Advanced Computer Architecture

Synchronization

Fall 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi and Wenisch of CMU, EPFL, Michigan, Wisconsin

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

- This Lecture
  - Synchronization
- Next Lecture:
  - TM

Lock Elision

Slides are courtesy Dan Sorin of Duke University

Speculative Lock Elision

- Multithreaded Programming gains Importance
  - SMPs, Multithreaded Architectures
  - Used in non-traditional fields (Desktop)
- Synchronization Mechanisms required for exclusive access
  - Serialize access to shared data structures
  - Necessary to prevent conflicting updates

Dynamically Unnecessary Synchronization

```
LOCK(locks->error_lock)
if (local_error>multi-
  >err_multi)
  multi-
  >err_multi=local_err;
UNLOCK(locks-
  >error_lock);
```

- Require no lock, if
  - no write access
  - different fields accessed

```
Thread 1
LOCK(hash_tbl.lock)
var=hash_tbl.lookup(X)
if(!var)
  hash_tbl.add(X);
UNLOCK(hash_tbl.lock)

Thread 2
LOCK(hash_tbl.lock)
var=hash_tbl.lookup(Y)
if(!var)
  hash_tbl.add(Y);
UNLOCK(hash_tbl.lock)
```

Multithreaded Programming

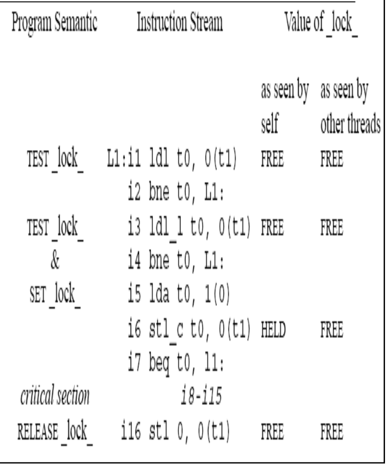
- Conservative Locking
  - Easier to show correctness
  - Lock more often than necessary
- Locking Granularity
  - Trade-Off between Performance and Complexity
- Thread-unsafe legacy libraries
  - Require global locking

False Dependencies

- Locks introduce Control and Data Dependence
- Removal
  - Key is *Appearance of Instantaneous Change*
  - Lock can be elided, if memory operation remain atomic
    - ▲Data read is not modified by other threads
    - ▲Data written is not access by other threads
  - Any instruction that violates these conditions must not be retired

Silent store pairs

- i16 & i6 are silent pair
  - i16 *undoes* i6
- Not silent individually
  - No write to _lock_
  - Read _lock_ is ok
- SLE does not depend on semantic info
  - Simply observe silent pairs



Program Semantic	Instruction Stream	Value of _lock_	
		as seen by self	as seen by other threads
TEST _lock_	L1:i1 ldl t0, 0(t1)	FREE	FREE
	i2 bne t0, L1:		
TEST _lock_	i3 ldl_l t0, 0(t1)	FREE	FREE
&	i4 bne t0, L1:		
SET _lock_	i5 lda t0, 1(0)		
	i6 stl_c t0, 0(t1)	HELD	FREE
	i7 beq t0, l1:		
critical section	*i8-i15*		
RELEASE _lock_	i16 stl 0, 0(t1)	FREE	FREE

Two SLE Predictions

- Silent pair prediction
  - On a store
    - Predict another store will undo changes
    - Don't perform stores
    - Monitor memory location
  - If validated, elide two stores
- Atomicity prediction
  - Predict all memory access within silent pair occur atomically
  - No partial updates visible to other threads

Initiating Speculation

- Detect candidate pairs
  - Filter indexed by program counter
  - Add confidence metric for each pair
- Predict, if lock held
  - If yes, don't speculate

Buffering Speculative State

- Register state
  - Reorder Buffer
    - Already used for branch prediction
  - Register Checkpoint
- Memory state
  - Augment write buffers
    - Keep speculative writes in write buffer…
    - …until validated
    - Allows merging of speculative writes

Misspeculation conditions

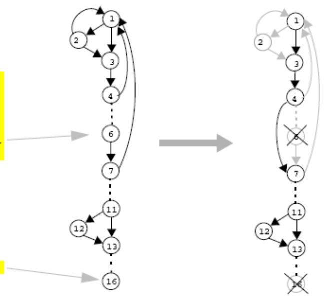
- Atomicity violation
  - Detected by coherence protocol
    - Sufficient for ROB register state
    - For register checkpoint add speculative access bit to L1 cache
- Resource constraints
  - Cache Size
  - Write Buffer Size

