UCONN COMPUTER ARCHITECTURE GROUP

# Exploring the Performance Implications of Memory Safety Primitives in Many-core Processors Executing Multi-threaded Workloads

Masab Ahmad, Syed Kamran Haider, Farrukh Hijaz,
Marten van Dijk, Omer Khan

University of Connecticut

# Agenda

Motivation

Characterization Methodology

Characterization Results

Insights and Possible Improvements

# Agenda

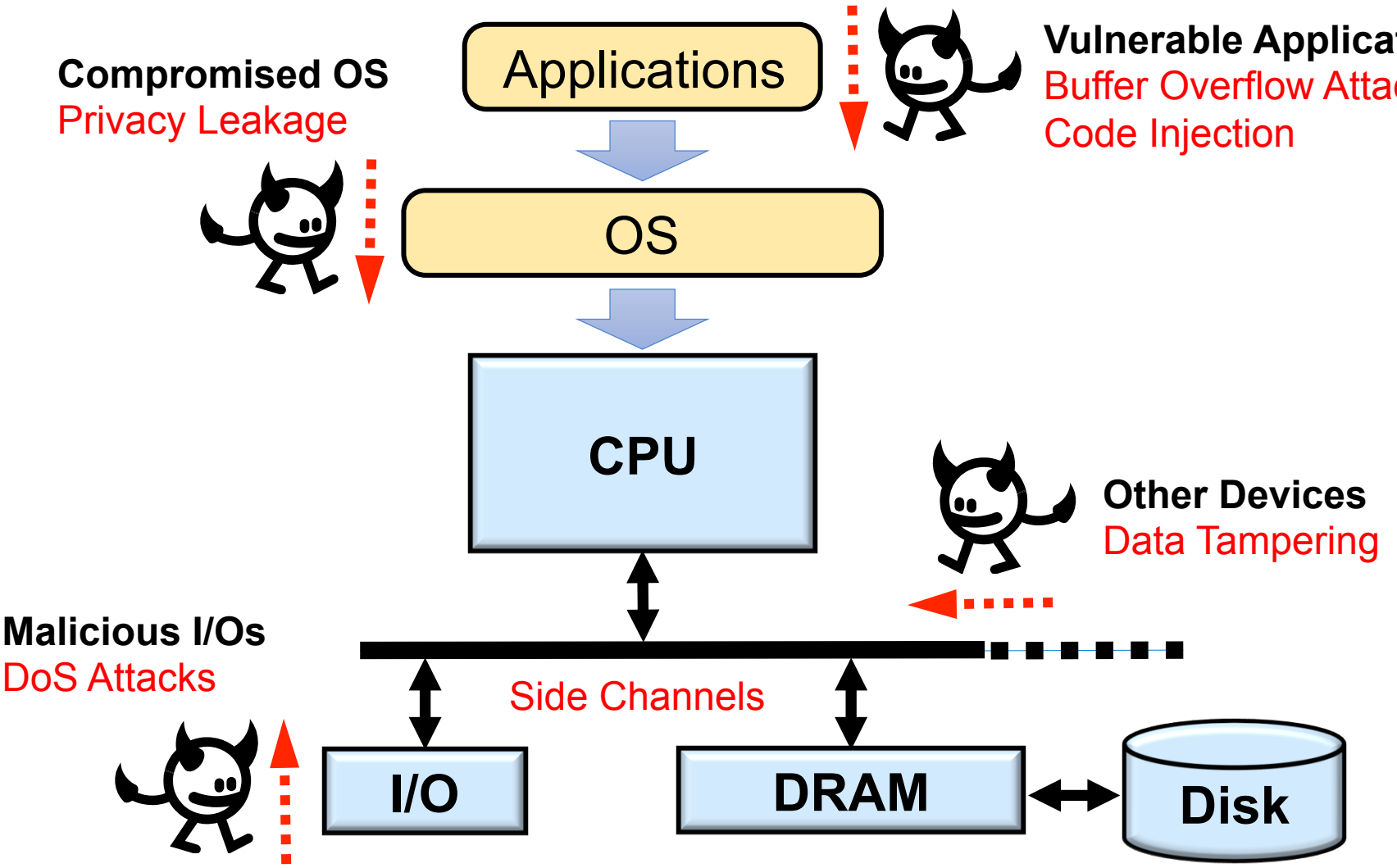
**Motivation**

Characterization Methodology

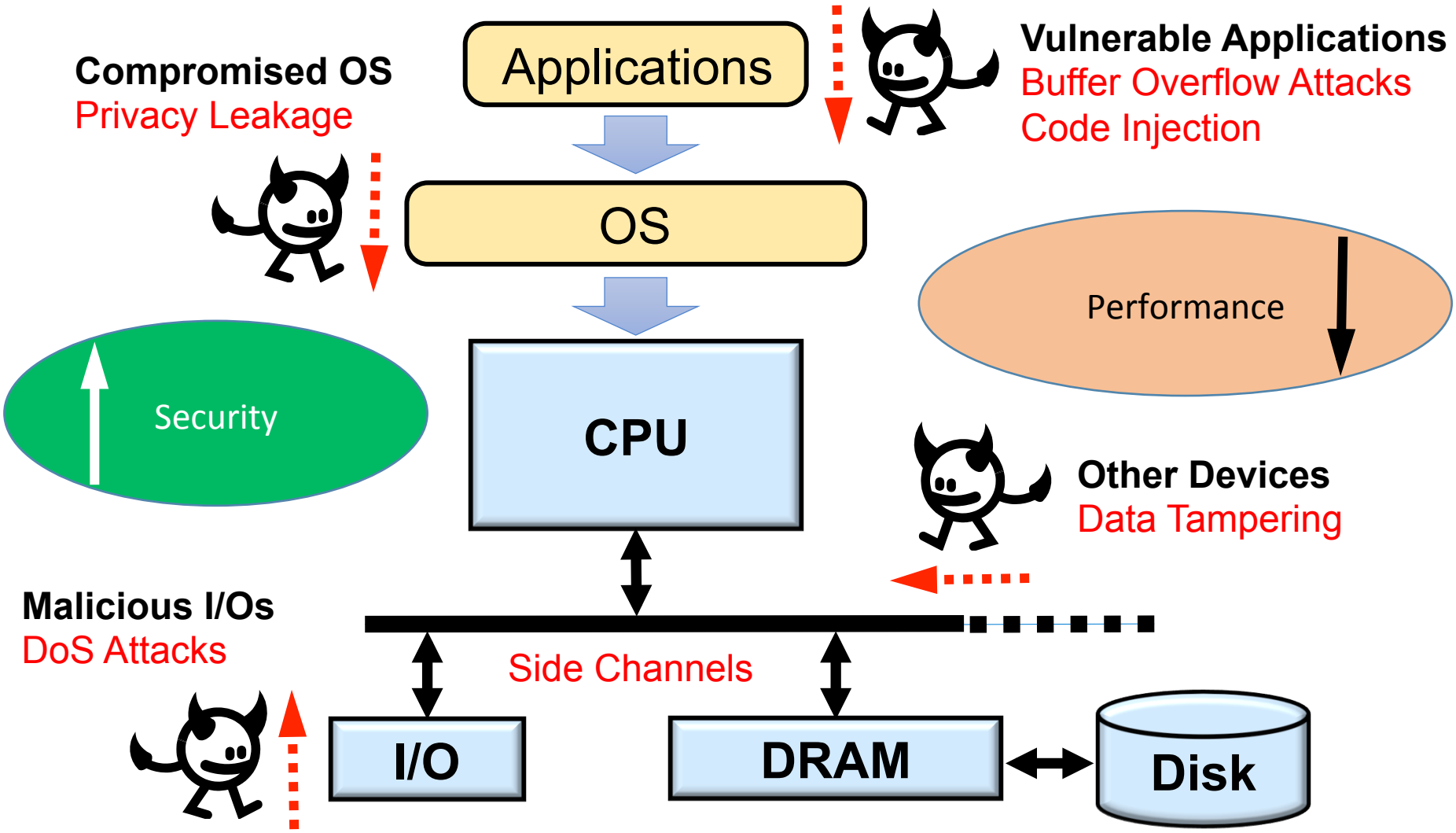
Characterization Results

Insights and Possible Improvements

# Security Vulnerabilities in Processors



# Security Vulnerabilities in Processors

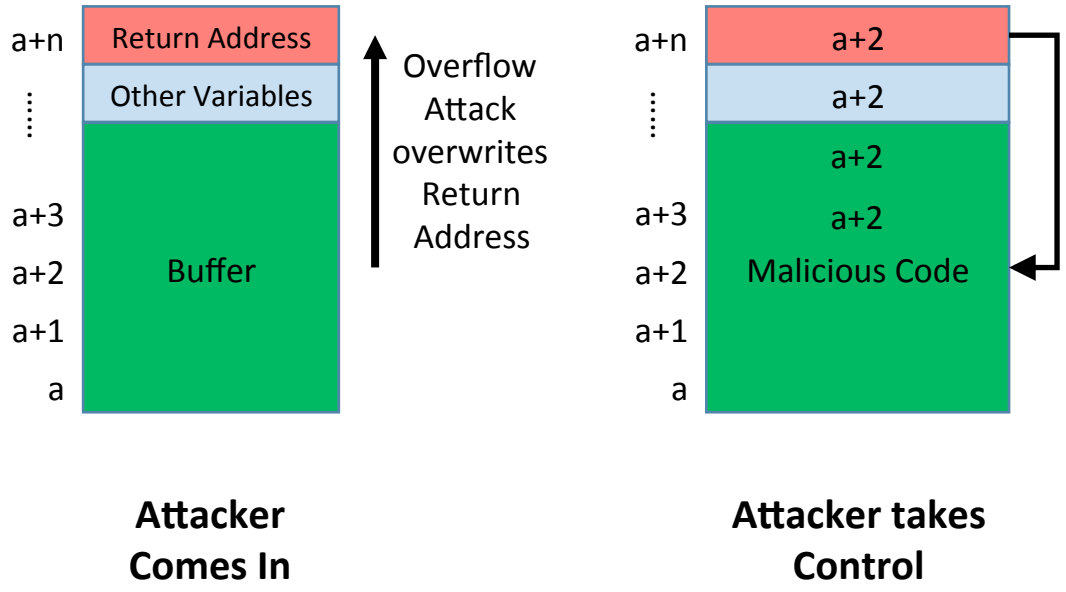


# Key Questions

- What are the performance implications of these security schemes?
- How do they affect performance in the context of multicores?
  - IEEE Computer Vision 2022 predicts that Multicores as a key enabling technology by 2022
- To get started, we look at *Buffer Overflow Protection Schemes*
  - How do they affect tradeoffs in Multicores?

# A Primer on Buffer Overflow Attacks

- The stack return address is located after the program contents
- An attacker enters a modified string, overwrites the return address
- The return address now points to the malicious code



# Security Solutions and Prior Works

- Dynamic Information Flow Tracking (DIFT) :
  - Flags and Tags all data from I/O
  - Raises exceptions of tagged data propagates into program control flow
  - High False positive rates, Zero false negatives (Hence not used in applications)
- Context based Checking :
  - Creates a "context", an identifier for each variable/data structure in the program
  - Checks on each variable read/write
  - Large data structure required, 1 element for each array element. Pointer aliasing problems (Hence also not used in applications)
- Bounds Checking :
  - Creates a base and bounds Metadata data structure for a program
  - Reads and writes are checked and updated as meta data
  - Close to Zero False positives and negatives (Used a lot in Safe languages etc)

# Security Solutions and Prior Works

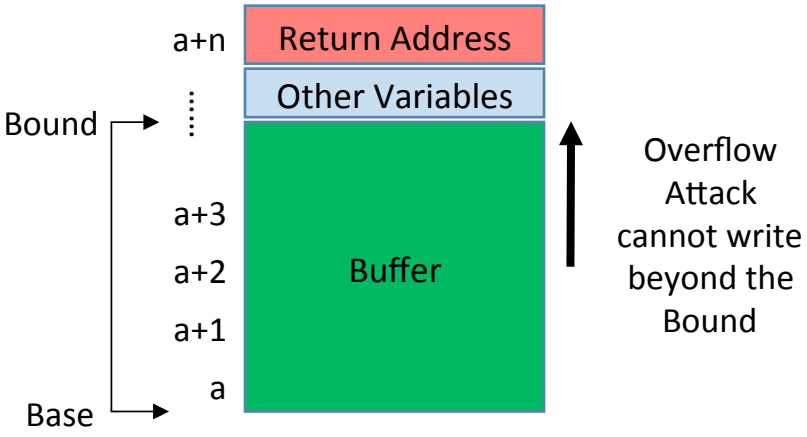
- Dynamic Information Flow Tracking (DIFT) :
  - Flags and Tags all data from I/O
  - Raises exceptions of tagged data propagates into program control flow
  - High False positive rates, Zero false negatives (Hence not used in applications)

