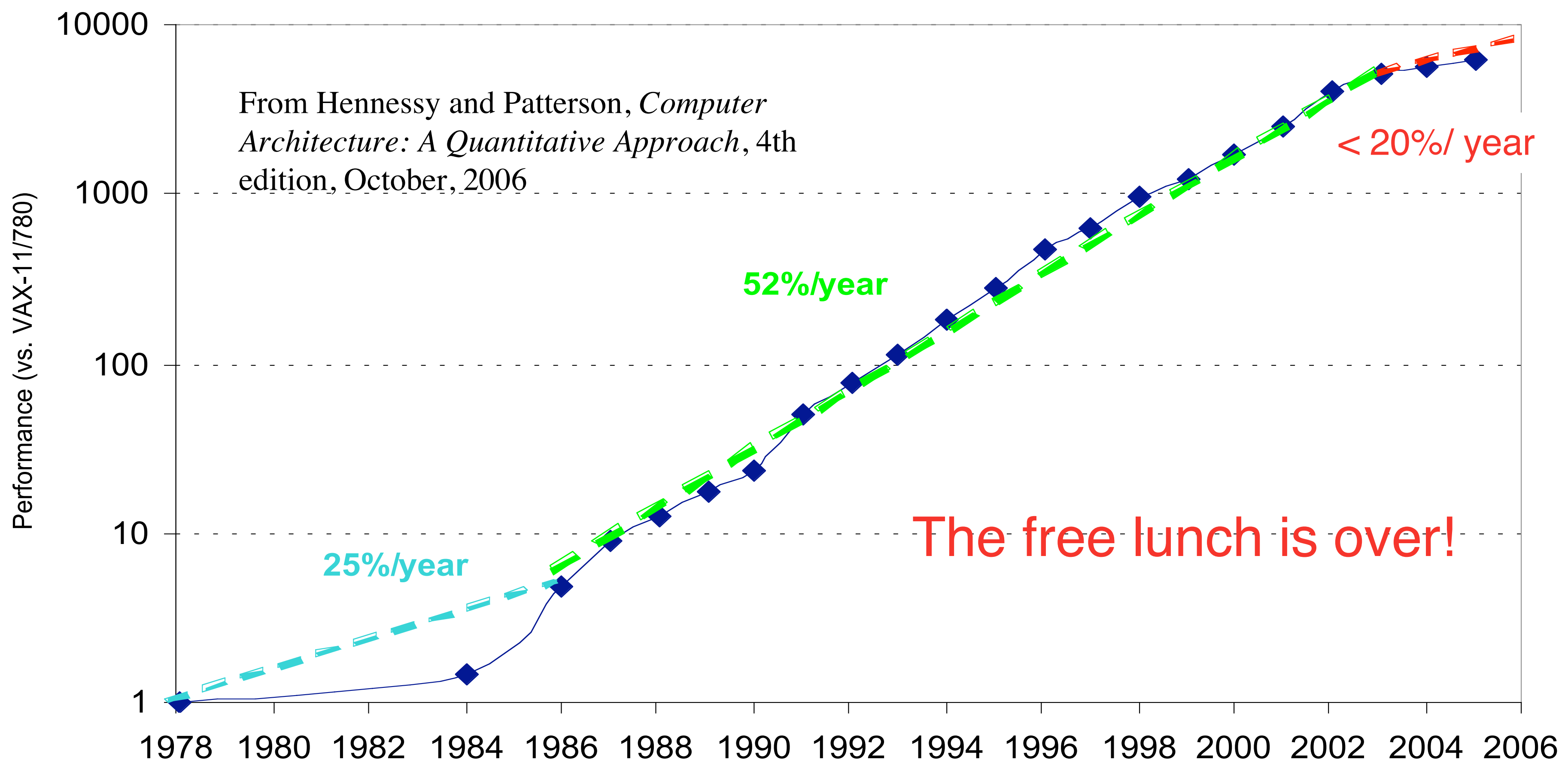
Towards Pervasive Parallelism

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University

UT Austin, October 2008

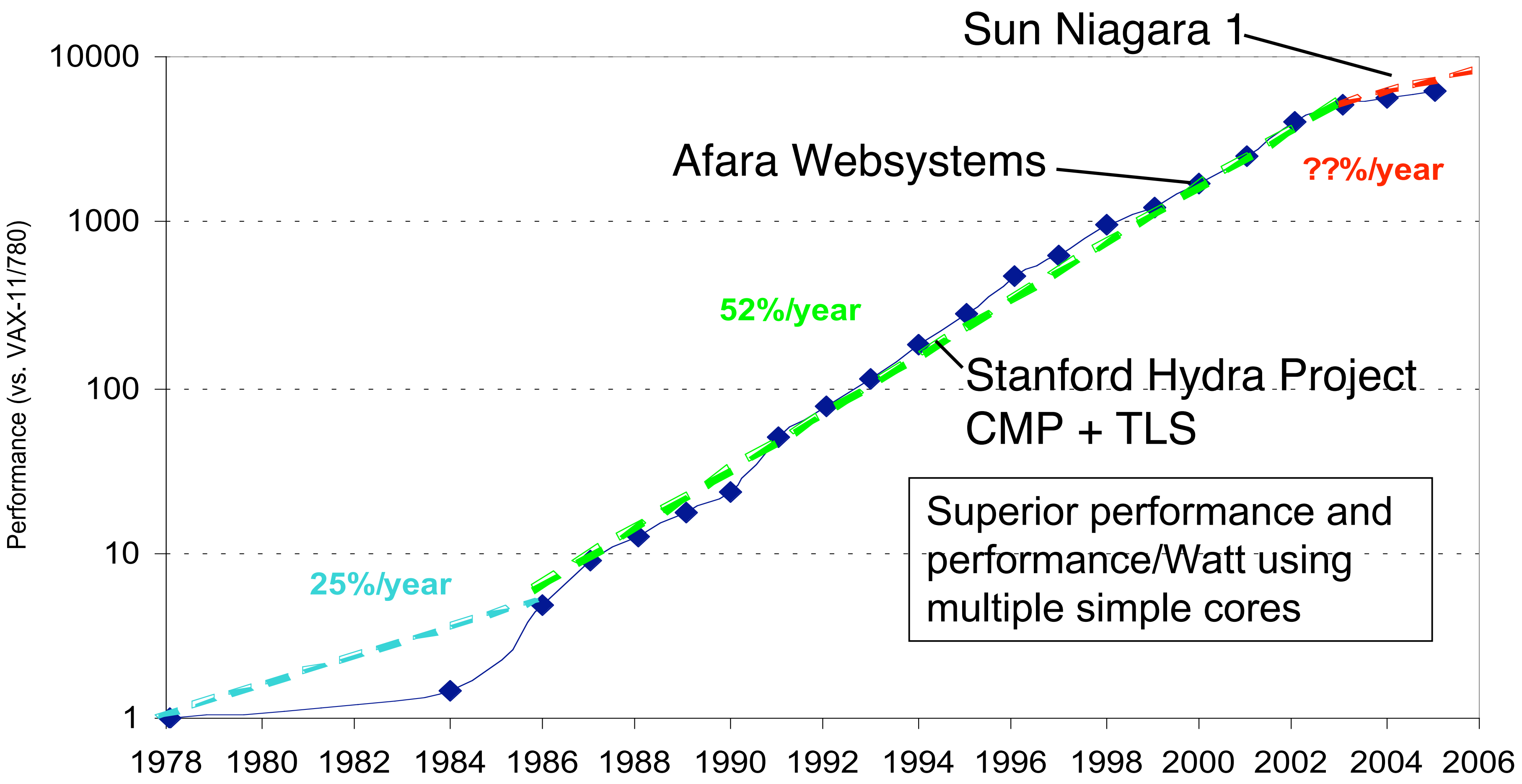
End of Uniprocessor Performance



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, October, 2006

The free lunch is over!

