Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs?

Eddy Zheng Zhang

Yunlian Jiang

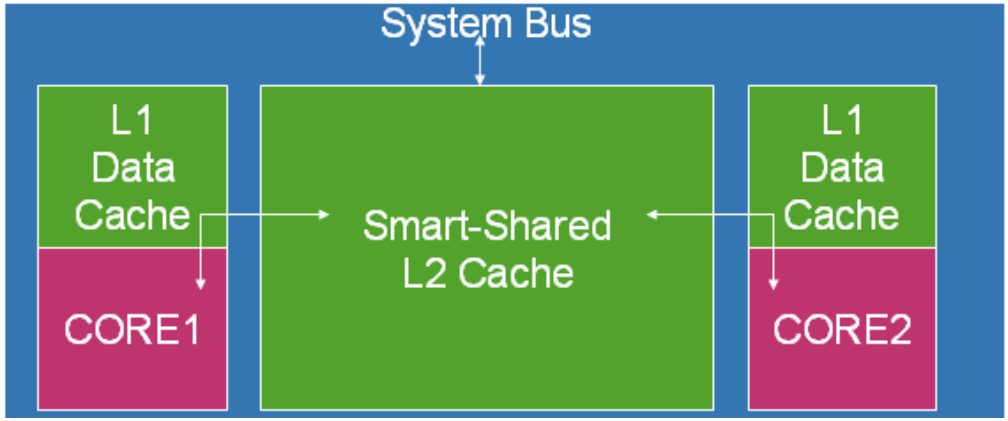
Xipeng Shen (presenter)

Computer Science Department
The College of William and Mary, VA, USA

Cache Sharing

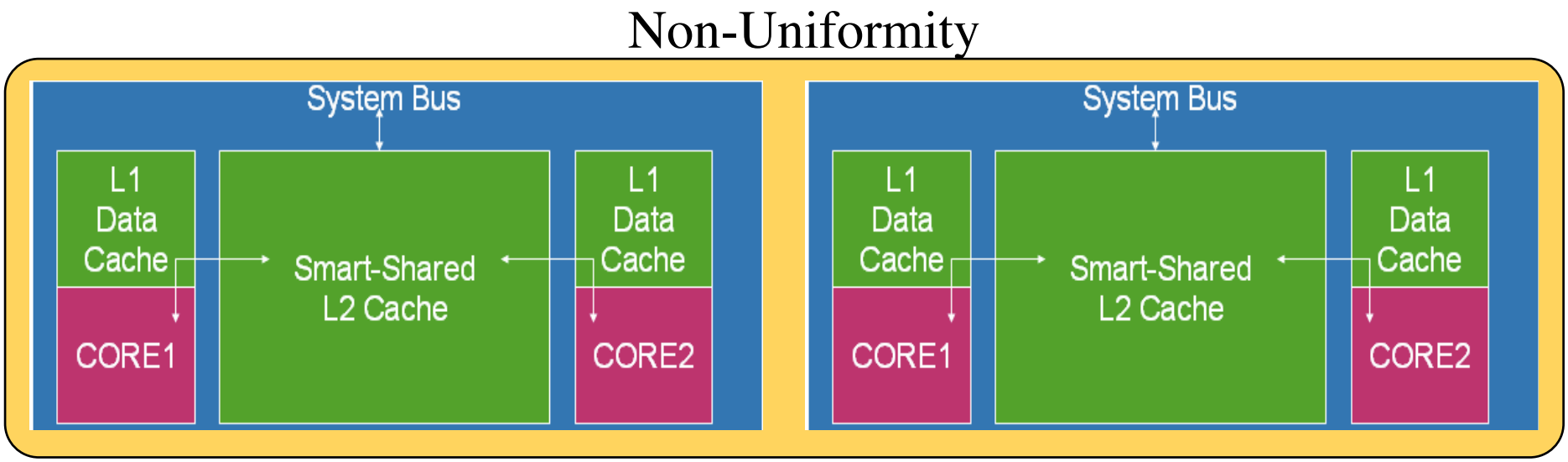
- A common feature on modern CMP

Cache Sharing on CMP



- A double-edged sword
  - Reduces communication latency
  - But causes conflicts & contention

Cache Sharing on CMP



- A double-edged sword
  - Reduces communication latency
  - But causes conflicts & contention

Many Efforts for Exploitation

- Example: shared-cache-aware scheduling
  - Assigning suitable programs/threads to the same chip
    - Independent jobs
      - Job Co-Scheduling [Snavely+:00, Snavely+:02, El-Moursy+:06, Fedorova+:07, Jiang+:08, Zhou+:09]
    - Parallel threads of server applications
      - Thread Clustering [Tam+:07]

Overview of this Work (1/3)

- A surprising finding
  - Insignificant effects from shared cache on a recent multithreaded benchmark suite (PARSEC)
  - Drawn from a systematic measurement
    - thousands of runs
    - 7 dimensions on levels of programs, OS, & architecture
    - derived from timing results
    - confirmed by hardware performance counters

Overview of this Work (2/3)

- A detailed analysis
  - Reason
    - three mismatches between executables and CMP cache architecture
  - Cause
    - the current development and compilation are oblivious to cache sharing

Overview of this Work (3/3)

- An exploration of the implications
  - Exploiting cache sharing deserves not less but more attention.
  - But to exert the power, cache-sharing-aware transformations are critical
    - Cuts half of cache misses
    - Improves performance by 36%.

Outline

- Experiment design
- Measurement and findings
- Cache-sharing-aware transformation
- Related work, summary, and conclusion.

Benchmarks (1/3)

- PARSEC suite by Princeton Univ [Bienia+:08]

> *"focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors"*

Benchmarks (2/3)

- Composed of
  - RMS applications
  - Systems applications
  - ……
- A wide spectrum of
  - working sets, locality, data sharing, synch., off-chip traffic, etc.

Benchmarks (3/3)

Program	Description	Parallelism	Working Set
Blackscholes	Black-Scholes equation	data	2MB
Bodytrack	body tracking	data	8MB
Canneal	sim. Annealing	unstruct.	256MB
Facesim	face simulation	data	256MB
Fluidanimate	fluid dynamics	data	64MB
Streamcluster	online clustering	data	16MB
Swaptions	portfolio pricing	data	0.5MB
X264	video encoding	pipeline	16MB
Dedup	stream compression	pipeline	256MB
Ferret	image search	pipeline	64MB