Committing Speculative Memory State

- ◆ Commits must appear instantaneously
- ◆ Cache state and Cache data
  - ● Can speculate on state
  - ● Can’t speculate on data
- ◆ Speculative store
  - ● Send GETX request
  - ● Block already exclusive when speculative writes commit

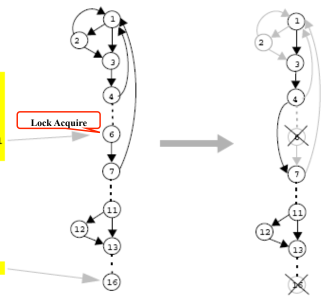
Example (from Lang Yiu)

```
        LOCK(locks->error_lock)
        if (local_err > multi->err_multi)
        multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

L1:1.ldl t0, 0(t1)           #t0 = lock
   2.bne t0, L1:             #if not free, goto L1
   3.ldl_l t0, 0(t1)         #load locked, t0 = lock
   4.bne t0, L1:             #if not free, goto L1
   5.lda t0, 1(0)            #t0 = 1
   6.stl_c t0, 0(t1)         #conditional store, lock = 1
   7.beq t0, L1:             #if stl_c failed, goto L1,
   8.ldq t0, 0(s4)
   9.ldt $f10, 0(t0)
  10.cmplt $f10,$f11,$f10
  11.fbeq $f10, L2:
  12.stt $f11, 0(t0)
  13.ldq t1,-31744(gp)
  14.ldq t0, 0(t1)
  15.ldq t1, 32(t0)
  16.stl 0 ,0(t1)            #lock = 0, release lock
```



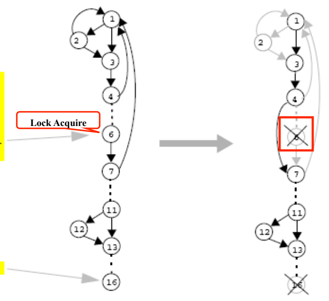
How to elide locks? - Example

```
        LOCK(locks->error_lock)
        if (local_err > multi->err_multi)
        multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

L1:1.ldl t0, 0(t1)           #t0 = lock
   2.bne t0, L1:             #if not free, goto L1
   3.ldl_l t0, 0(t1)         #load locked, t0 = lock
   4.bne t0, L1:             #if not free, goto L1
   5.lda t0, 1(0)            #t0 = 1
   6.stl_c t0, 0(t1)         #conditional store, lock = 1
   7.beq t0, L1:             #if stl_c failed, goto L1,
   8.ldq t0, 0(s4)
   9.ldt $f10, 0(t0)
  10.cmplt $f10,$f11,$f10
  11.fbeq $f10, L2:
  12.stt $f11, 0(t0)
  13.ldq t1,-31744(gp)
  14.ldq t0, 0(t1)
  15.ldq t1, 32(t0)
  16.stl 0 ,0(t1)            #lock = 0, release lock
```



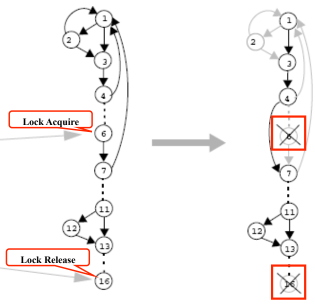
How to elide locks? - Example

```
        LOCK(locks->error_lock)
        if (local_err > multi->err_multi)
        multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

L1:1.ldl t0, 0(t1)           #t0 = lock
   2.bne t0, L1:             #if not free, goto L1
   3.ldl_l t0, 0(t1)         #load locked, t0 = lock
   4.bne t0, L1:             #if not free, goto L1
   5.lda t0, 1(0)            #t0 = 1
   6.stl_c t0, 0(t1)         #conditional store, lock = 1
   7.beq t0, L1:             #if stl_c failed, goto L1,
   8.ldq t0, 0(s4)
   9.ldt $f10, 0(t0)
  10.cmplt $f10,$f11,$f10
  11.fbeq $f10, L2:
  12.stt $f11, 0(t0)
  13.ldq t1,-31744(gp)
  14.ldq t0, 0(t1)
  15.ldq t1, 32(t0)
  16.stl 0 ,0(t1)            #lock = 0, release lock
```