Predicting The End of Uniprocessor Performance



Superior performance and performance/Watt using multiple simple cores

The Looming Crisis

- Software developers will soon face systems with
  - > 1 TFLOP of compute power
  - 32+ of cores, 100+ hardware threads
  - Heterogeneous cores (CPU+GPUs), app-specific accelerators
  - Deep memory hierarchies
- Challenge: harness these devices productively
  - Improve performance, power, reliability and security
- The parallelism gap
  - Threads, locks, messages
    - Pthreads, OpenMP, MPI
  - Too difficult find parallelism, to debug, maintain and get good performance for the masses
  - Yawning divide between the capabilities of today's programming environments, the requirements of emerging applications, and the challenges of future parallel architectures

The Stanford Pervasive Parallelism Laboratory

- Goal: the parallel computing platform for 2012
  - Make parallel application development practical for the masses
  - Not parallel programming as usual

- 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 Team

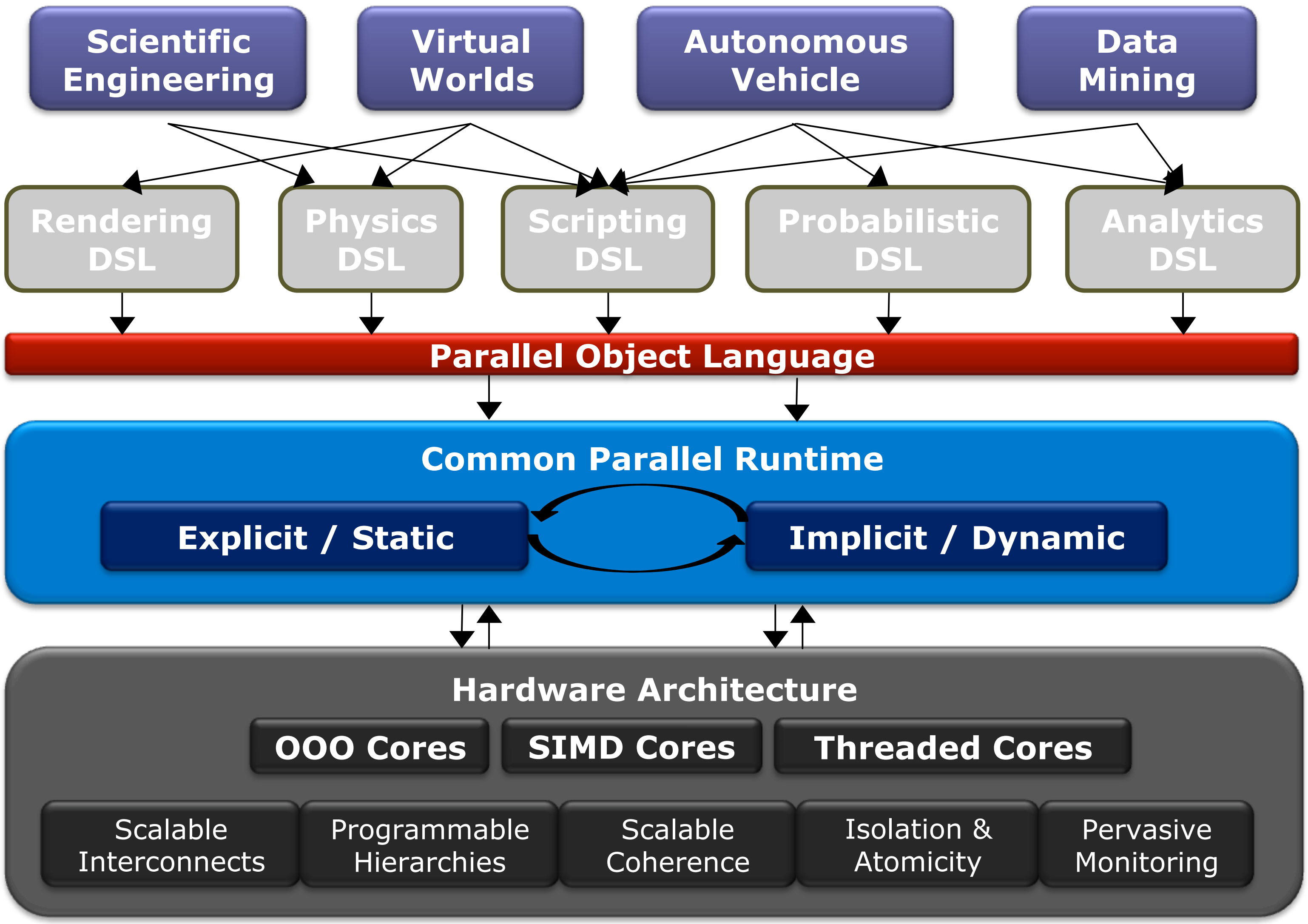
- Applications
  - Ron Fedkiw, Vladlen Koltun, Sebastian Thrun
- Programming & software systems
  - Alex Aiken, Pat Hanrahan, Mendel Rosenblum
- Architecture
  - Bill Dally, Mark Horowitz, Christos Kozyrakis, Kunle Olukotun (Director), John Hennessy

John Hennessy’s View on Future of Parallelism

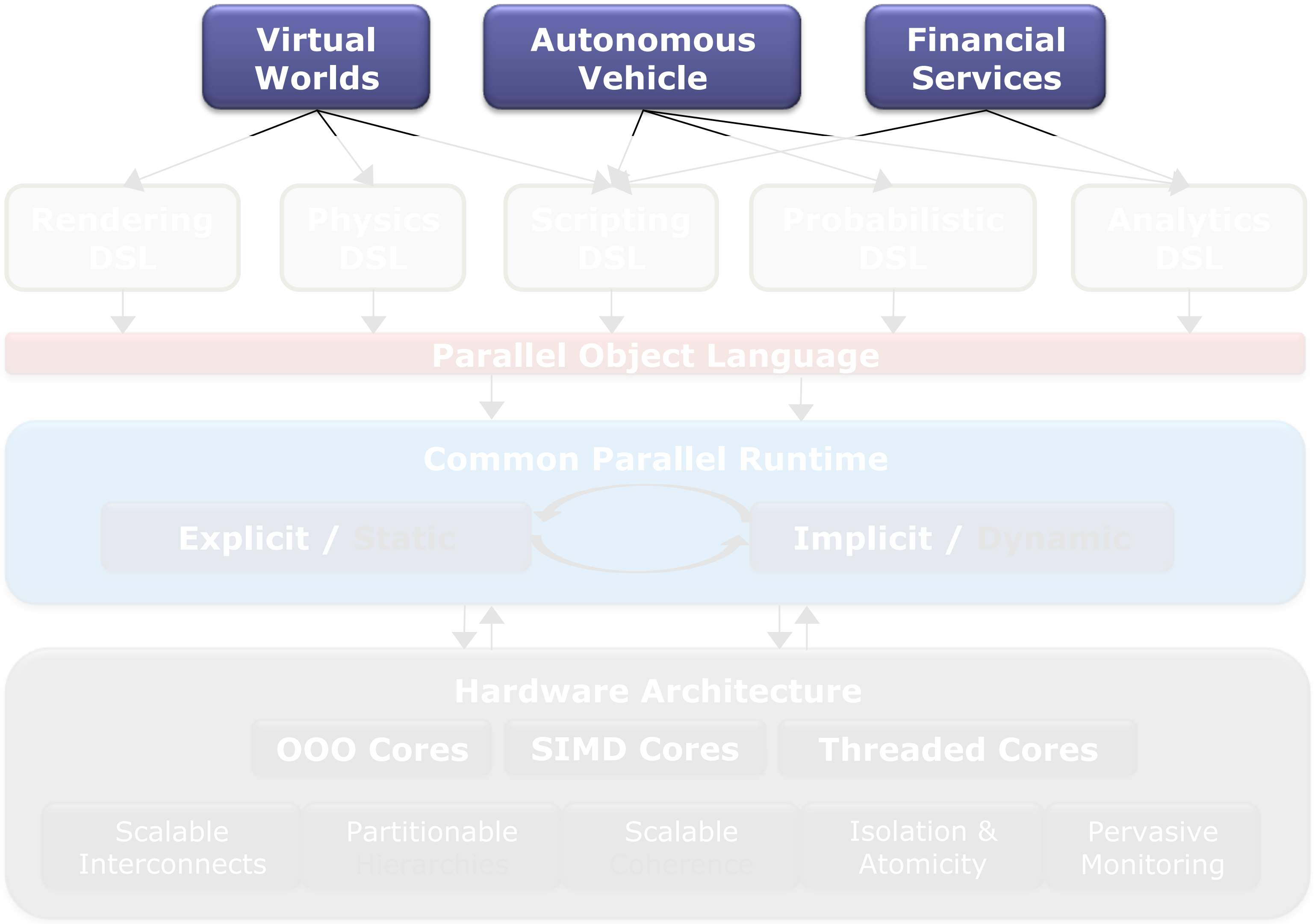
- We are ten years behind and need to catch up
- Don't look to the methods developed for high-end scientific computing to solve the problem
  - 10 procs. up instead of 10K procs. down
- Don’t Focus on scientific and engineering apps
  - These will be bulk of new applications and programmers
- Don't focus on absolute parallel efficiency
  - Focus on ease of use for programmer

