An Asymmetric Multi-core Architecture for Accelerating Critical Sections

M. Aater Suleman

Advisor: Yale Patt

HPS Research Group
The University of Texas at Austin

Acknowledgements

Moinuddin Qureshi (IBM Research, HPS)

Onur Mutlu (Microsoft Research, HPS)

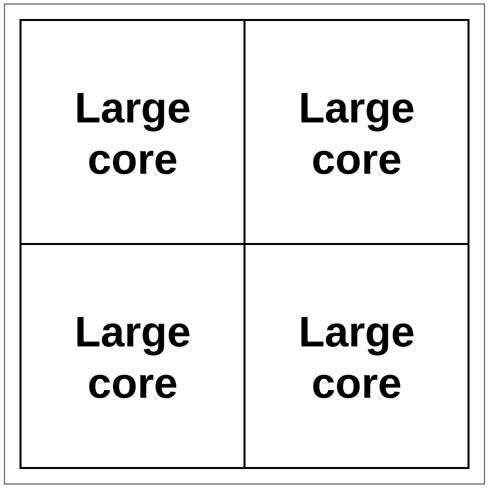
Eric Sprangle (Intel, HPS)

Anwar Rohillah (Intel)

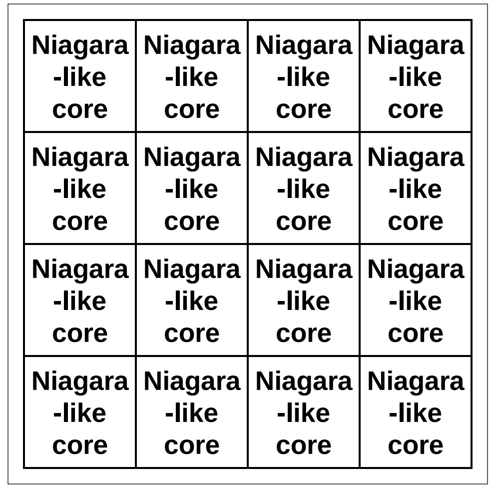
Anwar Ghuloum (Intel)

Doug Carmean (Intel)

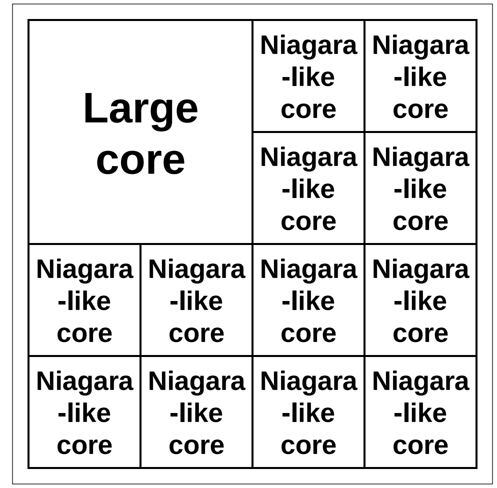
The Asymmetric Chip Multiprocessor (ACMP)



