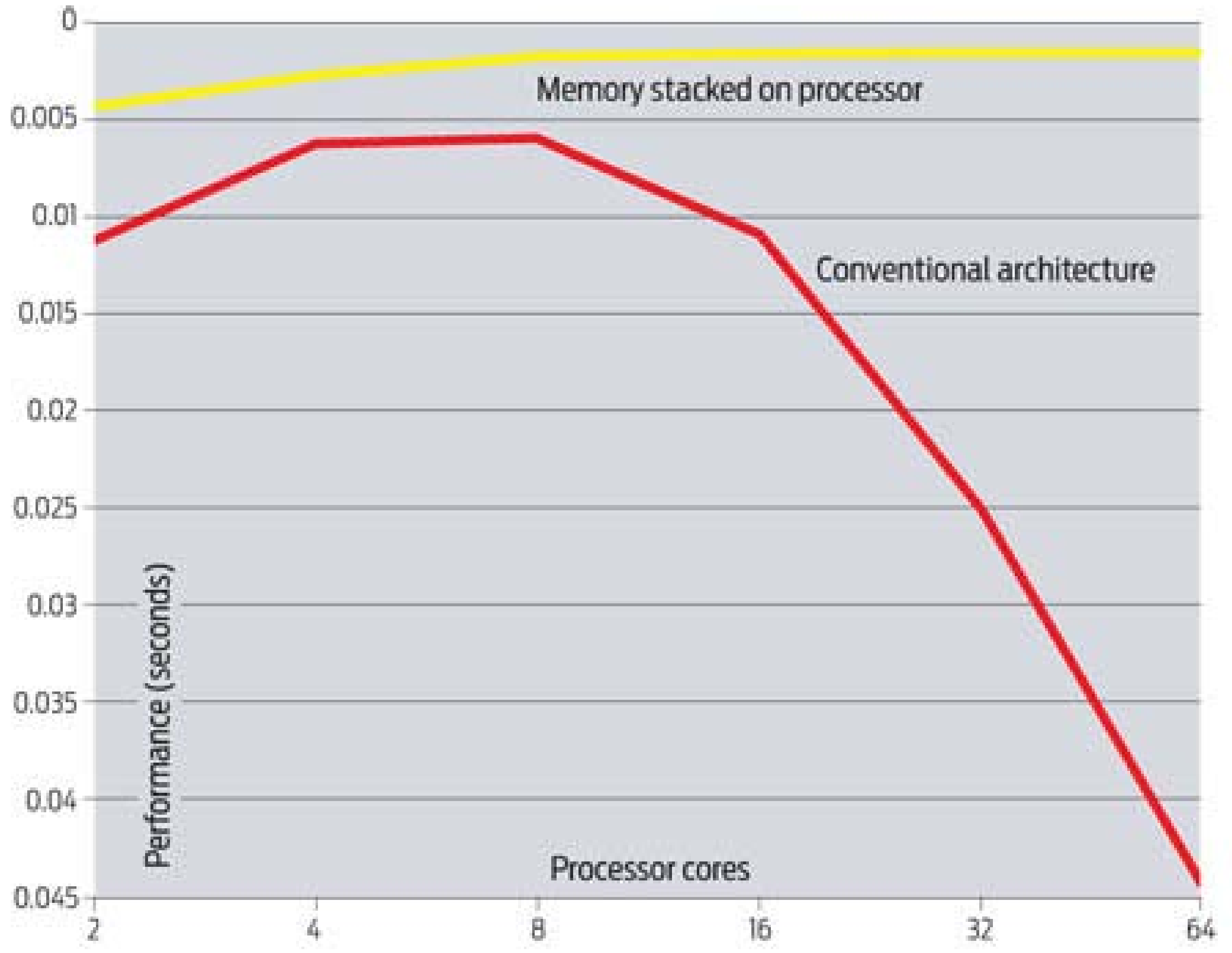
Programmable Transactional Memory

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University

SOS-13

The Memory Wall?



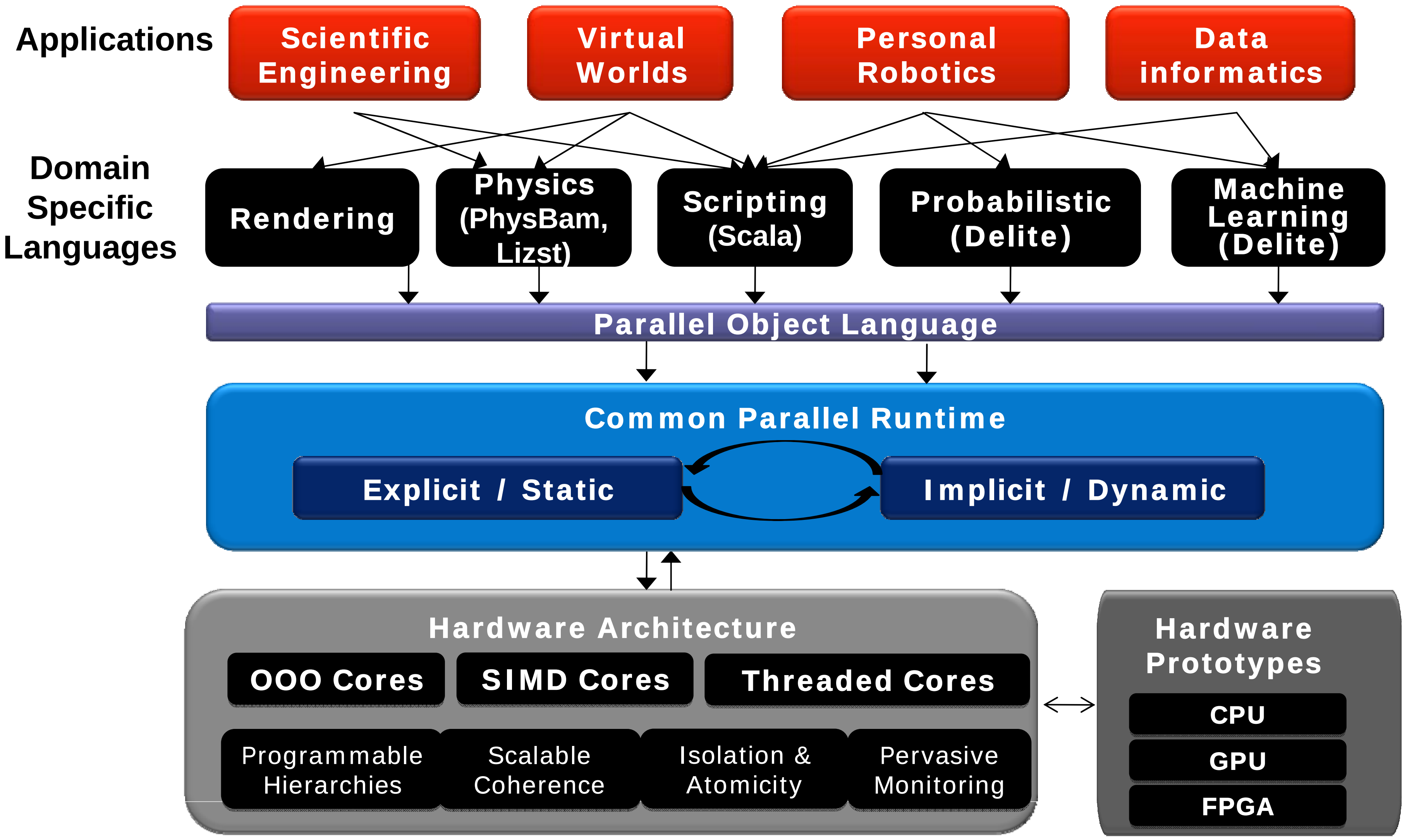
Multicore Is Bad News For Supercomputers
Samuel K. Moore

<u>Sharing</u> and scheduling can help!