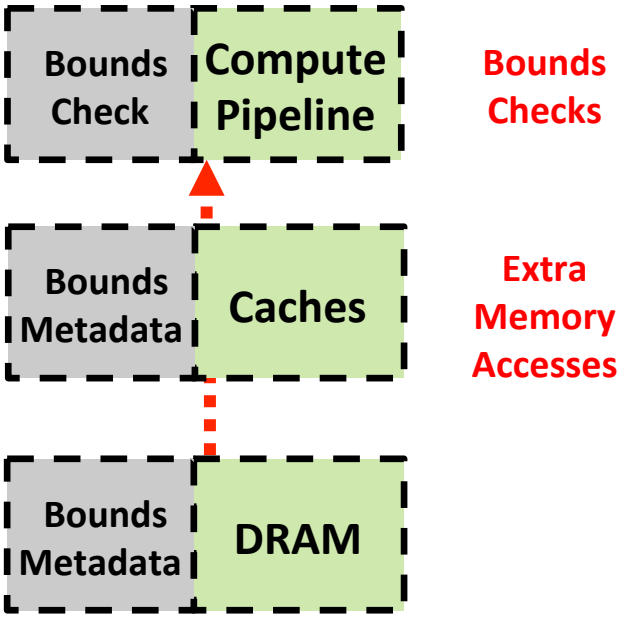
- Context based Checking :
  - Creates a "context", an identifier for each variable/data structure in the program
  - Checks on each variable read/write
  - Large data structure required, 1 element for each array element. Pointer aliasing problems (Hence also not used in applications)

- Bounds Checking :
  - Creates a base and bounds Metadata data structure for a program
  - Reads and writes are checked and updated as meta data
  - Close to Zero False positives and negatives (Used a lot in Safe languages etc)

# Bound Checking Schemes

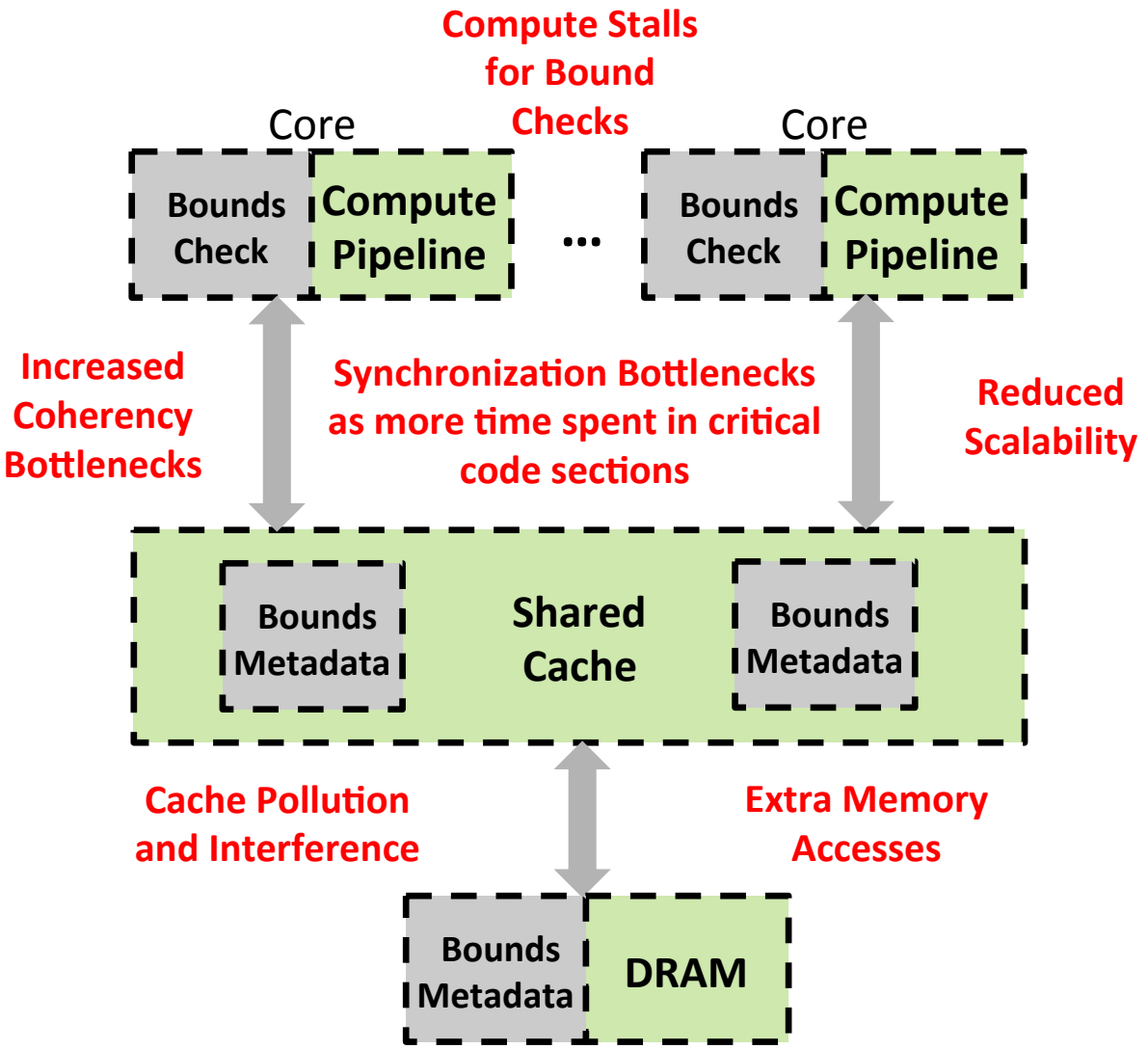
- Hardware based Schemes:
  - CHERI
  - HardBound
  - WatchdogLite
- Software based Schemes:
  - **SoftBound**
  - Cyclone
  - SafeCode
  - Ccured
  - And many others
  - Safe Languages such as Python and Java



# Performance Implications in Multicores

- Previous works have not analyzed memory safety schemes in the context of multicores
  - All prior works use sequential benchmarks such as SPEC



# Performance Implications in Multicores : **Critical Code**

- Critical code sections such as atomically locked program functions are most affected.

- Due to bounds checking stalls, a thread keeps a locked section for a longer time, depriving other threads from exploiting scalability.

**Code Without SoftBound**

```
1. int a = 1;
2. pthread_mutex_lock(&lock);
3. do_parallel_work();
4. pthread_mutex_unlock(&lock);
5. return 0;
```

**Code With SoftBound**

```
1. int a = 1;
2. pthread_mutex_lock(&lock);
3. do_parallel_work();
4. Get Security Metadata();
5. SoftBound Checks ();
6. pthread_mutex_unlock(&lock);
7. return 0;
```

# Agenda

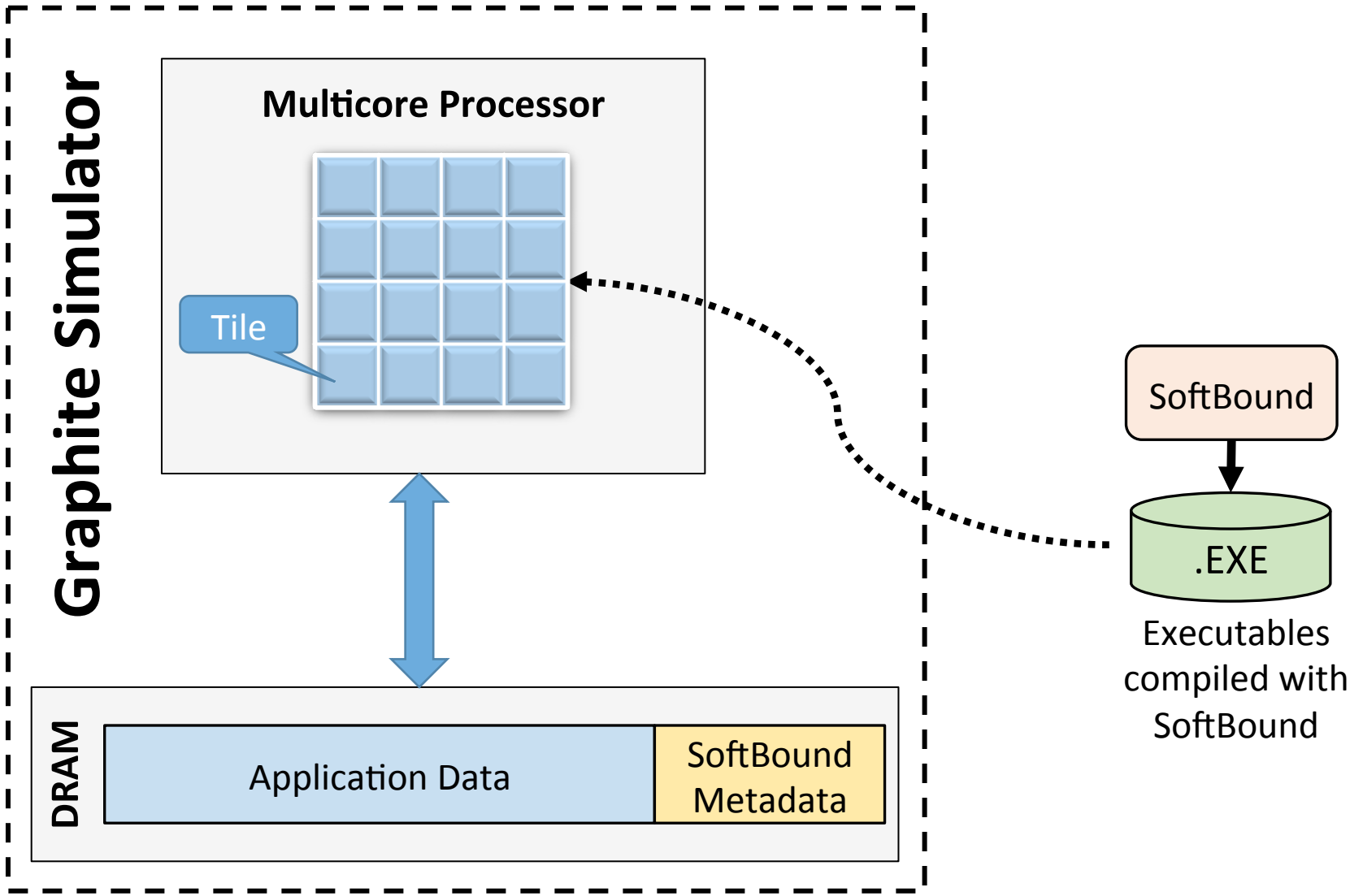
Motivation

**Characterization Methodology**

Characterization Results

Insights and Possible Improvements

# System Model



# System Parameters

- **The Graphite Simulator is used to analyze Parallel Benchmarks**
  - 256 Cores @ 1 GHz
  - Results for both in-order and Out-of-Order cores
  - L1-I Cache : 32 KB per core
  - L1-D Cache : 32 KB per core
  - L2 Cache : 256 KB per core
  - OOO ROB Buffer Size: 168
- **An 8 Core Intel machine used as well (Trends similar as Graphite results) (Results in Paper)**
- **POSIX Threading Model is used**
  - **Evaluation Includes results for 1-256 Threads**

| Architectural Parameter | Value |
|---|---|
| Number of Cores | 256 @ 1 GHz |
| In-Order Core Setup | |
| Compute Pipeline per Core | Single-Issue Core |
| Out-of-Order Core Setup | |
| Compute Pipeline per Core | Single-Issue Core<br>Out-of-Order Memory |
| Reorder Buffer Size | 168 |
| Load Queue Size | 64 |
| Store Queue Size | 48 |
| Memory Subsystem | |
| L1-I Cache per core | 32 KB, 4-way Assoc., 1 cycle |
| L1-D Cache per core | 32 KB, 4-way Assoc., 1 cycle |
| L2 Cache per core | 256 KB, 8-way Assoc., 8 cycle<br>Inclusive, NUCA |
| Cache Line Size | 64 bytes |
| Directory Protocol | Invalidation-based MESI<br>$ACKWise_4$ [11] limited directory |
| Num. of Memory Controllers | 8 |
| DRAM Bandwidth | 5 GBps per Controller |
| DRAM Latency | 100 ns |
| Electrical 2-D Mesh with XY Routing | |
| Hop Latency | 2 cycles (1-router, 1-link) |
| Contention Model | Only link contention<br>(Infinite input buffers) |
| Flit Width | 64 bits |

# Evaluated Benchmarks

- **Prefix Scan – 16M elements**
  - Each Thread gets a chunk to scan, then the Master Thread reduces the scans of other threads to get the final solution
  - Barriers used to do explicit Synchronization
- **Matrix Multiply – 1K x 1K**
  - Each Thread Tiles a chunk of the matrix, and multiplies to get the final matrix
  - Barriers used to do explicit Synchronization
- **Breadth First Search (BFS) – 1M vertices, 16M edges**
  - Each Thread gets a chunk of the graph to search on
  - Atomic locks used on all graph vertices to ensure that no race conditions occur in shared vertices
- **Dijkstra – 16K vertices, 134M edges**
  - Checking neighboring nodes for shortest path distances parallelized among threads
  - Explicit Barriers to progress each node check