Factors Covered in Measurements

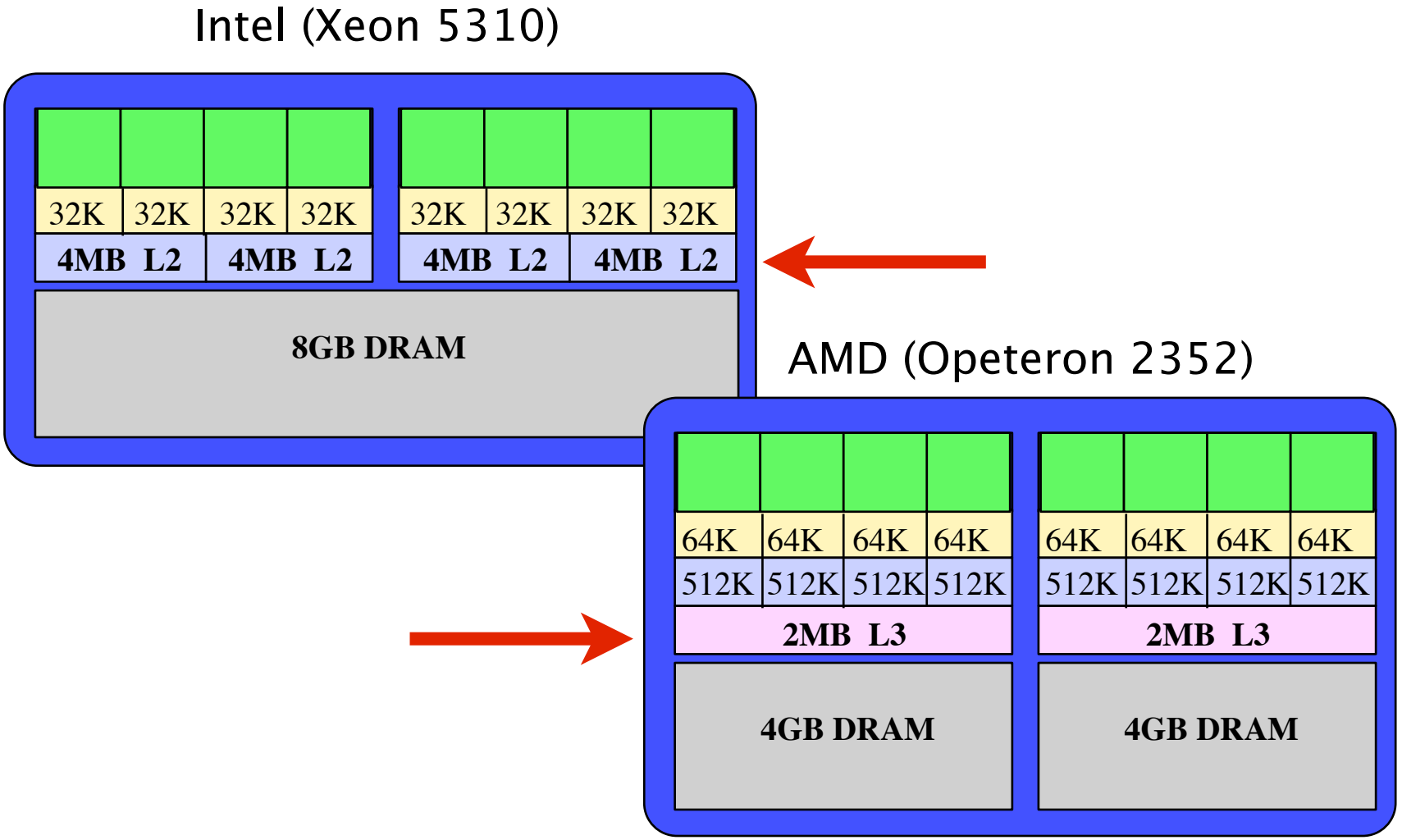
Dimension	Variations	Description
benchmarks	10	from PARSEC
parallelism	3	data, pipeline, unstructured
inputs	4	simsmall, simmedium, simlarge, native
# of threads	4	1,2,4,8
assignme	3	threads assignment to cores
binding	2	yes, no
subset of cores	7	The cores a program uses
platforms	2	Intel Xeon & AMD Operon

Program level

OS level

Arch. level

Machines


Intel (Xeon 5310)
32K
32K
32K
32K
32K
32K
32K
32K
4MB L2
4MB L2
4MB L2
4MB L2
8GB DRAM
AMD (Opeteron 2352)
64K
64K
64K
64K
64K
64K
64K
64K
512K
512K
512K
512K
512K
512K
512K
512K
2MB L3
2MB L3
4GB DRAM
4GB DRAM

Measurement Schemes

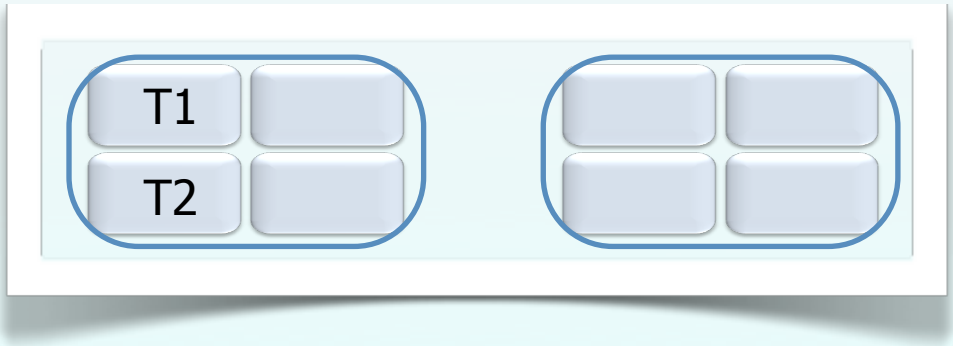
- Running times
  - Built-in hooks in PARSEC
- Hardware performance counters
  - PAPI
  - cache miss, mem. bus, shared data accesses

Outline

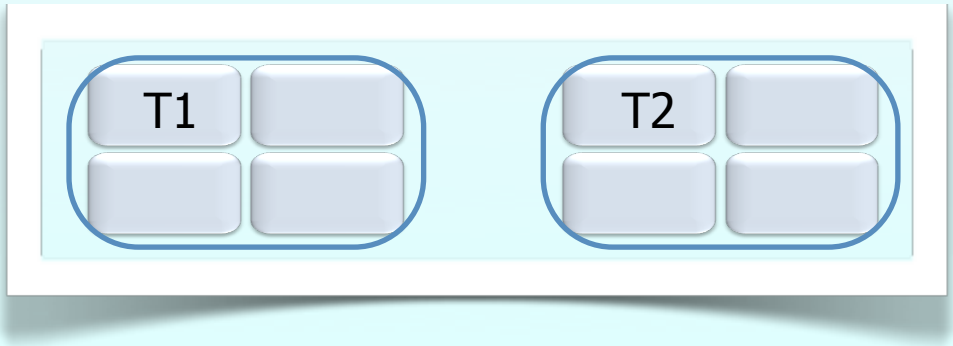
- Experiment design
- Measurement and findings
- Cache-sharing-aware transformation
- Related work, summary, and conclusions