How to elide locks? - Example

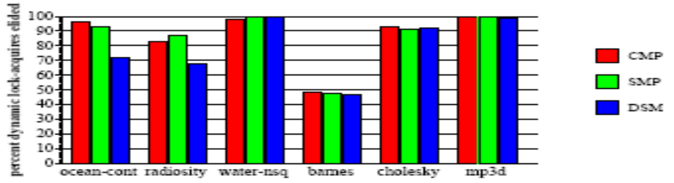
```
        LOCK(locks->error_lock)
        if (local_err > multi->err_multi)
        multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

L1:1.ldl t0, 0(t1)         #t0 = lock
   2.bne t0, L1:           #if not free, goto L1
   3.ldl_l t0, 0(t1)       #load locked, t0 = lock
   4.bne t0, L1:           #if not free, goto L1
   5.lda t0, 1(0)          #t0 = 1
   6.stl_c t0, 0(t1)       #conditional store, lock = 1
   7.beq t0, L1:           #if stl_c failed, goto L1,
   8.ldq t0, 0(s4)
   9.ldt $f10, 0(t0)
  10.cmplt $f10,$f11,$f10
  11.fbeq $f10, L2:
  12.stt $f11, 0(t0)
  13.ldq t1,-31744(gp)
  14.ldq t0, 0(t1)
  15.ldq t1, 32(t0)
  16.stl 0 ,0(t1)          #lock = 0, release lock
```

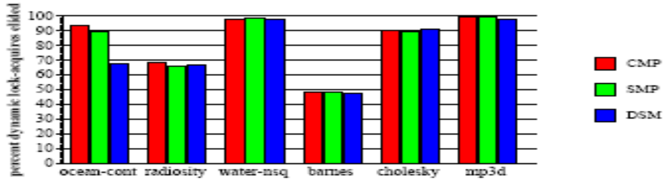


Evaluation

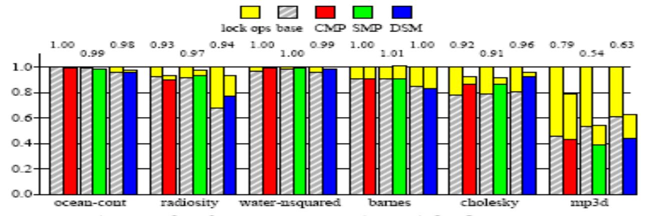
- 3 Systems
  - CMP, SMP and DSM
  - Total Store Order
  - Single Register Checkpoint
  - 32 entry lock predictor
- Run on SimpleMP
  - Based on Simplescalar
- Benchmarks
  - SPLASH
    - mp3d, barnes, cholesky
  - SPLASH-2
    - Radiosity, water-nsq, ocean
  - Microbenchmark

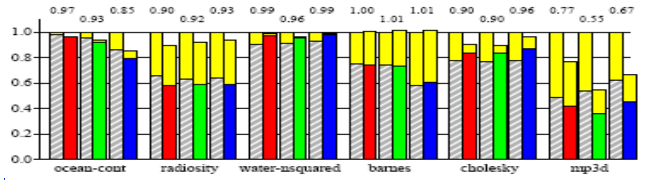
Lock acquires/releases elided


a) Percentage for 8 processors


b) Percentage for 16 processors

SPLASH Performance


a) normalized execution time (y axis) for 8 processors


b) normalized execution time (y axis) for 16 processors

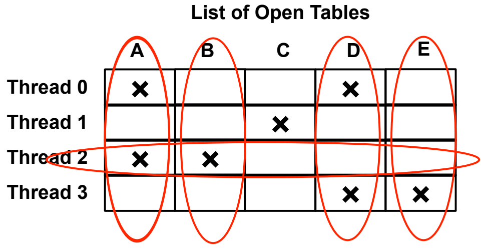
Asymmetric Critical Section Execution

Slides from M. Aater Suleman et. al., ASPLOS 2009

Background

- To leverage CMPs:
  - Programs must be split into *threads*
- Mutual Exclusion:
  - Threads are not allowed to update shared data concurrently
- Accesses to shared data are encapsulated inside ***critical sections***
- Only one thread can execute a critical section at a given time