The Stanford Pervasive Parallelism Laboratory

- Goal: the parallel computing platform for 2012
  - Make parallel application development practical for the average software developers
  - Parallel applications without parallel programming
- PPL is a combination of
  - Leading Stanford researchers across multiple domains
    - Applications, languages, software systems, architecture
  - Leading companies in computer systems and software
    - Sun, AMD, Nvidia, IBM, Intel, HP, NEC
  - An exciting vision for pervasive parallelism

The PPL Vision



Transactional Memory (TM)

- Memory transaction [Knight'86, Herlihy & Moss'93]
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions
- Atomicity (all or nothing)
  - At commit, all memory updates take effect at once
  - On abort, none of the memory updates appear to take effect
- Isolation
  - No other code can observe memory updates before commit
- Serializability
  - Transactions seem to commit in a single serial order

Programming with TM

```
int deposit(account, amount)
   synchronized(account) {
   atomic {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
        return (1);
   }
```

```
int withdraw(account, amount)
   synchronized(account) {
   atomic {
        int t = bank.get(account);
        t = t – amount;
        if (t<0) return (0);
        bank.put(account, t);
        return (1);
   }
```

- Declarative synchronization
  - Programmers says what but not how
  - No explicit declaration or management of locks
- System implements synchronization
  - Typically with optimistic concurrency [Kung'81]
  - Slow down only on true conflicts (R-W or W-W)

Advantages of TM

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements
- Performs as well as fine-grain locks
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness
- Failure atomicity & recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- Composability
  - Safe & scalable composition of software modules

Implementing Memory Transactions

- Data versioning for updated data
  - Manage new & old values for memory data
  - *Deferred(lazy) updates* (write-buffer) vs *direct updates* (undo-log)
- Conflict detection for shared data
  - Detect R-W and W-W for concurrent transactions
  - Track the *read-set* and *write-set* of each transaction
  - Check during execution (*pessimistic*) or at the end (*optimistic*)
- Ideal implementation
  - Software only: works with current & future hardware
  - Flexible: can modify, enhance, or use in alternative manners
  - High performance: faster than sequential code & scalable
  - Correct: no incorrect or surprising execution results

Software Transactional Memory

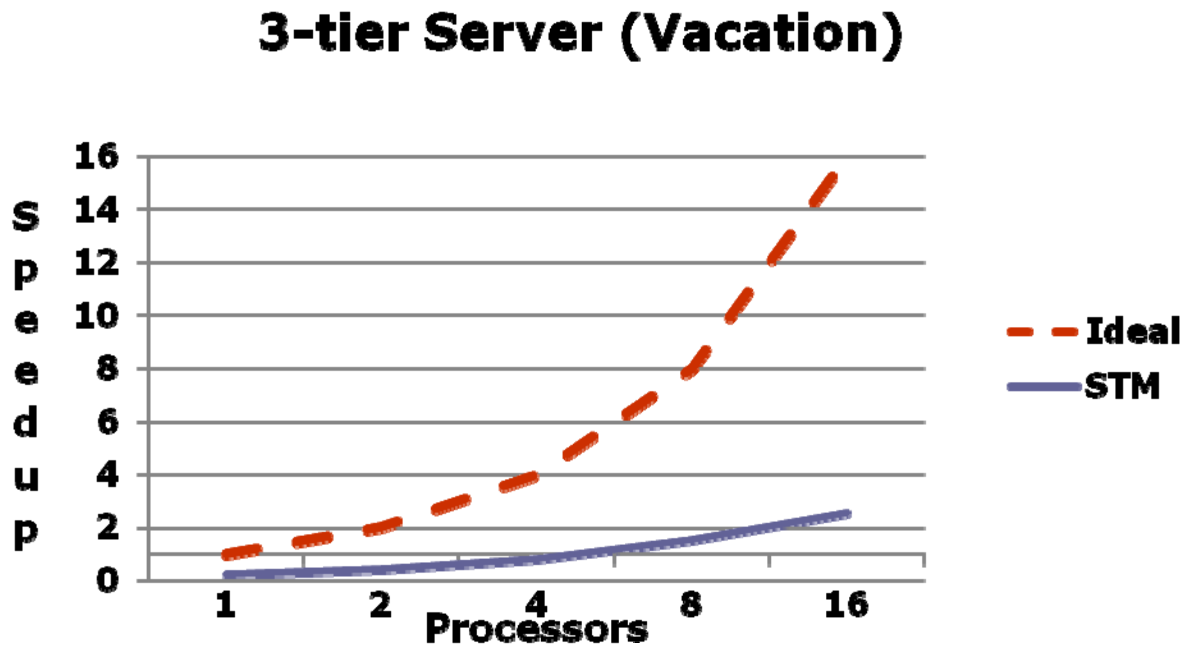
High-level $\xrightarrow{\text{STM Compiler}}$ Low-level

```
ListNode n;
atomic {
  n = head;
  if (n != NULL) {

    head = head.next;

  }
}
```

```
ListNode n;
STMstart();
  n = STMread(&head);
  if (n != NULL) {
    ListNode t;
    t = STMread(&head.next);
    STMwrite(&head, t);
  }
STMcommit();
```

- Software barriers for TM bookkeeping
  - Versioning, read/write-set tracking, commit, ...
  - Using locks, timestamps, object copying, ...
- Can be optimized by compilers [Adl-Tabatabai'06, Harris'06]
- Requires function cloning or dynamic translation

STM Performance Challenges

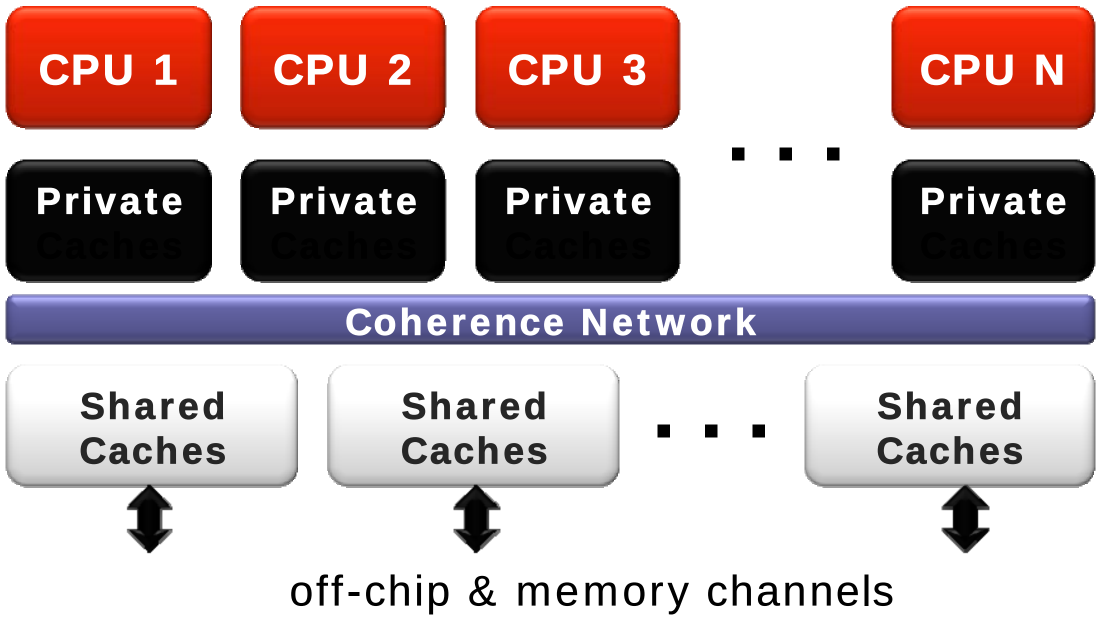


- 2x to 8x overhead due to SW barriers
  - After compiler optimizations, inlining, …
- Short term: demotivates parallel programming
  - TM coding easier than locks but harder than sequential…
- Long term: energy wasteful