Observation I:
Sharing vs. Non-sharing

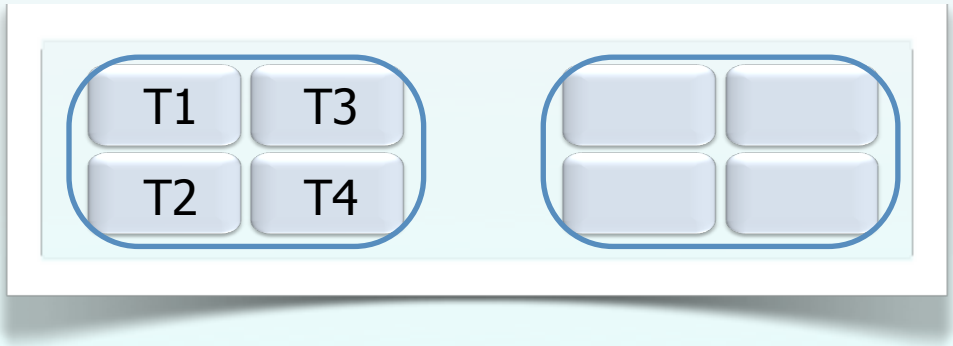
Sharing vs. Non-sharing



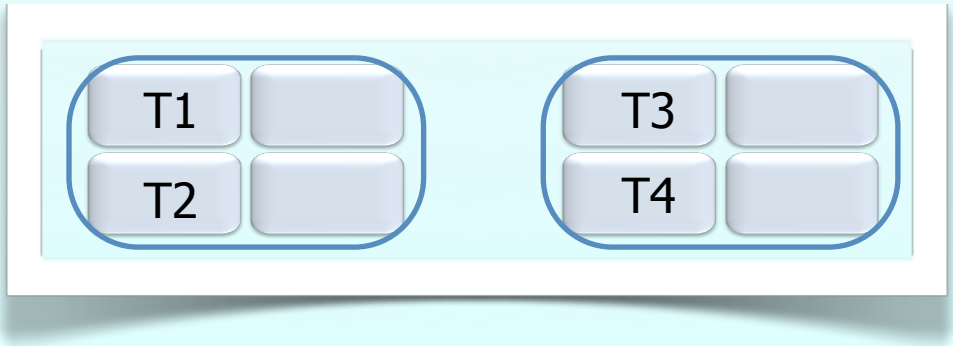
VS.



Sharing vs. Non-sharing

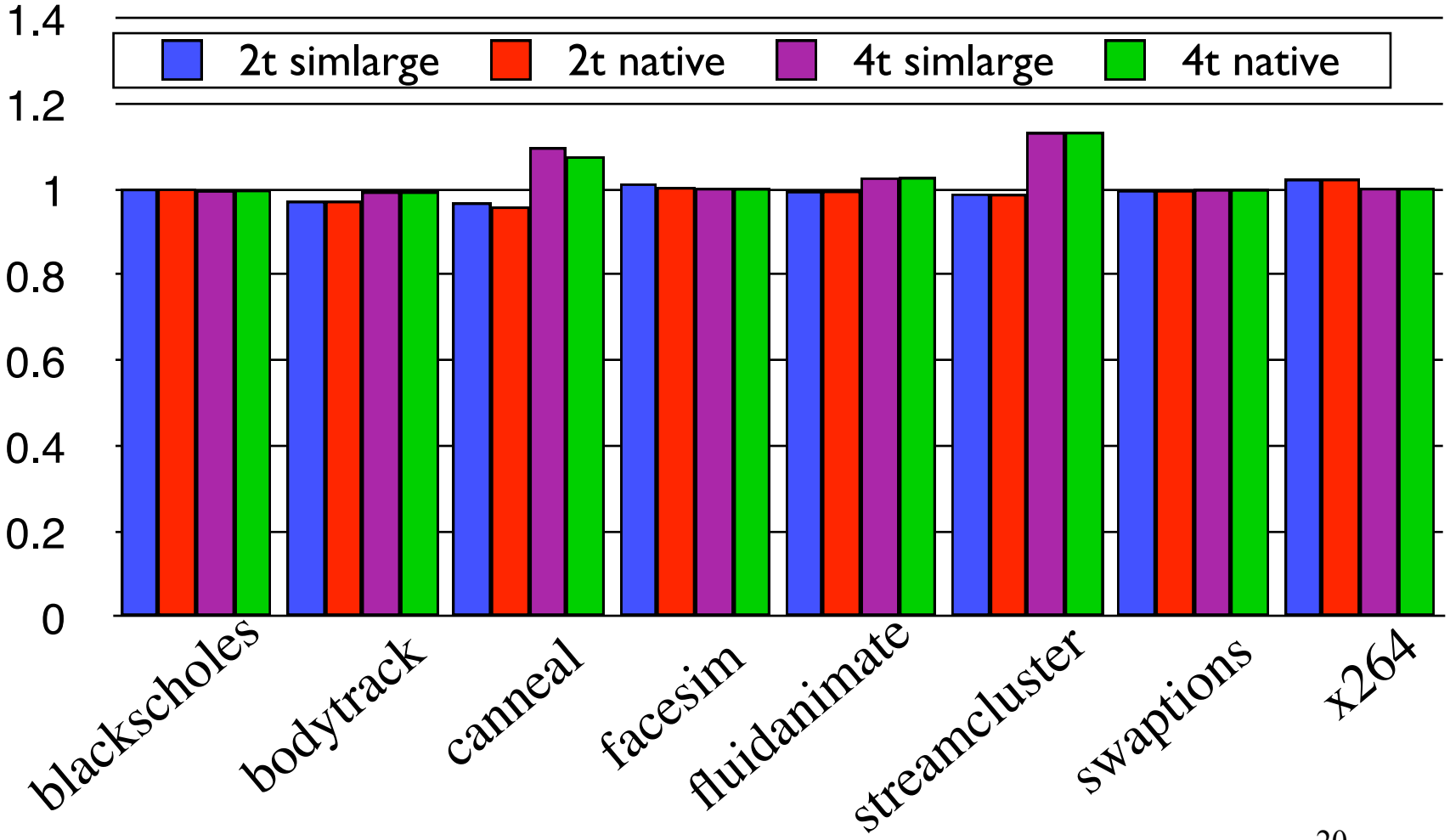


VS.