“Tile-Large” Approach



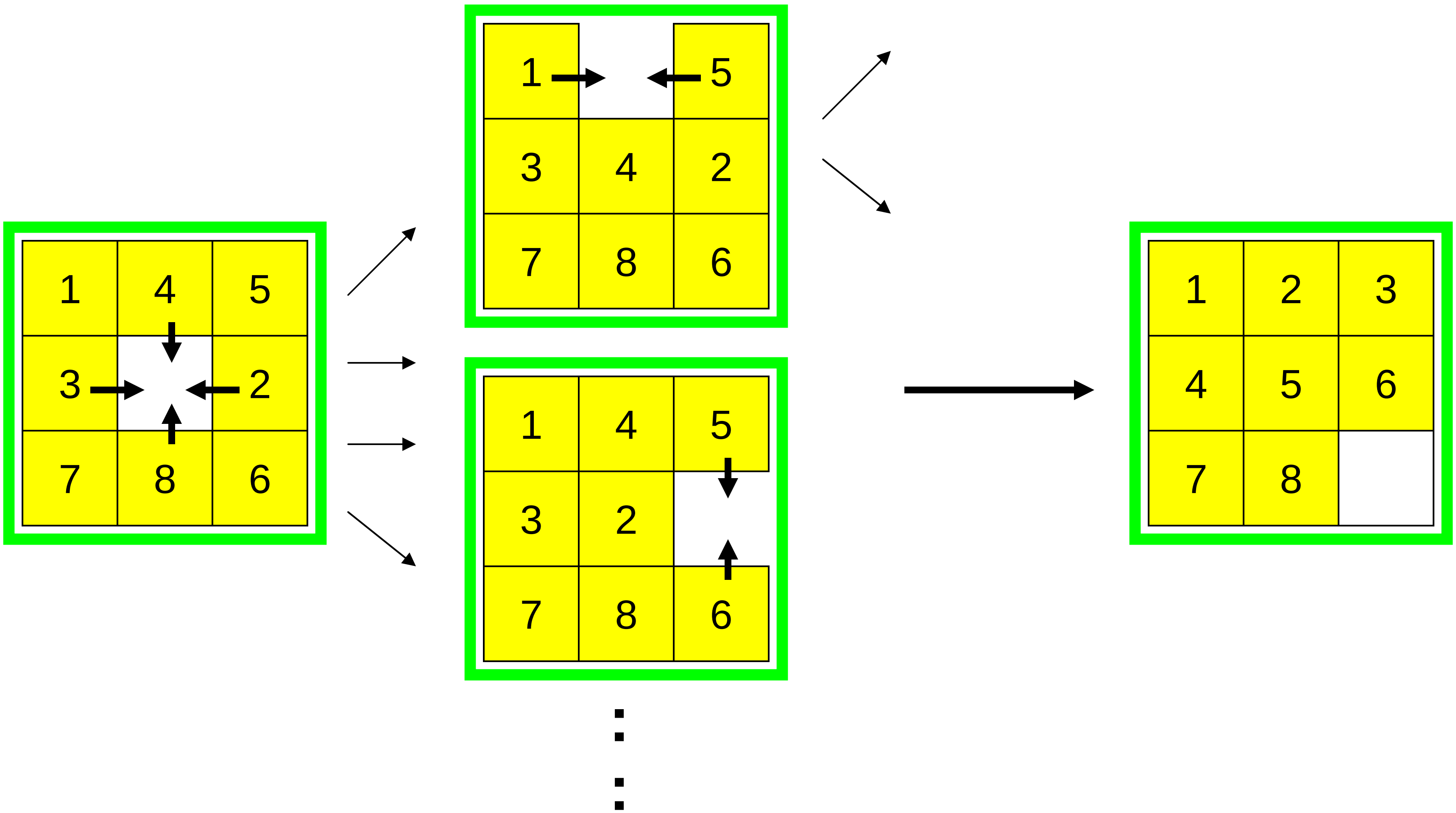
“Niagara” Approach



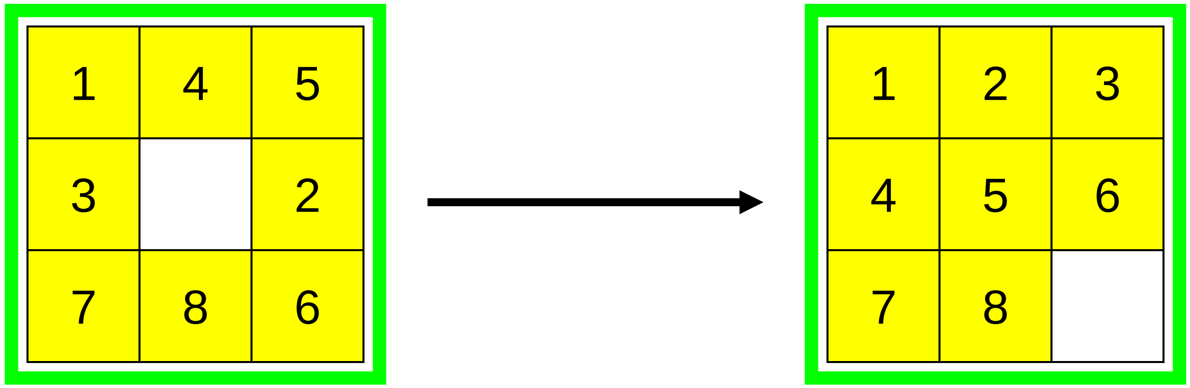
ACMP Approach

- Provide one large core and many small cores
- Accelerate serial part using the large core
- Execute parallel part on small cores for high throughput

