Thread-level Parallelism for the Masses

Kunle Olukotun
Computer Systems Lab
Stanford University
Feb 2007

The World has Changed

- Process Technology Stops Improving
  - Moore's law but …
  - Transistors don't get faster and they leak more (65nm vs. 45nm)
  - Wires are much worse
- Single Thread Performance Plateau
  - Design and verification complexity is overwhelming
  - Power consumption increasing dramatically
  - Instruction-level parallelism (ILP) has been mined out

From Intel Developer Forum, September 2004

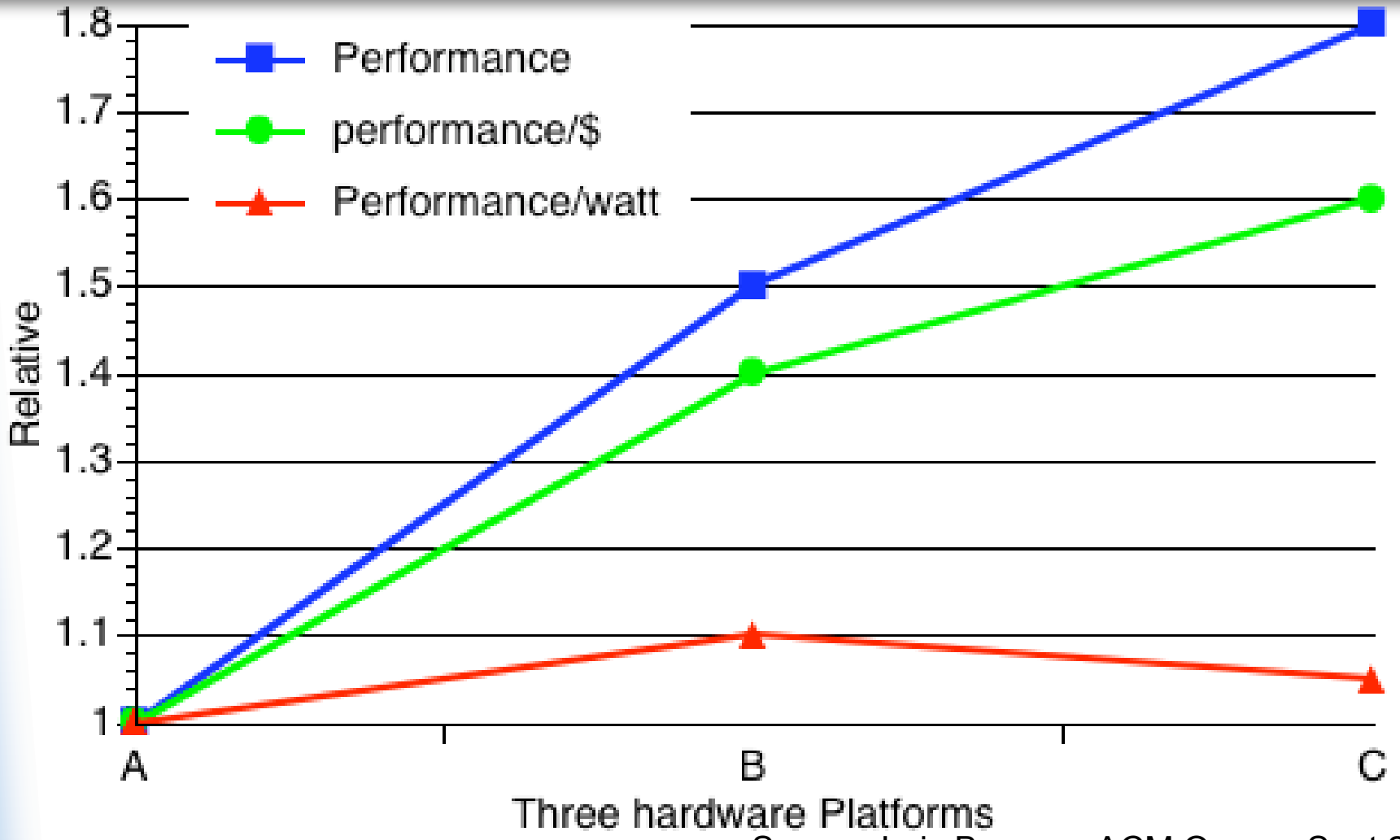
The Era of Single-Chip Multiprocessors

- Single-chip multiprocessors provide a scalable alternative
  - Relies on scalable forms of parallelism
    - Request level parallelism
    - Data level parallelism
  - Modular design with inherent fault-tolerance and match to VLSI technology
- Single-chip multiprocessors systems are here
  - All processor vendors are following this approach
  - In embedded, server, and even desktop systems
- How do we architect CMPs to best exploit thread-level parallelism?
  - Server applications: throughput
  - General purpose and scientific applications: latency

Outline

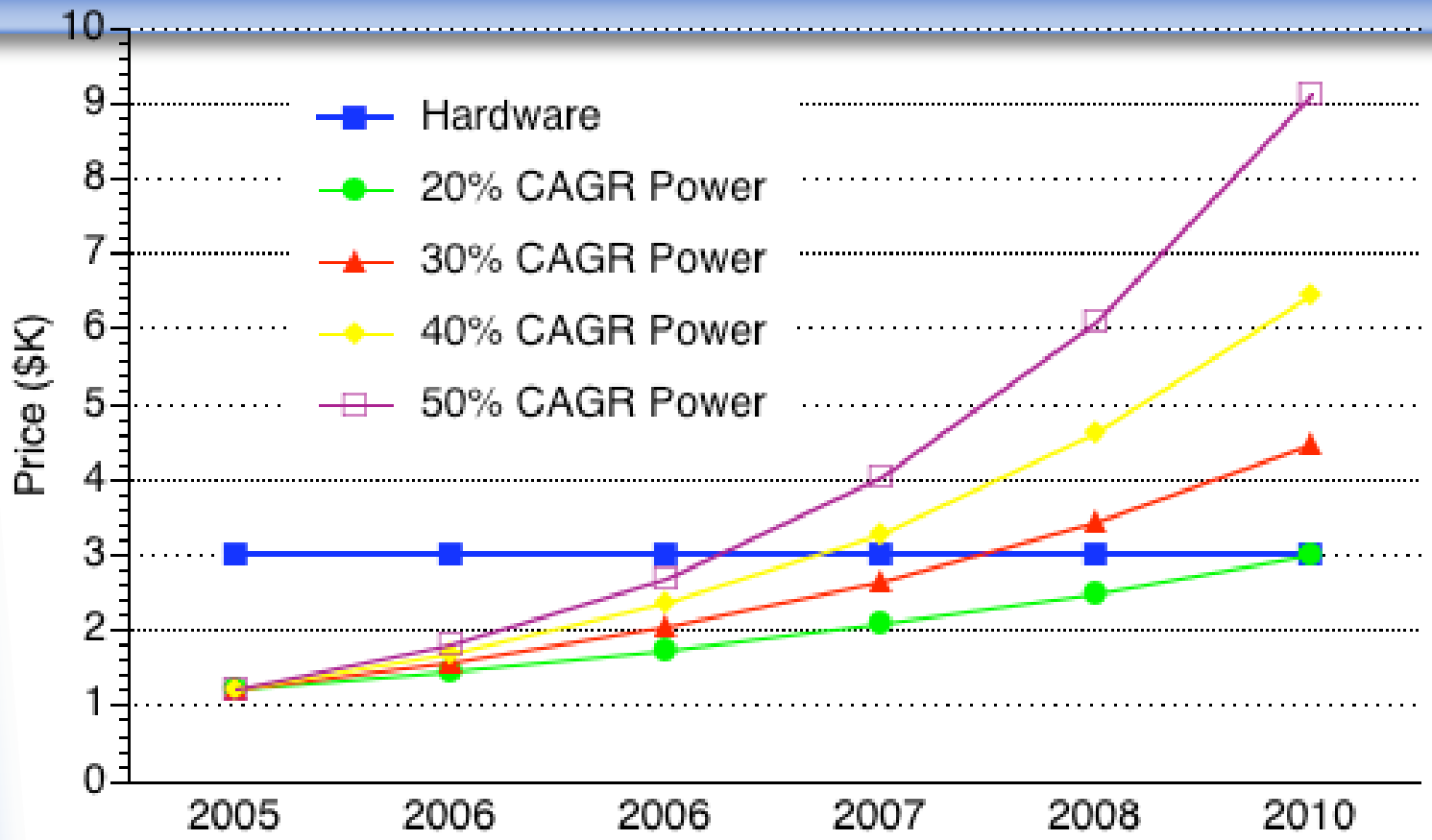
- Motivation: The era of chip multiprocessors
- Throughput and low power: Sun Niagara
- Latency: Stanford TCC

TLP for the Masses (Google)



Source: Luiz Barroso, ACM Queue, Sept 2005

Constant Performance/Watt



- TCO dominated by power costs
  - 4 year server life cycle @ $ 0.09 KWh
- We must improve performance/watt

Source: Luiz Barroso, ACM Queue, Sept 2005

Commercial Server Workloads

	Web99	JBB	TPC-C	TPC-H	SAP
Domain	Web server	Java App. server	OLTP	DSS	ERP
Instruction-level parallelism	low	low	low	high	low
Thread-level Parallelism	high	high	high	high	high
Working set	large	large	large	large	large
Data sharing	low	med	large	med	large