# Agenda

Motivation

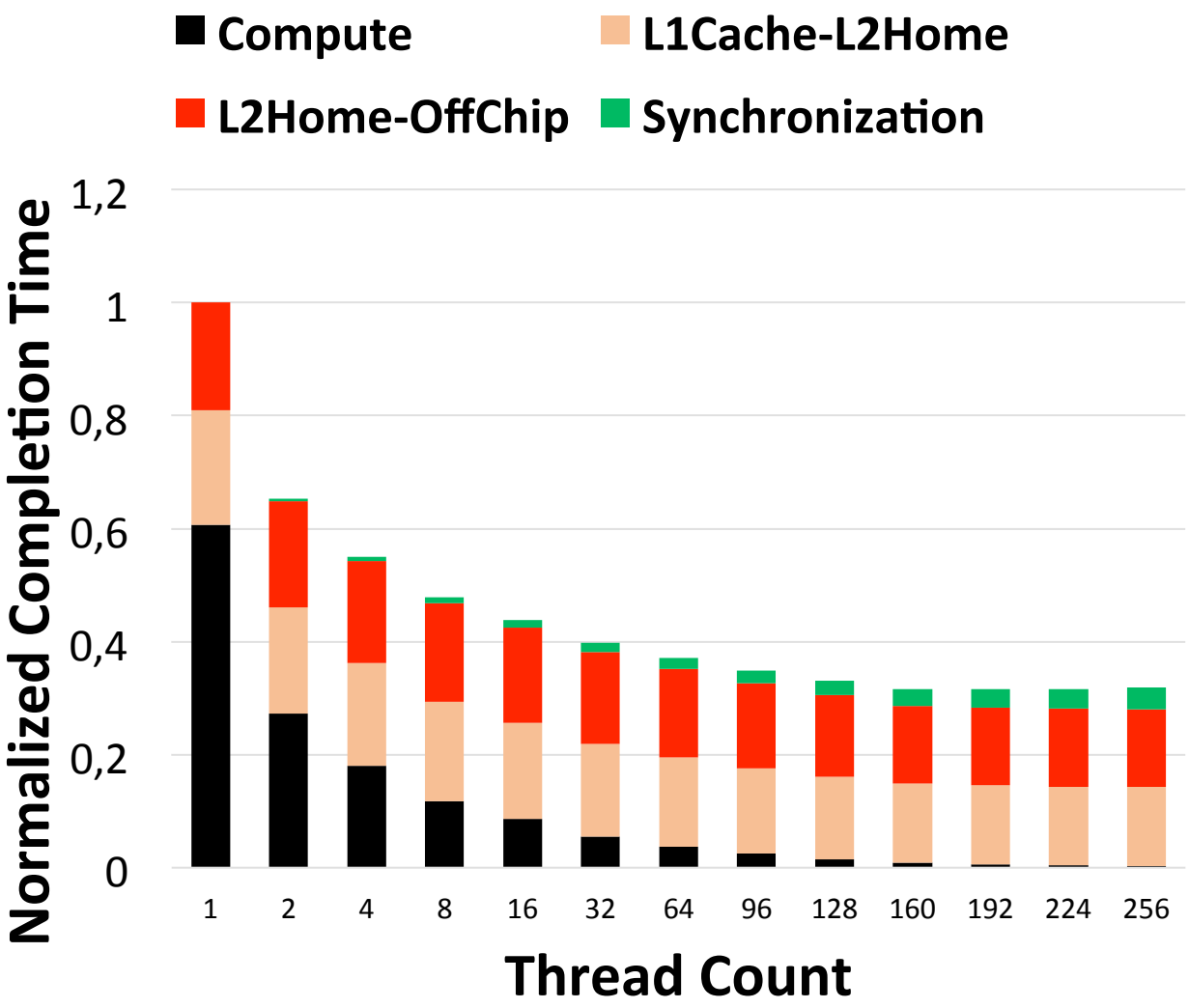
Characterization Methodology

**Characterization Results**

Insights and Possible Improvements

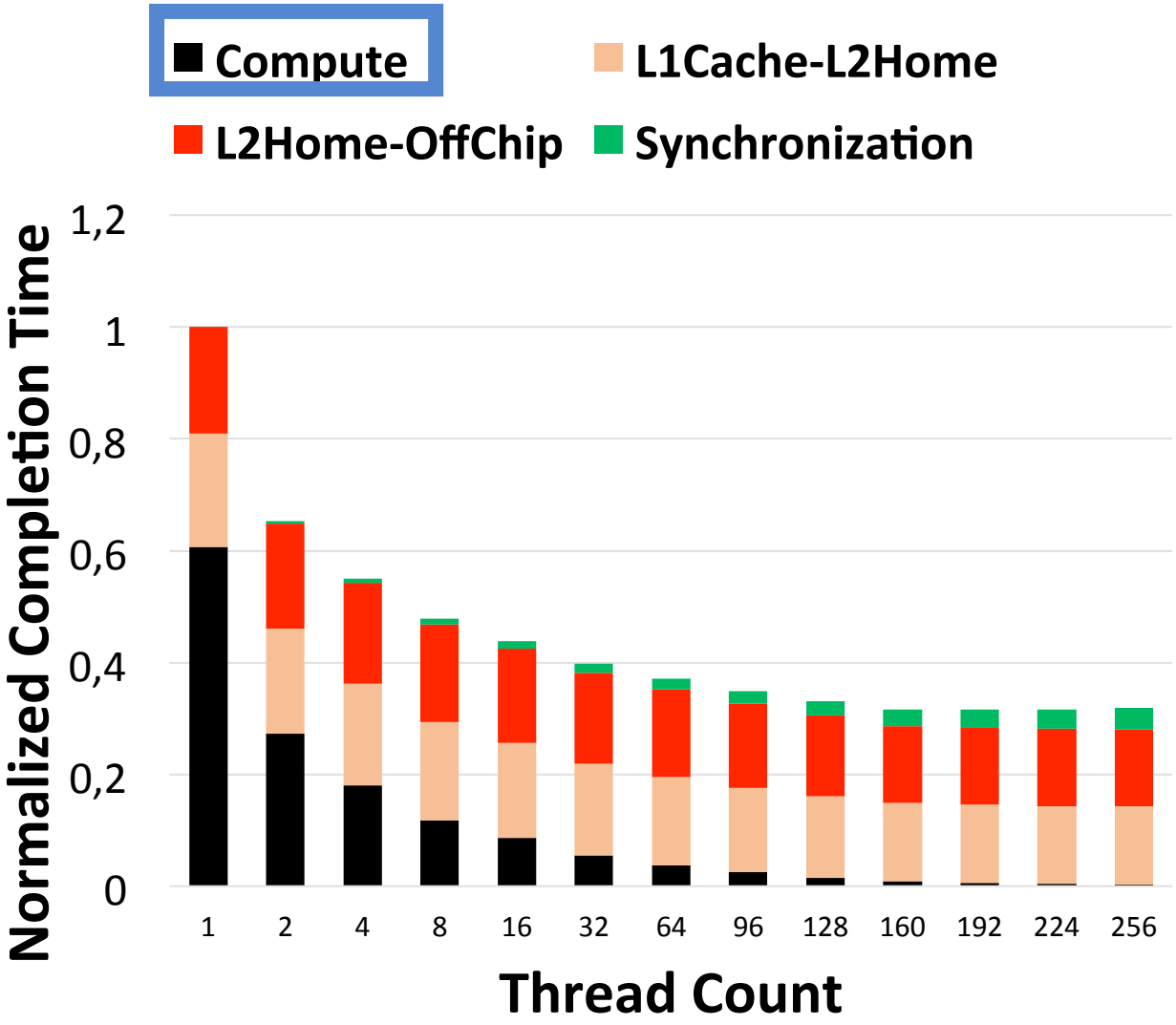
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



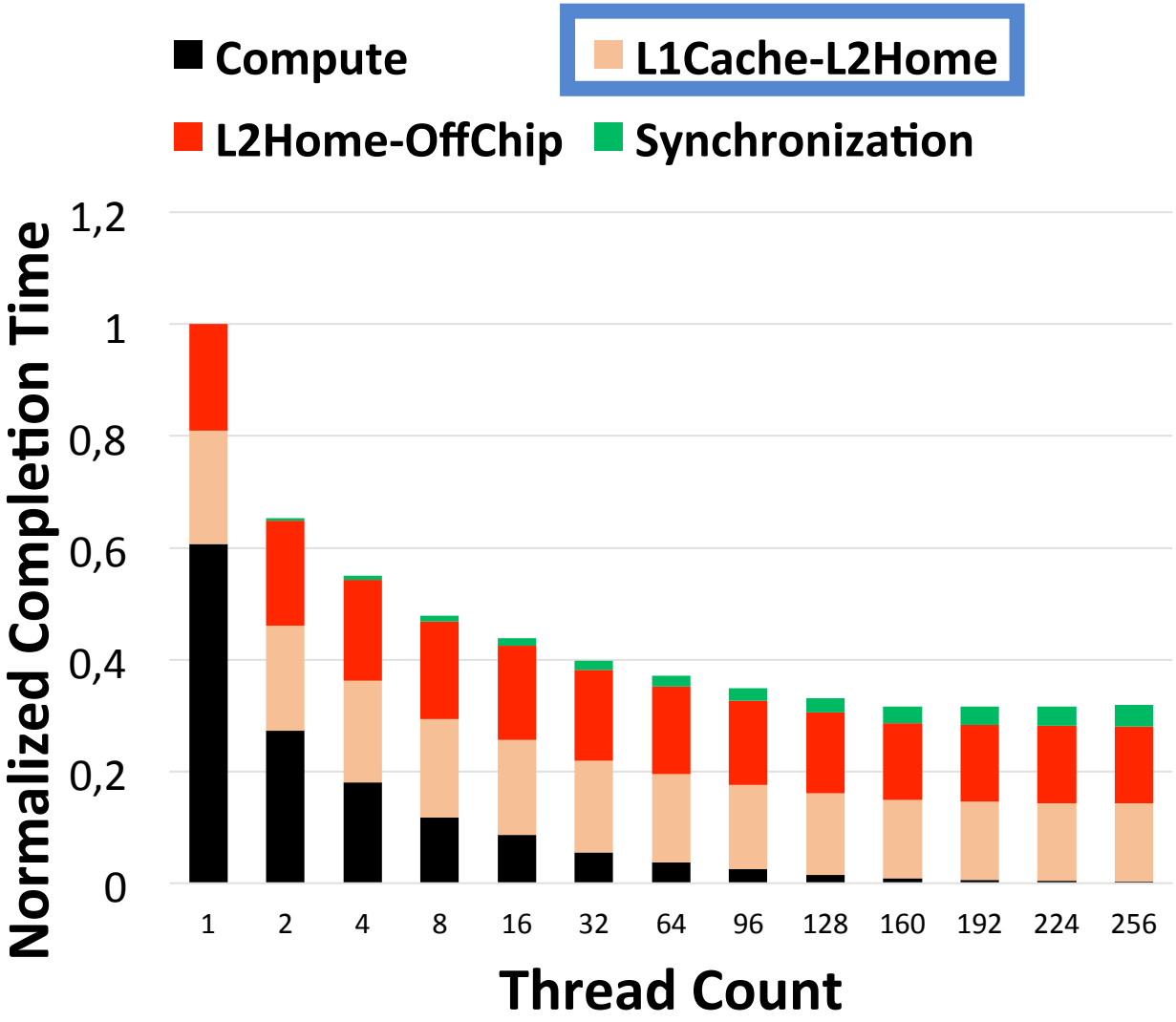
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



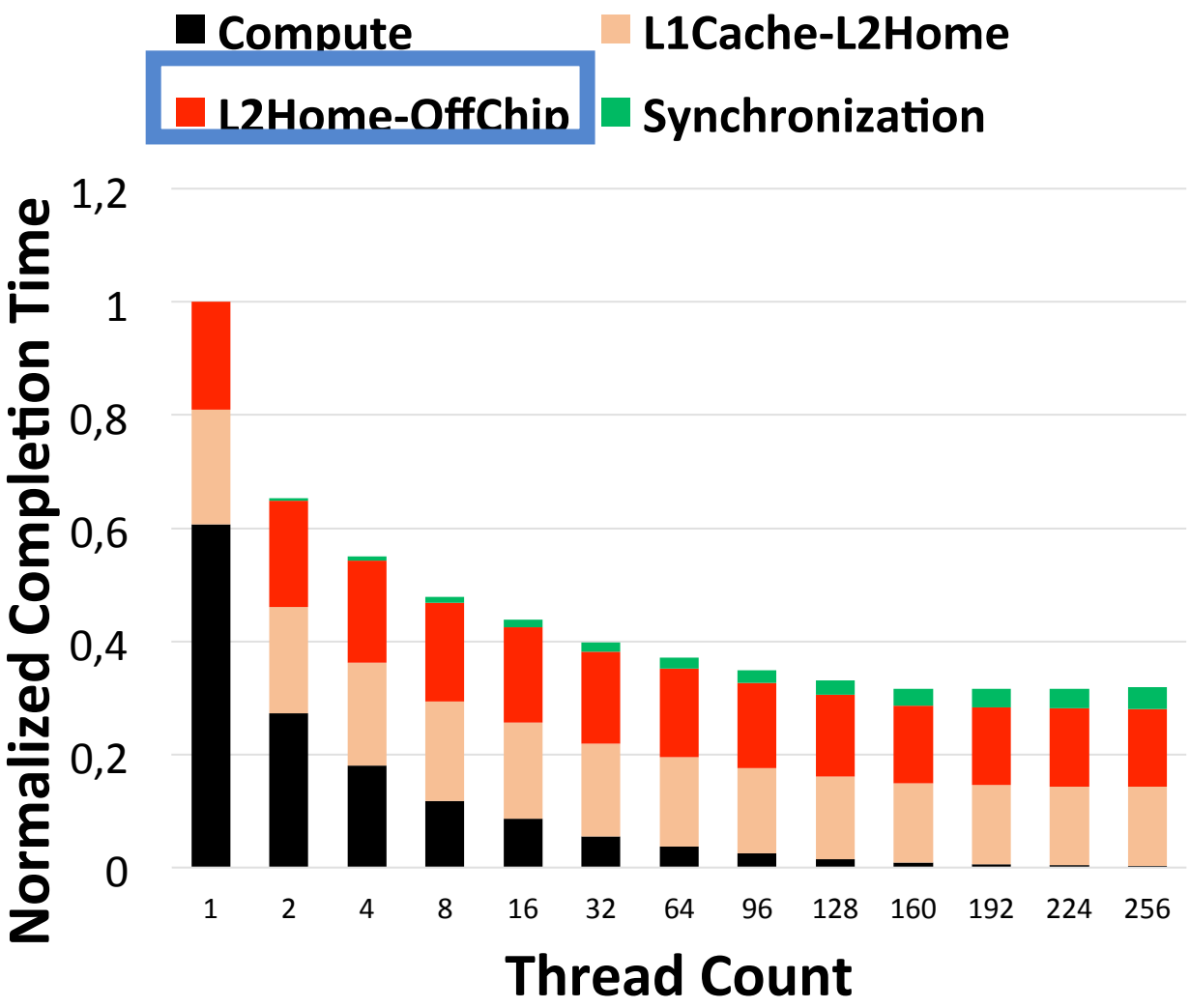
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