The 8-Puzzle Problem



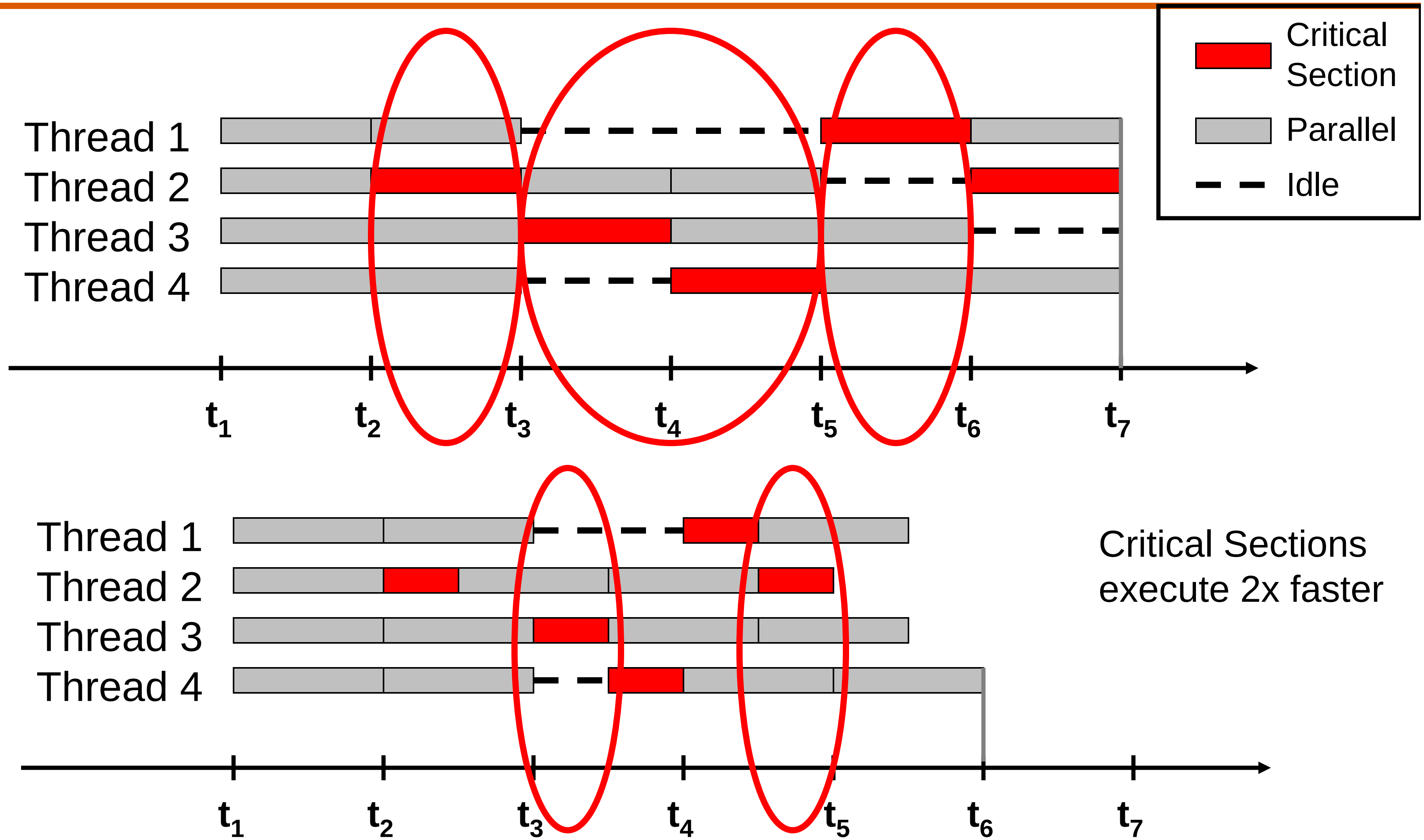
The 8-Puzzle Problem




```
while(problem not solved)

        SubProblem = PriorityQ.remove()

        Solve(SubProblem)
        if(solved)
                break
        NewSubProblems = Partition(SubProblem)

        PriorityQ.insert(NewSubProblems)
```