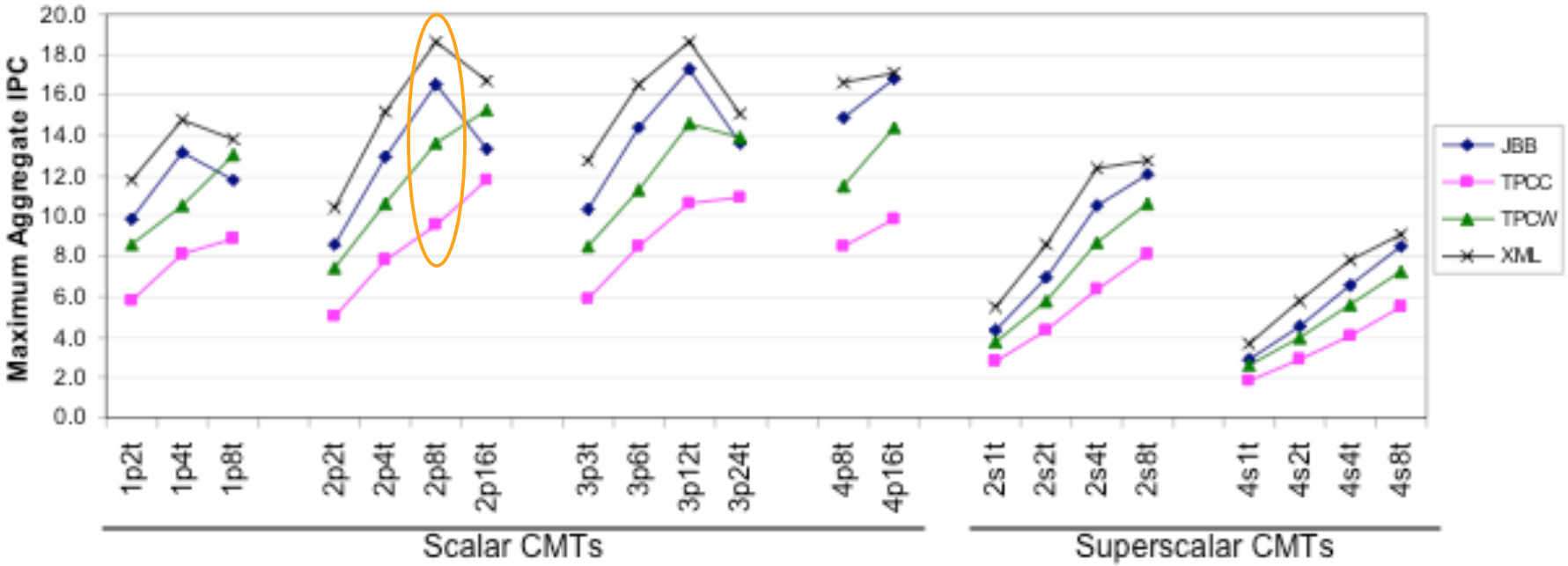
Server Throughput Computing Design

- Commercial server applications
  - Lots of tasks → multiple threads
  - Low ILP and high memory stall
- Best performance(throughput) achieved with multiple threads
  - Scalable form of parallelism
  - Tradeoff latency of single thread for throughput of multiple threads
  - Forgo single wide OOO CPU for multiple simple CPUs
  - Medium to large caches
  - Lots of memory bandwidth

Maximizing CMP Throughput with Simple Cores

- J. Davis, J. Laudon, K. Olukotun PACT '05 paper
- Examined several UltraSPARC II, III, IV, and Niagara designs, accounting for differing technologies
- Constructed an area model based on this exploration
- Assumed a fixed-area large die (400 mm^2), and accounted for pads, pins, and routing overhead
- Looked at performance for a broad swath of scalar and in-order superscalar processor core designs

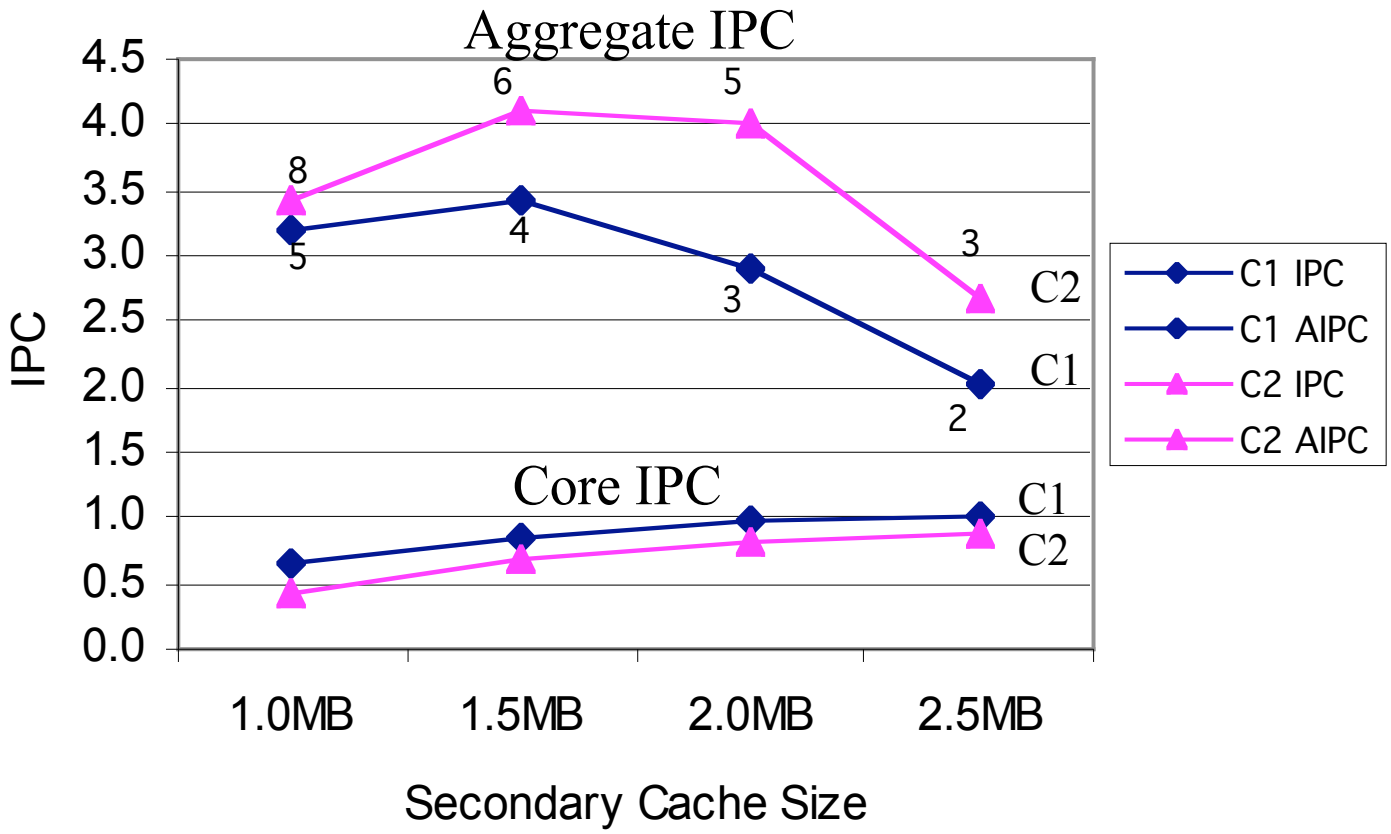
Simpler Cores Offer Higher Performance



- Optimize for Chip IPC
  - 400 mm^2 area architectures (4–20 cores)
  - L2 caches (1.5MB – 2.5MB)
  - Multiple pipes and multithreading is important
  - Scalar pipes are 37%–46% better than superscalar pipes (12 vs. 7 cores)