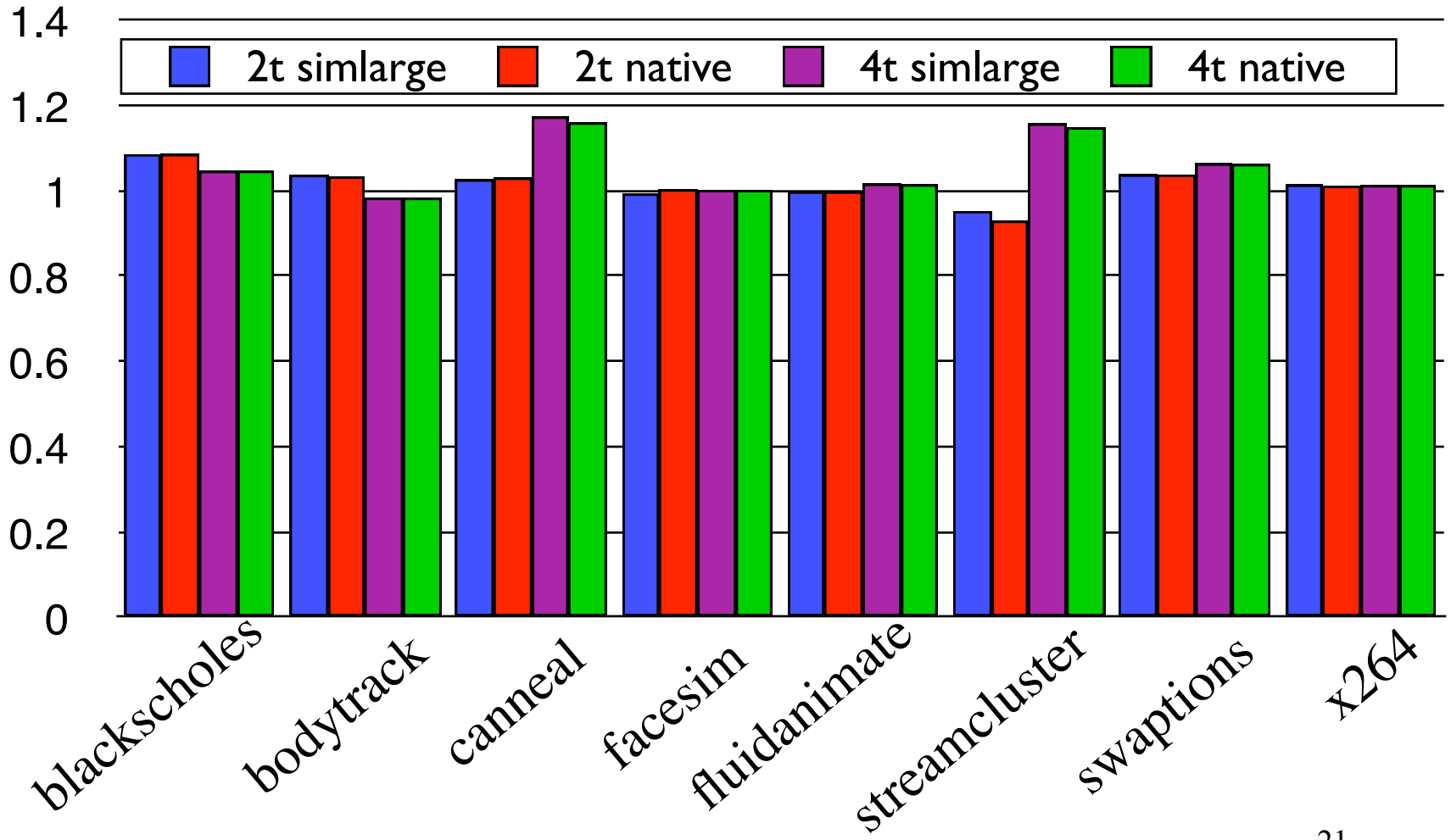
Sharing vs. Non-sharing

- Performance Evaluation (Intel)



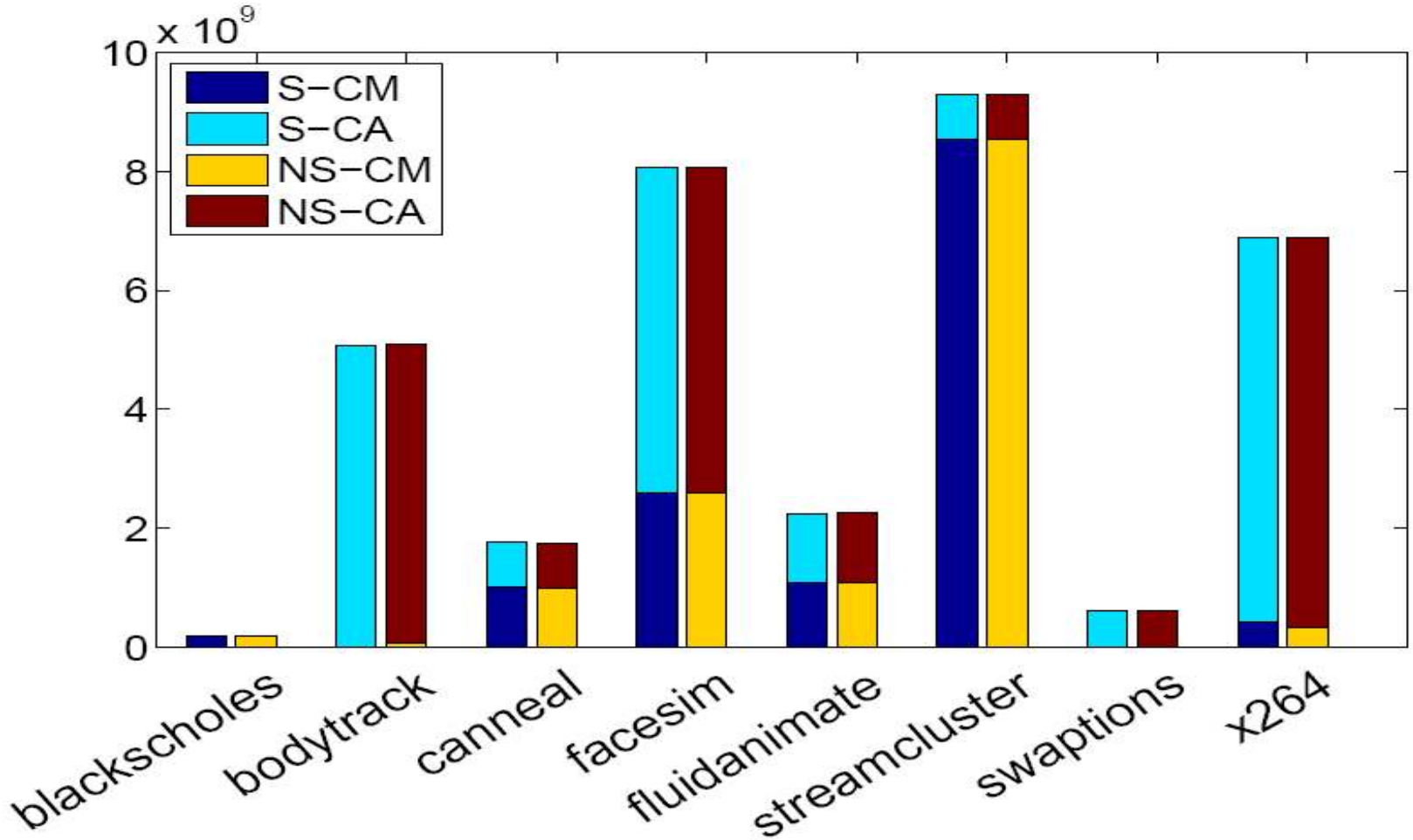
Sharing vs. Non-sharing

- Performance Evaluation (AMD)