Critical Sections

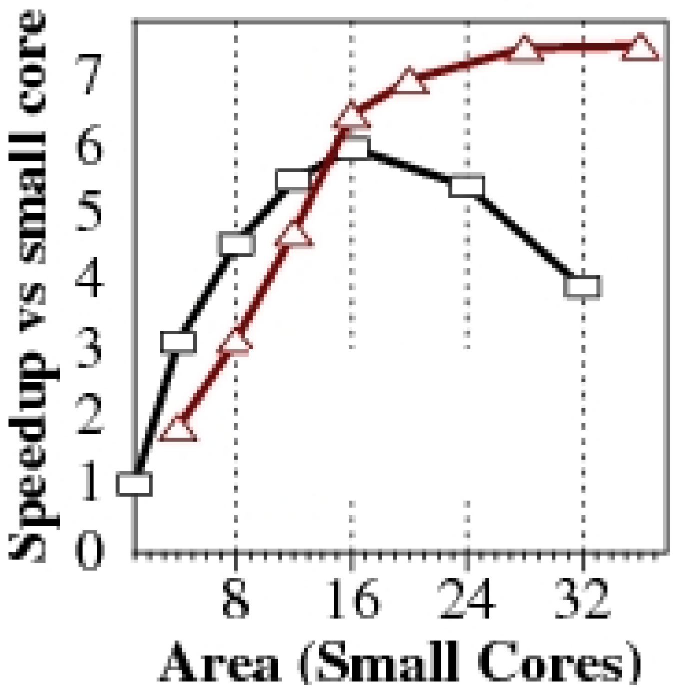
Contention for Critical Sections



Critical Sections execute 2x faster

MySQL Database

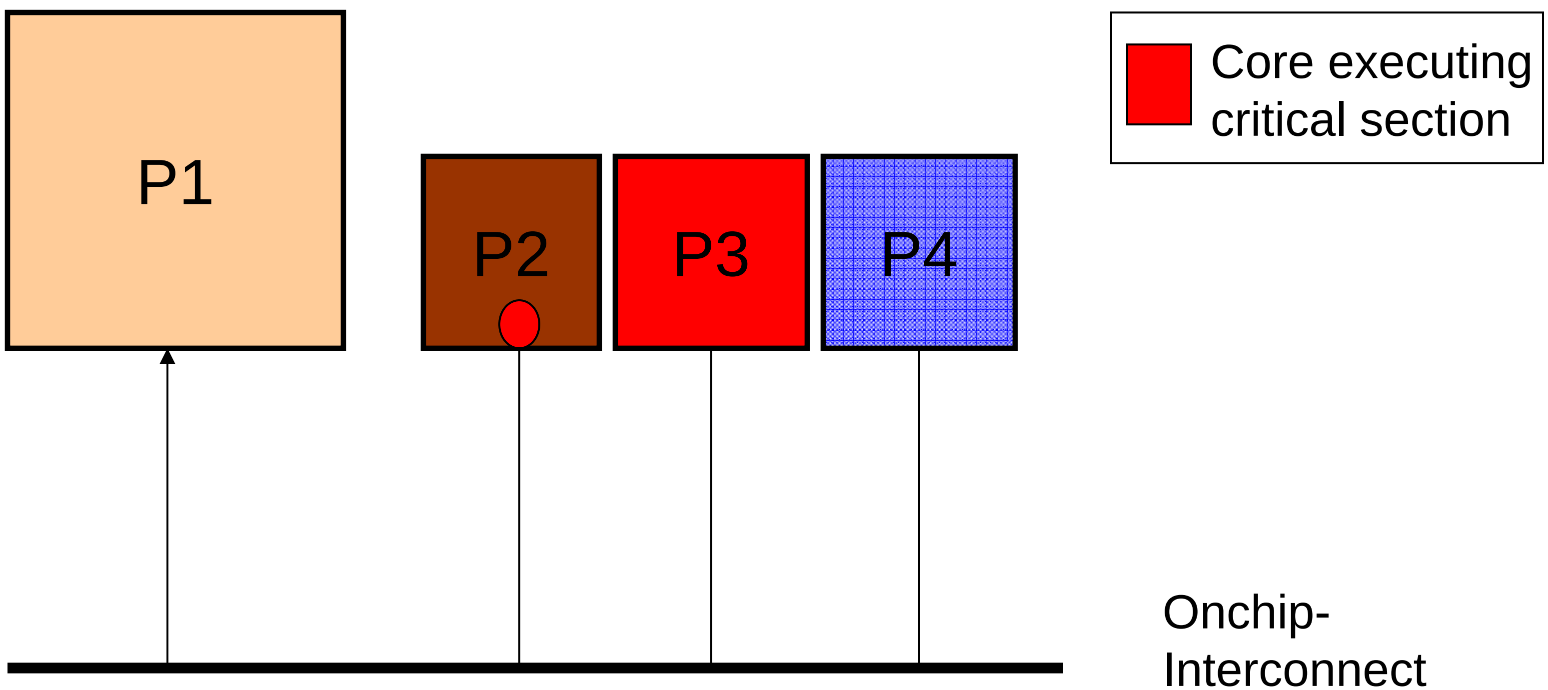
```
LOCK_open→Acquire()
foreach (table locked by thread)
        table.lock→release()
        table.file→release()
        if (table.temporary)
                table.close()
LOCK_open→Release()
```



Conventional ACMP

```
EnterCS()
        PriorityQ.insert(…)
LeaveCS()
```

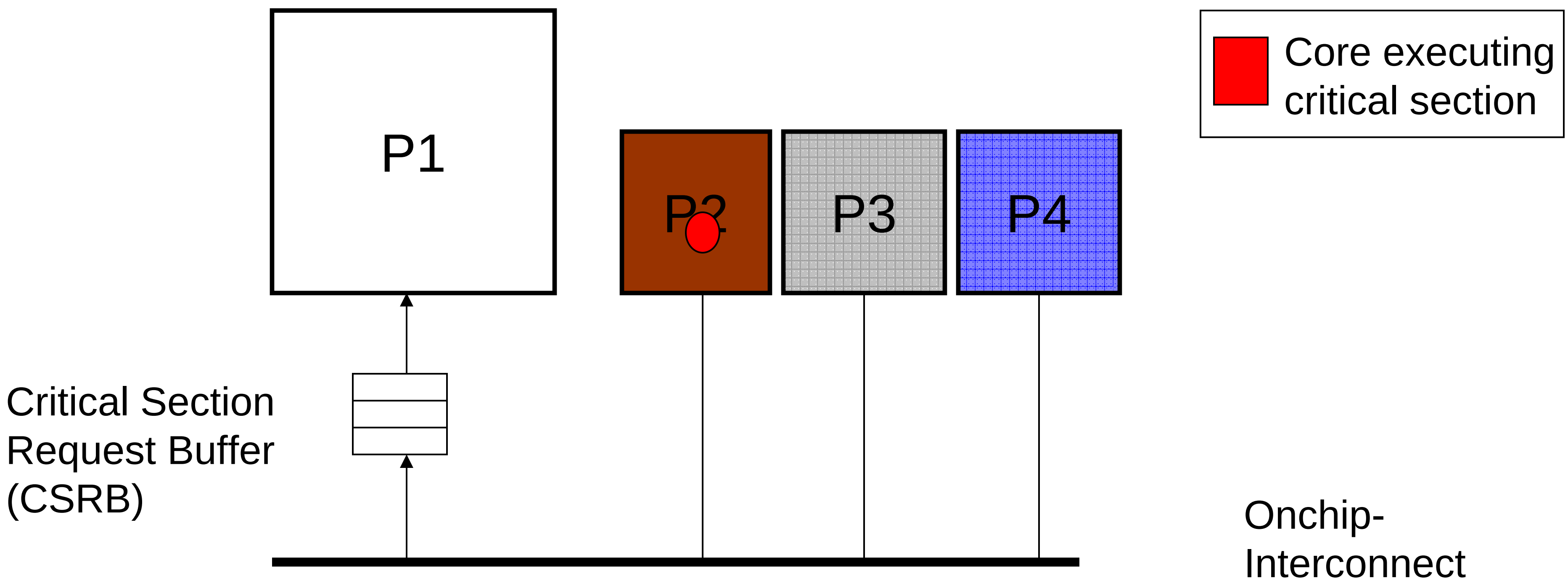
1. **P2 encounters a Critical Section**
2. **Sends a request for the lock**
3. **Acquires the lock**
4. **Executes Critical Section**
5. **Releases the lock**