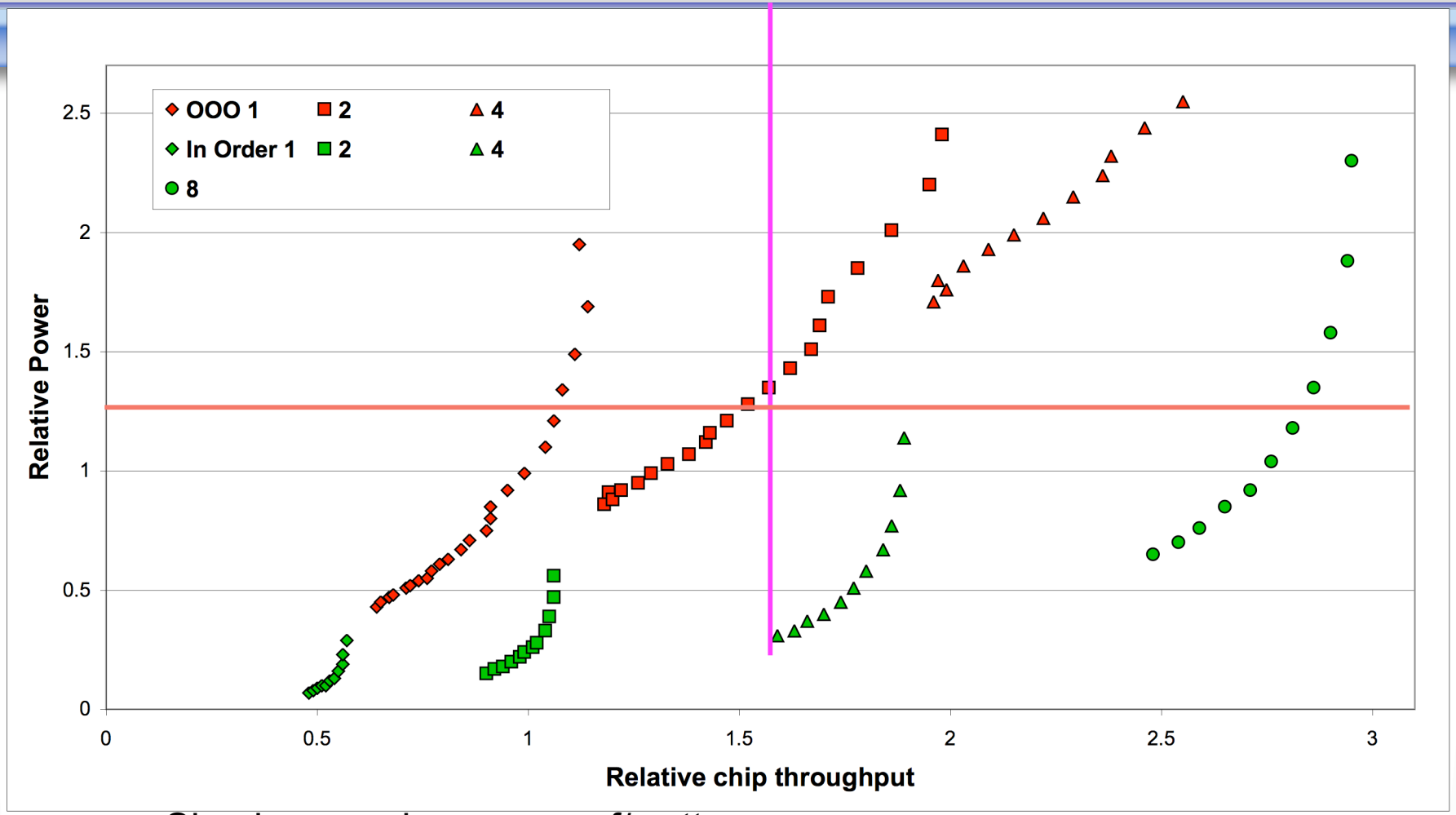
[PACT 2005] for details

Processor-Cache Balance is Important



- Performance on TPC-C
- C1: 64KB L1 caches, C2: 32 KB L1 caches, both 2p4t cores

Simpler Cores Offer Lower Power



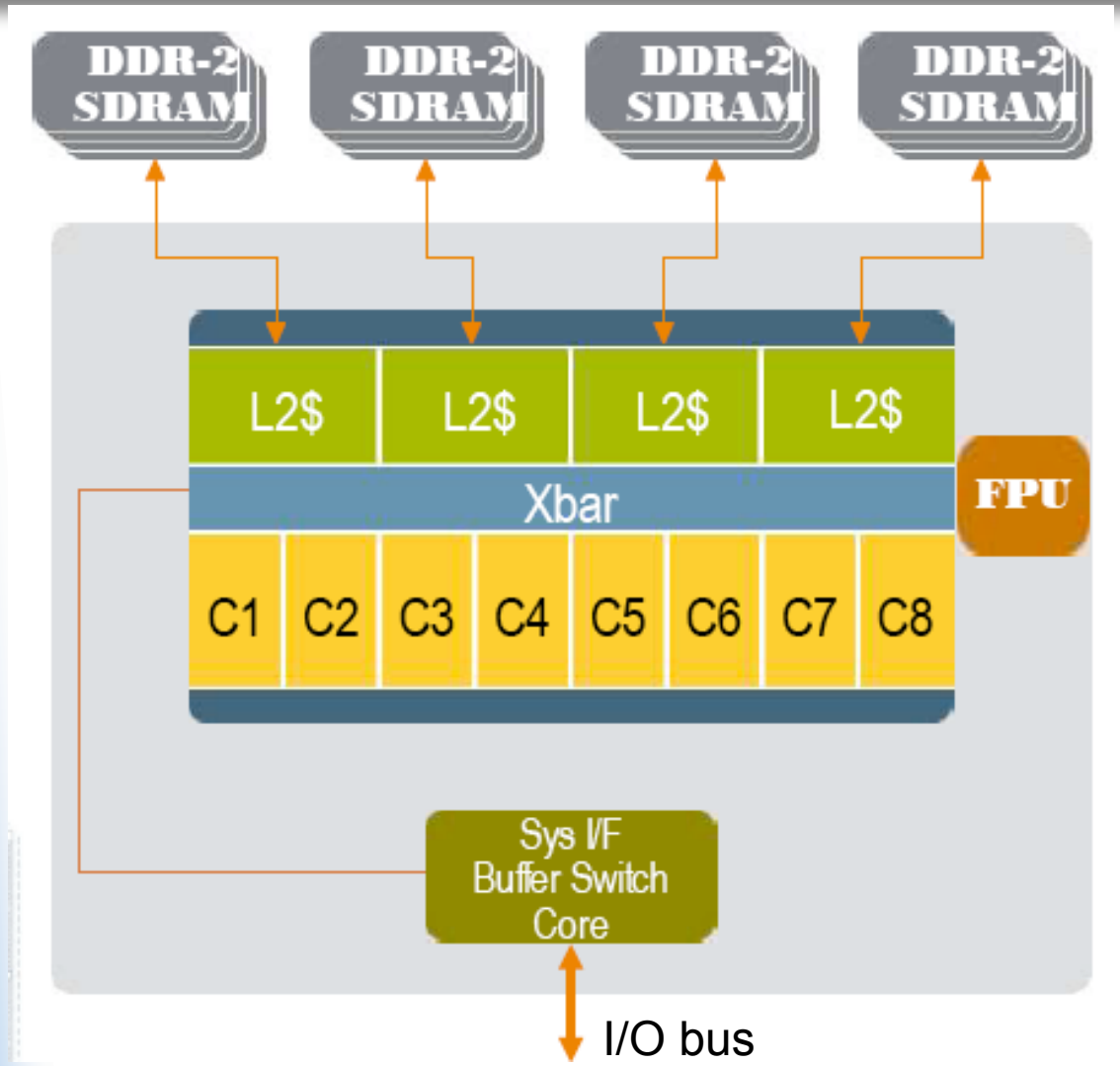
- Simple cores improve perf/watt
- Same performance at 20% of power
- 8 simple cores same power as 2 complex cores

Source: Tilak Agerwala, Micro May-June 2005

Niagara 1 Design Principles

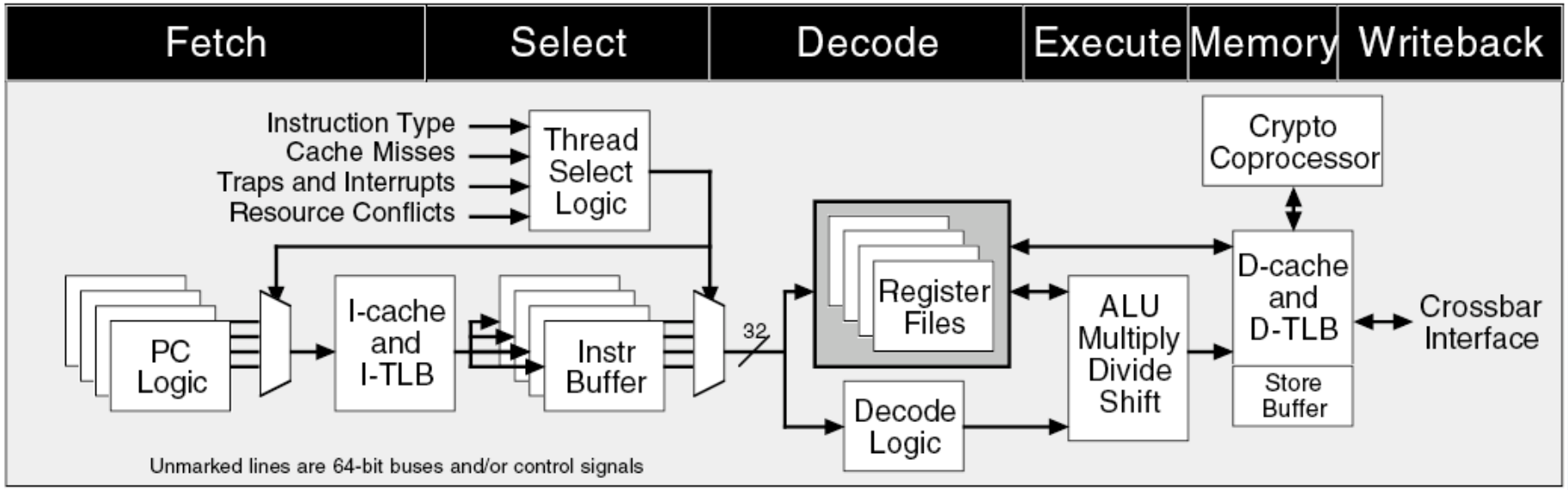
- Afara Websystems (2000)
  - Acquired by Sun Microsystems (2002)
- Designed for throughput and low power on commercial server applications
  - Niagara 1 (UltraSPARC T1) 2005
- Many simple cores vs. few complex cores
  - Exploit Thread (request) Level Parallelism vs. ILP
  - Improves power efficiency (MIPS/watt)
  - Lower development cost, schedule risk with simple pipeline
  - Improve yield by selling non-perfect parts
- Designed for good performance with cache misses
  - Lots of memory bandwidth per chip
  - Runs real apps even better than benchmarks
  - Hide cache, branch stalls with threads vs. ILP

Niagara CMP Overview



- SPARC V9 implementation
- 8 cores x 4 threads = 32 threads
- 90GB/sec crossbar switch
- High-bandwidth 4-way shared 3MB Level-2 cache on chip
- 4 DDR2 channels
- ~300M transistors
- 378 sq. mm die