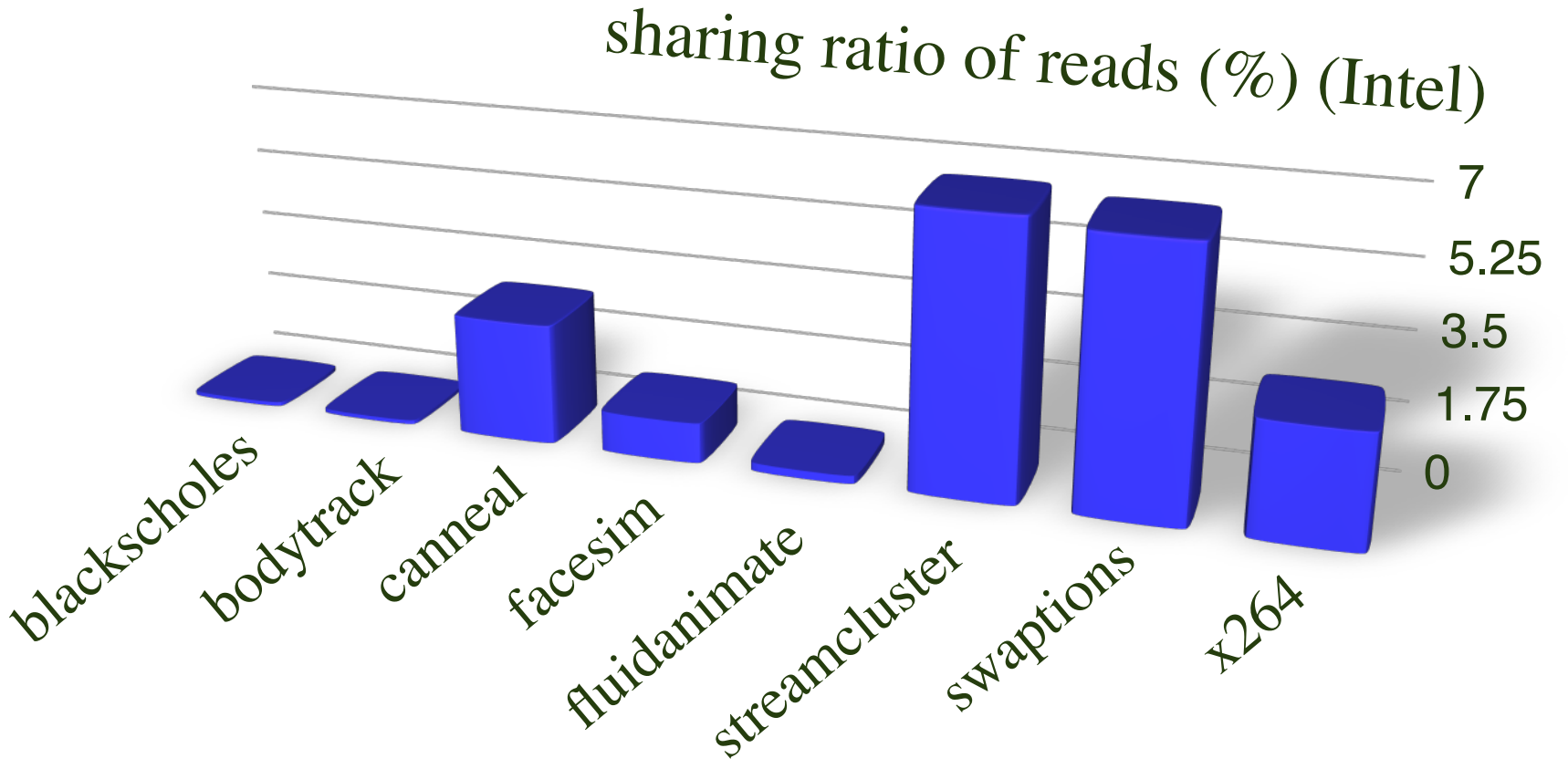
Sharing vs. Non-sharing

- L2-cache accesses & misses (Intel)



Reasons (1/2)

1) Small amount of inter-thread data sharing



Reasons (2/2)

2) Large working sets

Program	Description	Parallelism	Working Set
Blackscholes	Black-Scholes equation	data	2MB
Bodytrack	body tracking	data	8MB
Canneal	sim. Annealing	unstruct.	256MB
Facesim	face simulation	data	256MB
Fluidanimate	fluid dynamics	data	64MB
Streamcluster	online clustering	data	16MB
Swaptions	portfolio pricing	data	0.5MB
X264	video encoding	pipeline	16MB
Dedup	stream compression	pipeline	256MB
Ferret	image search	pipeline	64MB

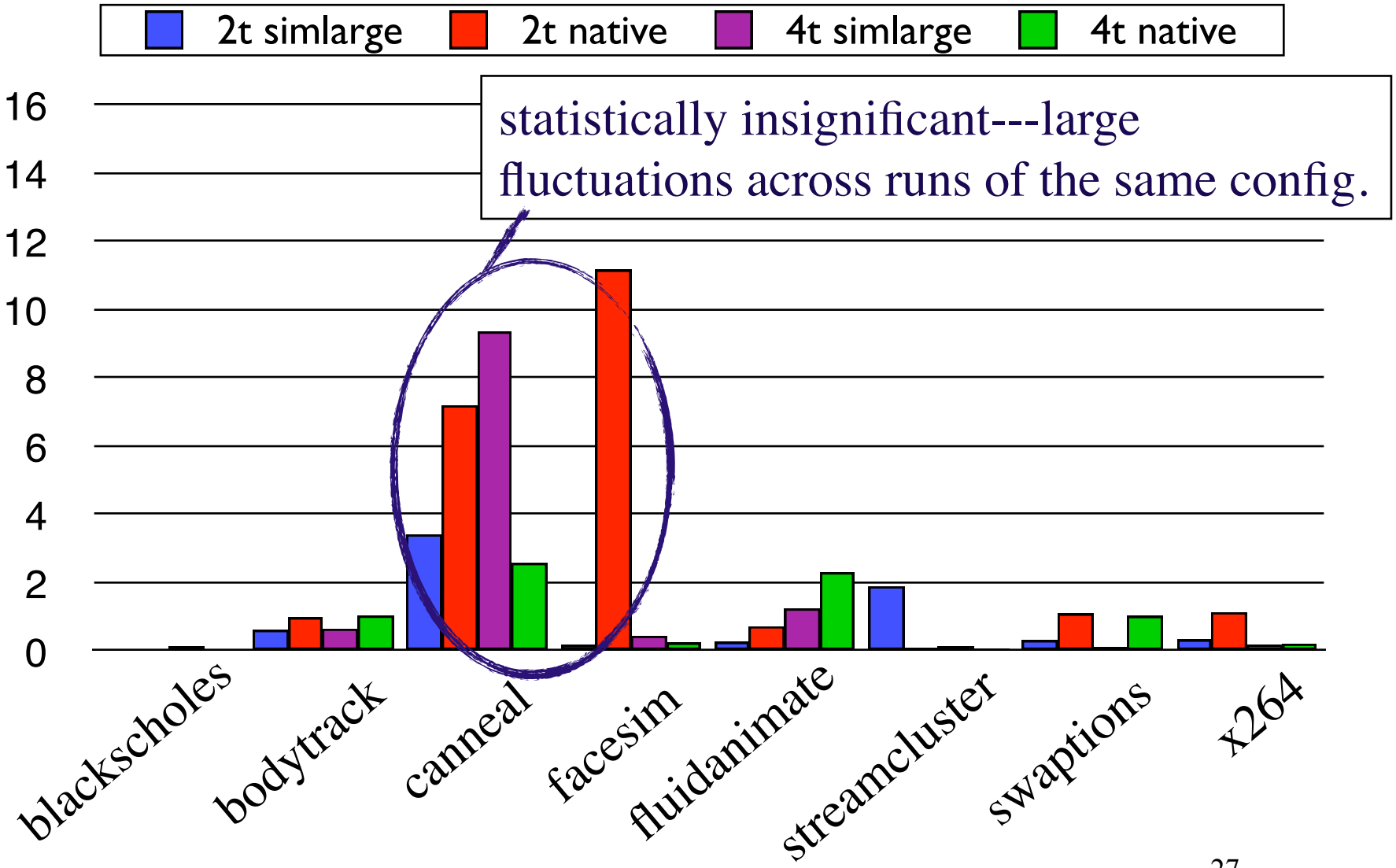
Observation II: Different Sharing Cases

- Threads may differ
  - Different data to be processed or tasks to be conducted.
  - Non-uniform communication and data sharing.