Hardware TM (HTM)

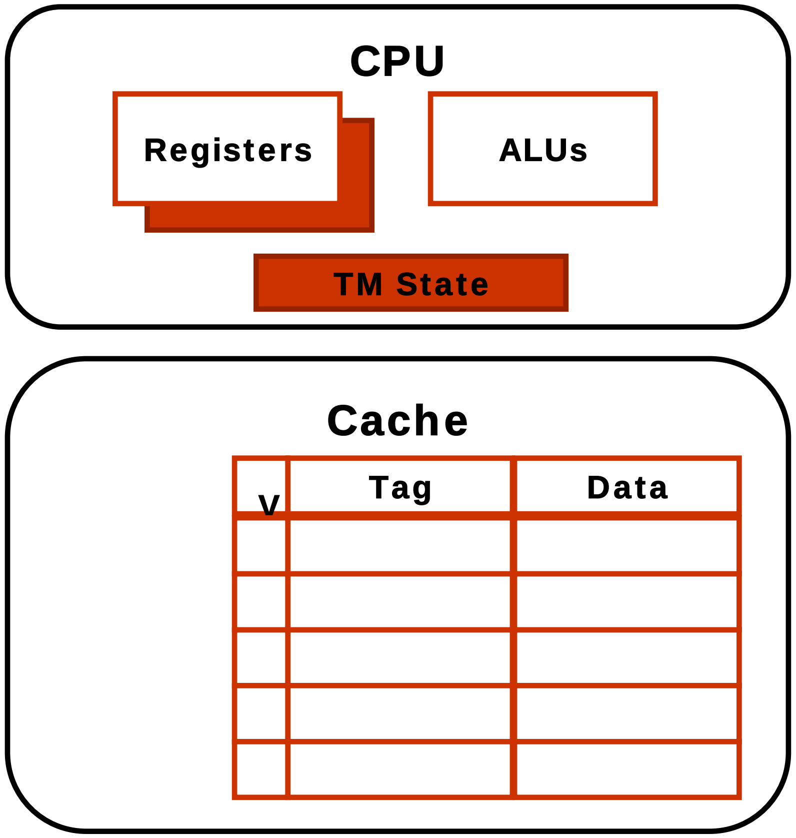
- HW support for common case TM behavior
  - Initial TMs used hardware [Knight'86, Herlihy & Moss'93]
  - All HTMs include software too...
- Rationale
  - HW can track all loads/stores transparently, w/o overhead
  - HW is good at fine-grain operations within a chip
  - We have transistors to spare in multi-core designs
    - Thanks to Moore's law...
- Basic HW mechanisms
  - Cache metadata track read-set & write-set
  - Caches buffer deferred updates
  - Coherence protocol does conflict detection

Multi-core Chip



- HTM works with bus-based & scalable networks
- HTM works with private & shared caches

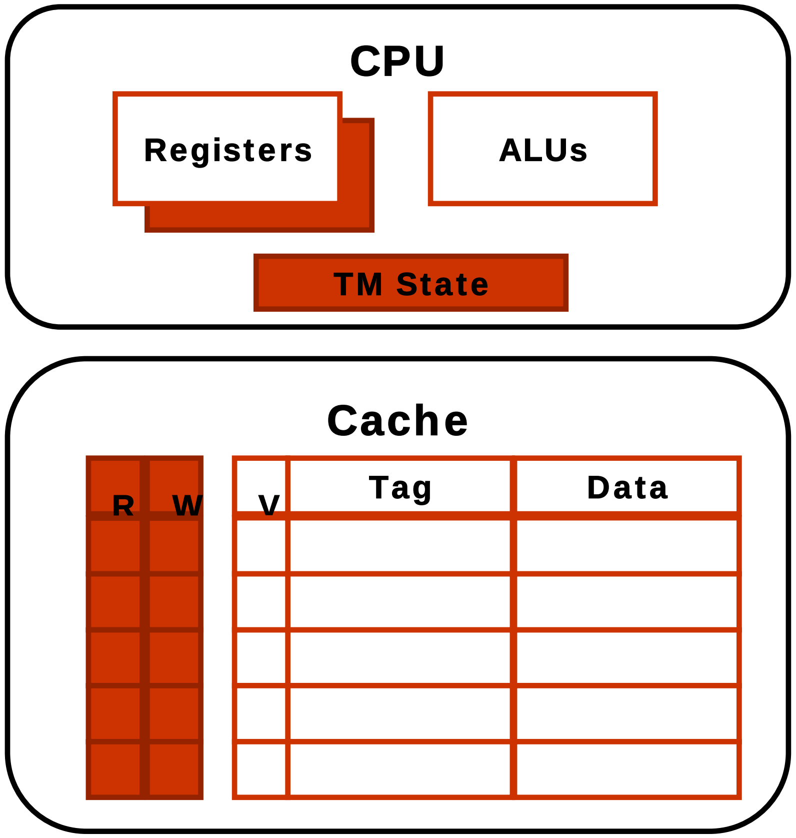
HTM Design



- The details in [ISCA’04, PACT’05, HPCA’07]

- CPU changes
  - Register checkpoint (available in many CPUs)
  - TM state registers (status, pointers to handlers, …)

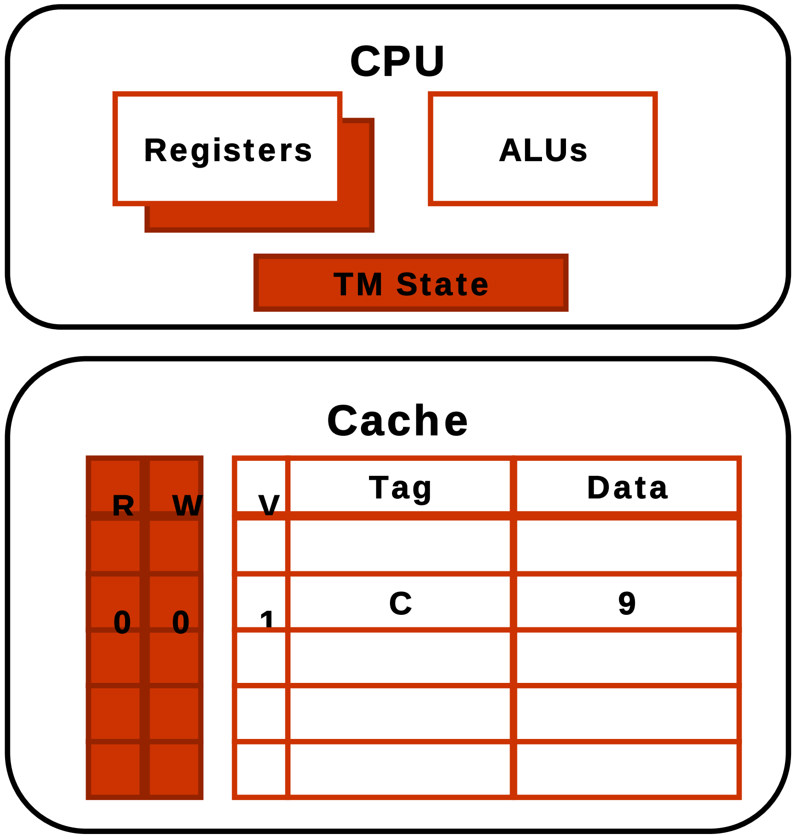
HTM Design



- The details in [ISCA’04, PACT’05, HPCA’07]

