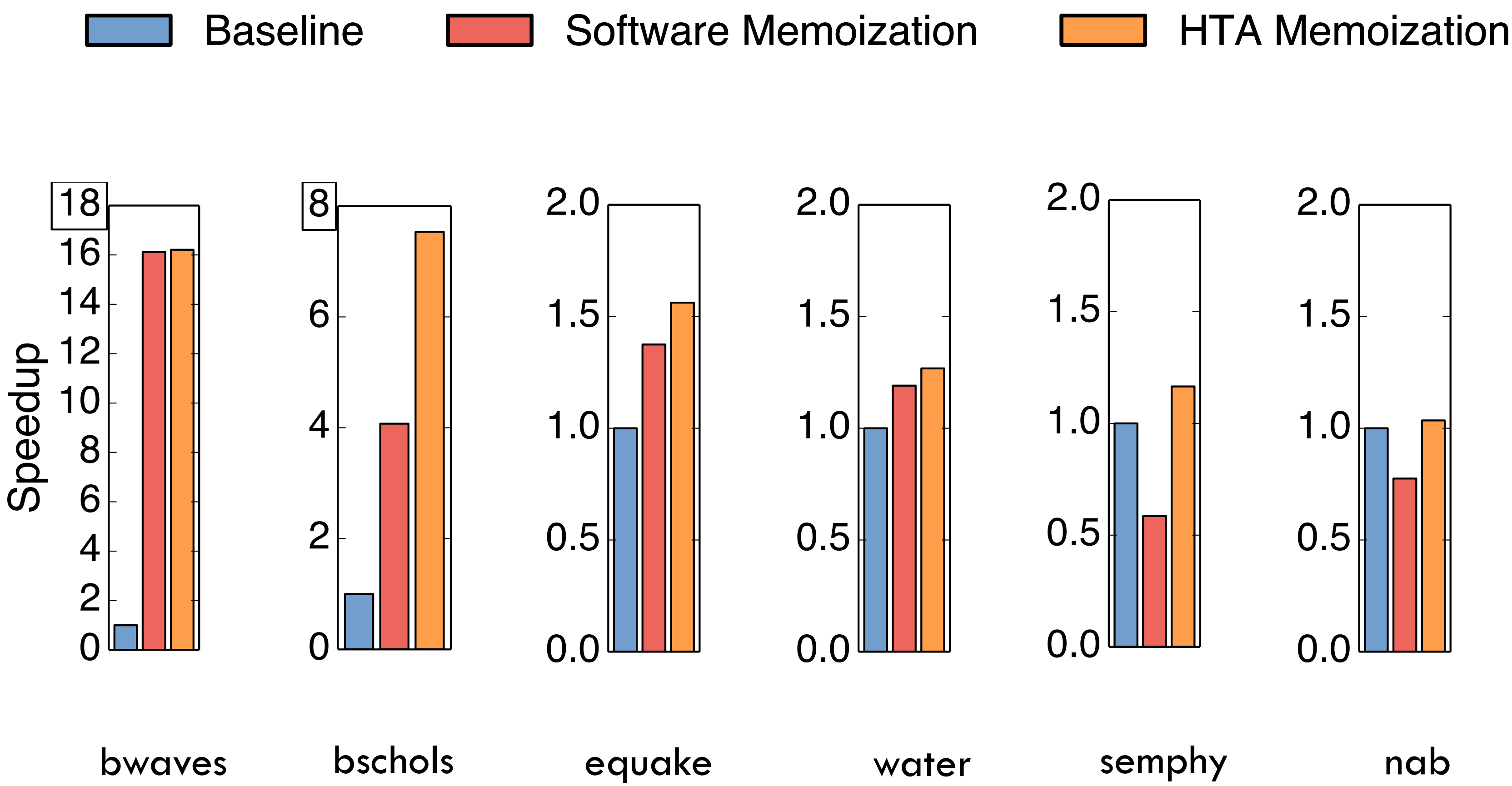
```
memo_exp: hta_lookup  <table id>, <key reg>, <value reg>, done
     call exp
     hta_swap      <table id>, <key reg>, <value reg>, done
done:     …
```

- Schemes
  - Baseline (no memoization)
  - Software memoization
  - HTA memoization
- Applications selected from
  - SPECCPU2006
  - SPECOMP2001
  - SPECOMP2012
  - PARSEC
  - SPLASH2
  - BioParallel

Flat-HTA speedups on memoization



Flat-HTA speedups on memoization



Conclusion

- **HTA** accelerates hash tables and memoization
  - Adopts a new hash table format
  - Accelerates common cases in HW; leaves rare cases to SW
- **Flat-HTA** reduces runtime overheads significantly
  - Requires minor (0.055% area) changes to cores
- **Hierarchical-HTA** improves spatial locality
  - Needs changes to cores and cache controllers
- HTA improves hash-table-intensive applications by up to 2x
- HTA enables **memoization** of small code regions

THANKS FOR YOUR ATTENTION!

QUESTIONS ARE WELCOME!

Massachusetts Institute of Technology

CSAIL