- Different thread placement may give different performance in the sharing case.

Different Sharing Cases

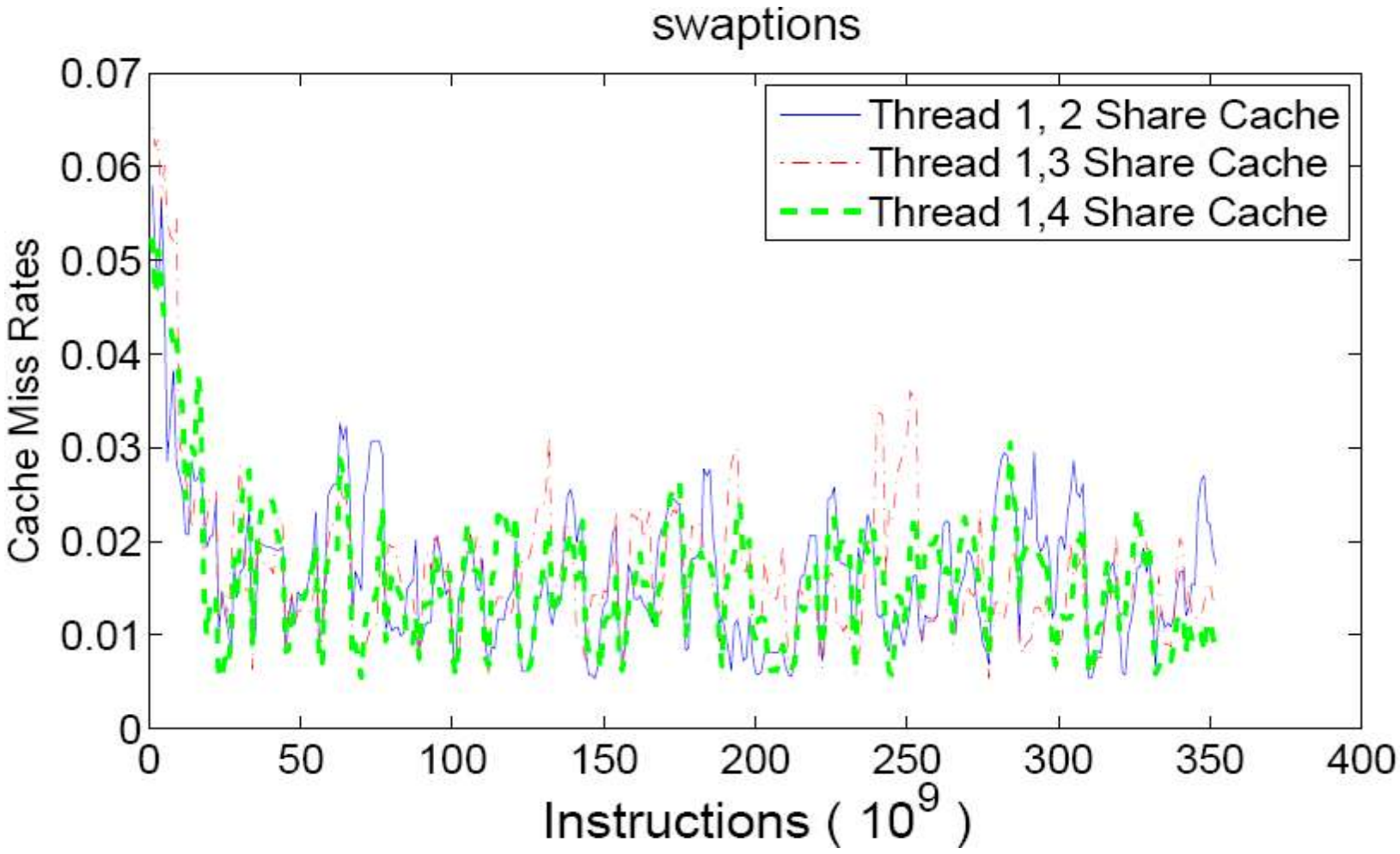
Max. Perf. Diff (%)