- Cache changes
  - R bit indicates membership in read-set
  - W bit indicates membership in write-set

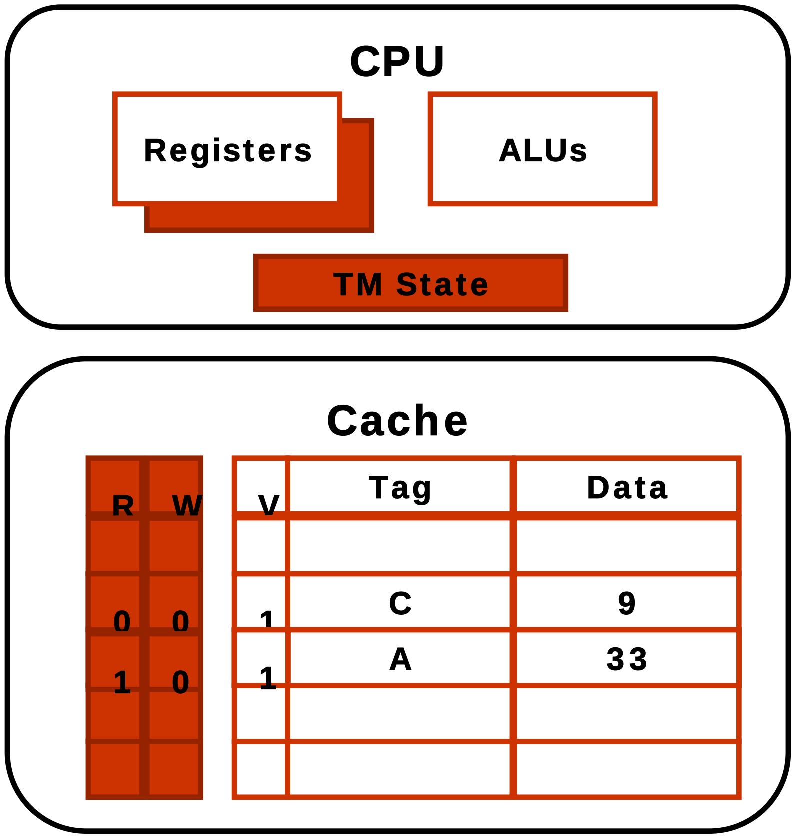
HTM Transaction Execution



```
Xbegin ⇦
    Load A
    Store 5 □ B
    Load C
Xcommit
```

- Transaction begin
  - Initialize CPU & cache state
  - Take register checkpoint

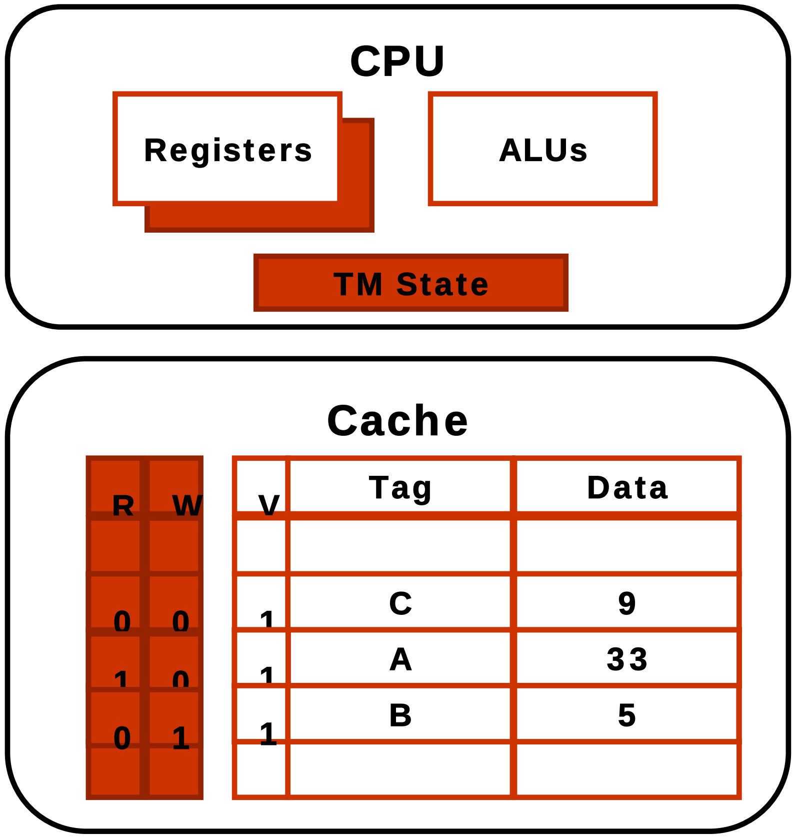
HTM Transaction Execution



```
Xbegin
   Load A ⇦
   Store 5 □ B
   Load C
Xcommit
```

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read-set

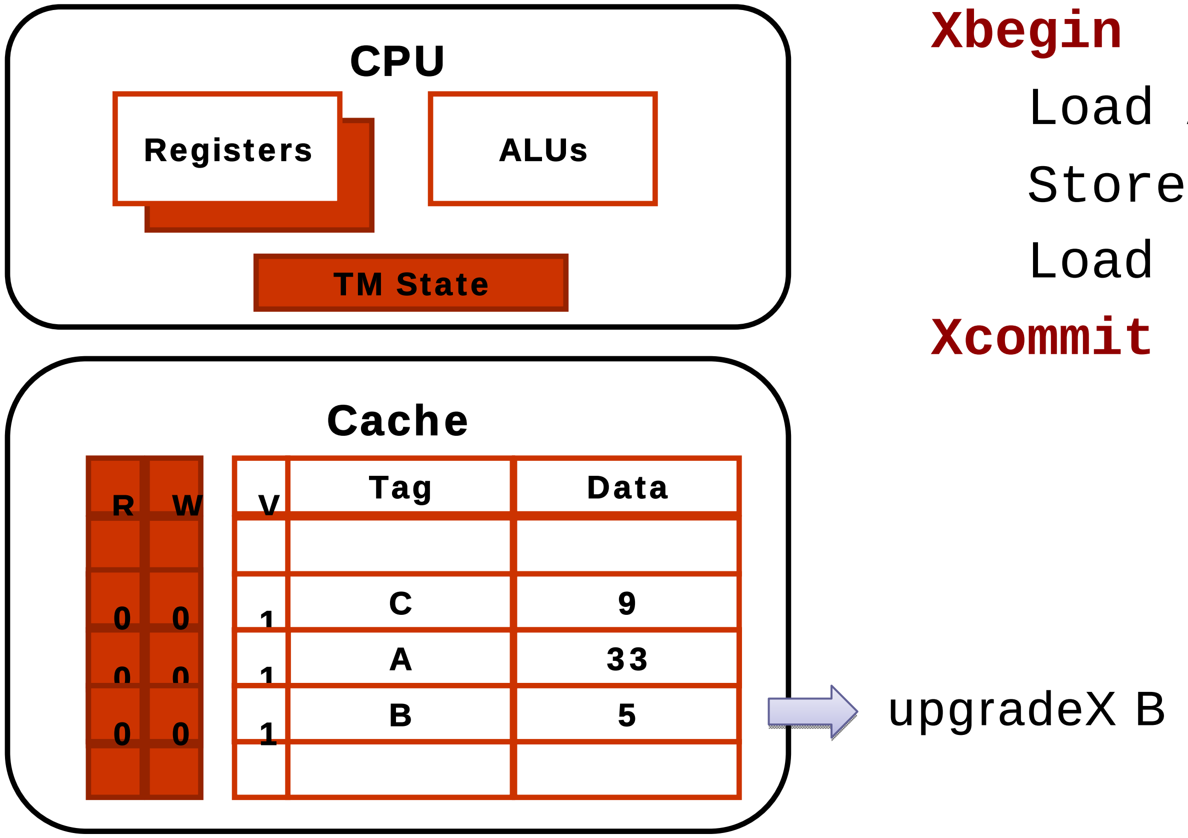
HTM Transaction Execution



```
Xbegin
   Load A
   Store 5 □ B ⇦
   Load C
Xcommit
```