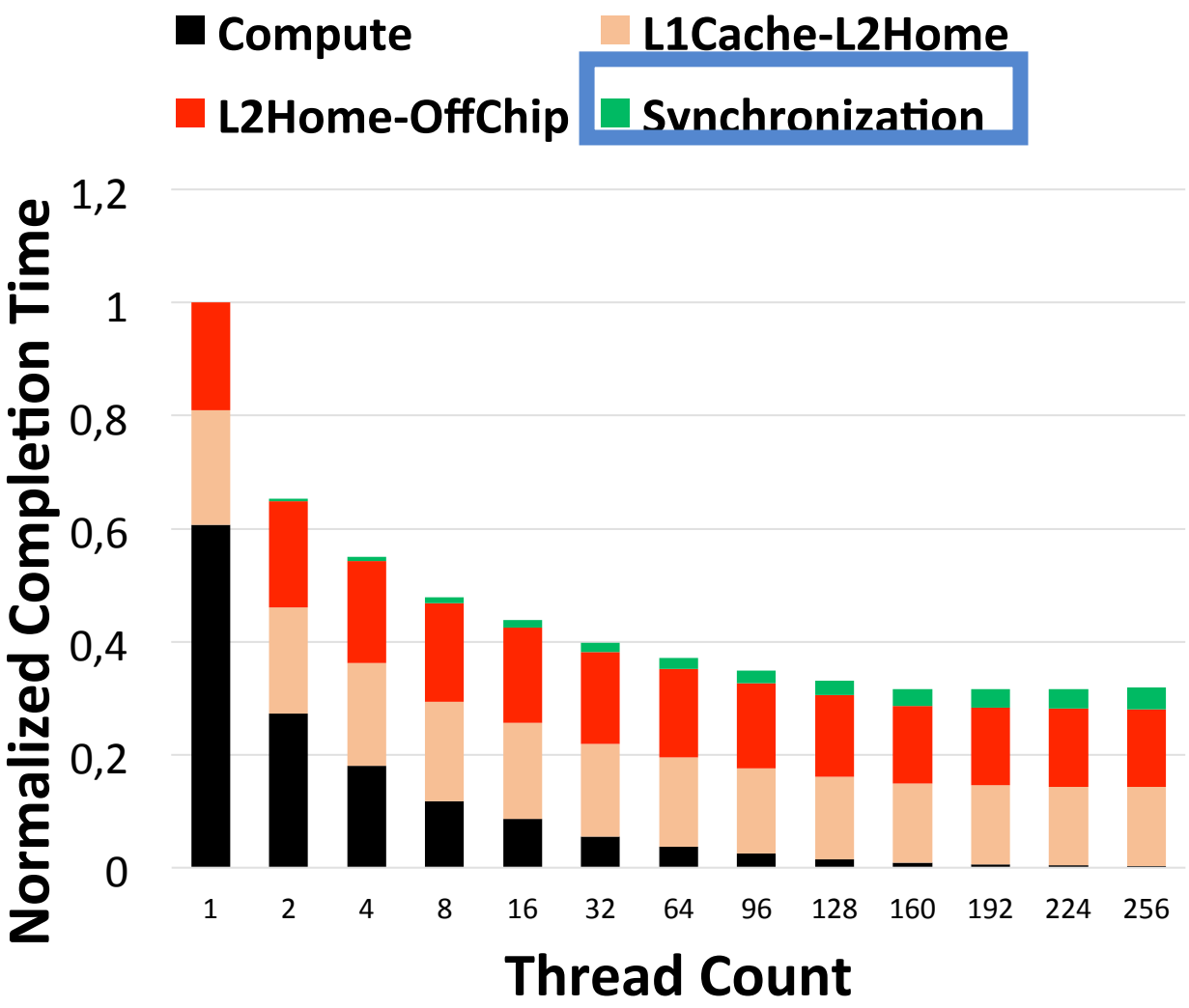
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



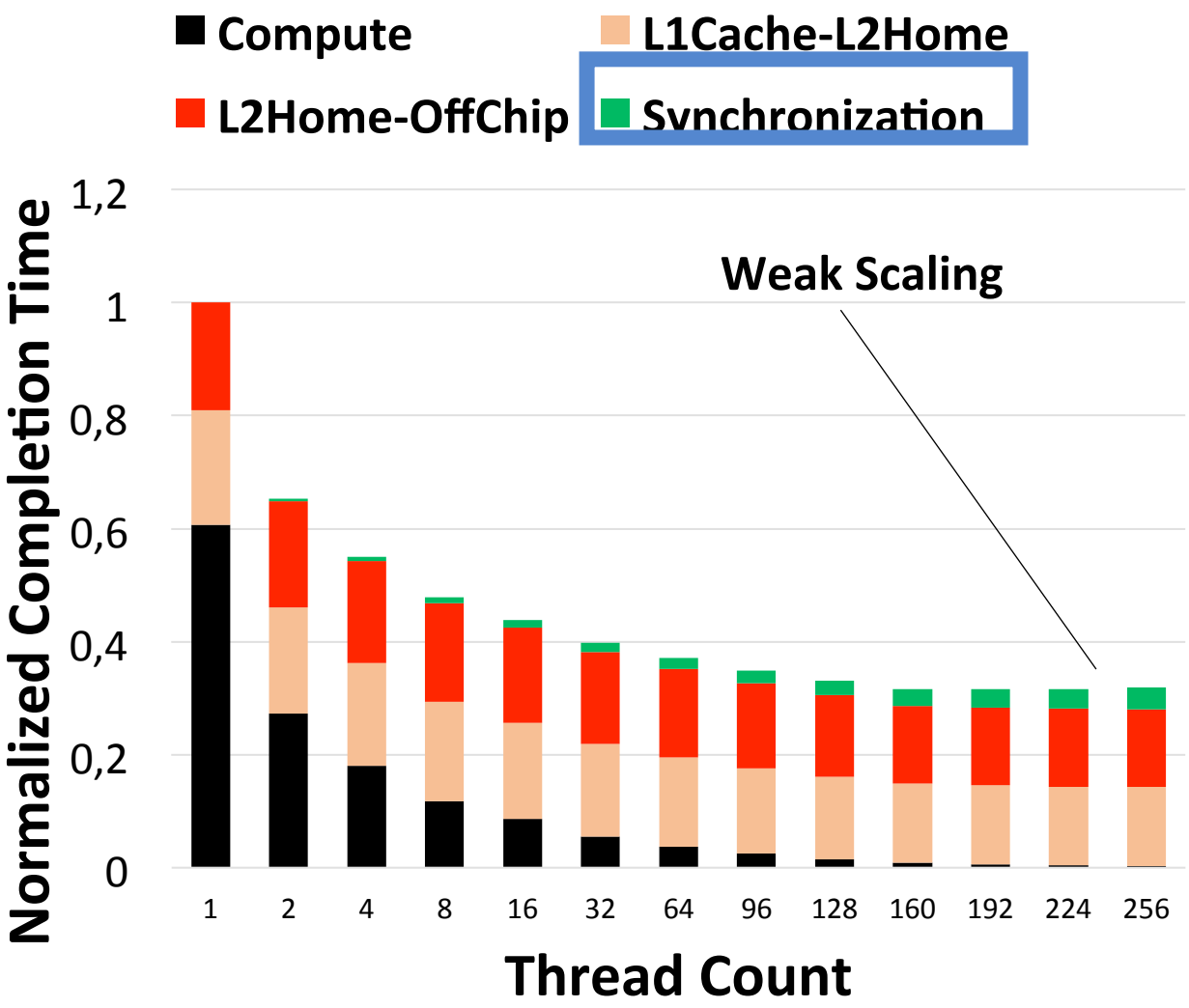
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



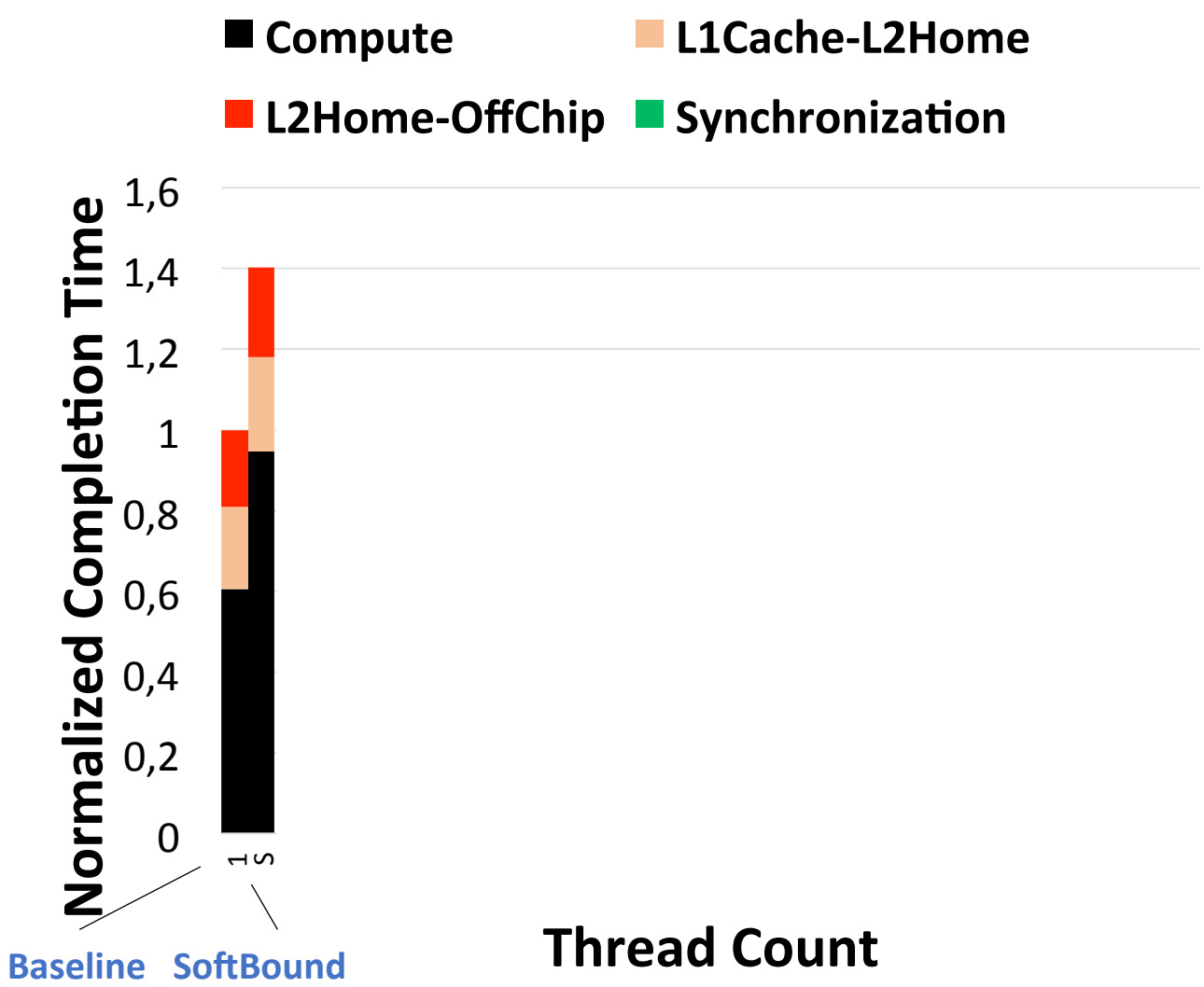
# Benchmark Characterization : Prefix Scan