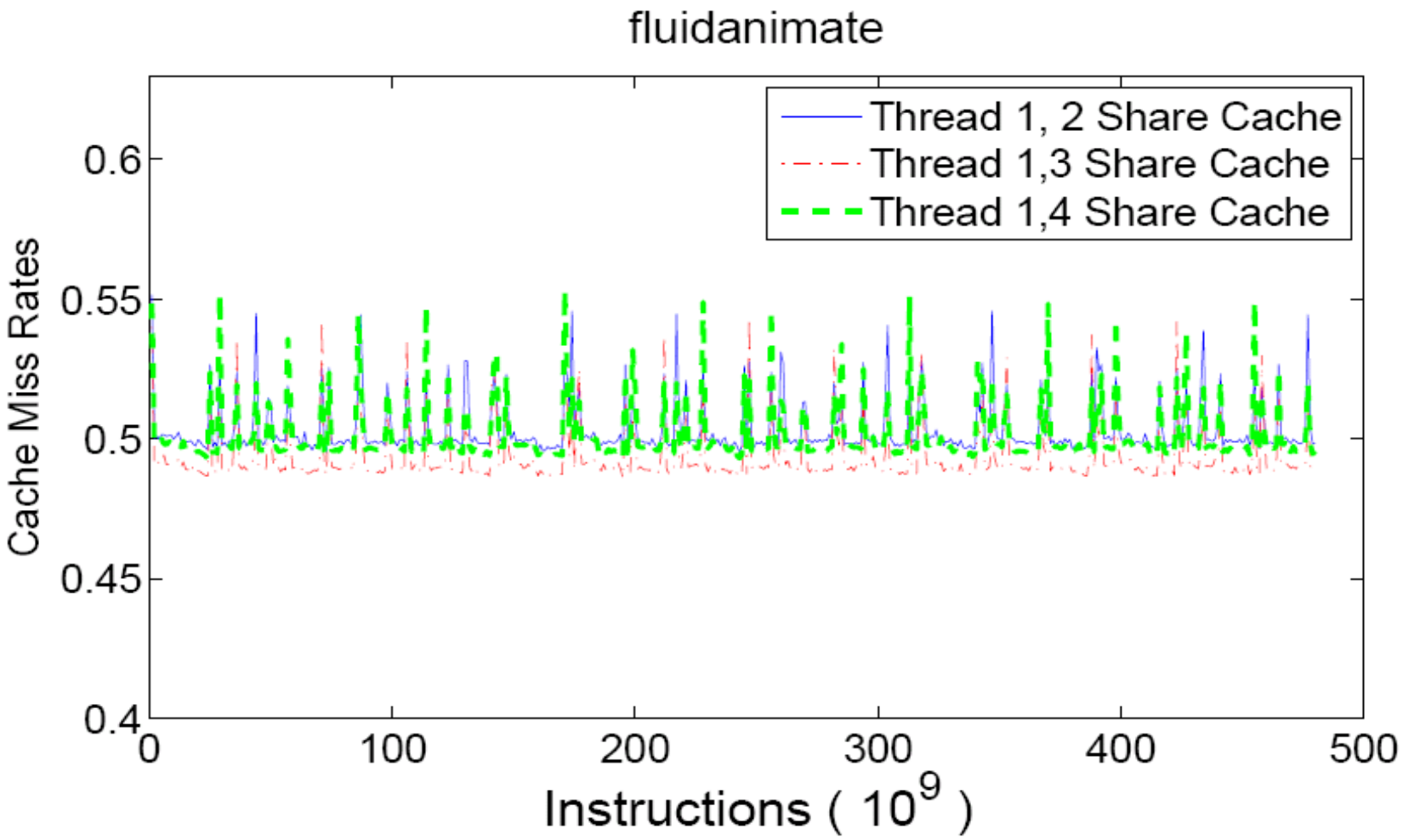
Two Possible Reasons

- Similar interactions among threads
- Differences are smoothed by phase shifts

Temporal Traces of L2 misses



Temporal Traces of L2 misses



Two Possible Reasons

- Similar interactions among threads
- ~~Differences are smoothed by phase shifts~~

Pipeline Programs

- Two such programs: ferret, and dedup
- Numerous concurrent stages
  - Interactions within and between stages

- Large differences between different thread-core assignments
- Mainly due to load balance rather than differences in cache sharing.

A Short Summary

- Insignificant influence on performance
  - Large working sets
  - Little data sharing
- Thread placement does not matter
  - Due to uniform relations among threads
- Hold across inputs, # threads, architecture, phases.

Outline

- Experiment design
- Measurement and findings
- Cache-sharing-aware transformation
- Related work, summary, and conclusions

Principle

- Increase data sharing among siblings
- Decrease data sharing otherwise

Non-uniform threads

Non-uniform cache sharing

Example: streamcluster

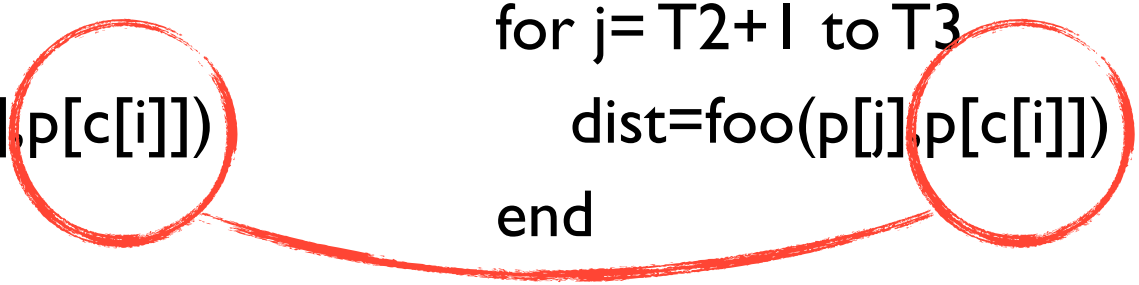
original code

thread 1

```
for i = 1 to N, step =1
… …
   for j= T1 to T2
      dist=foo(p[j],p[c[i]])
   end
… …
end
```

thread 2

```
for i = 1 to N, step =1
… …
   for j= T2+1 to T3
      dist=foo(p[j],p[c[i]])
   end
… …
end
```



Example: streamcluster

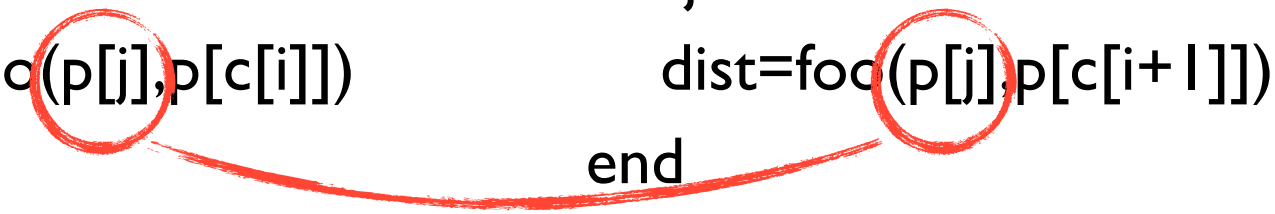
optimized code

thread 1

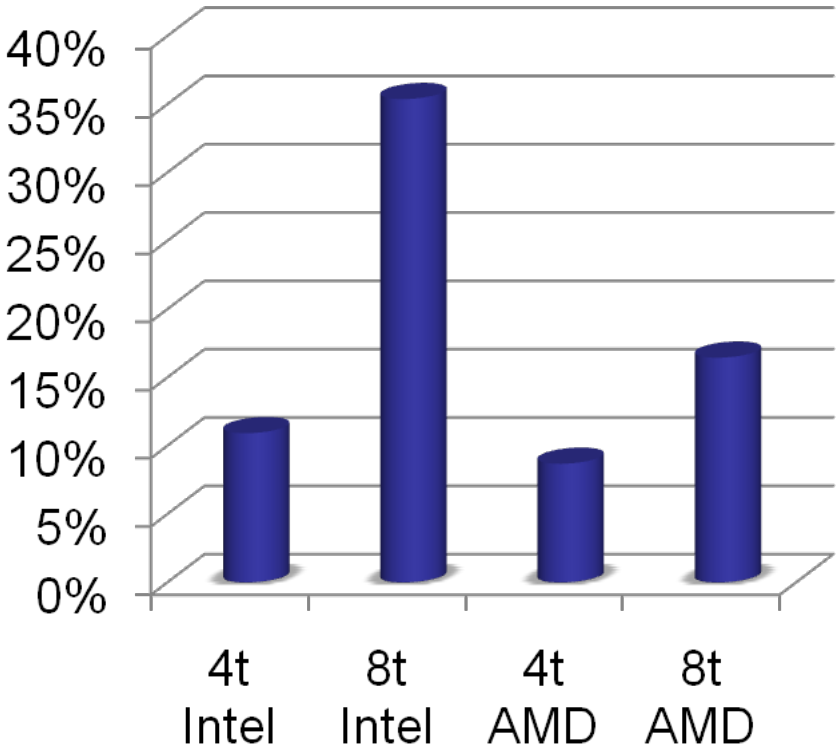
```
for i = 1 to N, step =2
… …
   for j= T1 to T3
      dist=foo(p[j],p[c[i]])
   end
… …
end
```