Accelerated Critical Sections (ACS)

```
EnterCS()
    PriorityQ.insert(…)
LeaveCS()
```

1. P2 encounters a Critical Section
2. P2 sends CSCALL Request to CSRB
3. P1 executes Critical Section
4. P1 sends CSDONE signal



Architecture Overview

- ISA extensions
  - CSCALL *LOCK_ADDR*, *TARGET_PC*
  - CSRET *LOCK_ADDR*

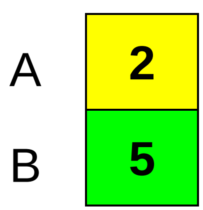
- Compiler/Library inserts CSCALL/CSRET

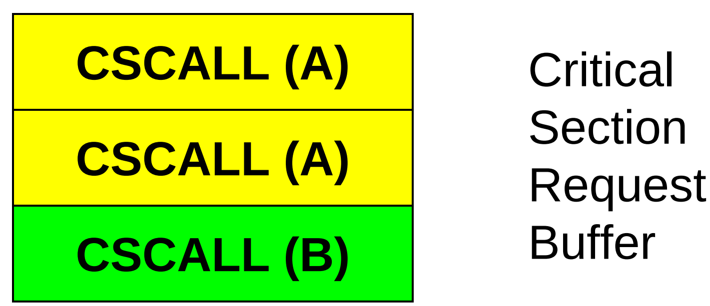
- On a CSCALL, the small core:
  - Sends a CSCALL request to the large core
    - Arguments: Lock address, Target PC, Stack Pointer, Core ID
  - Stalls and waits for CSDONE

- Large Core
  - Critical Section Request Buffer (CSRB)
  - Executes the critical section and sends CSDONE to the requesting core

“False” Serialization

- Independent critical sections are used to protect disjoint data
- Conventional systems can execute independent critical sections concurrently but ACS can artificially serializes their execution
- Selective Acceleration of Critical Sections (SEL)
  - Augment CSRB with saturating counters which track false serialization





Performance Trade-offs in ACS

- Fewer concurrent threads
  - As number of cores increase
    - Marginal loss in parallel performance decreases
    - More threads → Contention for critical sections increases which makes their acceleration more beneficial

- Overhead of CSCALL/CSDONE
  - Fewer cache misses for the lock variable

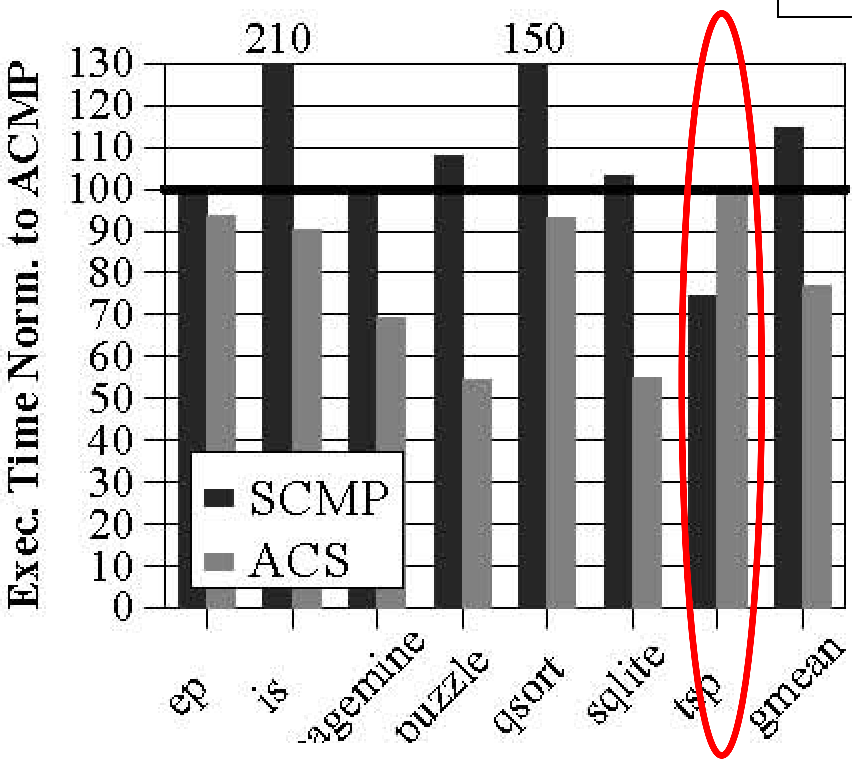
- Cache misses for private data
  - Fewer misses for shared data
    Cache misses reduce if Shared data > Private data
  - The large core can tolerate cache miss latencies better than small cores