Need a change: Parallel applications without parallel programming

The PPL Vision



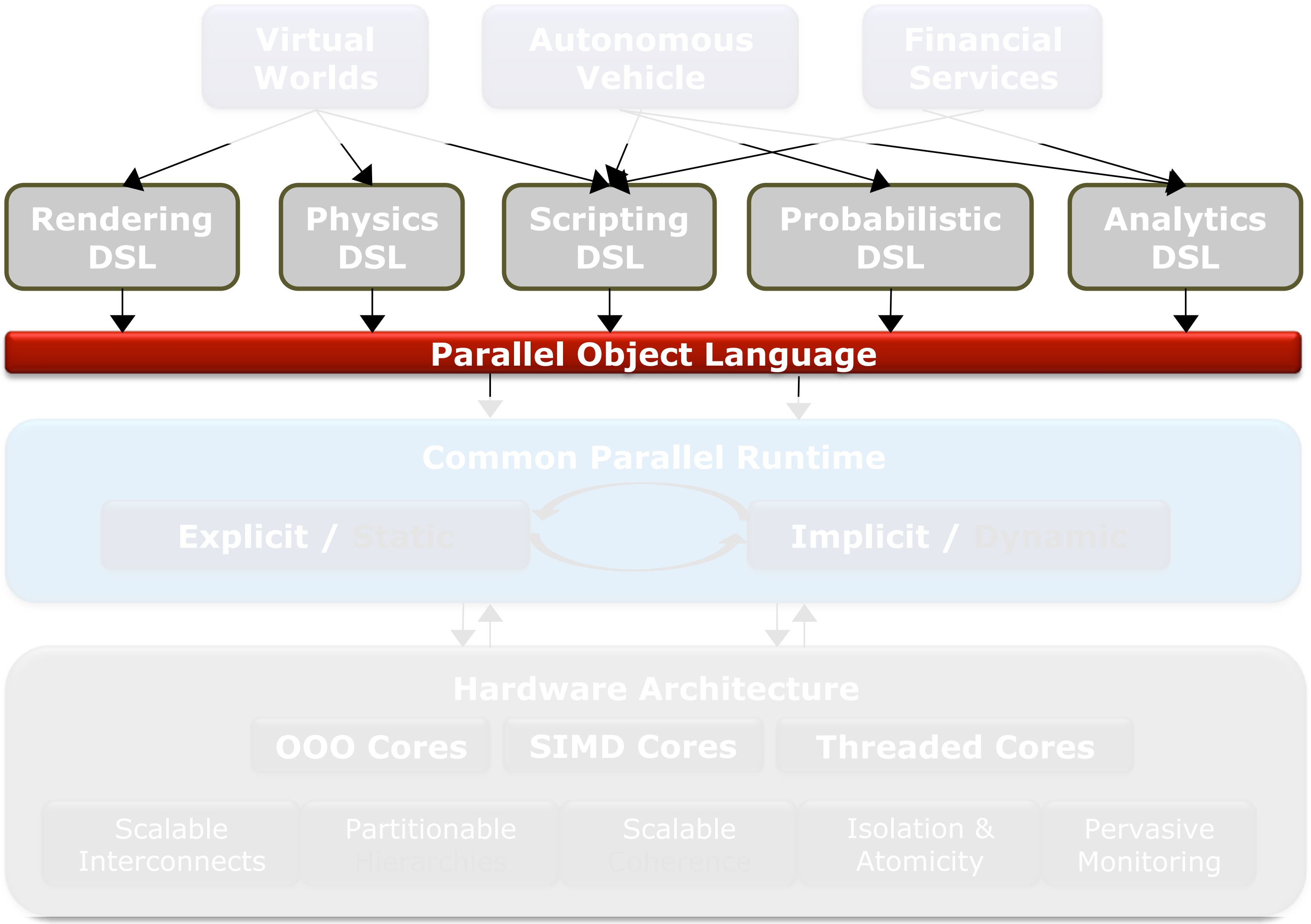
The PPL Vision



Virtual Worlds Application

- Next-gen web platform
  - Immersive collaboration
  - Social gaming
  - Millions of players in vast landscape
- Parallelism challenges
  - Client-side game engine
  - Server-side world simulation
  - AI, physics, large-scale rendering
  - Dynamic content, huge datasets
- More at http://vw.stanford.edu/

The PPL Vision



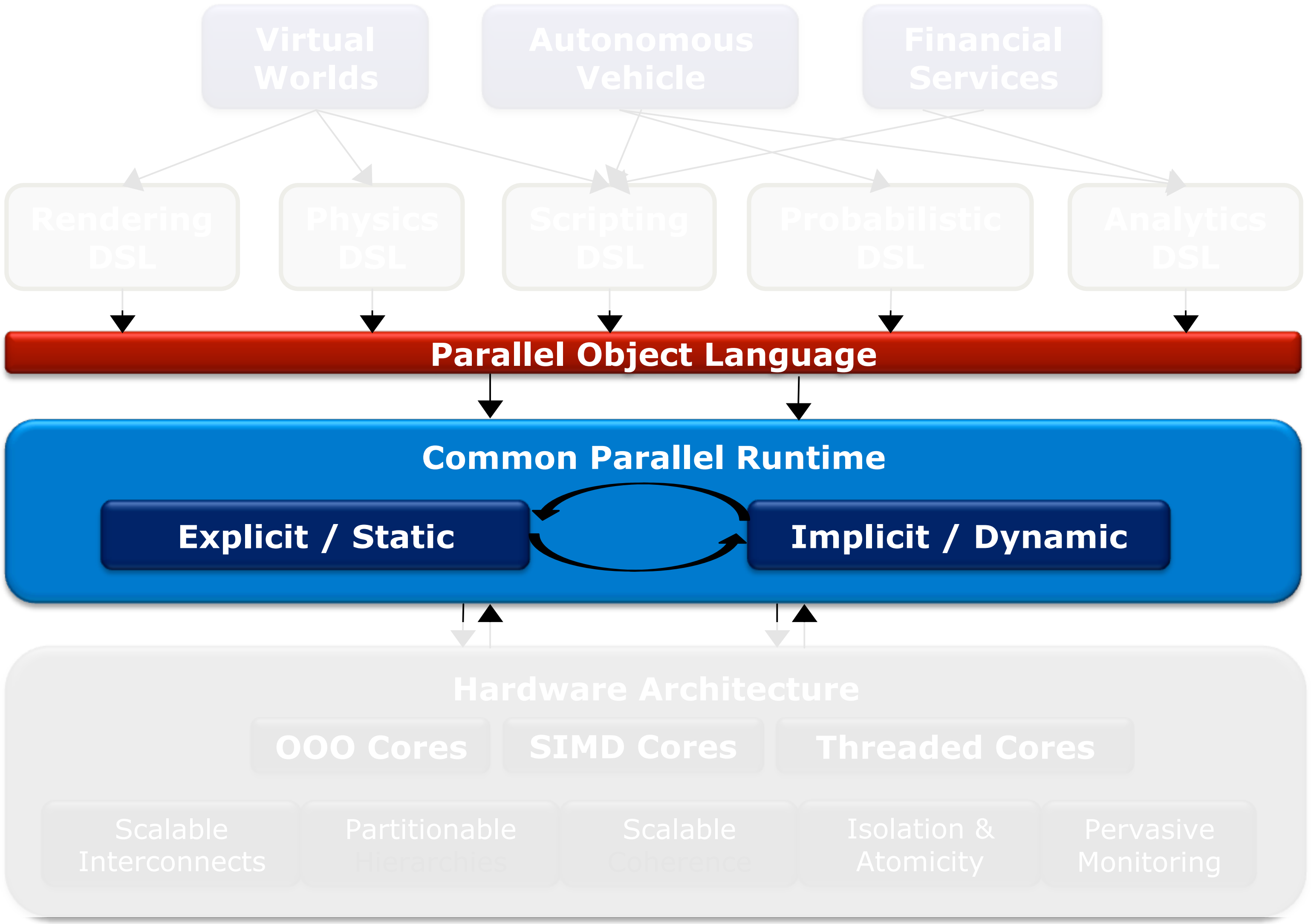
Domain Specific Languages (DSL)

- Leverage success of DSL across application domains
  - SQL (data manipulation), Matlab (scientific), Ruby/Rails (web),…
- DSLs ⇒ higher productivity for developers
  - High-level data types & ops tailored to domain
    - E.g., relations, matrices, triangles, …
  - Express high-level intent without specific implementation artifacts
    - Programmer isolated from details of specific system
- DSLs ⇒ scalable parallelism for the system
  - Allows aggressive optimization
  - Declarative description of parallelism & locality patterns
    - E.g., ops on relation elements, sub-array being processed, …
  - Portable and scalable specification of parallelism
    - Automatically adjust data structures, mapping, and scheduling as systems scale up