- Store operation
  - Serve cache miss if needed (eXclusive if not shared, Shared otherwise)
  - Mark data as part of write-set

HTM Transaction Execution

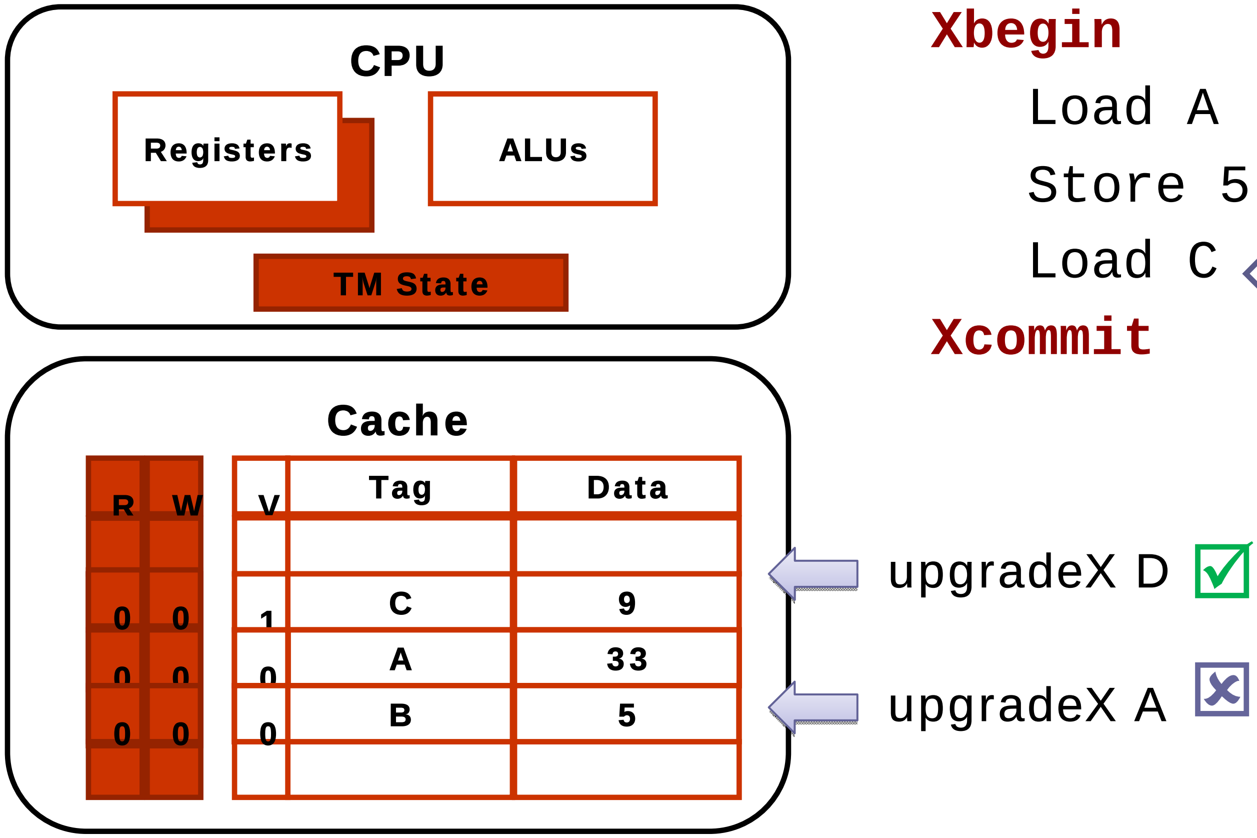


```
Xbegin
    Load A
    Store 5 □ B
    Load C
Xcommit ⇦
```

- Fast, 2-phase commit
  - Validate: request exclusive access to write-set lines (if needed)
  - Commit: gang-reset R & W bits, turns write-set data to valid (dirty) data

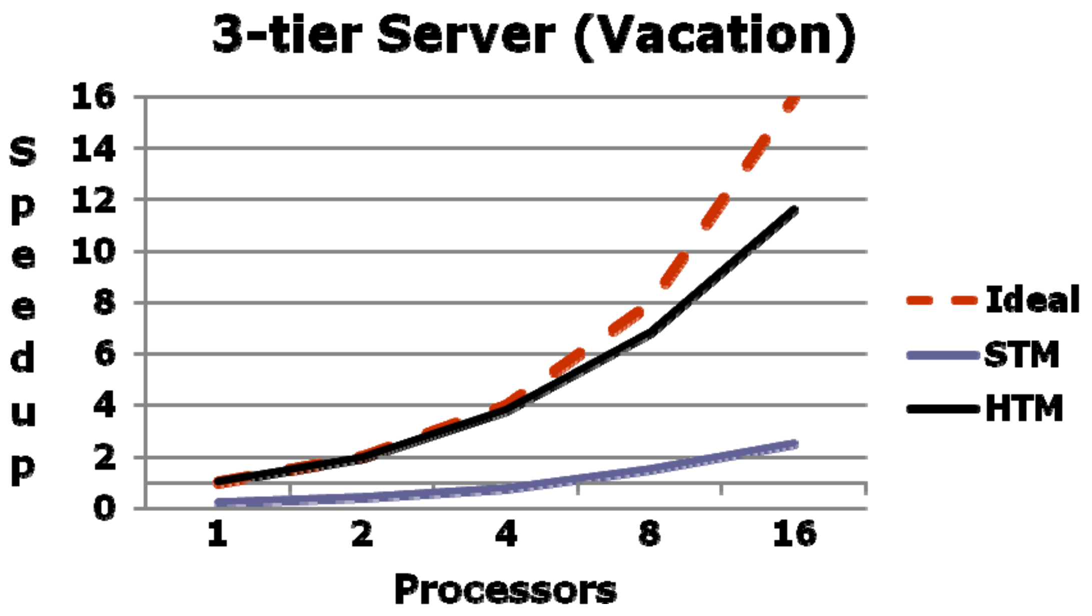
HTM Conflict Detection

```
Xbegin
    Load A
    Store 5 □ B
    Load C ⇦
Xcommit
```