SPARC Pipeline



- Single issue pipeline
- 4 threads, switch every cycle
- Per thread registers, instruction buffers and store buffers
  - 20% area overhead

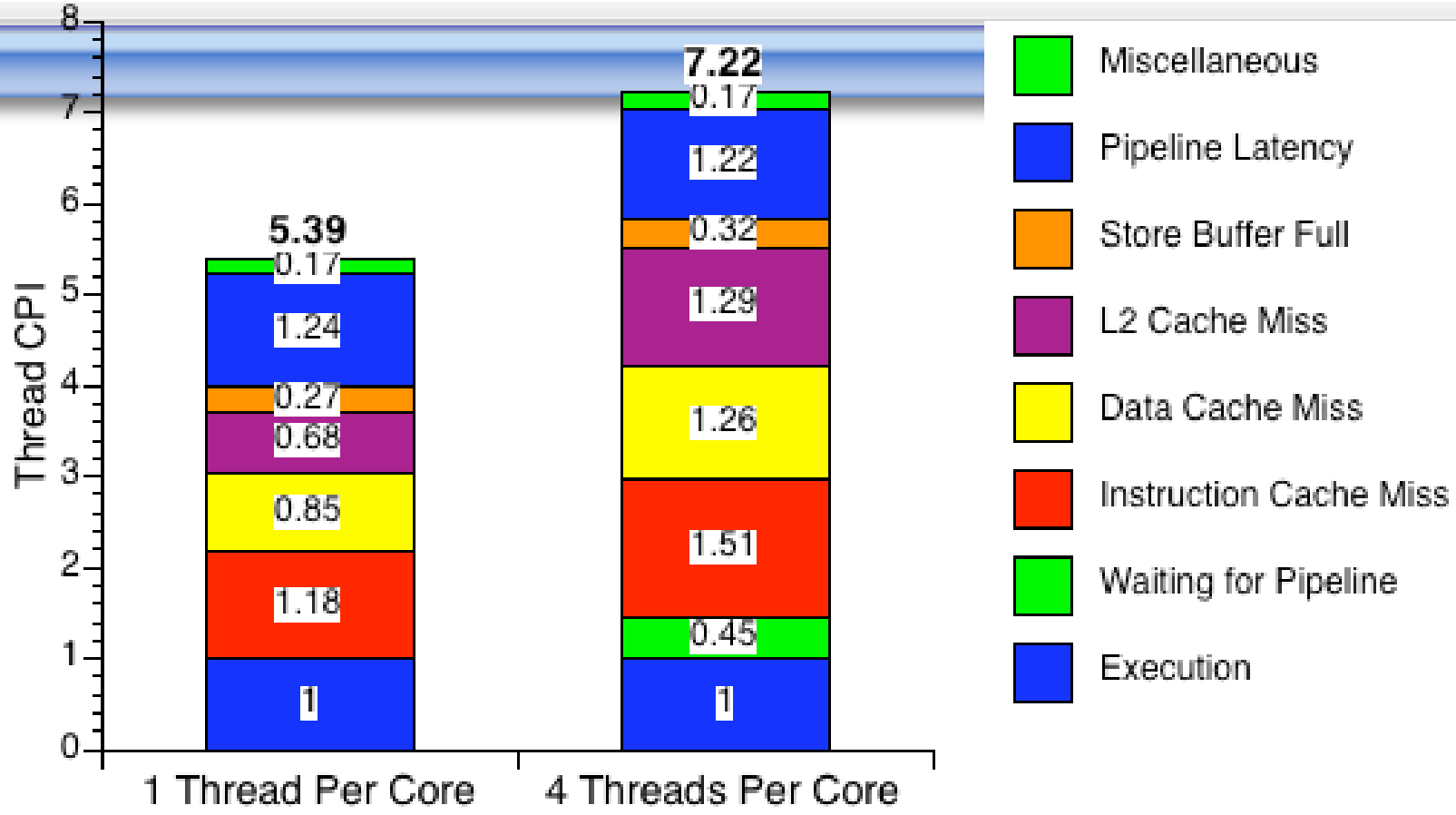
Niagara Memory System

- Instruction cache
  - 16kB, 4-way set associative, 32B line size
- Data cache
  - 8kB, 4-way set associative, 16B line size
  - Write-through cache, write-around on miss
- L2 cache
  - 3 MB, 12-way set associative, 64B line size
  - Write-through
  - 4-way banked by line
- Coherency
  - Data cache lines have 2 state: valid or invalid
  - Data cache is write through → no modified state
  - Caches kept coherent by tracking lines in directories in L2
- Consistency
  - TSO provided by crossbar
  - L2 is consistency point → threads that share L1 cache must wait to see to stores

Memory System Performance

- Latencies
  - Load-to-use: 3 cycles
  - Unloaded L2 latency: 22 cycles
  - Unloaded memory latency: ~90 ns
- Bandwidth
  - L2 bandwidth: 76.8 GB/s
  - Memory bandwidth: 25.6 GB/s