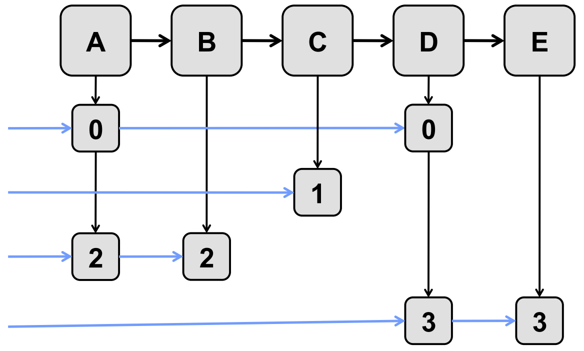
Example of Critical Section from MySQL



	A	B	C	D	E
Thread 0	✗			✗	
Thread 1			✗		
Thread 2	✗	✗			
Thread 3				✗	✗

Thread 2:
CloseAllTables()

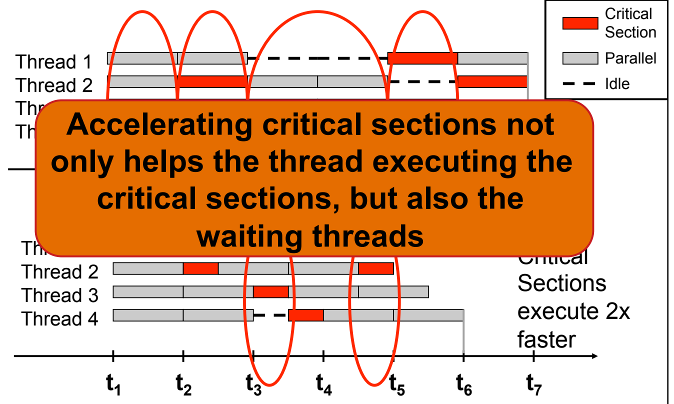
Example Critical Section from MySQL



Example Critical Section from MySQL

```
LOCK_open→Acquire()

  End of Transaction:
    foreach (table opened by thread)
        if (table.temporary)
            table.close()

LOCK_open→Release()
```

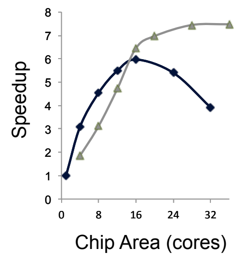
Contention for Critical Sections



Accelerating critical sections not only helps the thread executing the critical sections, but also the waiting threads

Impact of Critical Sections on Scalability

- Contention for critical sections increases with the number of threads and limits scalability

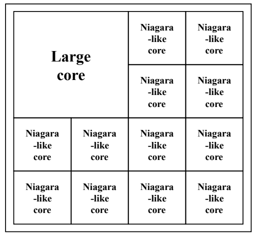


MySQL (oltp-1)

Outline

- Background
- ***Mechanism***
- Performance Trade-Offs
- Evaluation
- Related Work and Summary

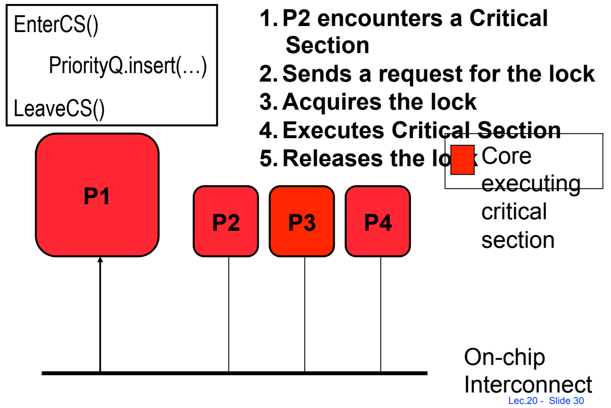
The Asymmetric Chip Multiprocessor (ACMP)



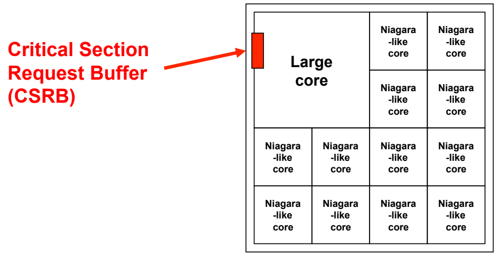
- Provide one large core and many small cores
- Execute parallel part on small cores for high throughput
- Accelerate serial part using the large core