- Fast conflict detection & abort
  - Check: lookup exclusive requests in the read-set and write-set
  - Abort: invalidate write-set, gang-reset R and W bits, restore checkpoint

Performance with HTM



- Scalable performance, up to 7x over STM [ISCA'07]
  - Within 10% of sequential for one thread
  - Uncommon HTM cases not a performance challenge

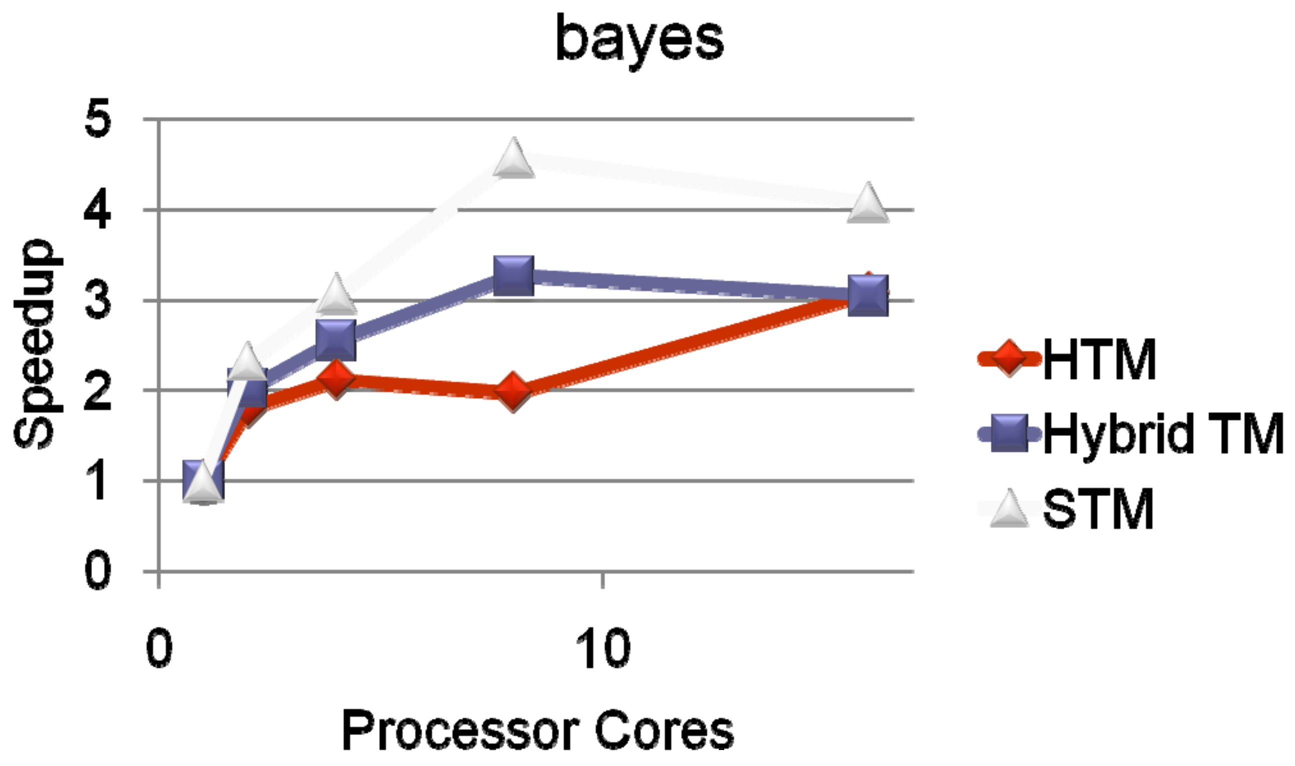
TCC Architecture Model

- TCC architecture approach
  - Programmer determines coherence and consistency points
    - At transaction boundaries
  - Use caches for fast local buffering
  - Use coherence protocol for commit & violation detection
- Distinguishing feature
  - *Re-design* the coherence protocol *just* for transactions
  - All communication at transaction commit points
  - No need to support other fine-grain communication
    - Simpler protocol, few short messages, ...

Scalable TCC Performance

- Scalable TCC implementation
  - Directory-based, 2-phase commit, traffic filtering
- Excellent scalability
  - Lazy commit not an overhead

Pure HTMs Have Limitations



Communication Landscape 3 Ws and 1 S

Comm. Mech.	When?	What?	Where?	Synch?
Message passing	Programmer: send/receive	Message buffer	Processor list	Yes
CC shared memory	System: load, store	Cache line	Broadcast/directory	No
Stanford TCC	Programmer: transactions	Write set	Broadcast/directory	Yes
Illinois Bulk	System: chunks	Write set	Broadcast/directory	No
Update_all	Programmer: **update_all**	Write set	Broadcast/directory	No
Update_some	Programmer: **update_some**	Write set	Processor list	No
Update_mem	Programmer: **update_mem**	Write set	memory	No
Data_update	Programmer: **data_update**	Address list	Broadcast, processor list, memory	No