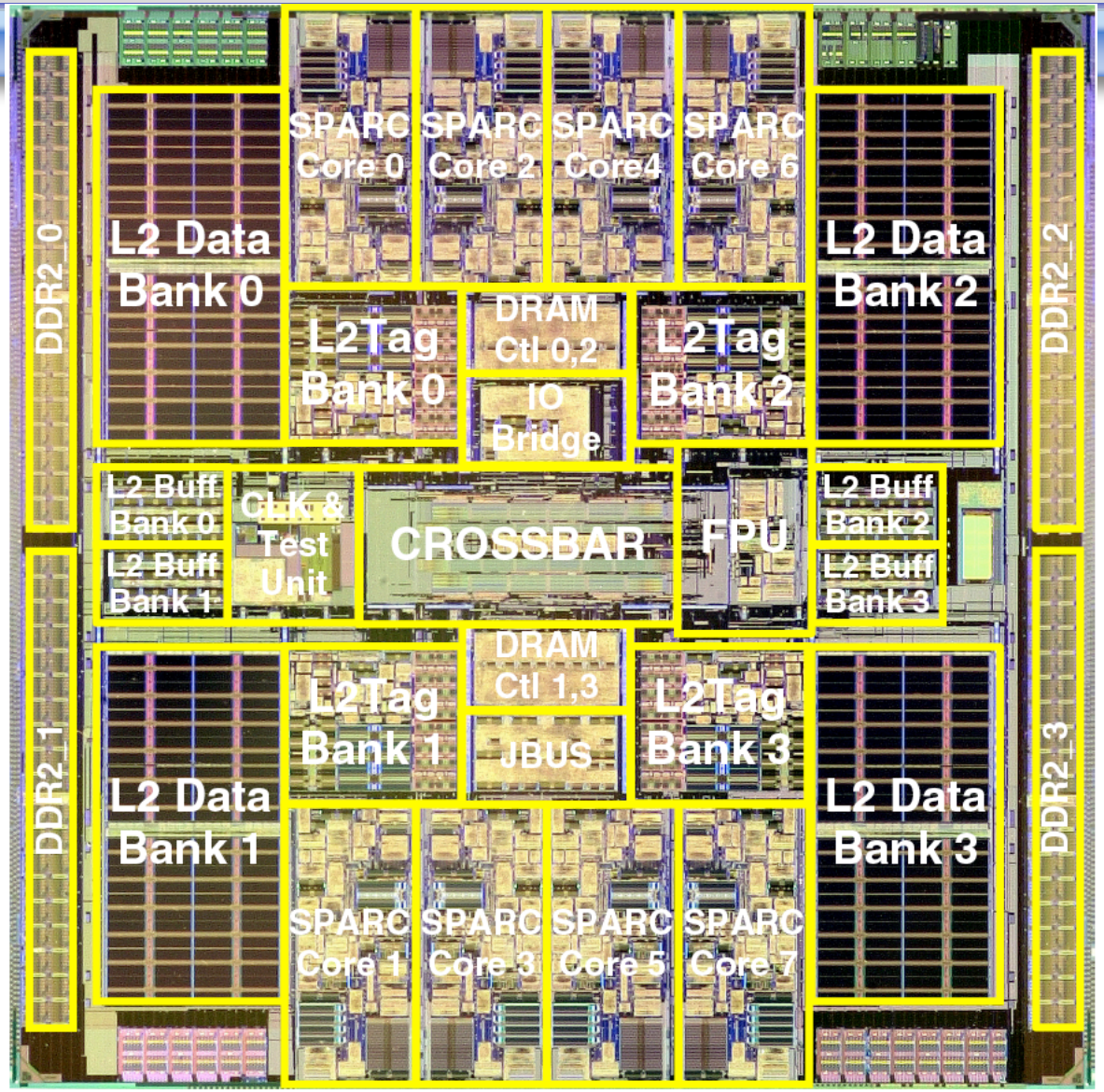
TPC-C Multithreading Performance



Core CPI	**5.39**	**1.8**

- 3x increase in throughput for 20% area increase
- 33% increase in latency

Niagara 1 Die Photo

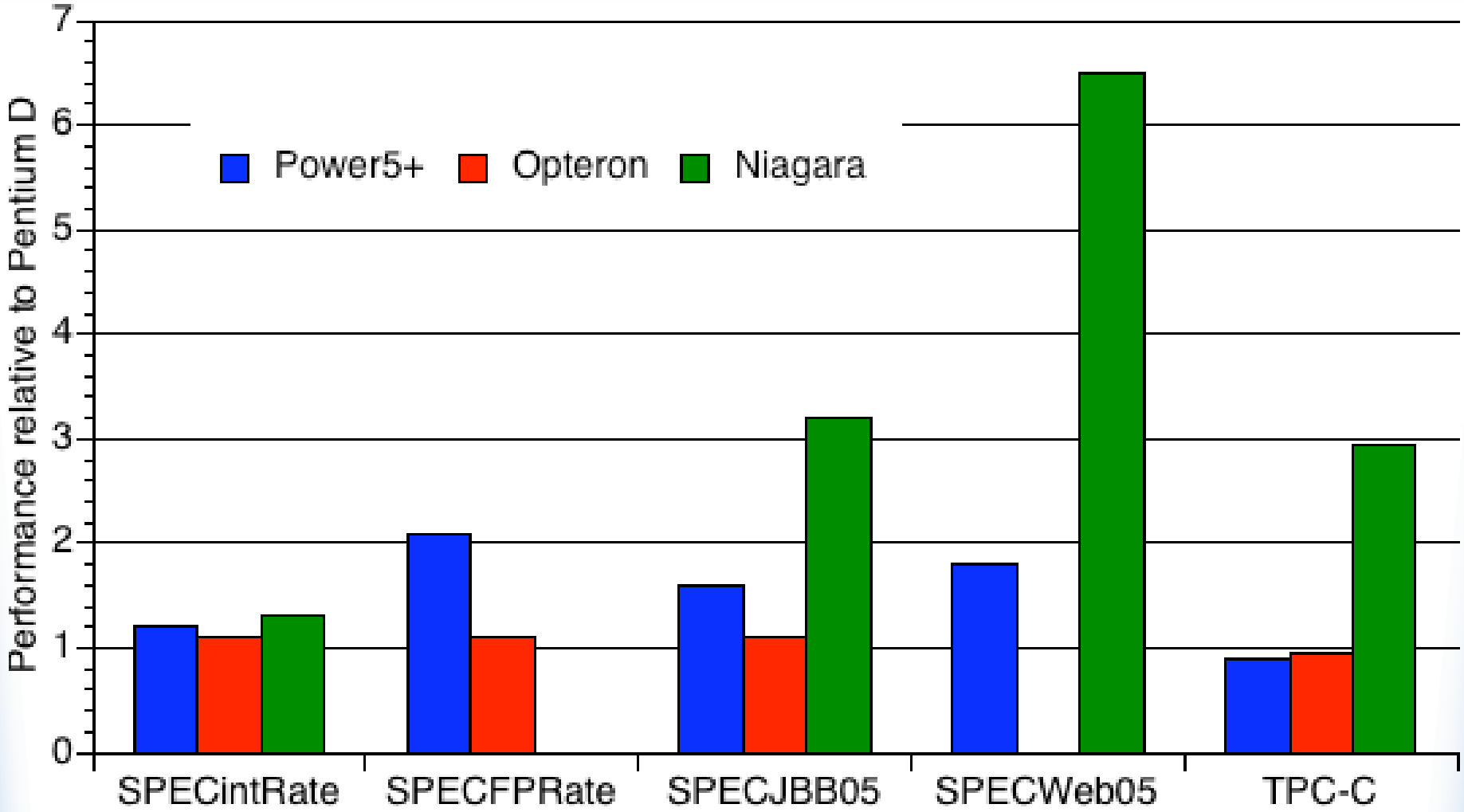


- 1 FPU
- 90 nm
- 279M transistors
- 378 mm^2

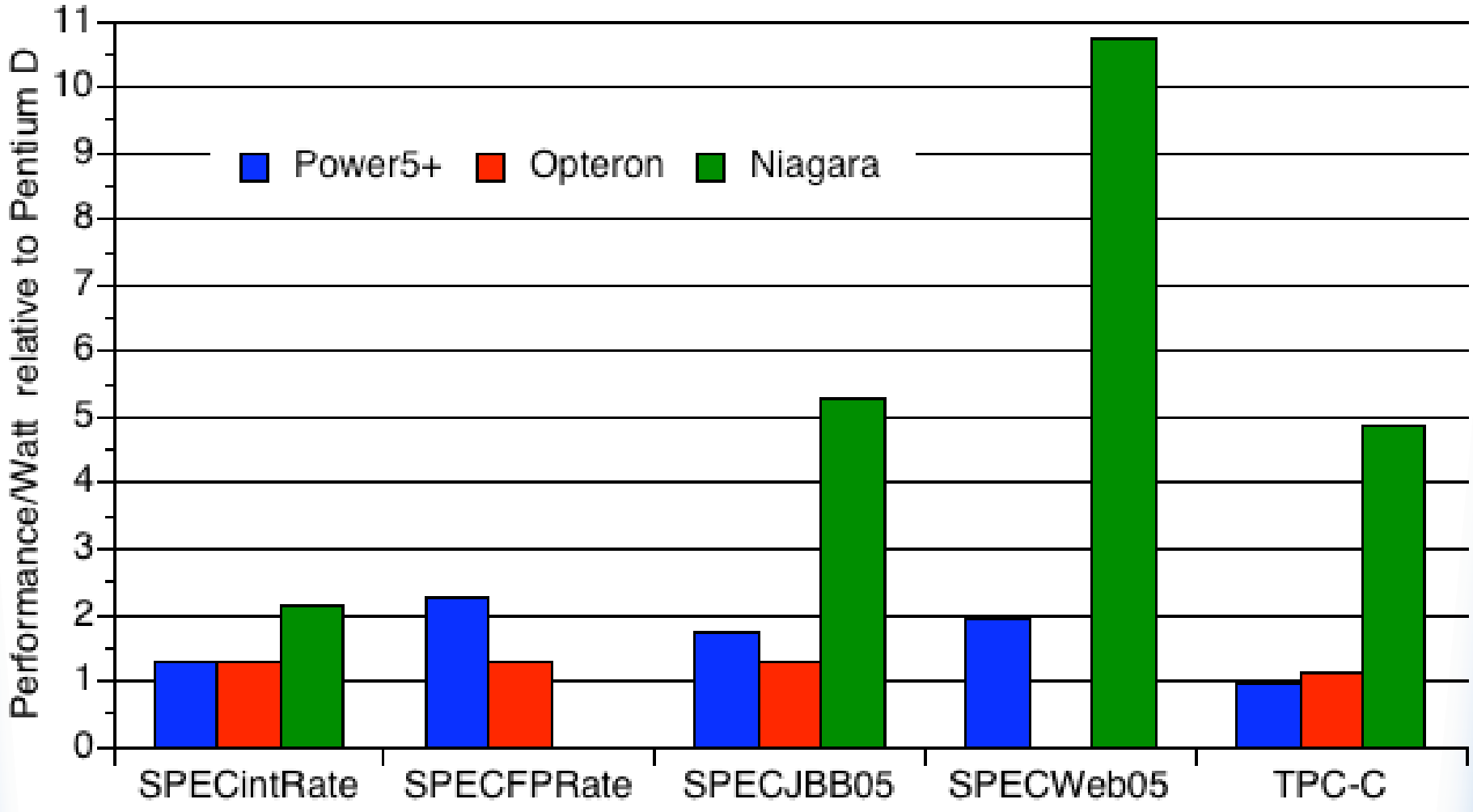
90nm Microprocessor Comparison

Processor	Niagara	Opteron	Pentium D	Power 5+
Cores/chip x threads/core	8 x 4	2 x 1	2 x 2	2 x 2
Peak IPC/core	1	3	3	4
Peak IPC/chip	8	6	6	8
L1 I/D (KB)	16/8	64/64	12/16	64/32
L2 (MB)	3 shared	1 per core	1 per core	1.9 shared
Memory BW (GB/s)	25.6	12.8	8.5	12.8
Frequency (GHz)	1.2	2.4	3.2	2.3
Power	79	110	130	120