- Shows Weak Scaling



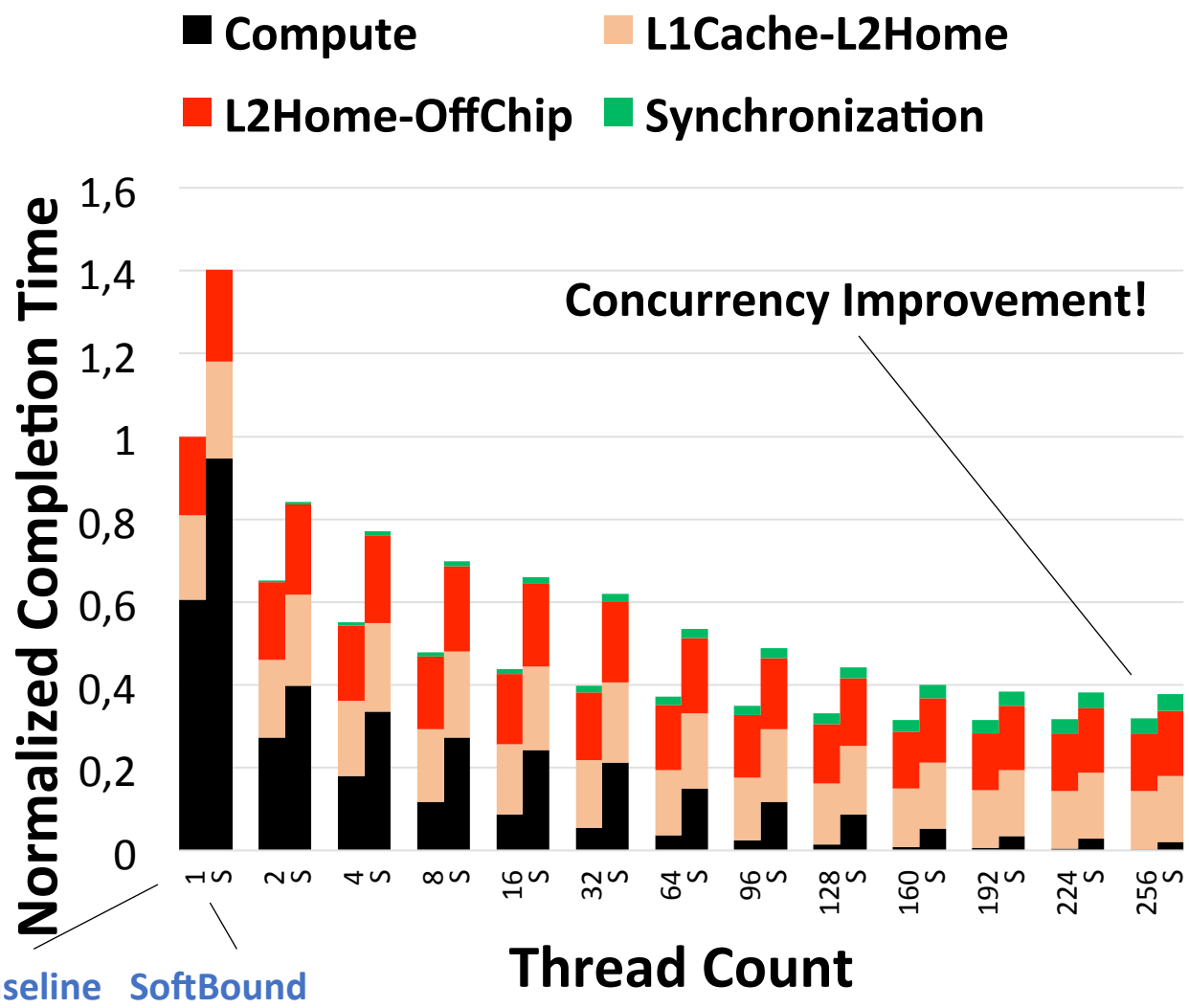
# Benchmark Characterization : Prefix Scan

- SoftBound has higher overheads due to extra compute and memory accesses
- ‘S’ shows results with SoftBound



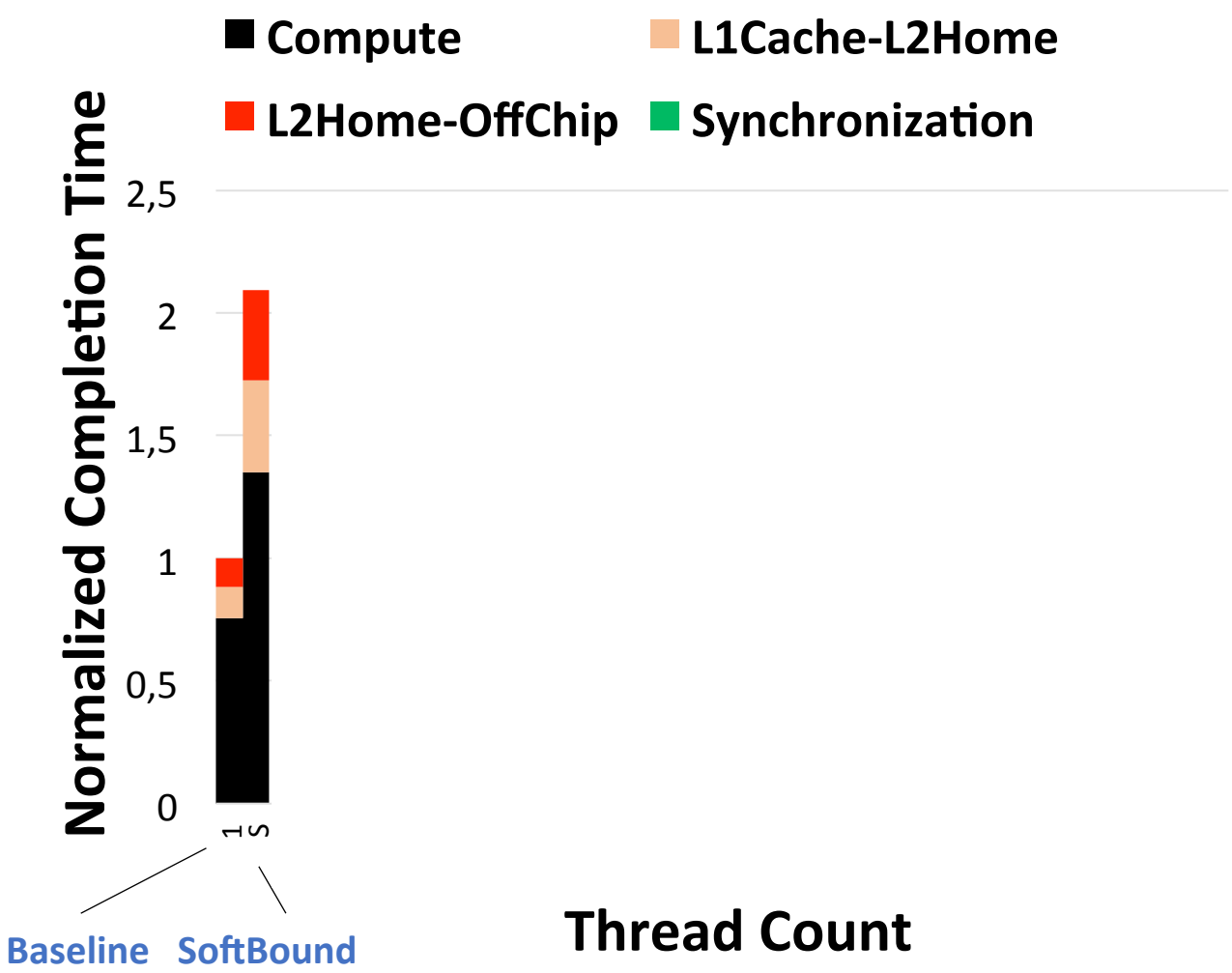
# Benchmark Characterization : Prefix Scan

- Concurrency hides SoftBound's overhead at high thread counts
  - More Compute than Communication in this Workload
- 'S' shows results with SoftBound



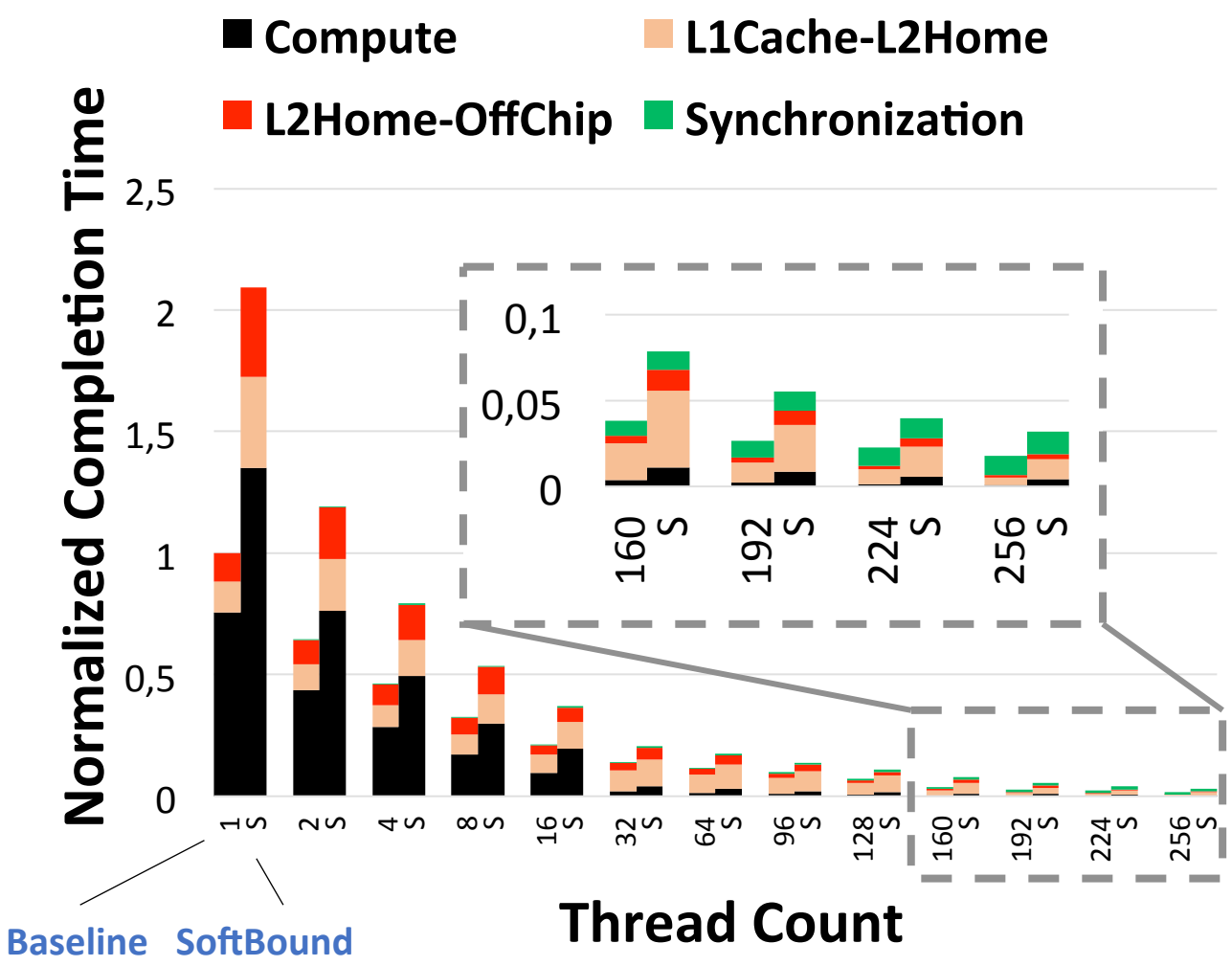
# Benchmark Characterization : Matrix Multiply

- Significant Overhead with SoftBound
- Memory Bounds Applications lead to more Metadata and security checks
- 'S' shows results with SoftBound



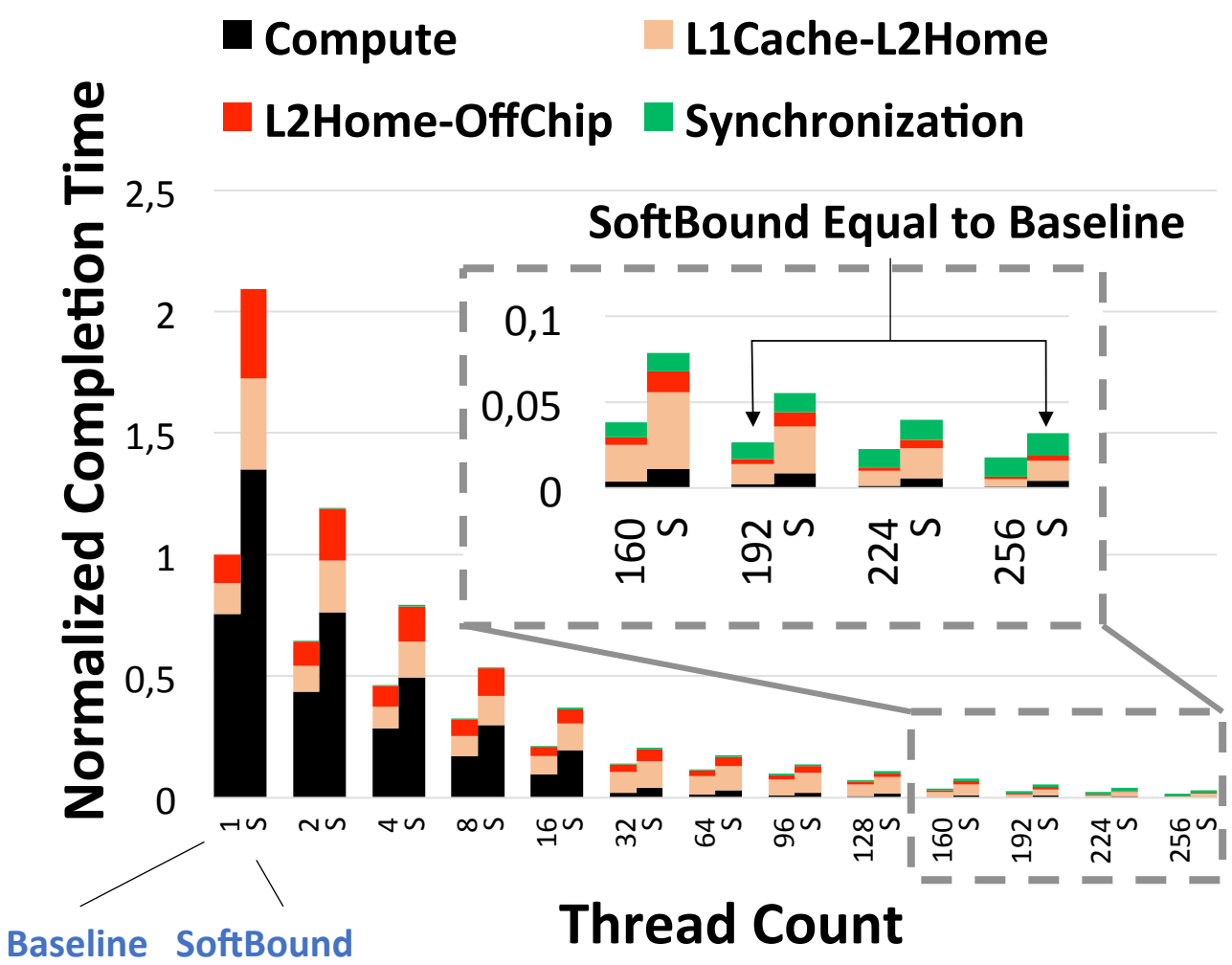
# Benchmark Characterization : Matrix Multiply