DSL Research

- Goal: create framework for DSL development
- Initial DSL targets
  - Rendering, physics simulation, probabilistic machine learning computations
- Approach
  - DSL implementation ⇒ embed in base PL
    - Start with Scala (OO, type-safe, functional, extensible)
    - Use Scala as a scripting DSL that also ties multiple DSLs
  - DSL-specific optimizations ⇒ active libraries
    - Use domain knowledge to optimize & annotate code

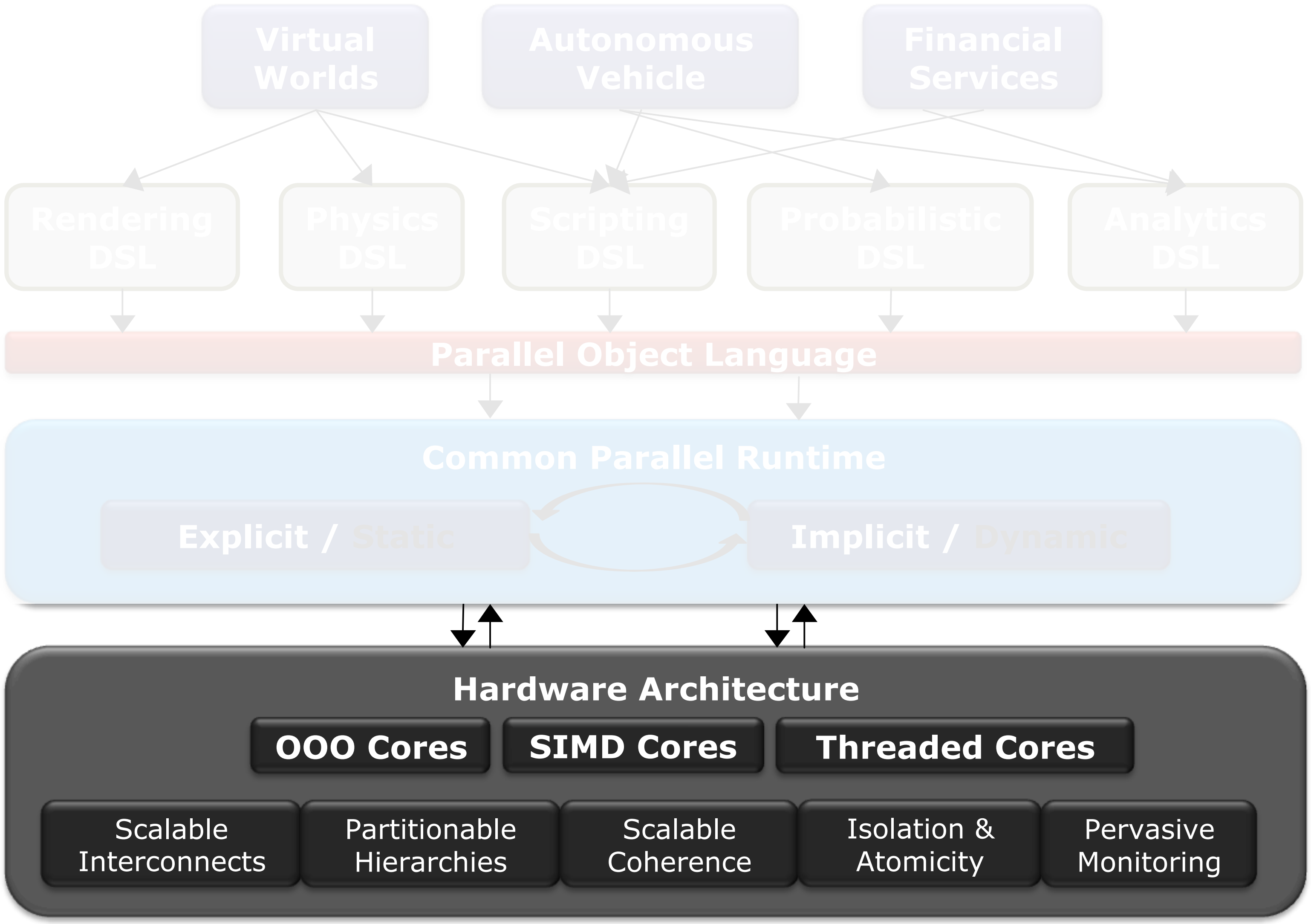
The PPL Vision



Common Parallel Runtime (CPR)

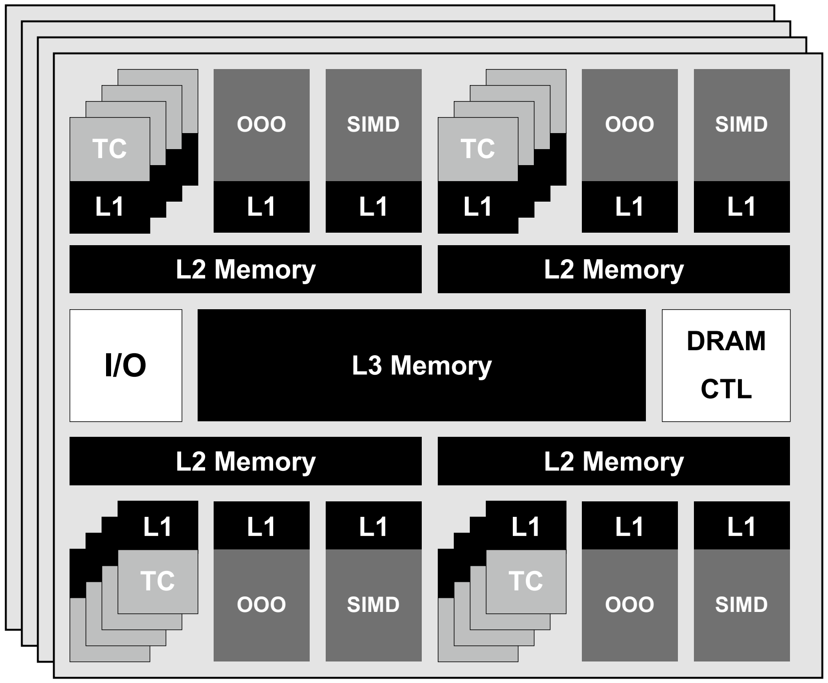
- Goals
  - Provide common, portable, abstract target for all DSLs
  - Manages parallelism & locality
    - Achieve efficient execution (performance, power, …)
    - Handles specifics of HW system
- Approach
  - Compile DSLs to common IR
    - Base language + low-level constructs & pragmas
      - Forall, async/join, atomic, barrier, …
    - Per-object capabilities
      - Read-only or write-only, output data, private, relaxed coherence, …
  - Combine static compilation + dynamic management
    - Static management of regular tasks & predictable patterns
    - Dynamic management of irregular parallelism

The PPL Vision



Hardware Architecture @ 2012

- The many-core chip
  - 100s of cores
    - OOO, threaded, & SIMD
  - Hierarchy of shared memories
  - Scalable, on-chip network
- The system
  - Few many-core chips
  - Per-chip DRAM channels
  - Global address space
- The data-center
  - Cluster of systems



Architecture Research

- Revisit architecture & microarchitecture for parallelism
  - Define semantics & implementation of key primitives
  - Communication, atomicity, isolation, partitioning, coherence, consistency, checkpoint
  - Fine-grain & bulk support
- Software-managed HW primitives
  - hardware provides key mechanisms, software synthesizes into useful execution systems
  - Exploit high-level knowledge from DSLs & CPR
- Software synthesizes primitives into execution systems
  - Streaming system: partitioning + bulk communication
  - Thread-level spec: isolation + fine-grain communication
  - Transactional memory: atomicity + isolation + consistency
  - Security: partitioning + isolation
  - Fault tolerance: isolation + checkpoint + bulk communication
- Challenges: interactions, scalability, cost, virtualization
  - 100s to 100s of cores

Architecture Research

- Revisit architecture & microarchitecture for parallelism
  - Define semantics & implementation of key primitives
  - Communication, atomicity, isolation, partitioning, coherence, consistency, checkpoint
  - Fine-grain & bulk support
- Software-managed HW primitives
  - hardware provides key mechanisms, software synthesizes into useful execution systems
  - Exploit high-level knowledge from DSLs & CPR
- Software synthesizes primitives into execution systems
  - Streaming system: partitioning + bulk communication
  - Thread-level spec: isolation + fine-grain communication
  - $\boxed{\text{Transactional memory: atomicity + isolation + consistency}}$
  - Security: partitioning + isolation
  - Fault tolerance: isolation + checkpoint + bulk communication
- Challenges: interactions, scalability, cost, virtualization
  - 100s to 100s of cores