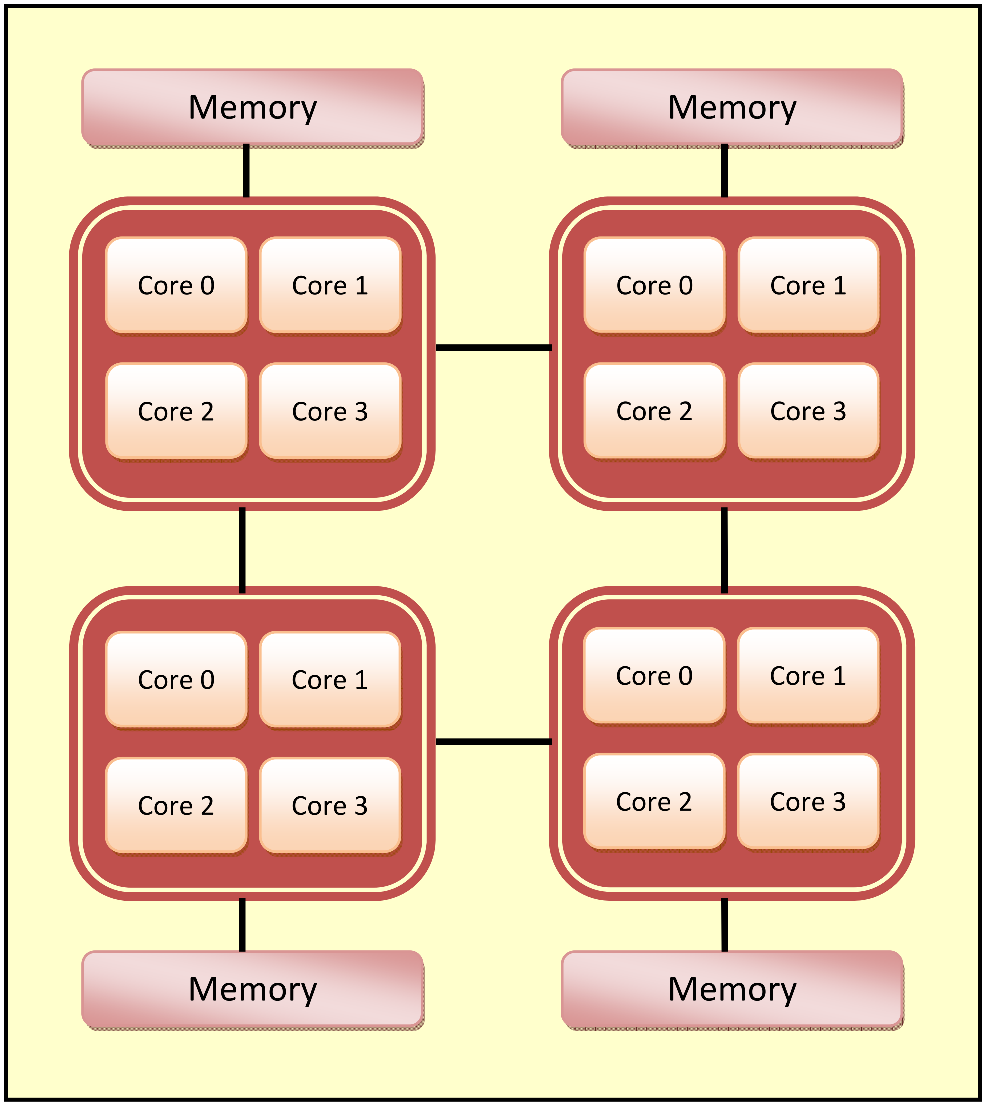
Architecture Research Methodology

- Conventional approaches are useful
  - Develop app & SW system on existing platforms
    - Multi-core, accelerators, clusters, …
  - Simulate novel HW mechanisms
    - No realistic software

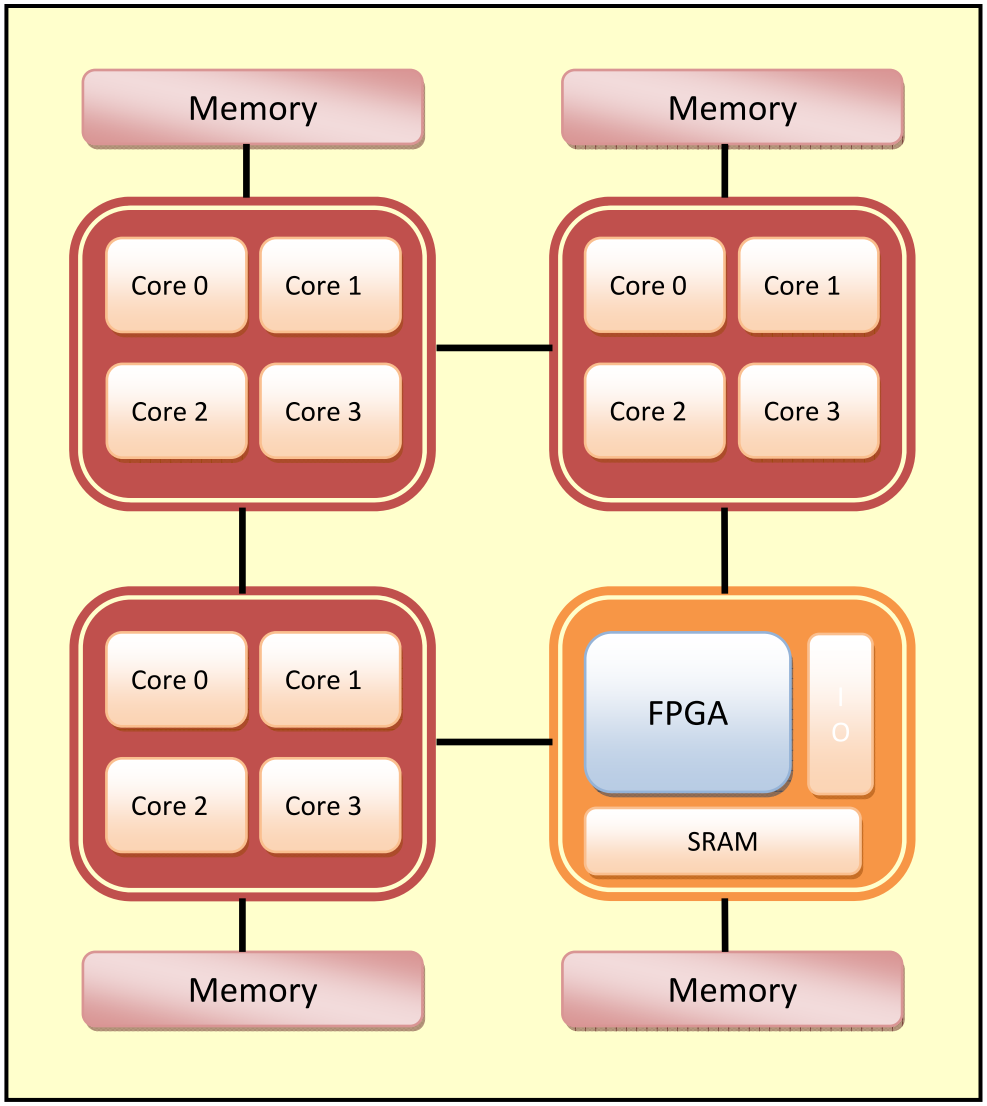
- Need some method that bridges HW & SW research
  - Makes new HW features available for SW research
  - Does not compromise HW speed, SW features, or scale
  - Allows for full-system prototypes
    - Needed for research, convincing for industry, exciting for students

- Approach: commodity chips + FPGAs in memory system
  - Commodity chips: fast system with rich SW environment
  - FPGAs: prototyping platform for new HW features
  - Scale through cluster arrangement

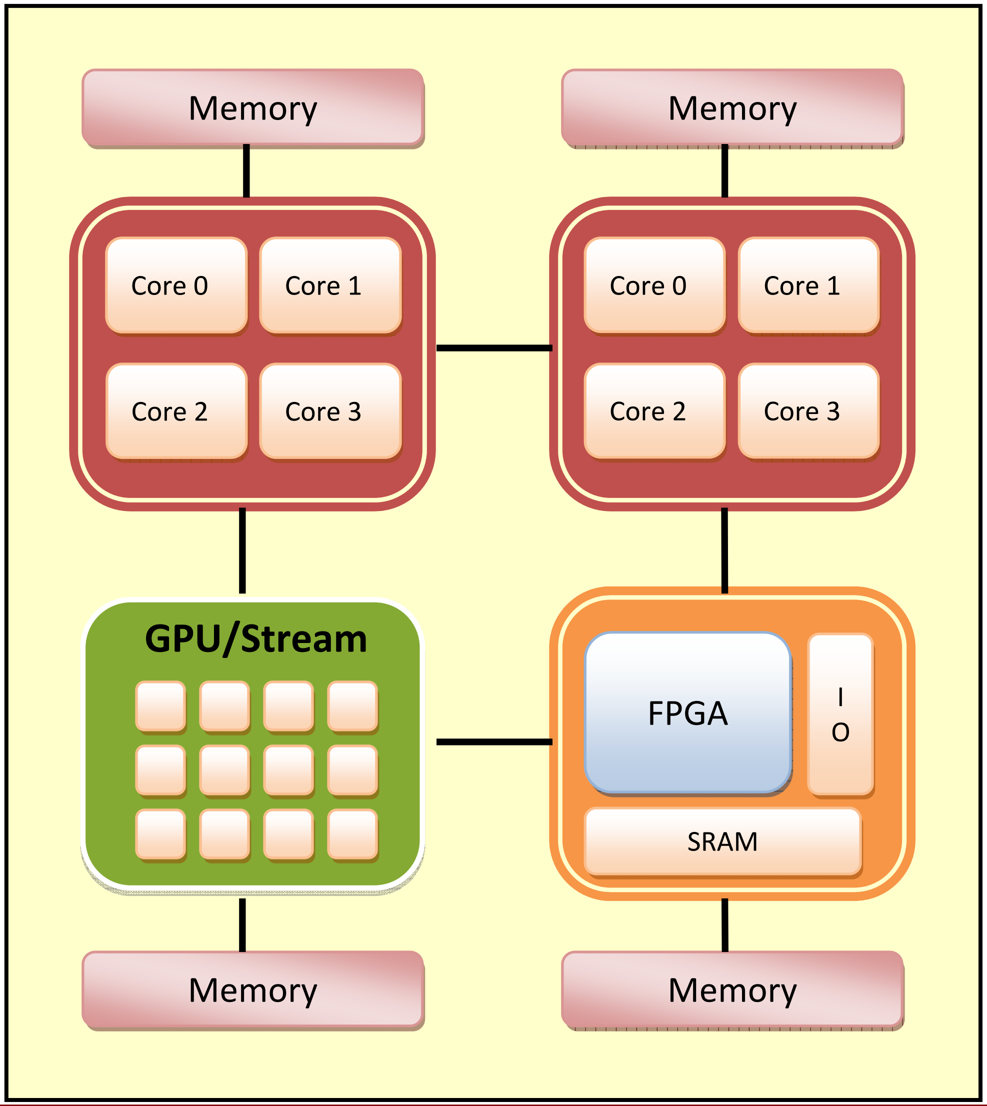
FARM: Flexible Architecture Research Machine