- Concurrency does reduce SoftBound's overhead at high thread counts
  - However, overhead still quite significant
- 'S' shows results with SoftBound



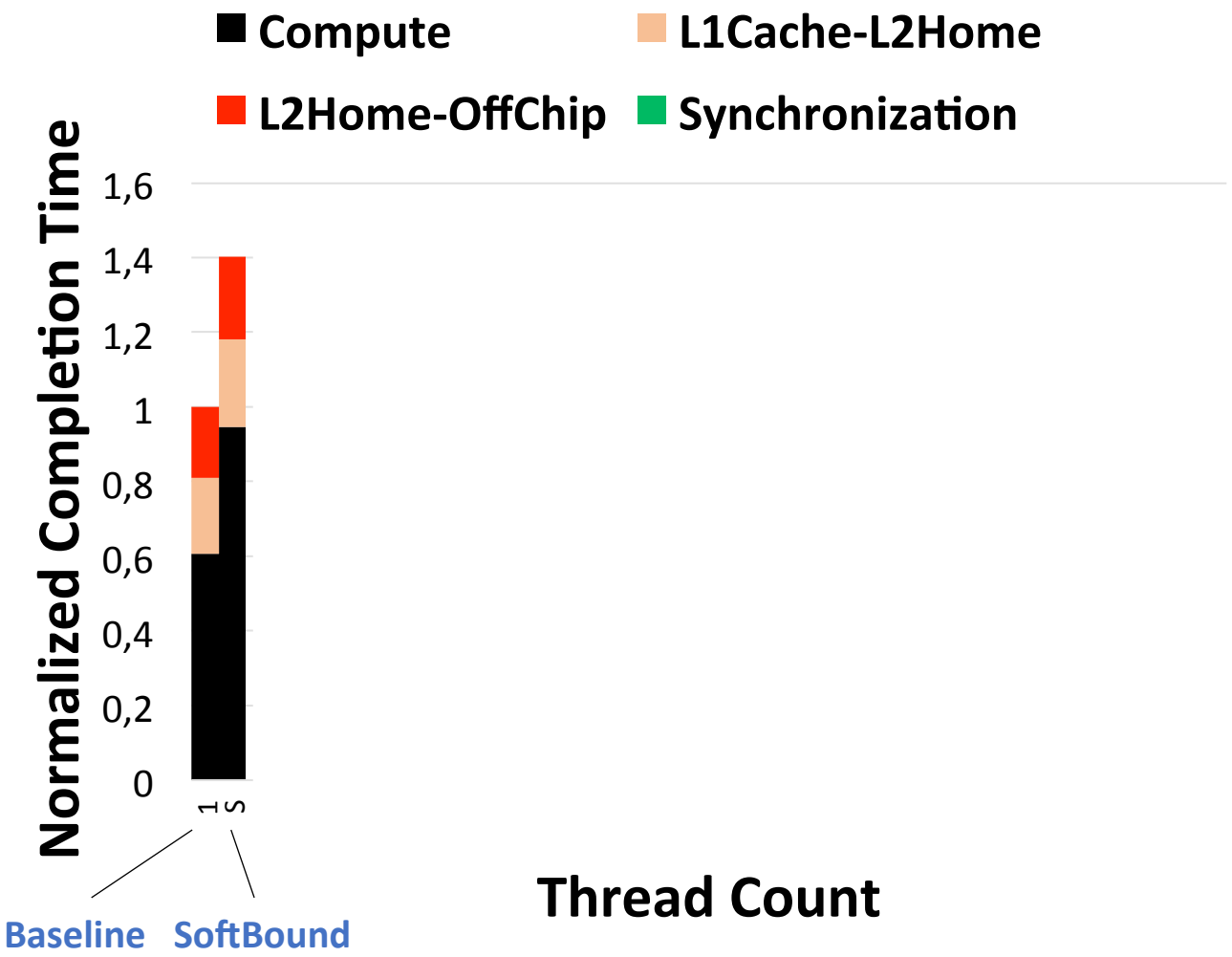
# Benchmark Characterization : Matrix Multiply

- One can get the Efficiency of the baseline by using additional Threads for SoftBound
- 'S' shows results with SoftBound



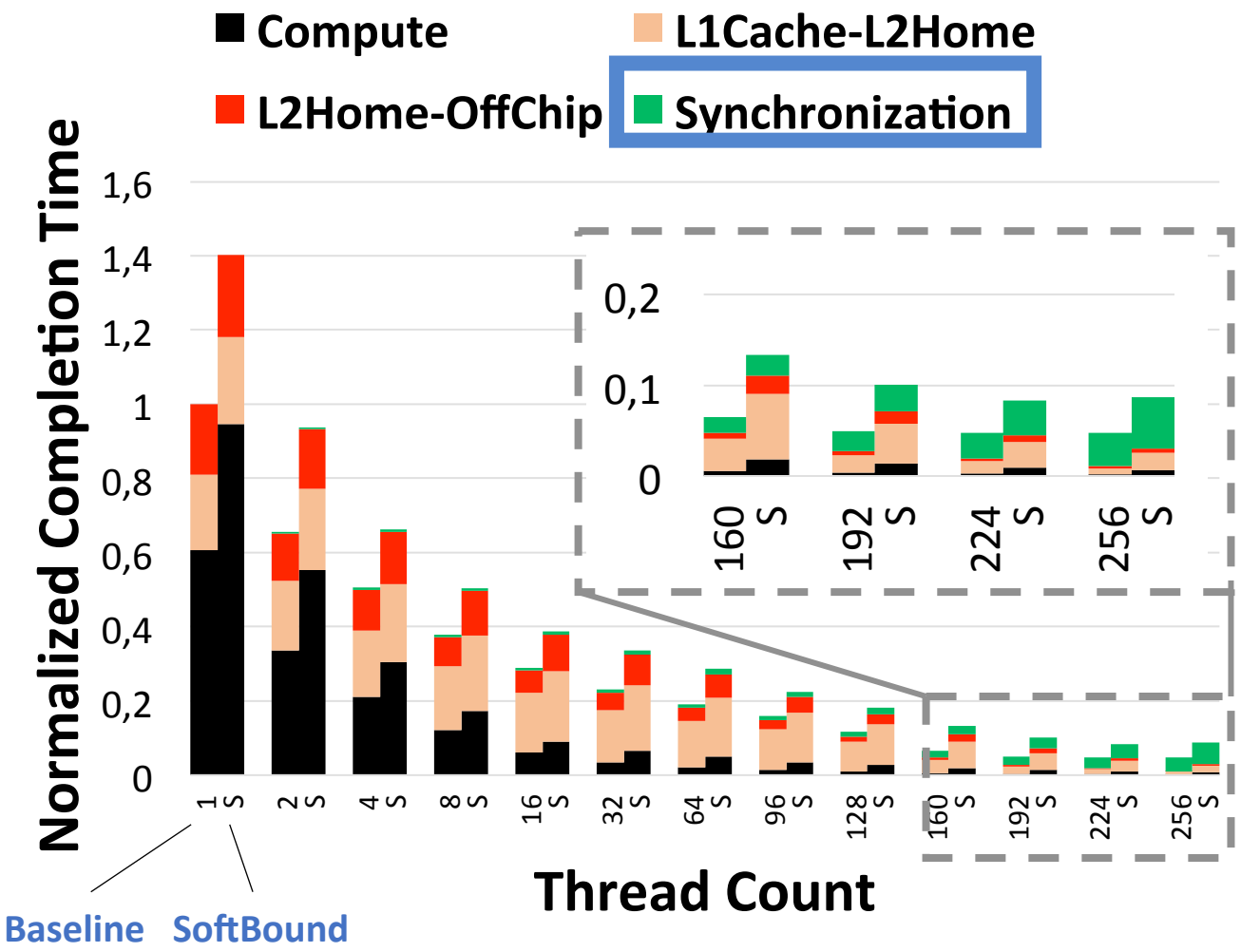
# Benchmark Characterization : Breadth First Search (BFS)

- Another Graph Workload, however with higher scalability and Locality
- Concurrency does reduce SoftBound's overhead at high thread counts
  - However, overhead still quite significant
- 'S' shows results with SoftBound



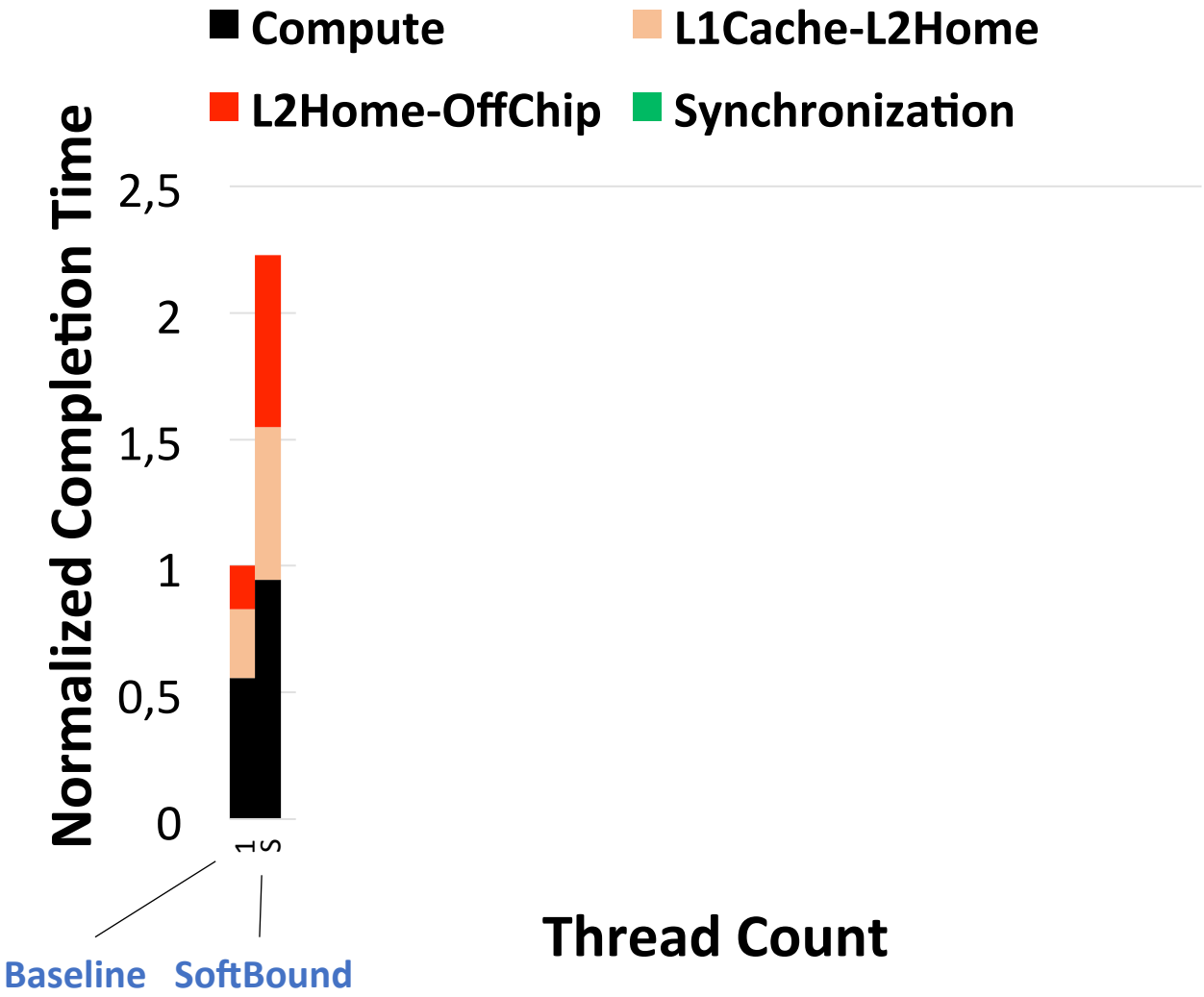
# Benchmark Characterization : Breadth First Search (BFS)

- Concurrency helps reduce SoftBound overhead at high thread counts
  - However, overhead still quite significant because of fine grained synchronization in this workload
- 'S' shows results with SoftBound