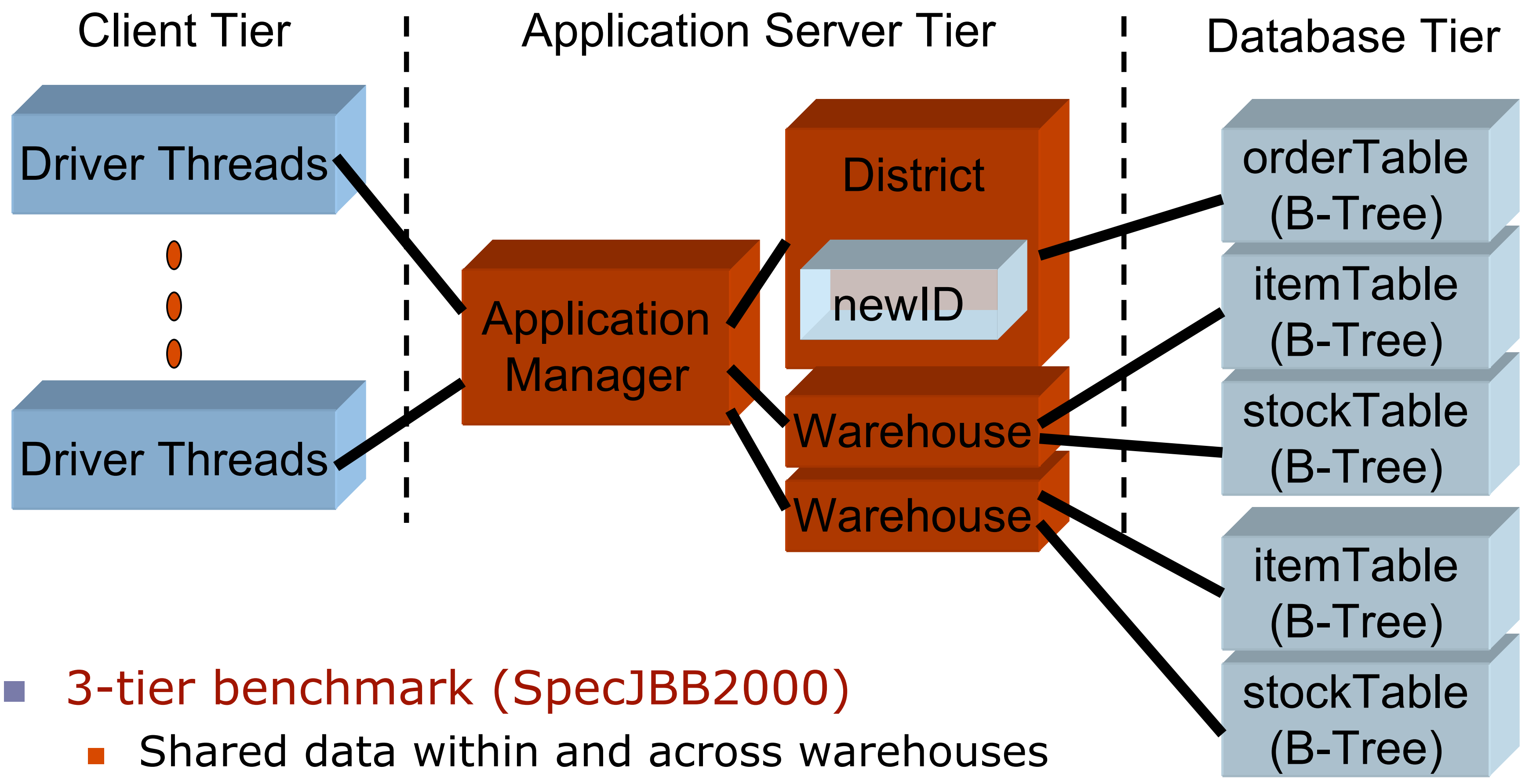
Transactional Memory (TM)

- 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

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

TM Example: 3-tier Server



- 3-tier benchmark (SpecJBB2000)
  - Shared data within and across warehouses
- Parallelized actions within one warehouse
  - Orders, payments, delivery updates, etc on shared data

Sequential Code for NewOrder

```
TransactionManager::go() {
      // 1. initialize a new order transaction
      newOrderTx.init();
      // 2. create unique order ID
      orderId = district.nextOrderId(); // newID++
      order = createOrder(orderId);
      // 3. retrieve items and stocks from warehouse
      warehouse = order.getSupplyWarehouse();
      item = warehouse.retrieveItem();   // B-tree search
      stock = warehouse.retrieveStock(); // B-tree search
      // 4. calculate cost and update node in stockTable
      process(item, stock);
      // 5. record the order for delivery
      district.addOrder(order); // B-tree update
      // 6. print the result of the process
      newOrderTx.display();
}
```

- Non-trivial code with complex data-structures
  - Fine-grain locking ➔ difficult to get right
  - Coarse-grain locking ➔ no concurrency

TM Code for NewOrder

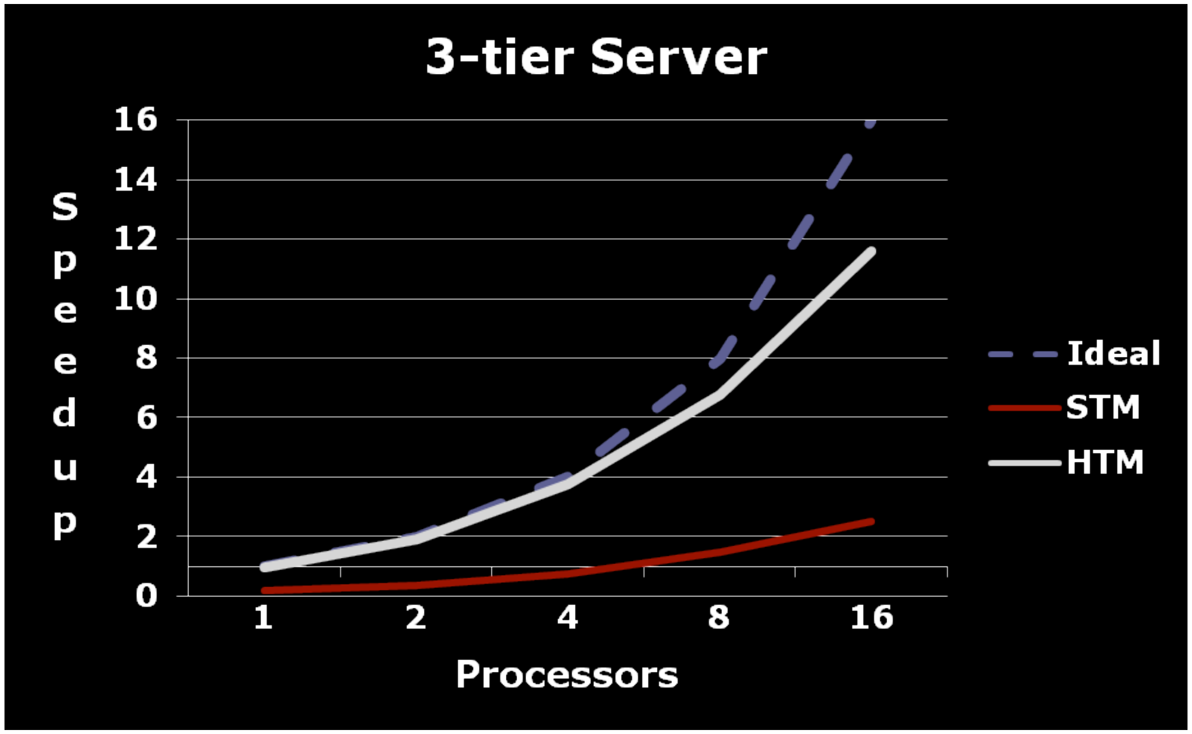
```
TransactionManager::go() {
    atomic { // begin transaction
      // 1. initialize a new order transaction
      // 2. create a new order with unique order ID
      // 3. retrieve items and stocks from warehouse
      // 4. calculate cost and update warehouse
      // 5. record the order for delivery
      // 6. print the result of the process
    } // commit transaction
}
```

- Whole NewOrder as one atomic transaction
  - 2 lines of code changed for parallelization
  - No need to analyze storage scheme, ordering issues, …

Implementing Memory Transactions

- Data versioning for updated data
  - Manage new & old values for memory data
  - *Deferred updates* $\boxed{\text{(lazy)}}$ vs *direct updates* (eager)
- 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 $\boxed{(\textit{optimistic})}$
- 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