Conventional ACMP

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



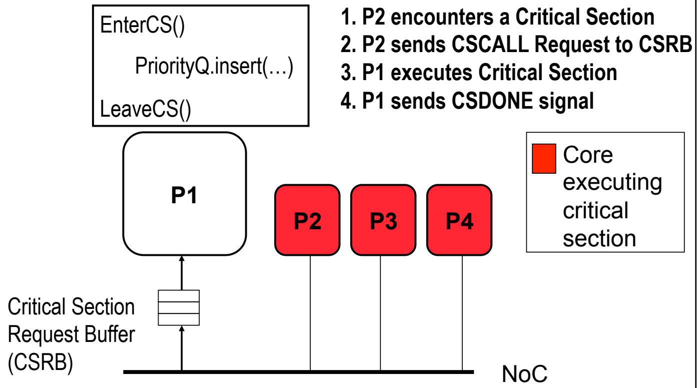
Accelerating Critical Sections (ACS)



Accelerate Amdahl's serial part & **critical sections** using the large core

Accelerated Critical Sections (ACS)

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

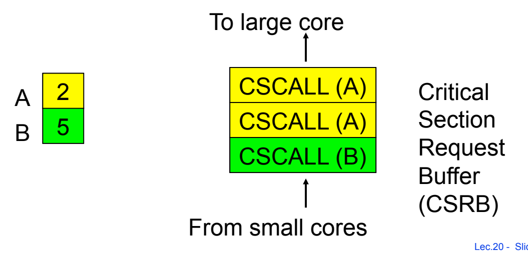

EnterCS()
PriorityQ.insert(…)
LeaveCS()
P1
P2
P3
P4
Core executing critical section
Critical Section Request Buffer (CSRB)
NoC

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

- ACS can serialize independent critical sections
- Selective Acceleration of Critical Sections (SEL)
  - Saturating counters to track false serialization



Outline

- Background
- Mechanism
- ***Performance Trade-Offs***
- Evaluation
- Related Work and Summary

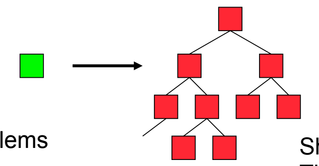
Performance Tradeoffs

- ***Fewer threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration more beneficial
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids "ping-ponging" of locks among caches by keeping them at the large core
- ***More cache misses for private data vs. fewer misses for shared data***

Cache misses for private data

PriorityHeap.insert(NewSubProblems)

Private Data:
NewSubProblems



Shared Data:
The priority heap

Puzzle Benchmark

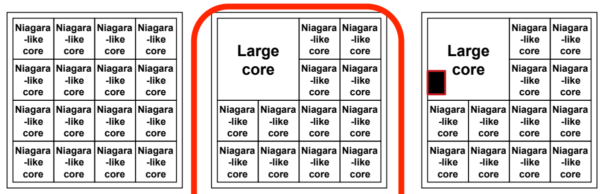
Performance Tradeoffs

- ***Fewer threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration more beneficial
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids “ping-ponging” of locks among caches by keeping them at the large core
- ***More cache misses for private data vs. fewer misses for shared data***
  - Cache misses reduce if shared data > private data

Outline

- Background
- Mechanism
- Performance Trade-Offs
- ***Evaluation***
- Related Work and Summary

Experimental Methodology



SCMP
- All small cores
- Conventional locking

ACMP
- One large core (area-equal 4 small cores)
- Conventional locking

ACS
- ACMP with a CSRB
- Accelerates Critical Sections

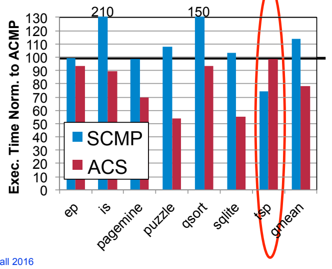
Experimental Methodology

- 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 ROB, 4-wide issue, 12-stage
  - Small core: Similar to Pentium 1, 2GHz, in-order, 2-wide issue, 5-stage
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring

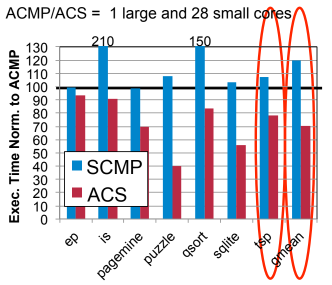
Workloads with Coarse-Grain Locks

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

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



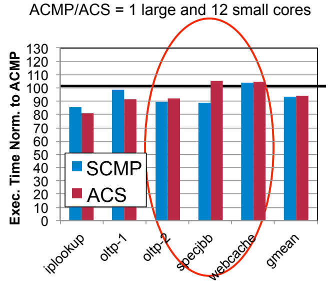
Chip Area = 32 small cores
SCMP = 32 small cores
ACMP/ACS = 1 large and 28 small cores



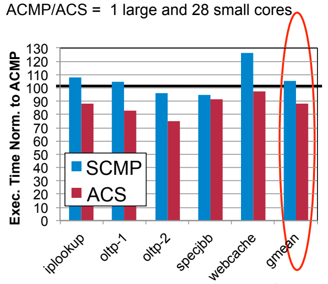
Workloads with Fine-Grain Locks

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

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

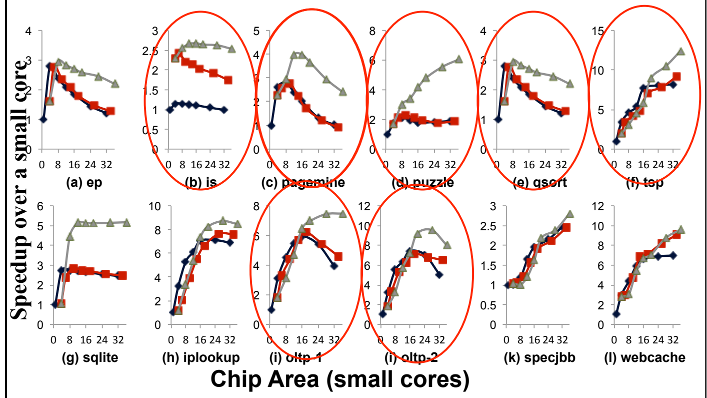


Chip Area = 32 small cores
SCMP = 32 small cores
ACMP/ACS = 1 large and 28 small cores

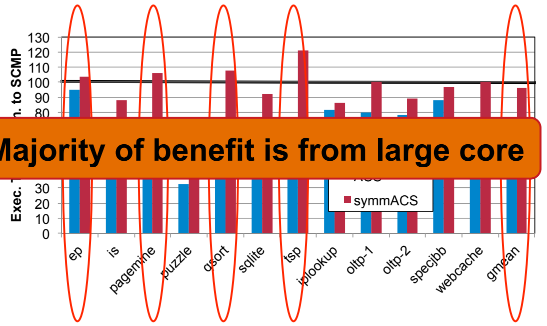


Equal-Area Comparisons

------ SCMP
------ ACMP
------ ACS



ACS on Symmetric CMP



Majority of benefit is from large core

Outline

- Background
- Mechanism
- Performance Trade-Offs
- Evaluation
- ***Related Work and Summary***

Related Work

- Improving locality of shared data by thread migration and software prefetching (Sridharan+, Trancoso+, Ranganathan+)
  ACS not only improves locality but also uses a large core to accelerate critical section execution
- Asymmetric CMPs (Morad+, Kumar+, Suleman+, Hill+)
  ACS not only accelerates the Amdahl's bottleneck but also critical sections
- Remote procedure calls (Birrell+)
  ACS is for critical sections among shared memory cores

Hiding Latency of Critical Sections

- Transactional memory (Herlihy+)
  ACS does not require code modification
- Transactional Lock Removal (Rajwar+) and Speculative Synchronization (Martinez+)
  - Hide critical section latency by increasing concurrency
    ACS reduces latency of each critical section
  - Overlaps execution of critical sections with no data conflicts
    ACS accelerates ALL critical sections
  - Does not improve locality of shared data
    ACS improves locality of shared data

$\rightarrow$ACS outperforms TLR (Rajwar+) by 18% (details in paper)

Conclusion

- Critical sections limit scalability
- Accelerate critical sections with a powerful core
- 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