Throughput Performance

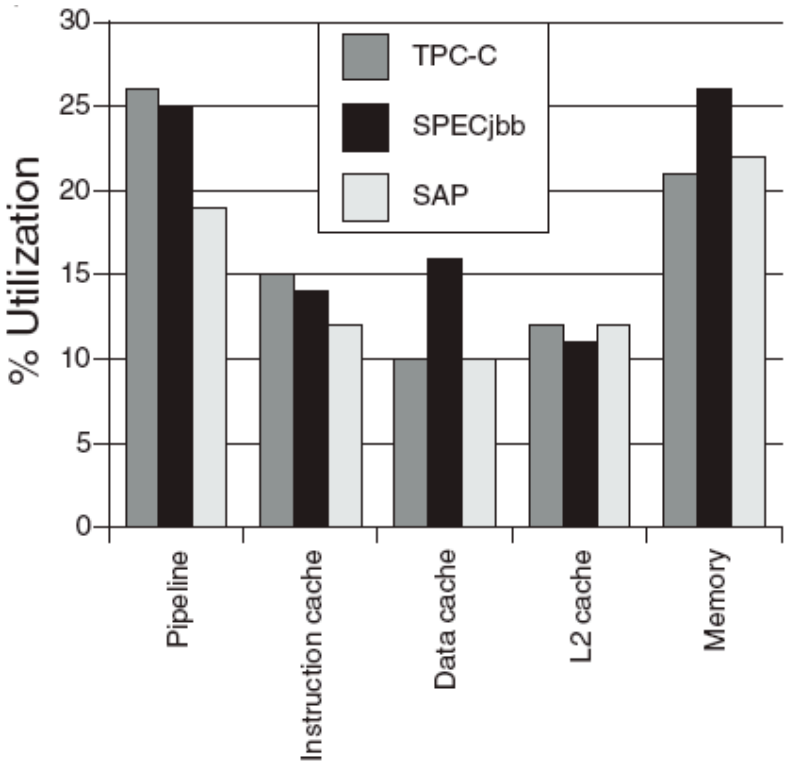


Performance/Watt

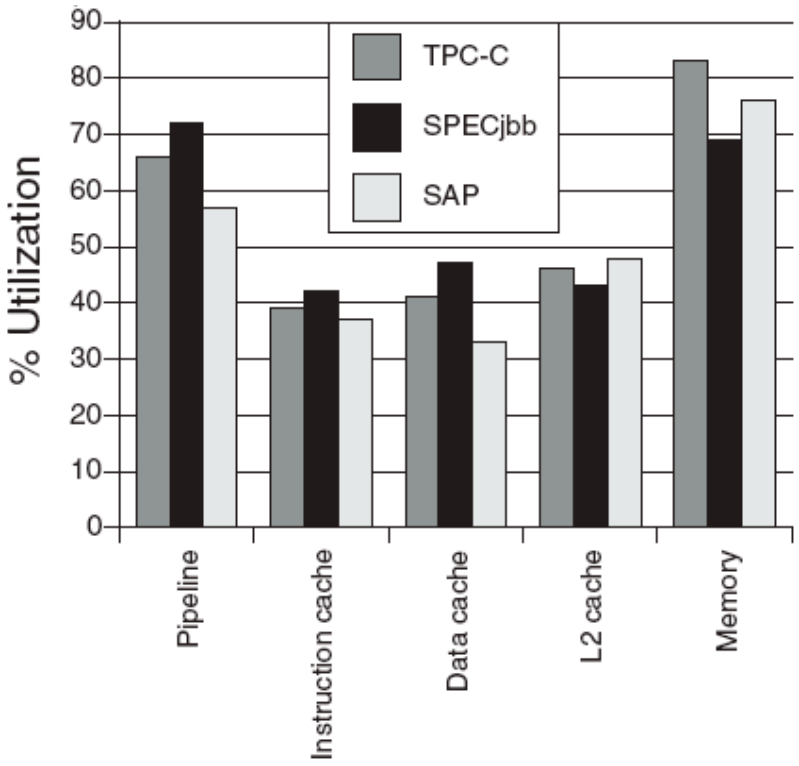


Niagara 1 Bottlenecks

1 thread/core



4 threads/core

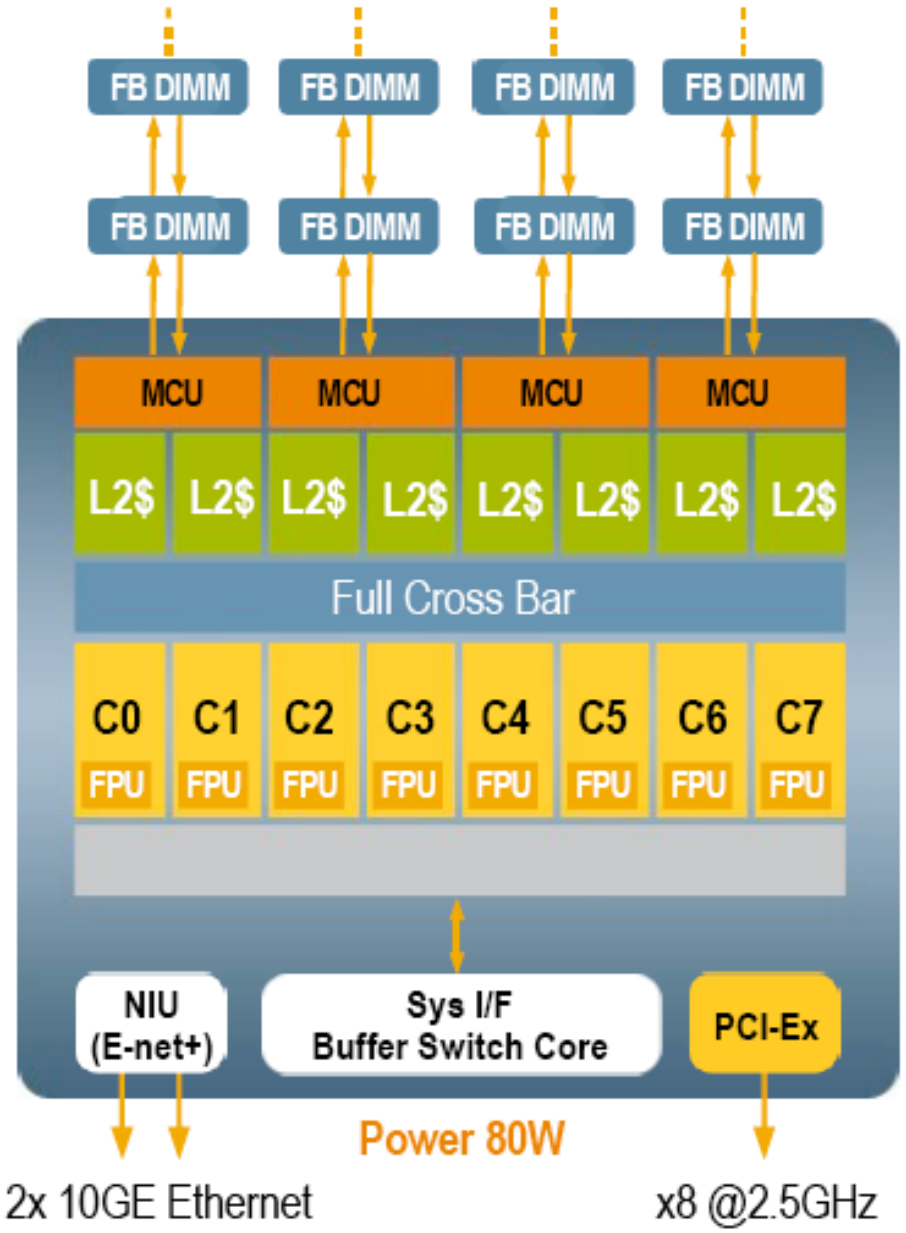


- Bottlenecks in pipeline and DRAM interface
- Spare bandwidth in caches

Transforming Niagara 1 into Niagara 2

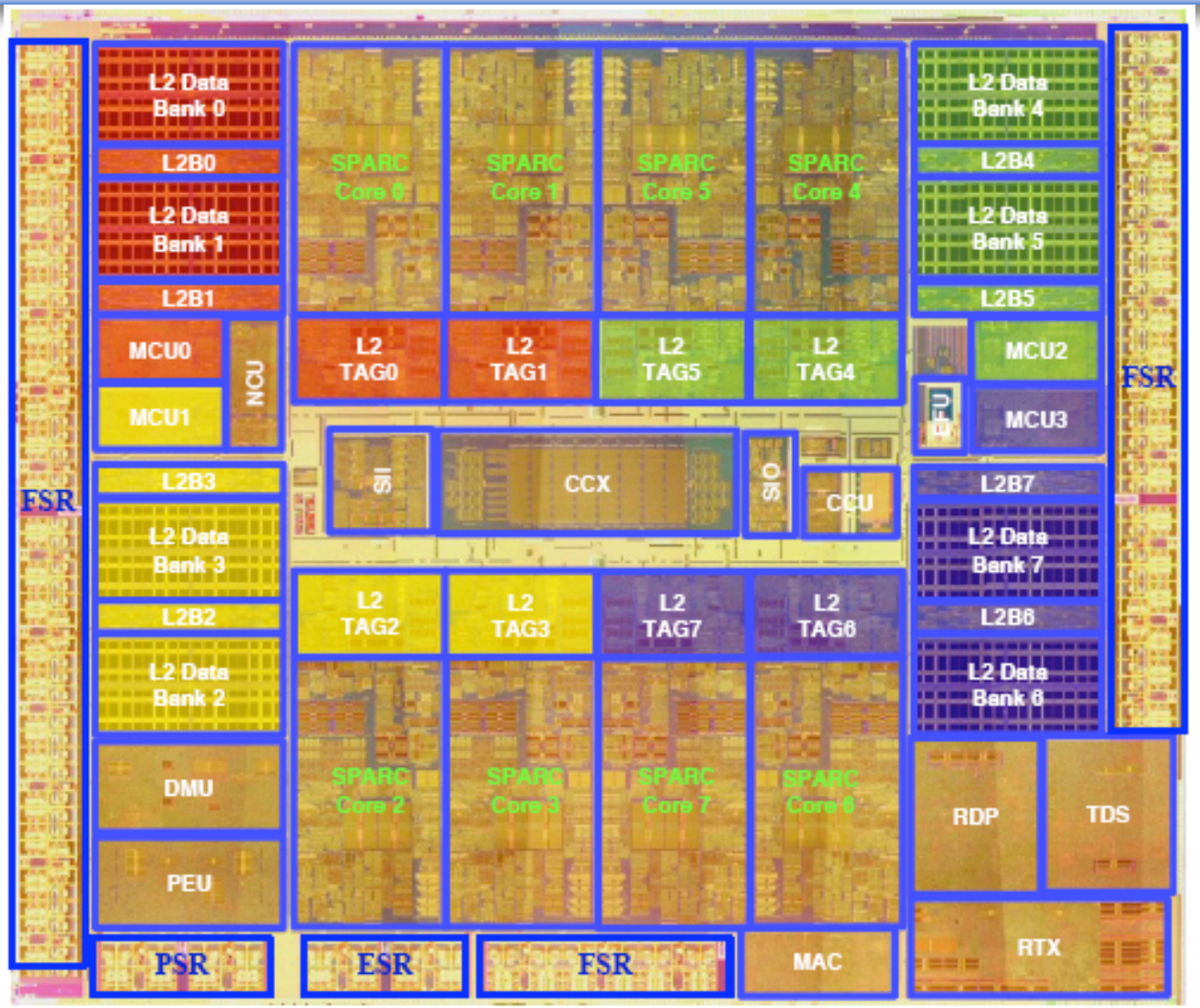
- Goal: double throughput with same power
  - Double number of cores?
- Double number of threads from 4 to 8
  - Choose 2 threads out of 8 to execute each cycle
- Double execution units from 1 to 2 and add FPU
- Double set associativity of L1 instruction cache to 8-way
- Double size of fully associative DTLB from 64 to 128 entries
- Double L2 banks from 4 to 8
  - 15 percent performance loss with only 4 banks and 64 thread