Experimental Methodology

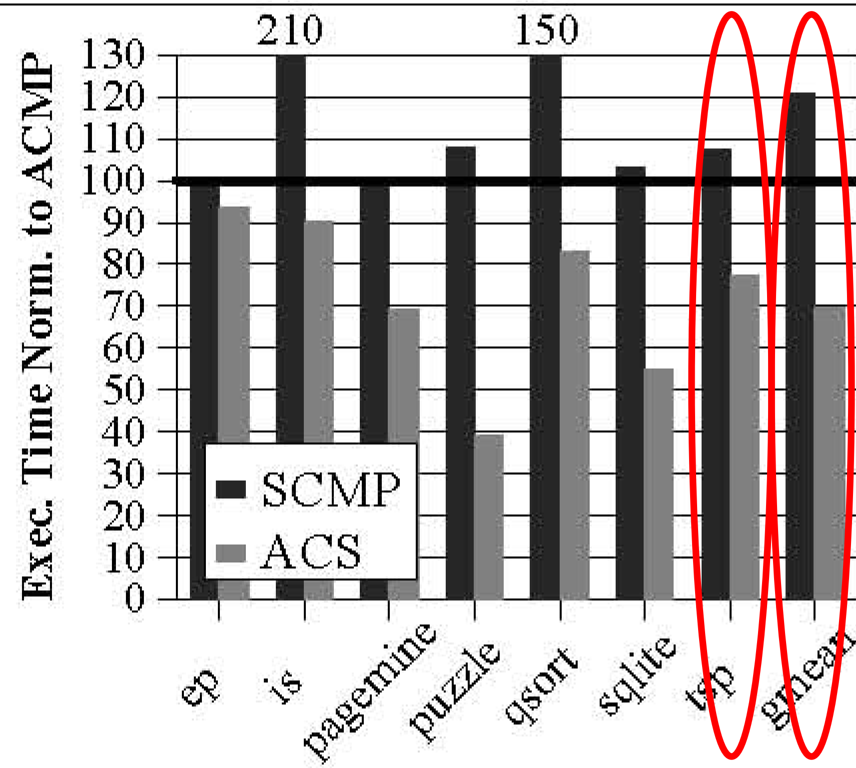
- Configurations
  - One large core is the size of 4 small cores
  - At chip area equal to N small cores
    - Symmetric CMP (SCMP): N small cores, conventional locking
    - Asymmetric CMP (ACMP): 1 large core, N – 4 small cores, conventional locking
    - ACS: 1 large core, N – 4 small cores, (N – 4)-entry CSRB.
- Workloads
  - 12 critical section intensive applications from various domains
  - 7 use coarse-grain locks and 5 use fine-grain locks
- Simulation parameters:
  - x86 cycle accurate processor simulator
  - Large core: Similar to Pentium-M with 2-way SMT. 2GHz, out-of-order, 128-entry, 4-wide, 12-stage
  - Small core: Similar to Pentium 1, 2GHz, in-order, 2-wide, 5-stage
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring

Workloads with Coarse-Grain Locks

Equal-area comparison
Number of threads = *Best threads*

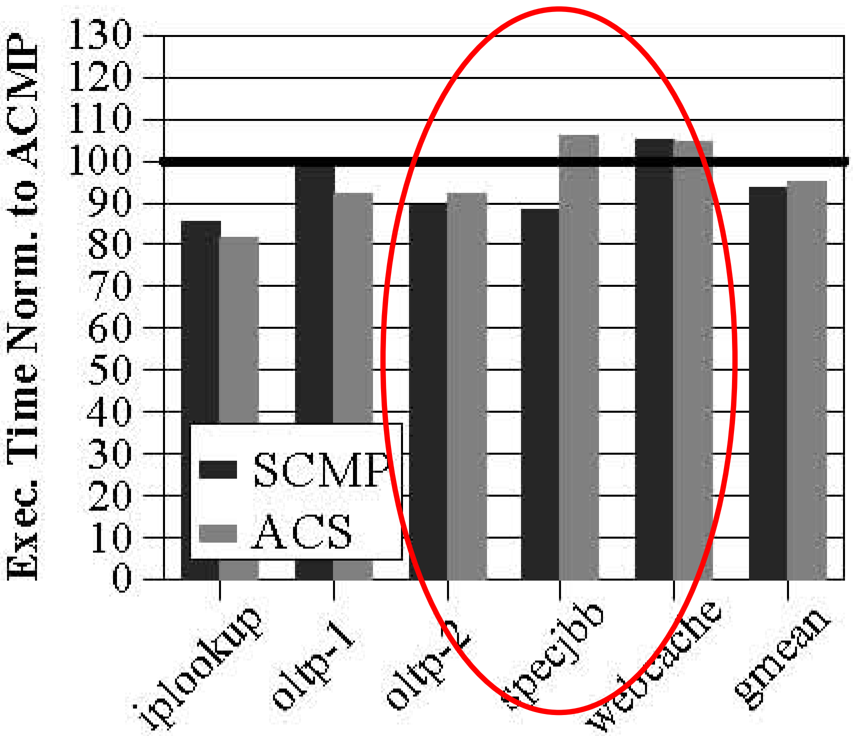


Chip Area = 16 cores
SCMP = 16 small cores
ACMP/ACS = 1 large and 12 small cores

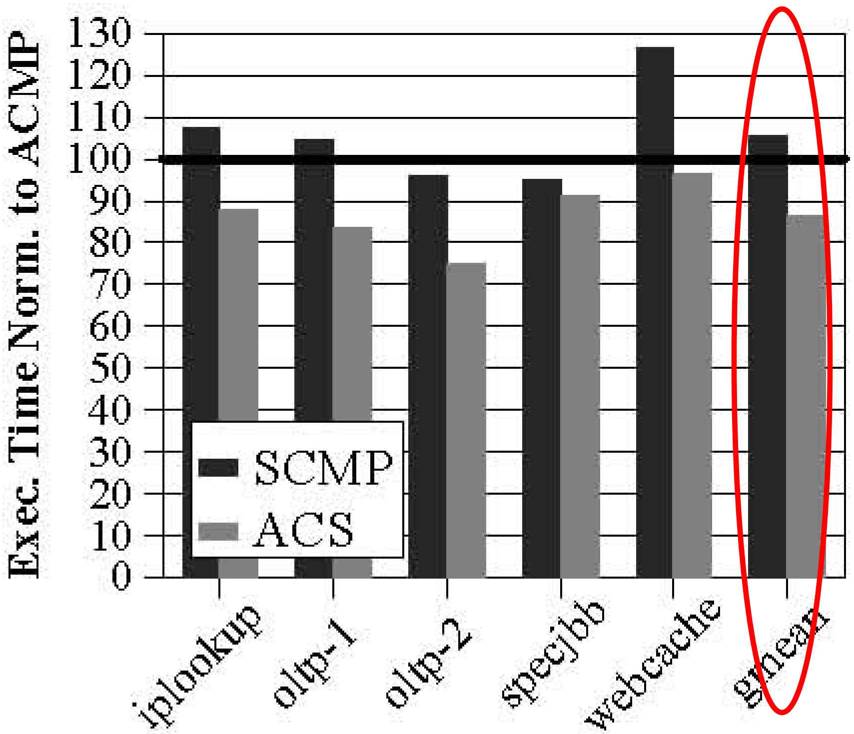


Chip Area = 32 small cores
SCMP = 32 small cores
ACMP/ACS = 1 large and 28 small cores