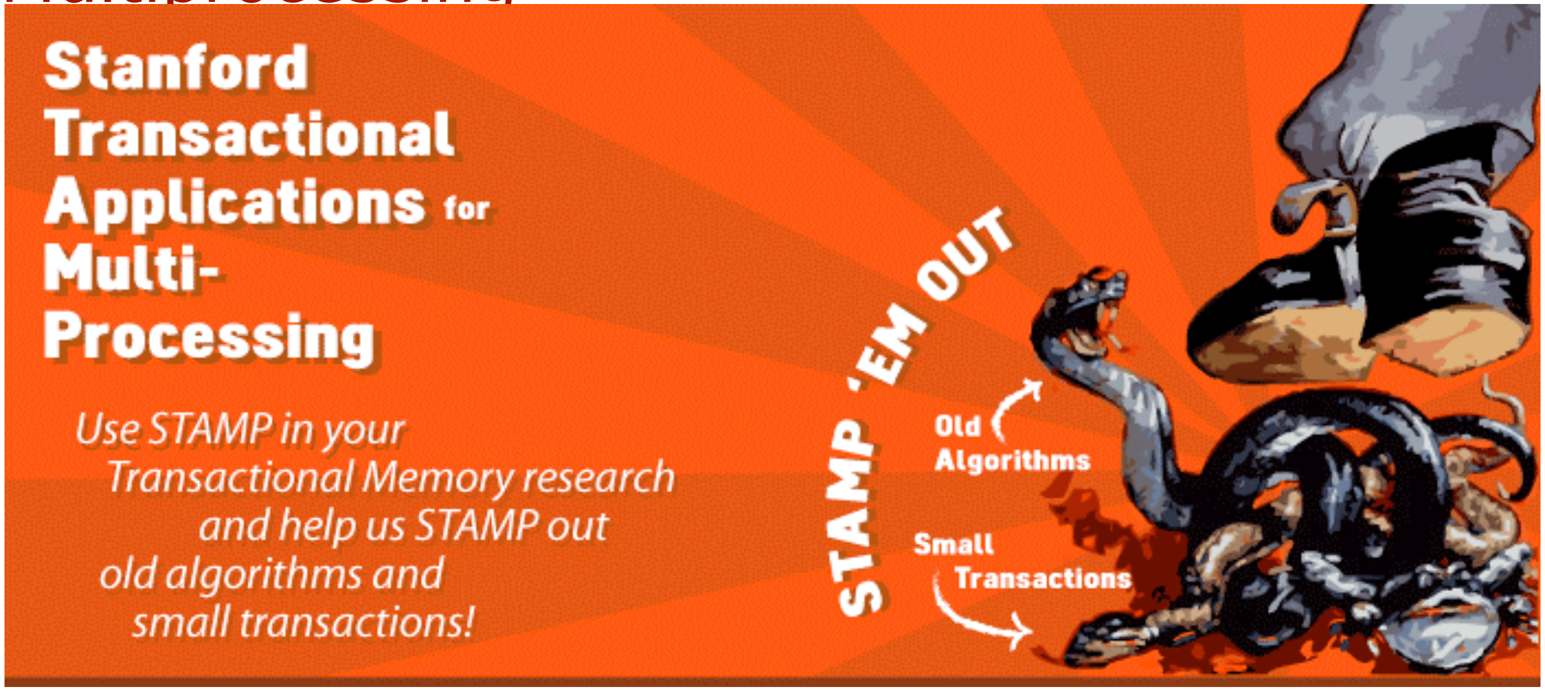
Performance with Hardware TM



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

STAMP Benchmark Suite

- Stanford Transactional Applications for Multiprocessing



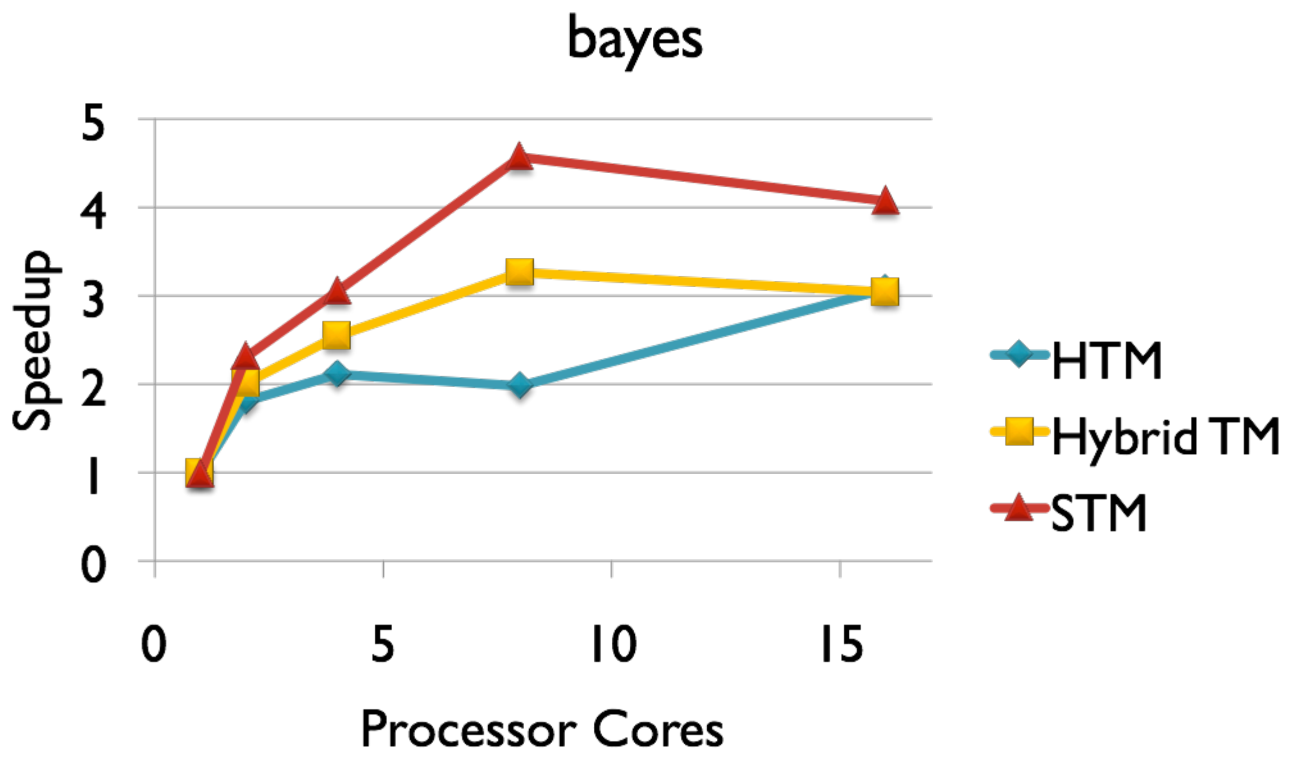
Stanford Transactional Applications for Multi-Processing
Use STAMP in your Transactional Memory research and help us STAMP out old algorithms and small transactions!
STAMP 'EM OUT
Old Algorithms
Small Transactions

- 8 applications from variety of domains
- http://stamp.stanford.edu

STAMP Applications

Application	Domain	Description
bayes	Machine learning	Learns structure of a Bayesian network
genome	Bioinformatics	Performs gene sequencing
intruder	Security	Detects network intrusions
kmeans	Data mining	Implements K-means clustering
labyrinth	Engineering	Routes paths in maze
ssca2	Scientific	Creates efficient graph representation
vacation	Online transaction processing	Emulates travel reservation system
yada	Scientific	Refines a Delaunay mesh