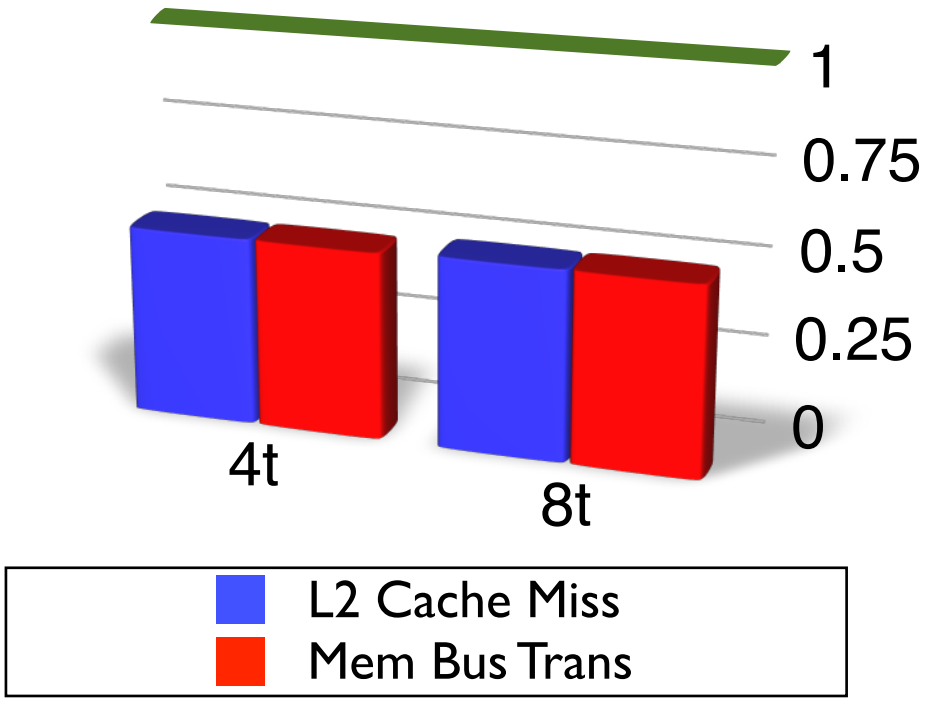
thread 2

```
for i = 1 to N, step =2
… …
   for j= T1 to T3
      dist=foo(p[j],p[c[i+1]])
   end
… …
end
```

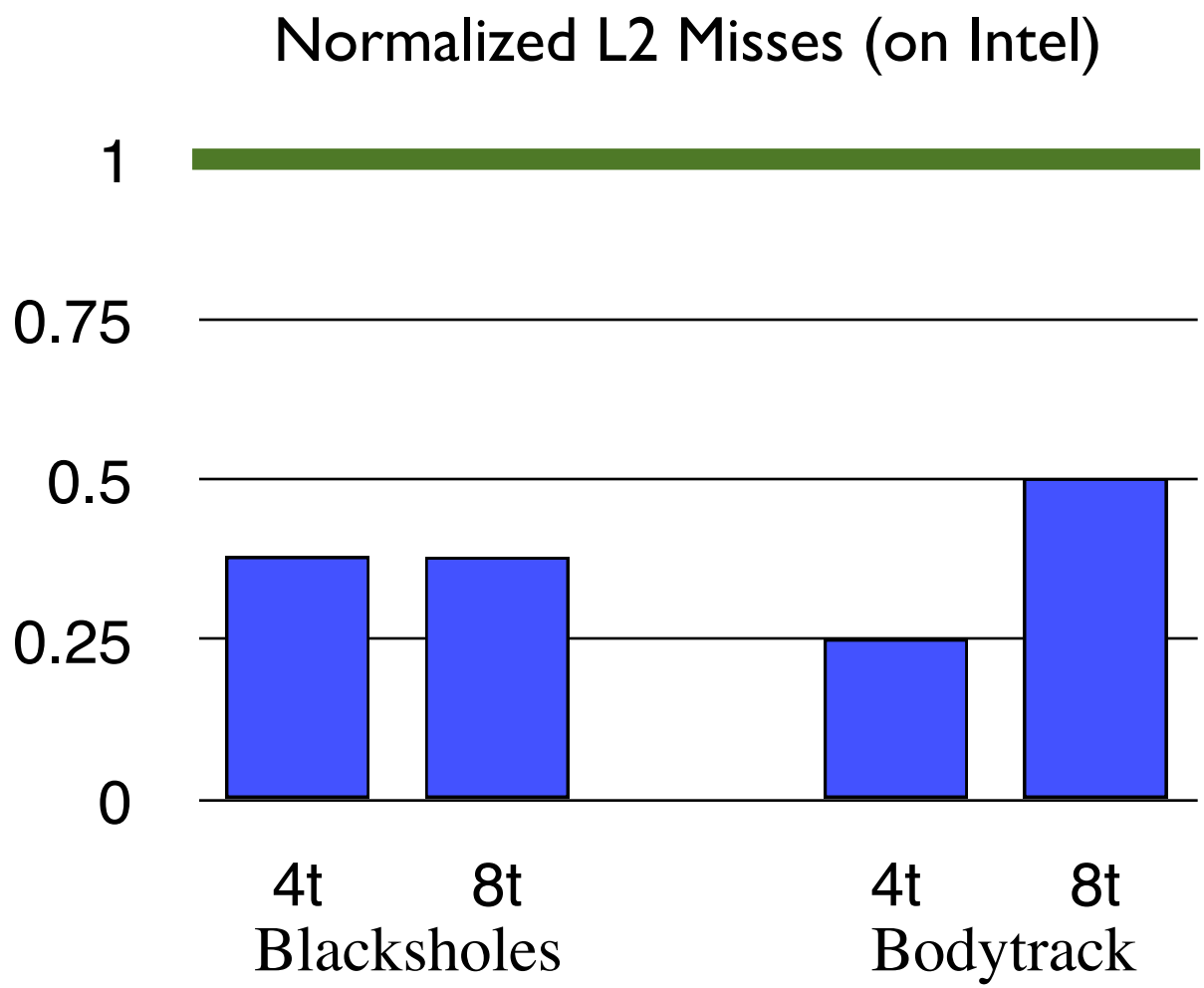


Performance Improvement (streamcluster)





Other Programs



Implication

- To exert the potential of shared cache, program-level transformations are critical.

- Limited existing explorations
  - Sarkar & Tullsen'08, Kumar& Tullsen'02, Nokolopoulos'03.

* A contrast to the large body of work in OS and architecture.

Related Work

First *systematic* examin. of the influence of cache sharing in modern CMP on the perf. of *contemporary multithreaded* apps.

- Co-runs of independent programs
  - Snavely+:00, Snavely+:02, El-Moursy+:06, Fedorova+:07, Jiang+:08, Zhou +:09, Tian+:09
- Co-runs of parallel threads of multithreaded programs
  - Liao+:05, Tuck+:03, Tam+:07
  - Have been focused on certain aspects of CMP
    - Simulators-based for cache design
    - Old benchmarks (e.g. SPLASH-2)
    - Specific class of apps (e.g., server apps)
    - Old CMP with no shared cache

Summary

Measurement

Insignificant influence from cache sharing despite inputs, arch, # threads, thread placement, parallelism, phases, etc.

Analysis

Mismatch between SW & HW causing the observations.

Transformation

Large potential of cache-share-aware code optimizations.

Conclusion

Does cache sharing on CMP matter to contemporary multithreaded programs?

Yes. But the main effects show up only after cache-sharing-aware transformations.

Thanks!

Questions?