Niagara 2 at a Glance



- 8 cores x 4 threads
  - 2 pipes/core
  - 2 thread groups
- 1 FPU per core
- Crypto coprocessor per core
  - DES, 3DES, AES, RC4, etc
- 4MB L2, 8-banks, 16-way S.A
- 4 x dual-channel FBDIMM ports (60+ GB/s)
- > 2x Niagara 1 throughput and throughput/watt
- 1.4 x Niagara 1 int
- > 10x Niagara 1 FP

Niagara 2 Die



- 65 nm
- 342 mm^2

The Looming Crisis

- By 2010, software developers will face…
- CPU's with: (Niagara 2 and follow ons)
  - 20+ cores
  - 100+ hardware threads
  - Heterogeneous cores and application specific accelerators
- GPU's with general computing capabilities
- **Parallel programming gap**: Yawning divide between the capabilities of today's programmers, programming languages, models, and tools and the challenges of future parallel architectures and applications

Challenges in Parallel Programming

- Finding independent tasks
- Mapping tasks to threads
- Defining & implementing synchronization protocol
- Race conditions
- Deadlock avoidance
- Memory model
- Composing tasks
- Scalability
- Parallel performance analysis
- Recovering from errors

- → Transactions address a lot of parallel programming problems

Transactional Coherence & Consistency (TCC)

- With Christos Kozyrakis
- Goal
  - Make shared memory parallel programming accessible to the average developer
- Shared-memory parallel programming with transactions
  - No threads, no locks, just transactions…
- Programmer defined transactions are the *only abstraction*:
  - Parallel work
  - Communication and synchronization
  - Memory coherence and consistency
  - Failure atomicity and recovery
  - Performance optimization

Transactional memory definition

- Memory transaction: A sequence of memory operations that either execute completely (commit) or have no effect (abort)
- An “all or nothing” sequence of operations
  - On commit, all memory operations appear to take effect as a unit (all at once)
  - On abort, none of the stores appear to take effect
- Transactions run in isolation
  - Effects of stores are not visible until transaction commits
  - No concurrent conflicting accesses by other transactions
- Similar to database ACID properties

Transactional memory language construct

- The basic atomic construct:

  lock(L); x++; unlock(L); → atomic {x++;}

- Declarative – user simply specifies, system implements "under the hood"
- Basic atomic construct universally proposed
  - HPCS languages (Fortress, X10, Chapel) provide atomic in lieu of locks
  - Research extensions to languages – Java, C#, Haskell, …
- Lots of recent research activity
  - Transactional memory language constructs
  - Compiling & optimizing atomic
  - Hardware & software implementations of transactional memory

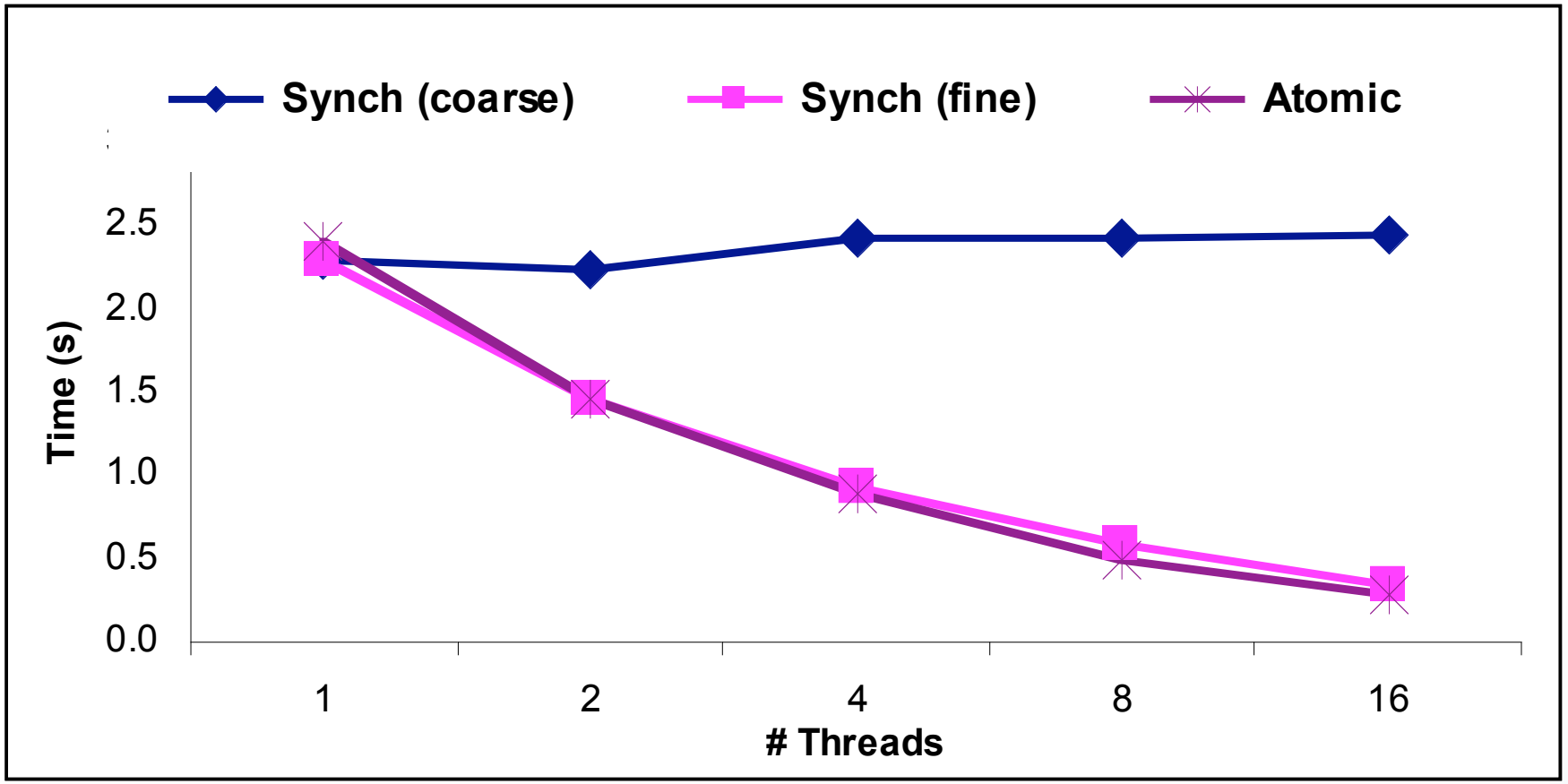
The Atomos Transactional Programming Language

- Atomos derived from Java
  - Transactional Memory for concurrency
    - `atomic` blocks define basic nested transactions
    - Removed `synchonized` and `volatile`
  - Transaction based conditional waiting
    - Derivative of Conditional Critical Regions and Harris `retry`
    - Removed `wait`, `notify`, and `notifyAll`
    - *Watch sets* for efficient implementation
  - Open nested transactions
    - `open` blocks committing nested child transaction before parent
    - Useful for language implementation but also available for apps
  - Violation handlers
    - Handle expected violations without rolling back in all cases
    - Not part of the language, only used in language implementation