Pure HTMs Have Limitations

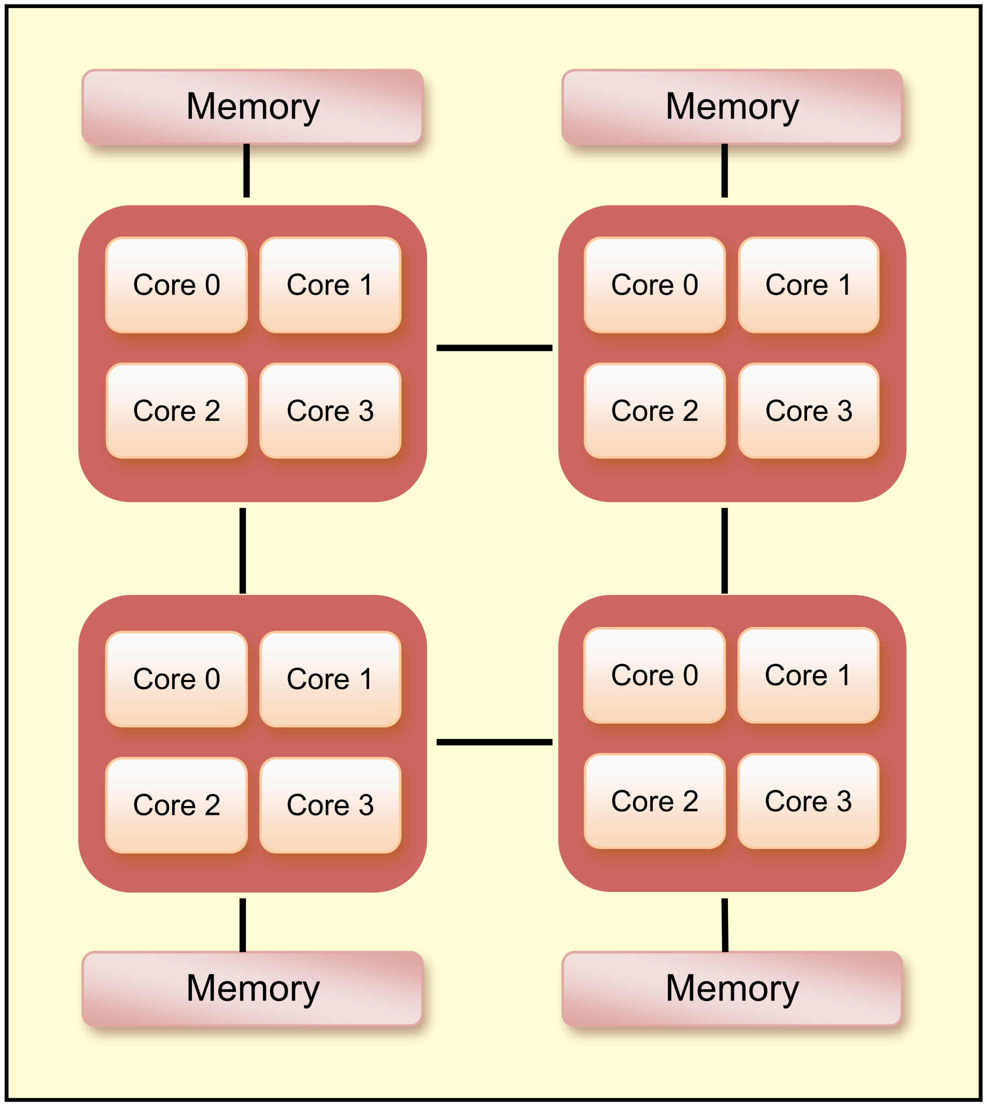


- Conflict detection granularity can be important

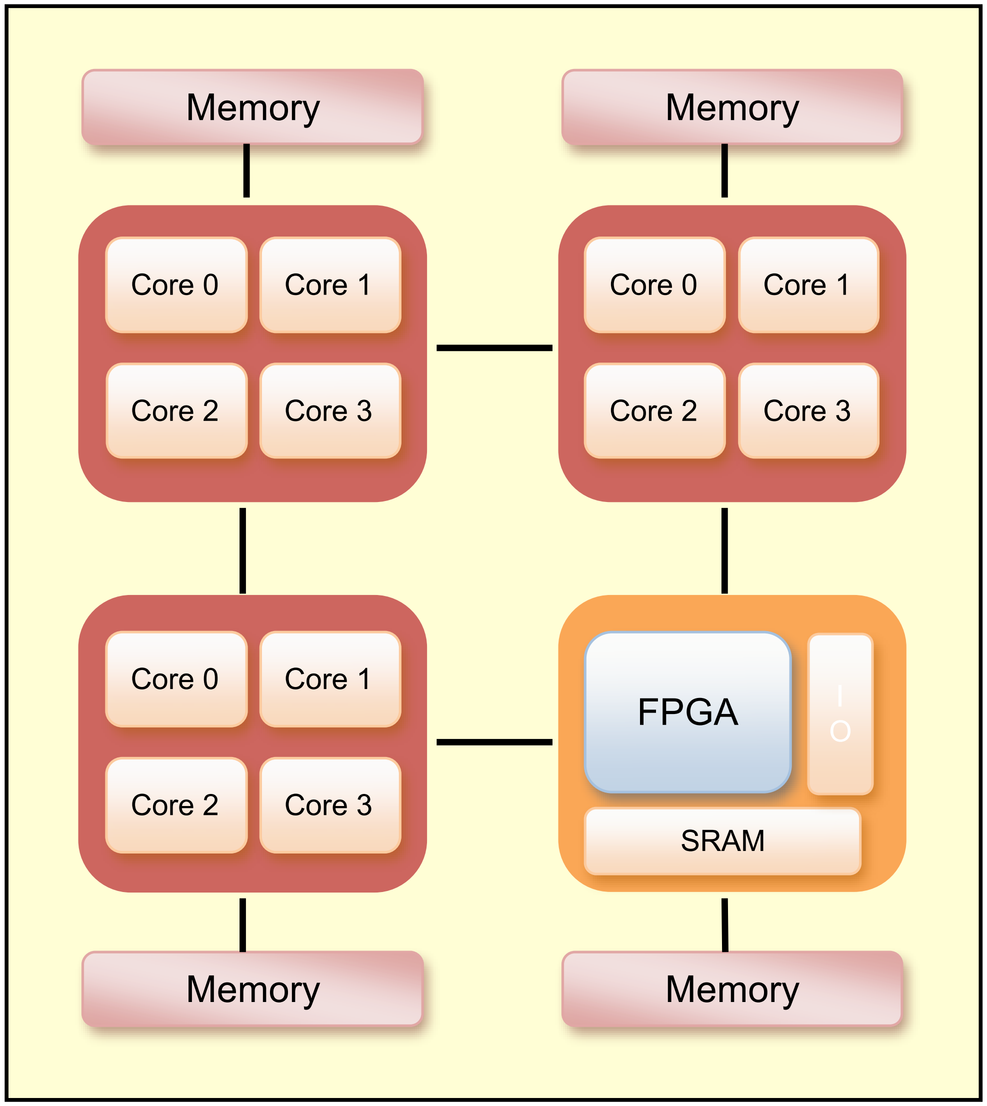
Architecture Research Methodology

- Conventional approaches are useful
  - Develop app & SW system on existing platforms
    - Multi-core, accelerators, clusters, …
  - Simulate novel HW mechanisms
- 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

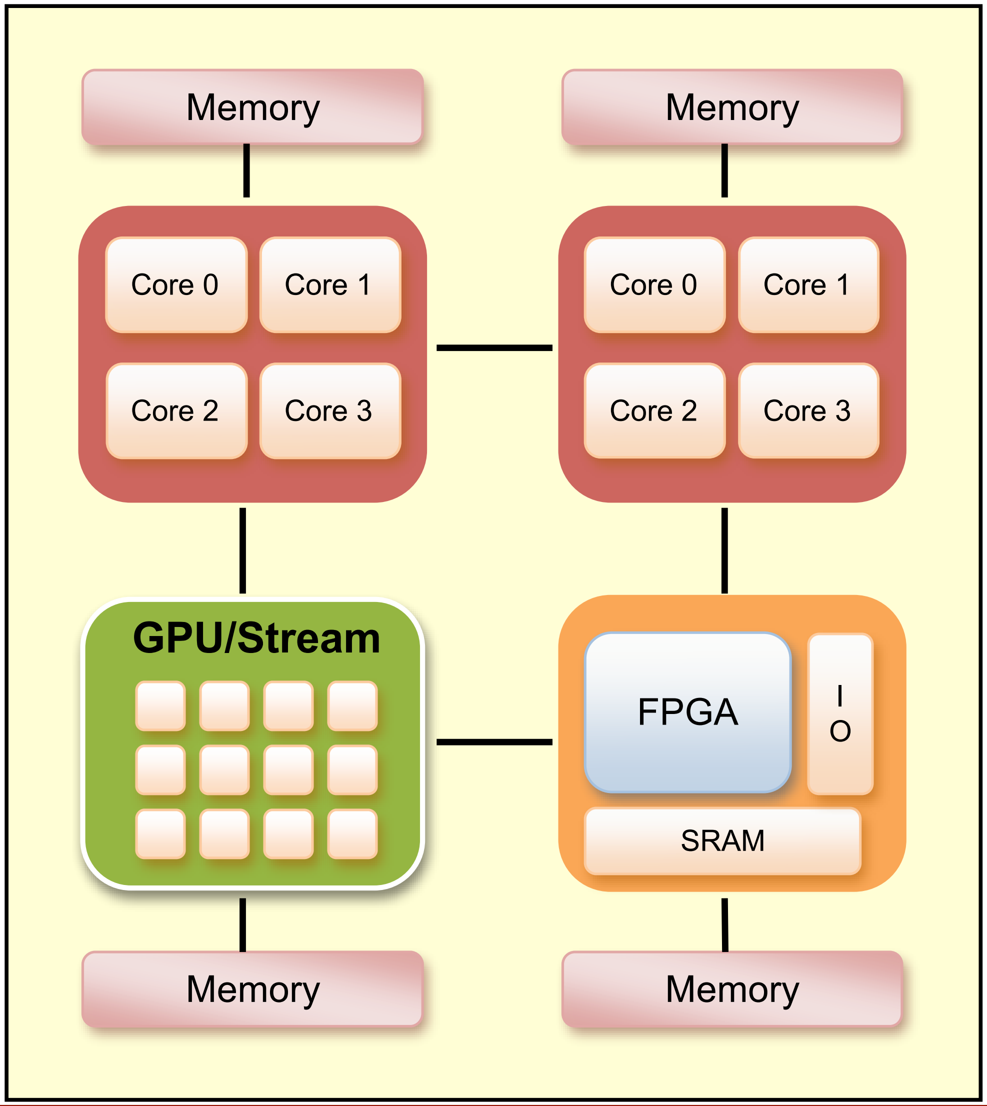
FARM: Flexible Architecture Research Machine