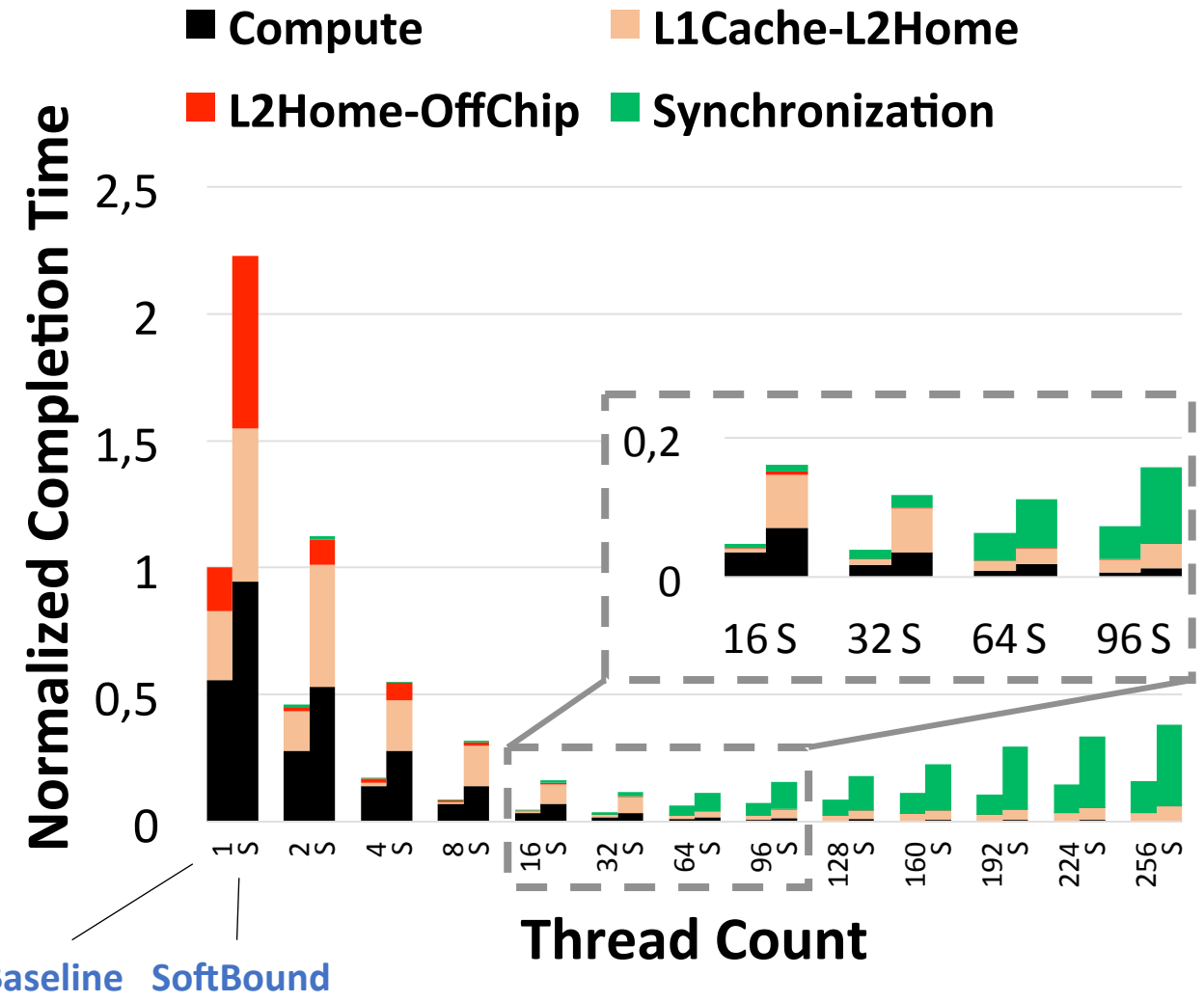
# Benchmark Characterization : Dijkstra

- SoftBound results in higher on-chip and off-chip data accesses, due to large working set
- ‘S’ shows results with SoftBound

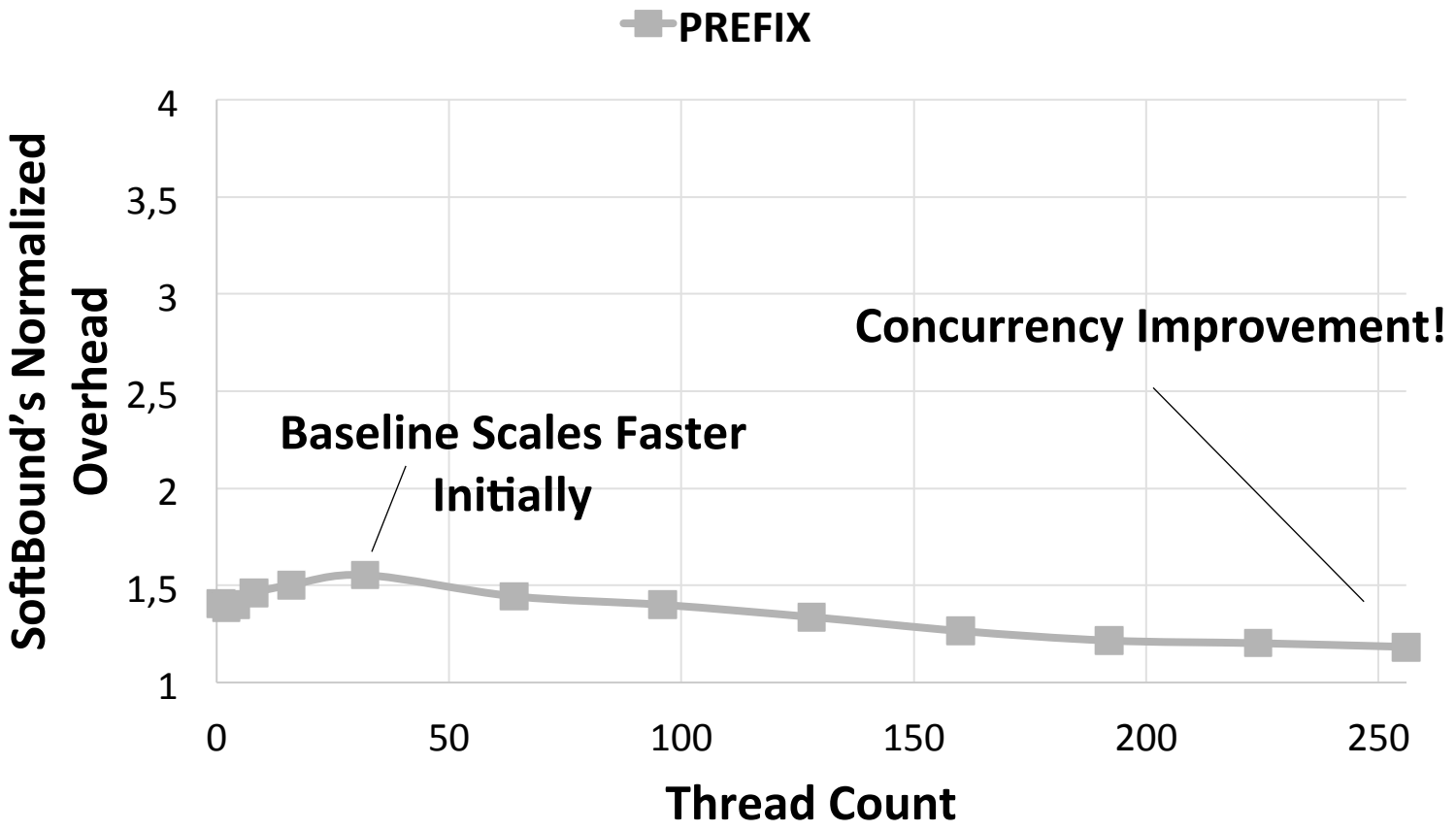


# Benchmark Characterization : Dijkstra

- Scalability is limited due to fine grained communication
- SoftBound checks within critical sections hurts performance
- 'S' shows results with SoftBound