Transactional Memory Benefits

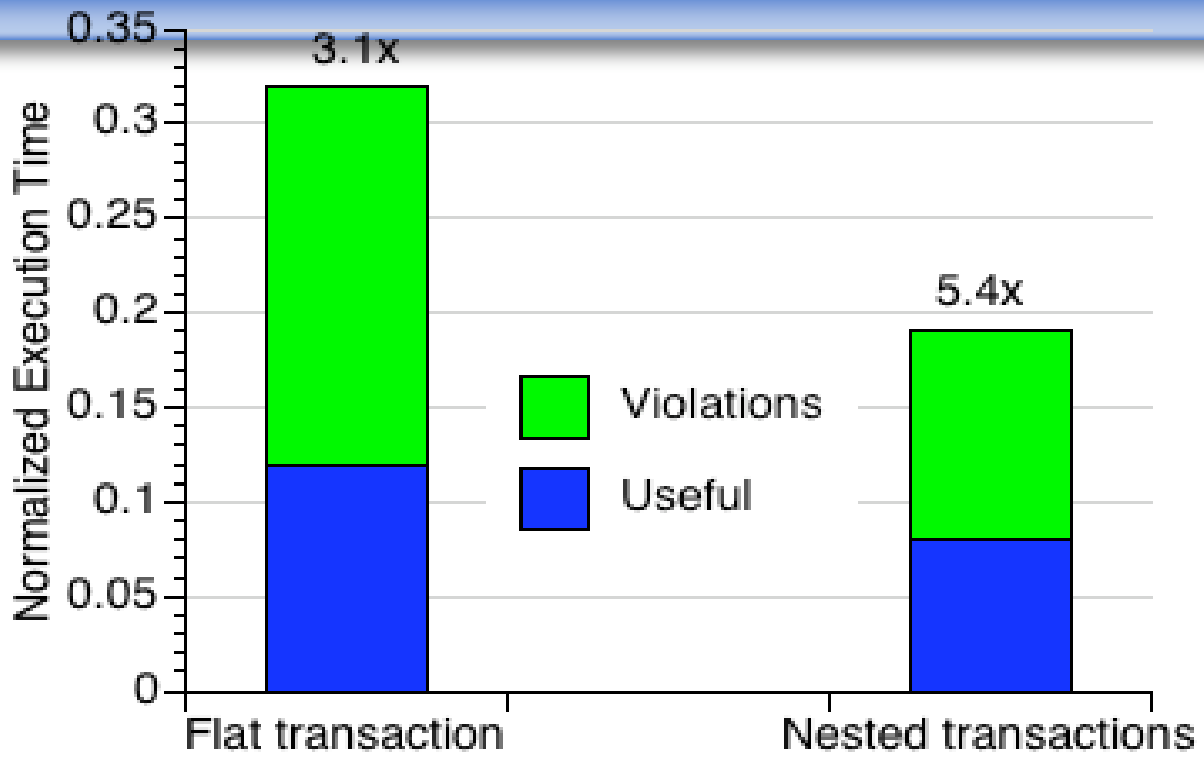
- Easier to achieve scalable results
  - Locks
    - Coarse-grained locks simple to use
      However limits scalability
    - Fine-grained locks can improve scaling
      However hard to get right
  - Transactions
    - As easy to use as coarse-grained locks
    - Scale as well as fine-grained locks

HashMap performance



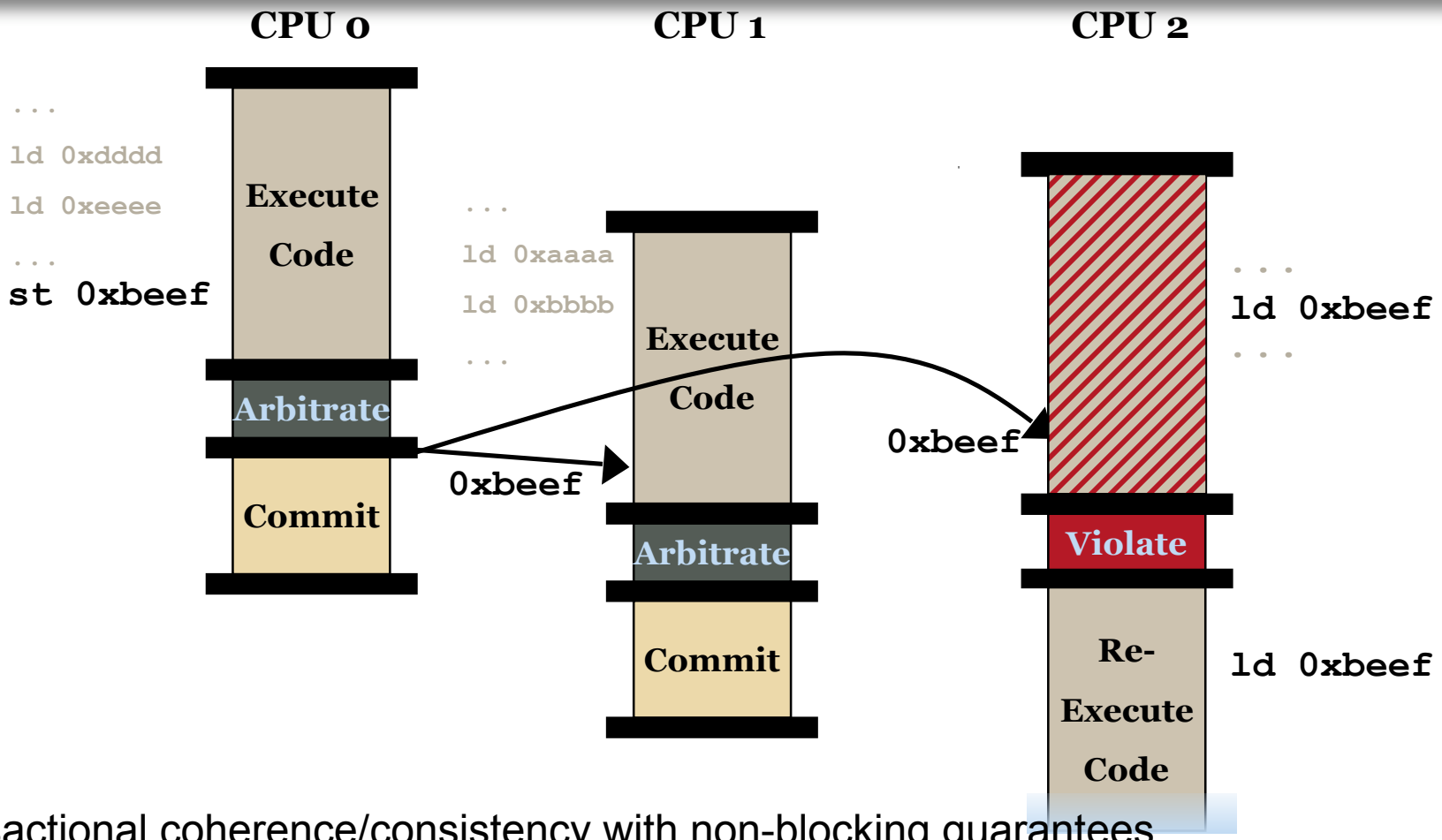
Transactions scales as well as fine-grained locks

SPECjbb2000 with Nested Transactions



- 5.4x speedup with 8 processor CMP
- Violation time reduced and improved cache behavior
- Details in [ISCA'06]

TCC Execution Model



Transactional coherence/consistency with non-blocking guarantees

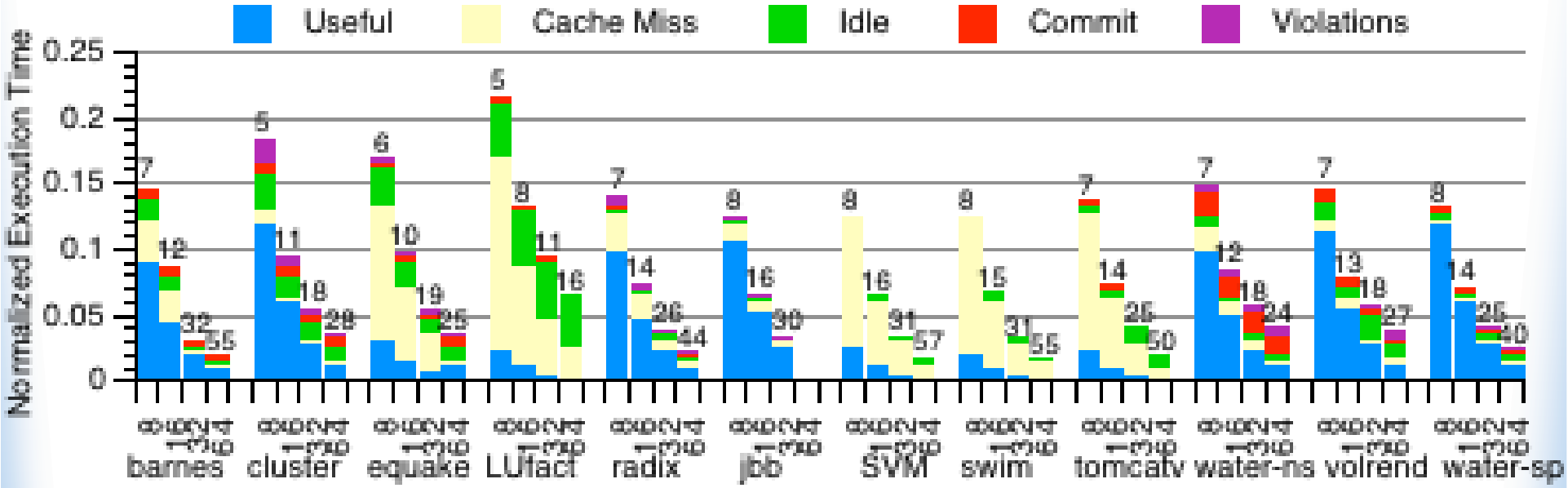
See [ISCA'04] for details

Scaling TCC

- Eliminate commit broadcast of original TCC
- Eliminate write through cache design
- Supports parallel commits
- First scalable TM implementation of a directory based distributed shared memory that is live-lock free

Scalable TCC Performance
64 proc DSM

- Write-through commit no longer works
  - 2-phase commit ⇒ parallel commit
  - Commit is not a bottleneck
- Excellent scalability



Conclusions

- Sun Niagara
  - CMP optimized for performance/watt on commercial server applications
  - Design approach: trade latency for throughput → low power
  - Simple pipelines, CMP and multithreading, high bandwidth memory hierarchy
  - 2-4x improvement in performance/watt compared to conventional microprocessors on commercial applications
- Stanford TCC: a shared-memory parallel model with transactions
  - Software-defined transactions as the only abstraction for parallelism, communication, failure atomicity, and optimization
  - Simple path to correct code:
    - Through speculative parallelism and HW-based atomicity/ordering
  - Intuitive, feedback-driven performance tuning
    - Through continuous data tracking and logging
  - Initial results are encouraging
    - 90% of performance at 10% of programming effort