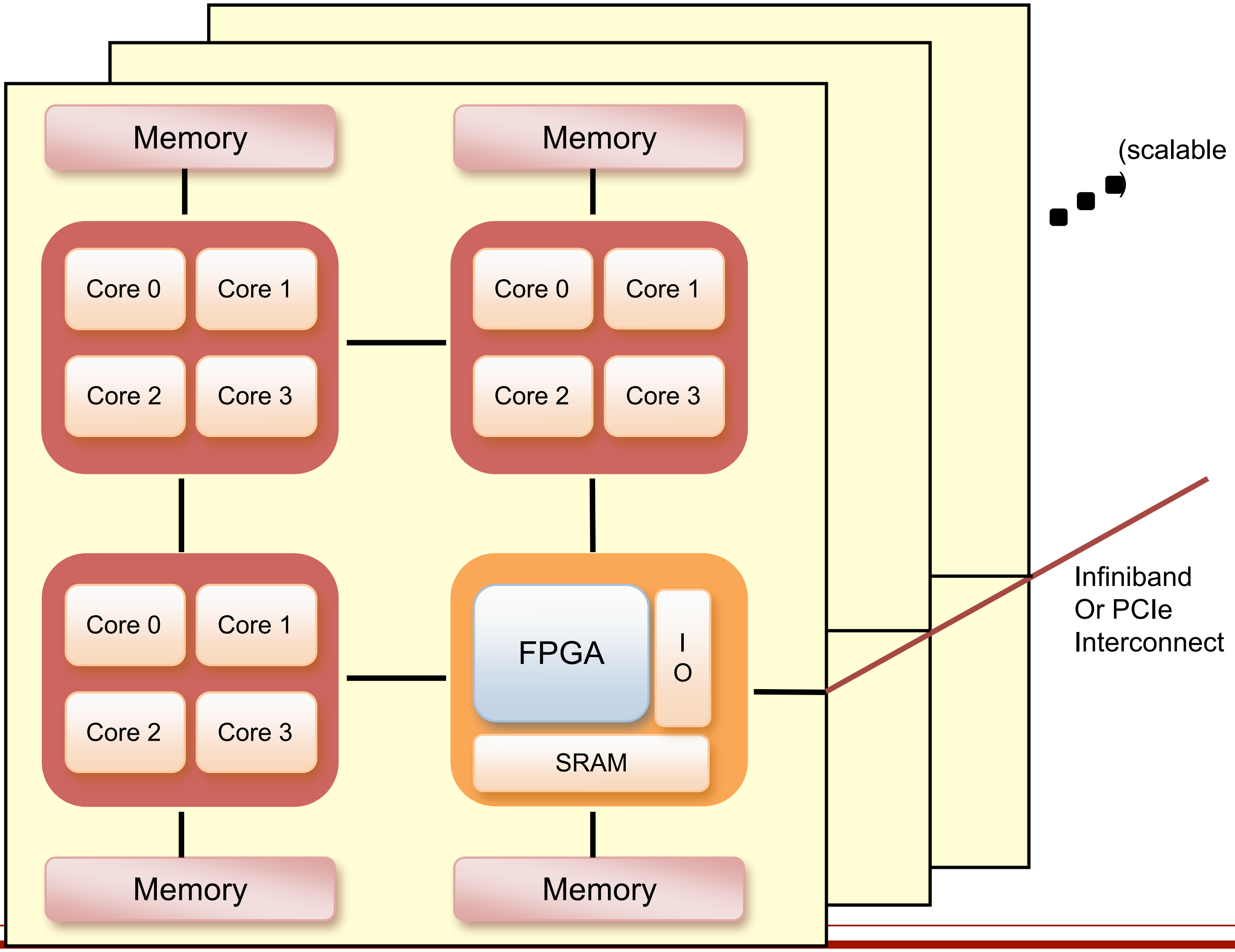
FARM: Flexible Architecture Research Machine



FARM: Flexible Architecture Research Machine

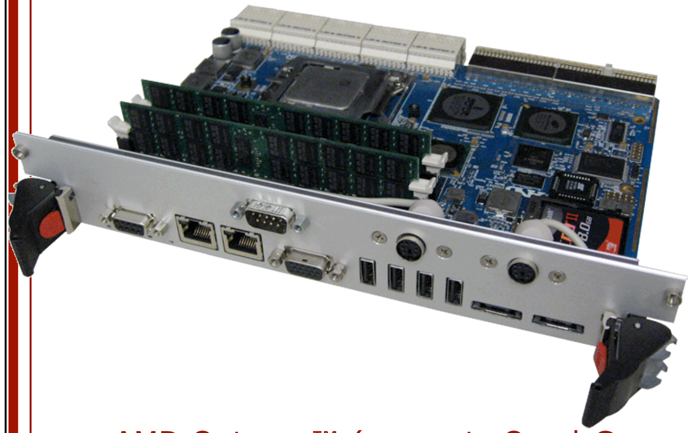


FARM: Flexible Architecture Research Machine



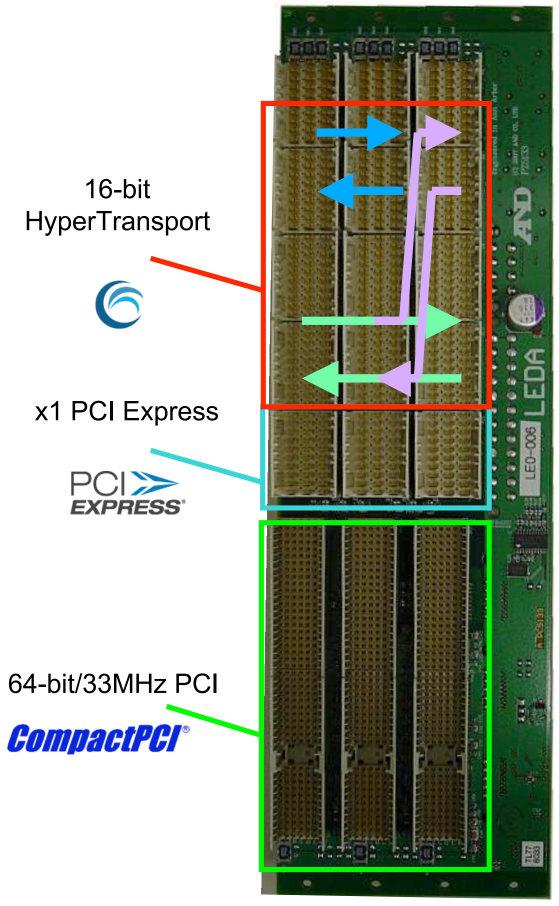
FARM Prototype Procyon System

AD7003 : Single Board computer



- AMD Opteron™ (supports Quad-Core “Barcelona” processor family)
- Broadcom HT2100/HT1000 Chipset
- Connects to peripheral boards via HyperTransport or PCI Express
- DDR2 DIMM x 2
- Gigabit Ethernet x 2
- USB2.0 x 4
- VGA
- SATA and eSATA
- System monitor function
- AMI BIOS
- Supports CompactPCI peripheral boards

Leda : Procyon Evaluation Backplane



Pollux : Procyon Interface Evaluation Board

- Altera FPGA Stratix II
- Interface to backplane
  - Hyper Transport I/F x16 *2
  - x1 PCIe
- Panel I/O Ports
  - 8 DO / 8 DI (TTL)
- DDR2 Memory (32 x 16M bit)

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
  - Combine implicit & explicit resource management
  - Flexible HW primitives
  - Real system prototypes