Workloads with Fine-Grain Locks



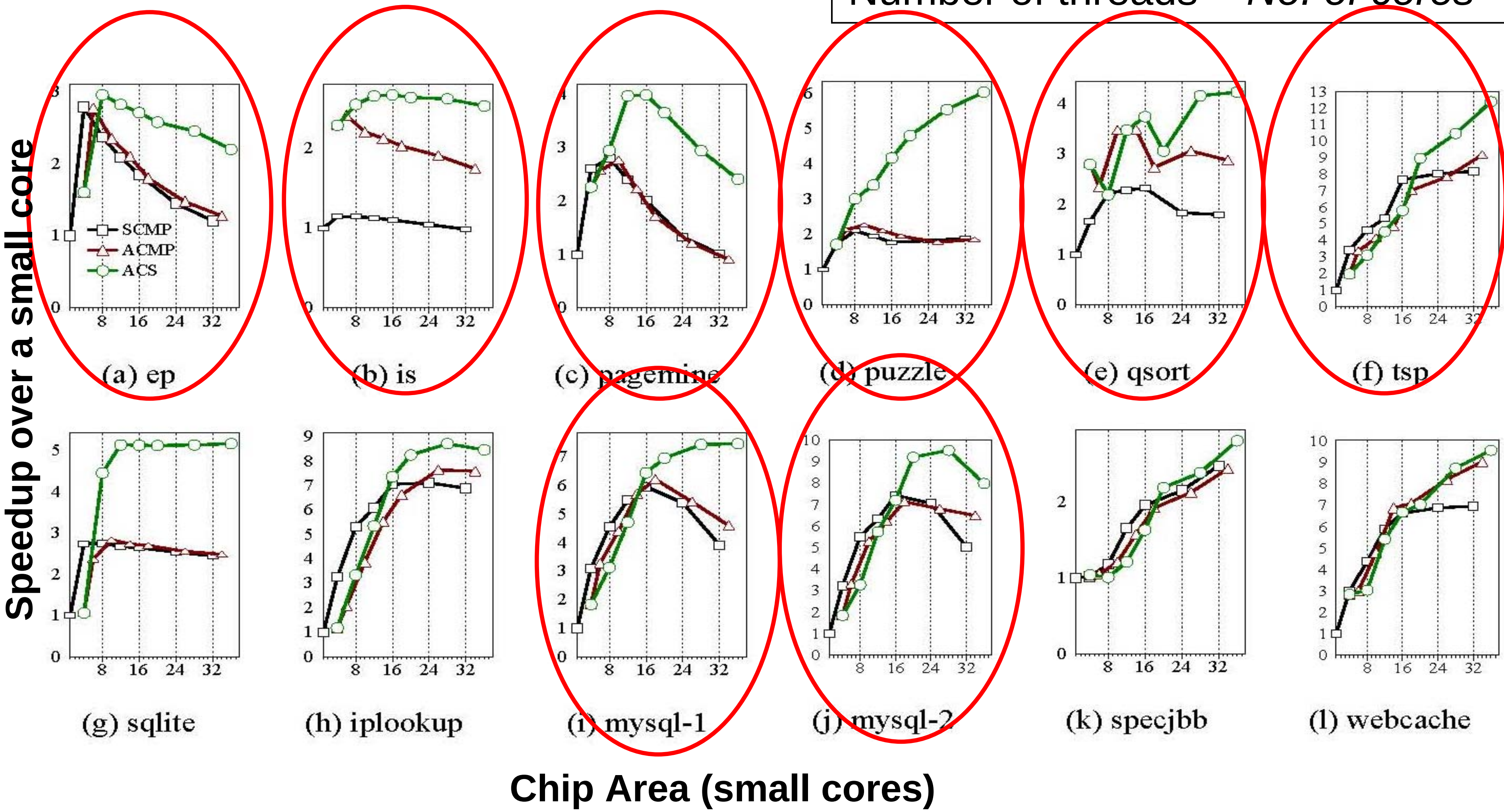
Area = 16 small cores



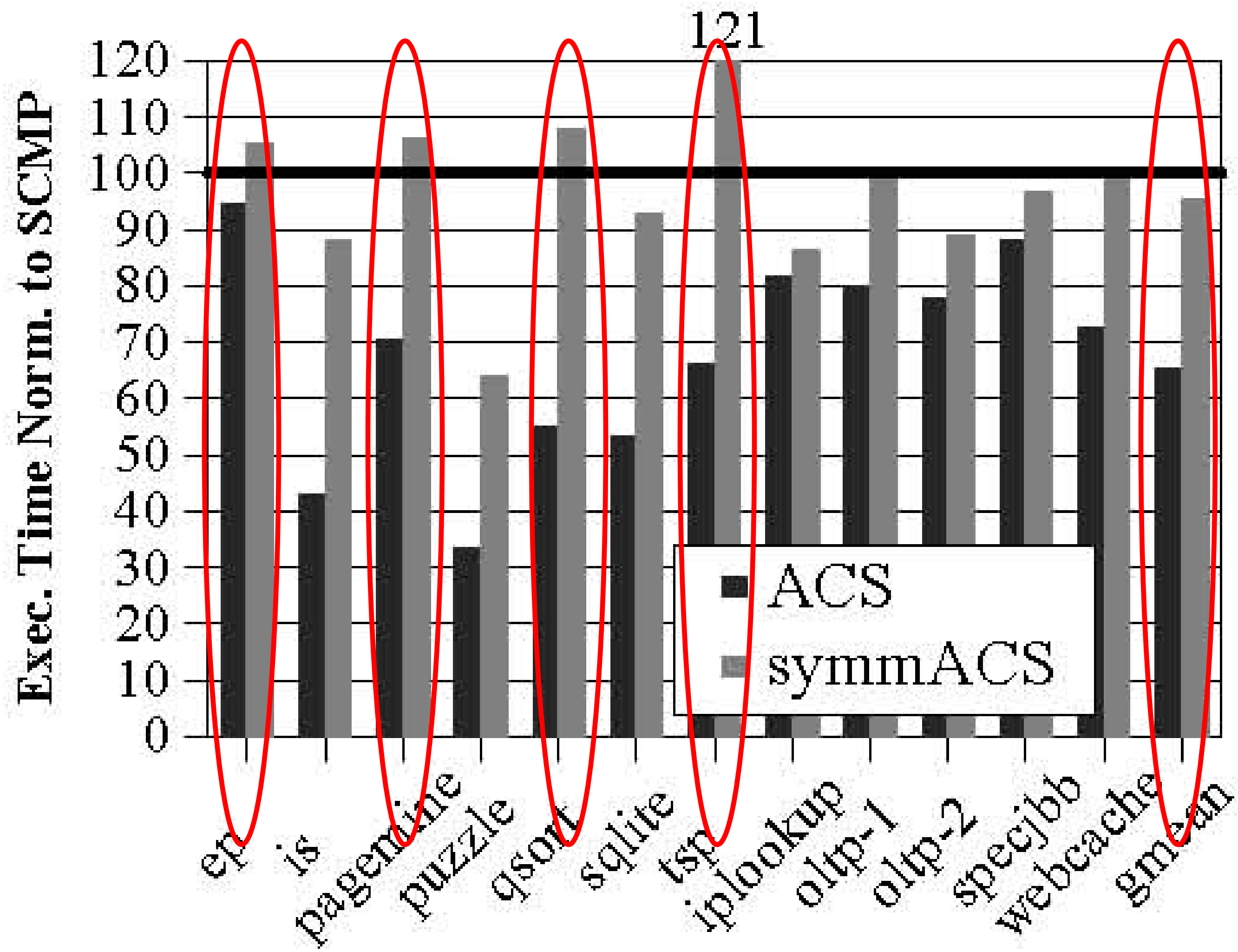
Area = 32 small cores

Equal-Area Comparisons

Number of threads = *No. of cores*



ACS on Symmetric CMP



Conclusion

- ACS reduces average execution time by:
  - 34% compared to an equal-area SCMP
  - 23% compared to an equal-area ACMP
- ACS improves scalability of 7 of the 12 workloads
- Future work will examine resource allocation in ACS in presence of multiple applications