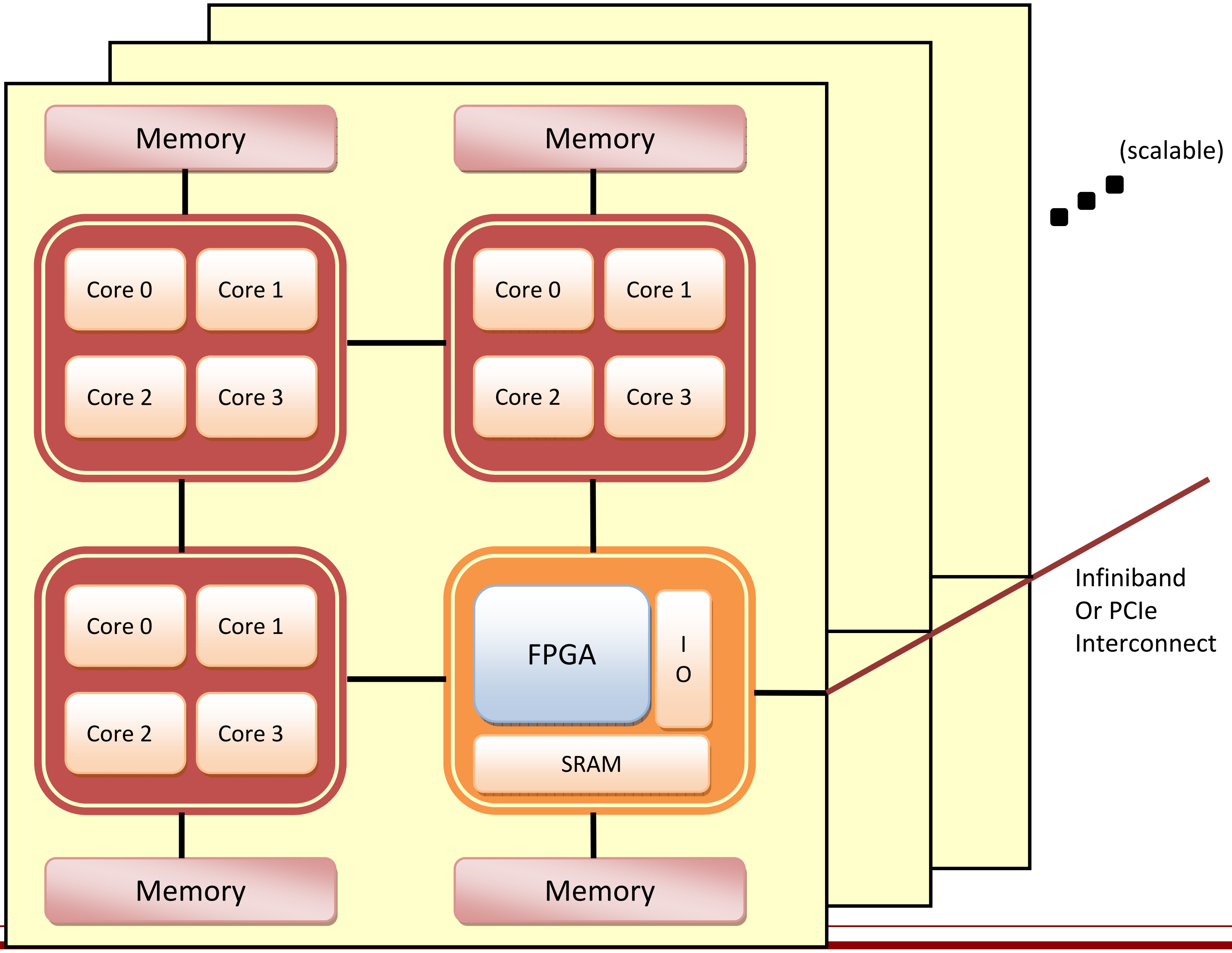
FARM: Flexible Architecture Research Machine



FARM: Flexible Architecture Research Machine

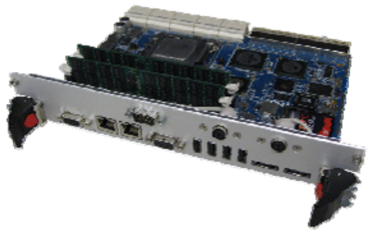
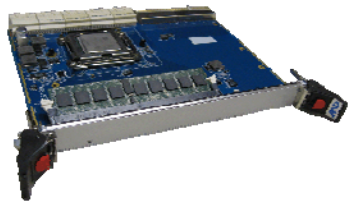


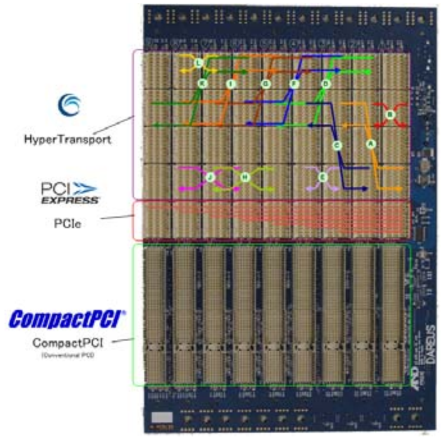
FARM: Flexible Architecture Research Machine



FARM Prototype Procyon System

- Hardware platform for FARM
- From A&D Technology, Inc.
- Full system board
  - AMD Opteron Socket F
  - Two DDR2 DIMMs
  - USB/eSATA/VGA/GigE
  - Sun OpenSolaris OS
- Extra CPU board
  - AMD Opteron Socket F
- FPGA Board
  - Altera Stratix II FPGA
- All connected via HT backplane
  - Also provides PCIe and PCI



Conclusions

- Need a full system vision for pervasive parallelism
  - Applications, programming models, programming languages, software systems, and hardware architecture

- Key initial ideas
  - Domain-specific languages
  - **Programmable memory systems (TM)**
  - Real system prototypes