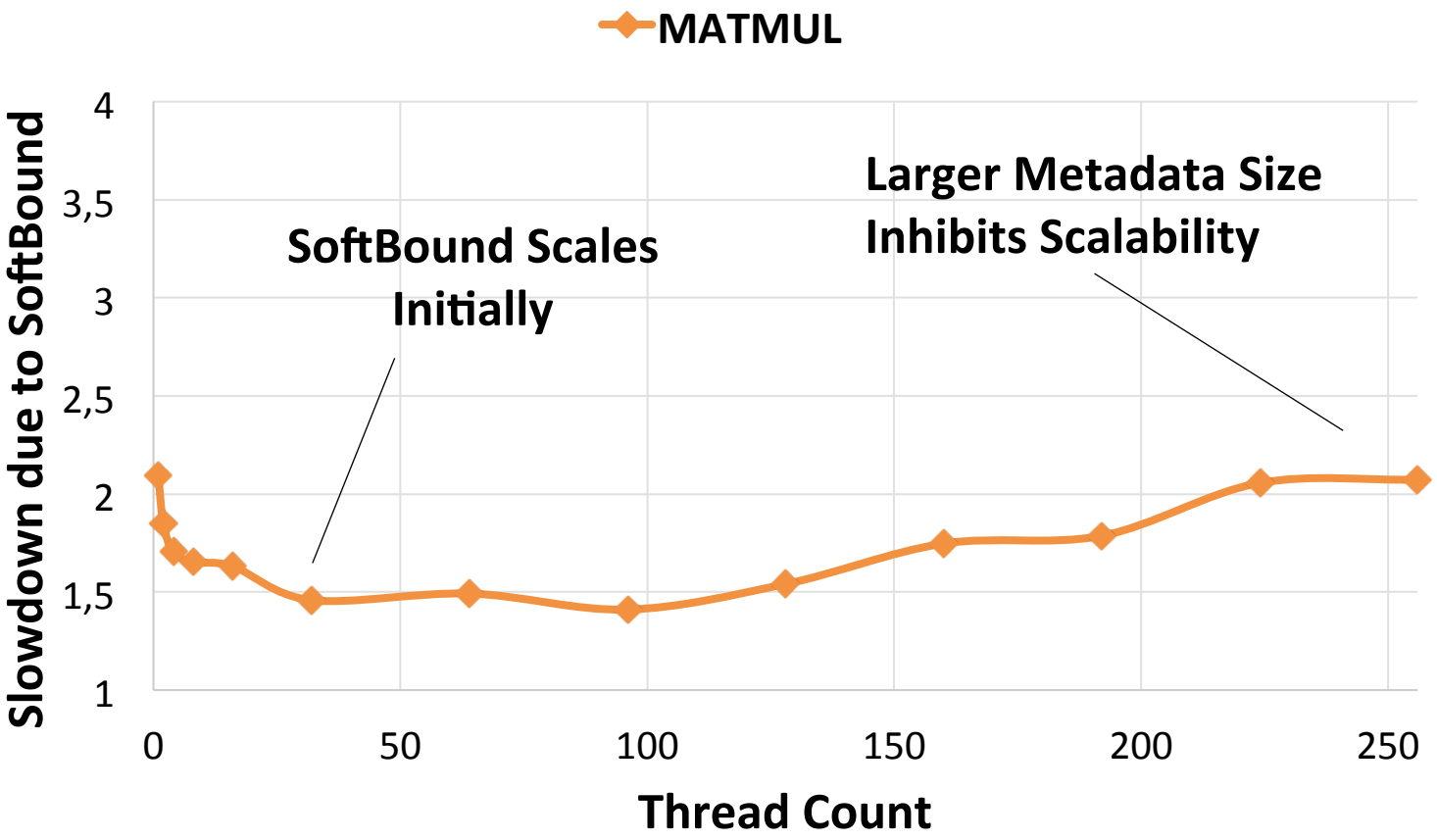


# Benchmark Characterization : Summary of Slowdowns



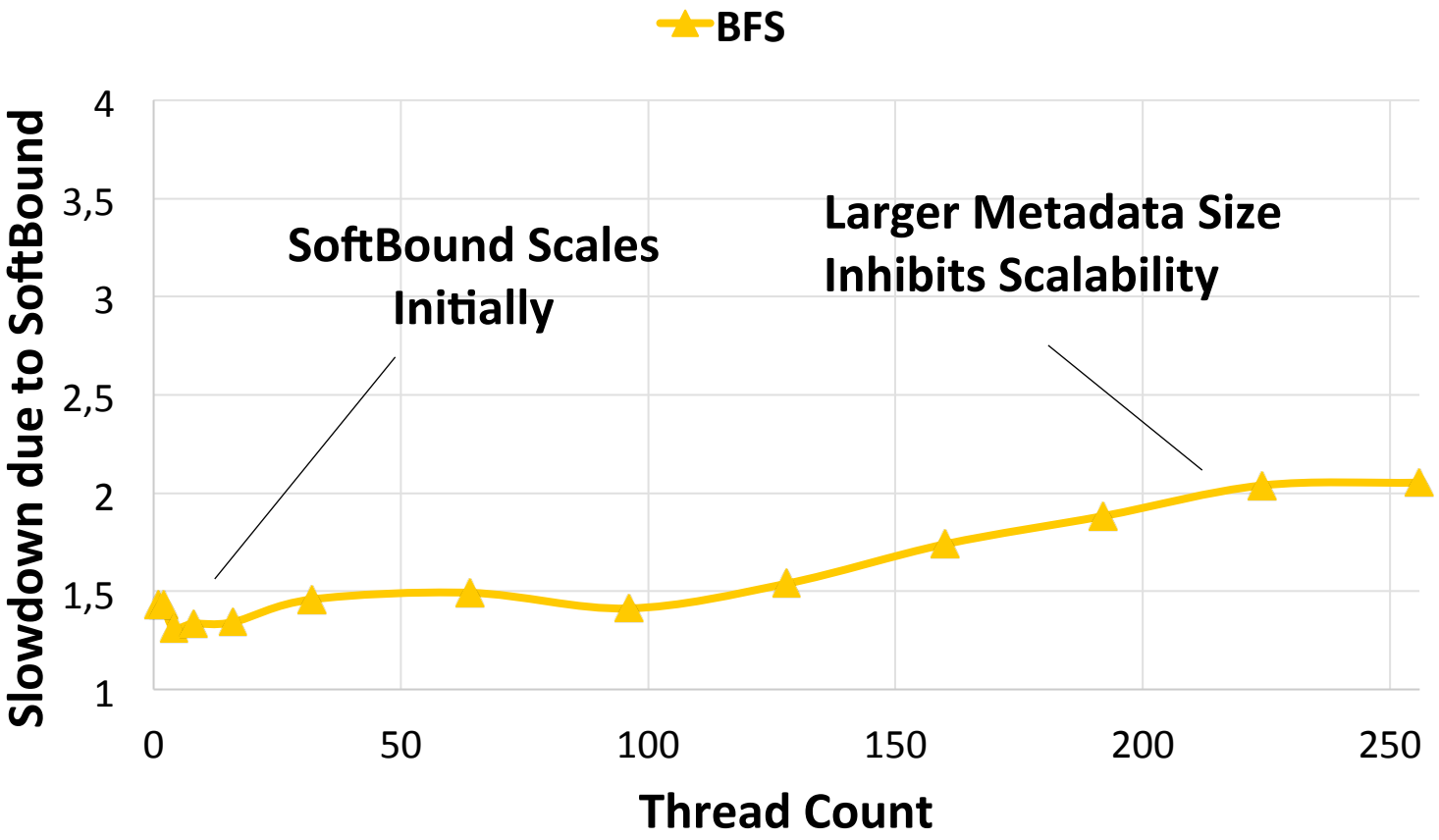

- Concurrency helps hide SoftBound Overheads

# Benchmark Characterization : Summary of Slowdowns



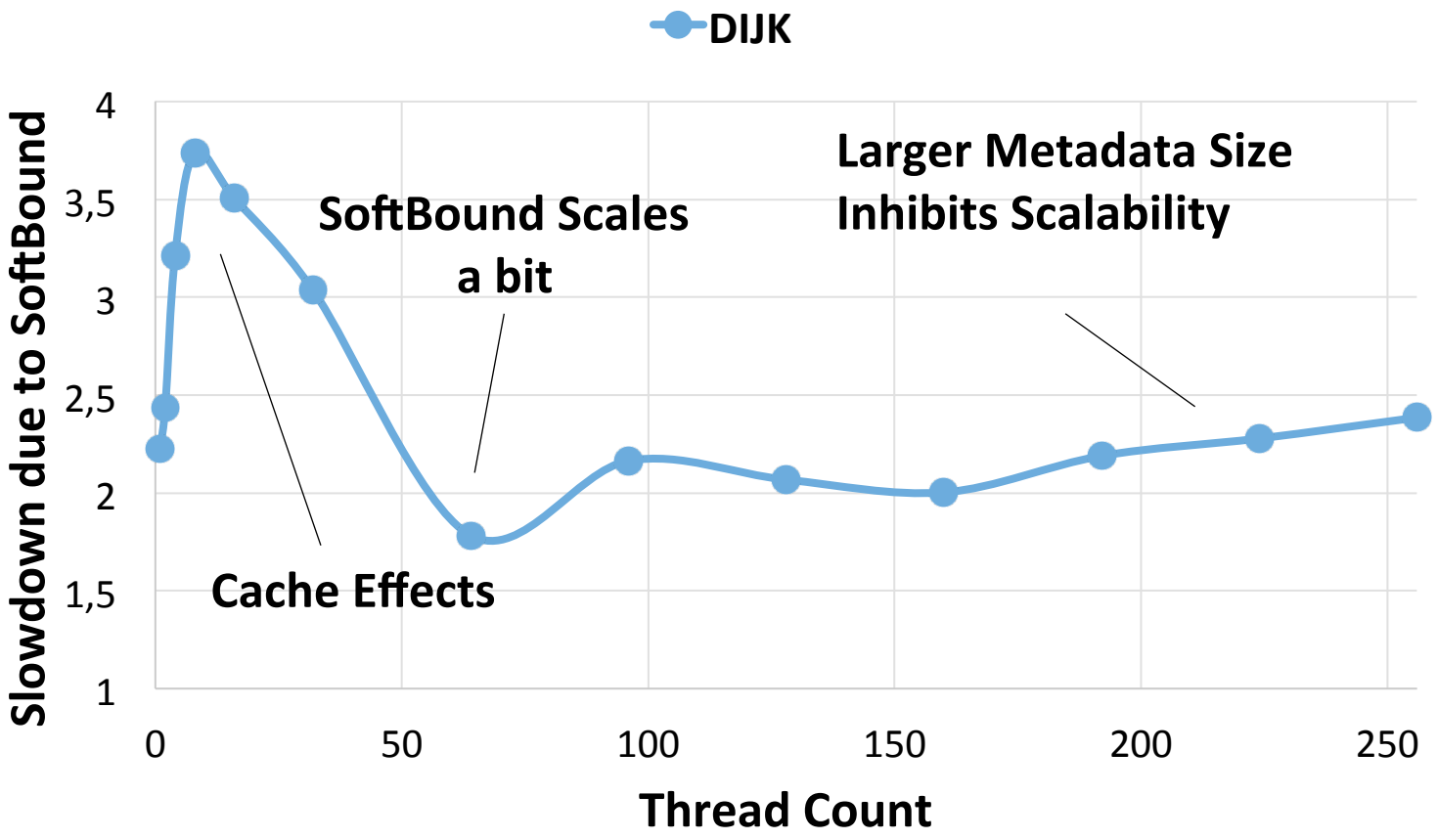

- Concurrency helps hide SoftBound Overheads initially at ~32-64 threads
- Then worsens since metadata impacts the available local cache size

# Benchmark Characterization : Summary of Slowdowns




- Concurrency helps hide SoftBound Overheads initially at ~2-8 threads
- Then worsens since metadata impacts critical sections

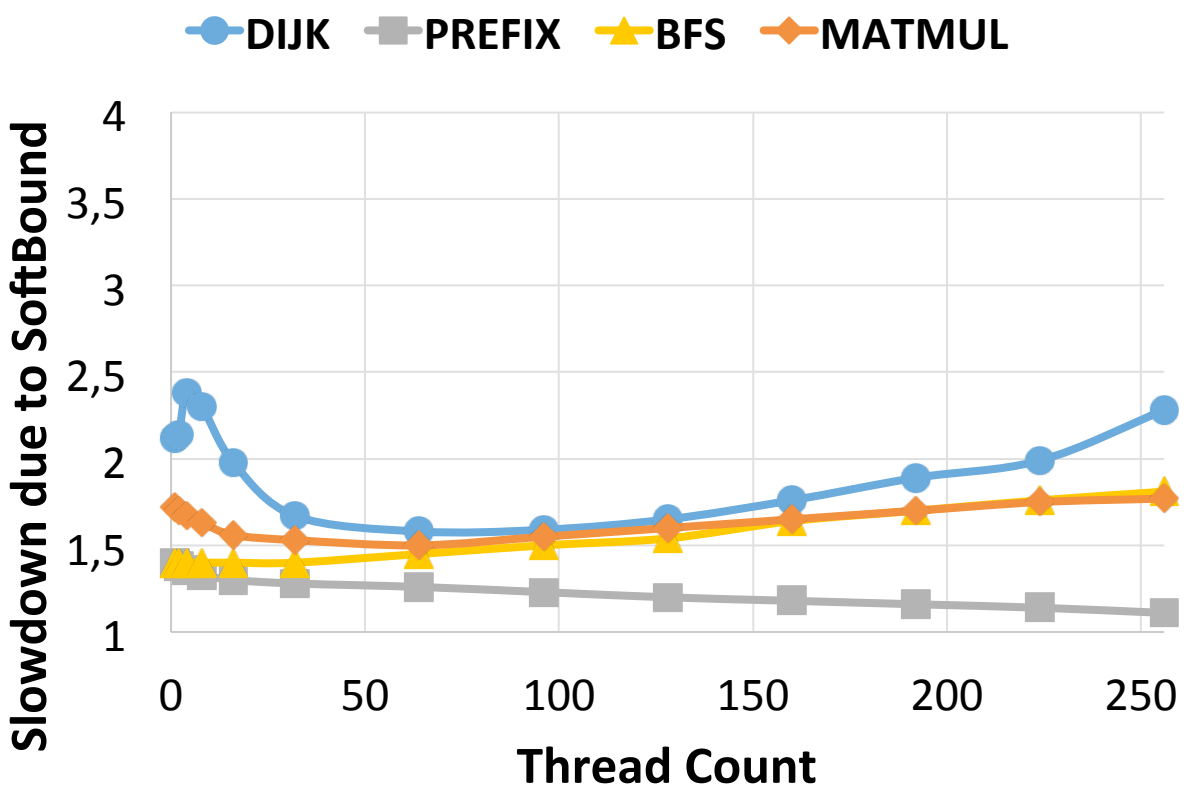
# Benchmark Characterization : Summary of Slowdowns




- Concurrency helps hide SoftBound Overheads initially at ~2-8 threads
- Then worsens since metadata impacts critical sections

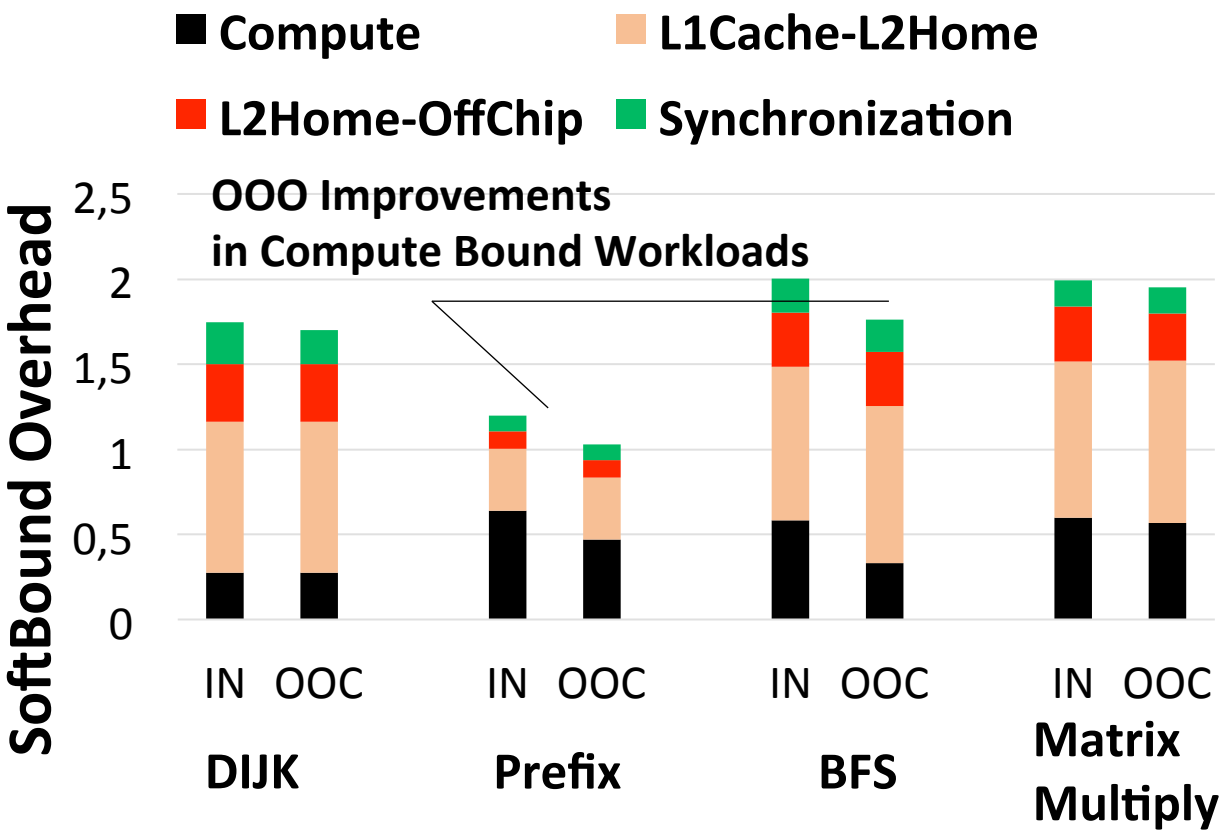
# Benchmark Characterization : Slowdowns in **Out-of-Order Cores**

- All Prior Results had In-Order Cores
- Reduction in performance degradation observed
  - OOOs improve critical code sections
- **Latency hiding helps SoftBound scale better**



# Benchmark Characterization : In-Order vs. Out-of-Order

- Parallel SoftBound results for both In-Order and OOO core types
  - At thread counts showing highest speedups
- **OOOs can not improve parallel performance much**
  - **Alternative architectural improvements needed**



# Agenda

Motivation

Characterization Methodology

Characterization Results

**Insights and Possible Improvements**

# Insights from Characterization

- Most bottlenecks in Multi-threaded SoftBound workloads stem from Memory Accesses and Synchronization
- Latency Hiding does improve SoftBound using OoO Cores slightly
- Increase in thread count allows SoftBound to perform as well as a baseline at a lower thread counts
- **However, additional software/architectural mechanisms are needed to further improve Performance**

# Potential Future Improvements

- Improve SoftBound's **Metadata structures**
  - Compression
  - Better Layout (e.g. a better tree type of structure)

- Use Parallelization Strategies (e.g. Blocking) that are aware of Security Metadata's presence

- Devise a **Prefetcher for Security Metadata**
  - Prefetch SoftBound metadata from DRAM
  - Hides SoftBounds latency