The Re-design Imperative: Manycore Changes Everything

Bruce Shriver
University of Tromsø, Norway

November 2010
Óbuda University, Hungary

Continuation of the Set of Three Lectures

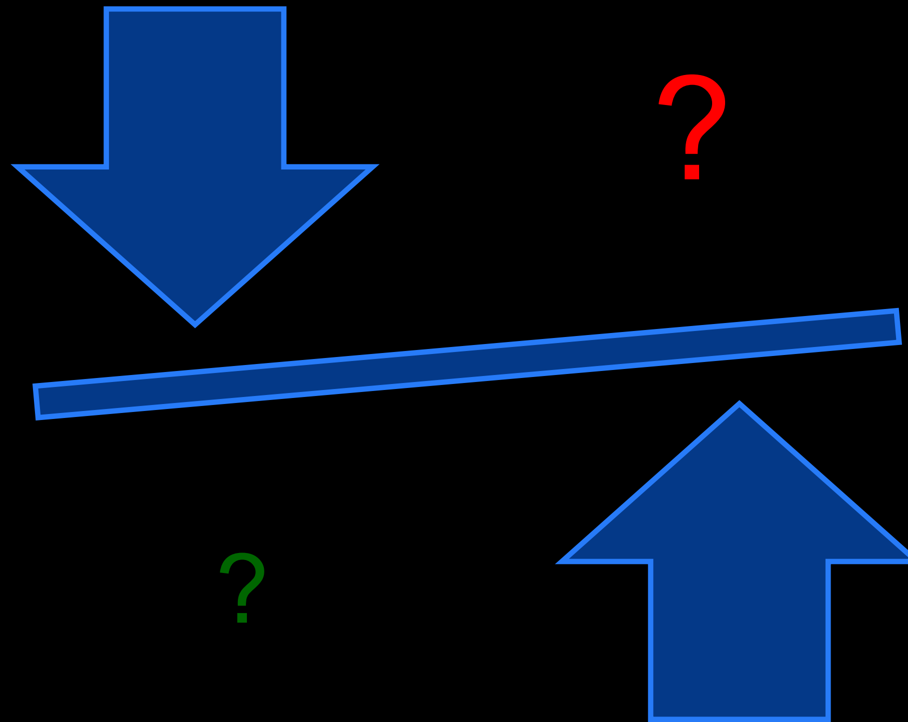
Reconfigurability Issues in Multi-core Systems

- The first lecture explored the thesis that reconfigurability is an integral design goal in multi-/many-core systems.

The Re-Design Imperative: Why Many-core Changes Everything

- The next two lectures explore the impact the multi-/many-core systems have on algorithms, programming language, compiler and operating system support and vice-versa.

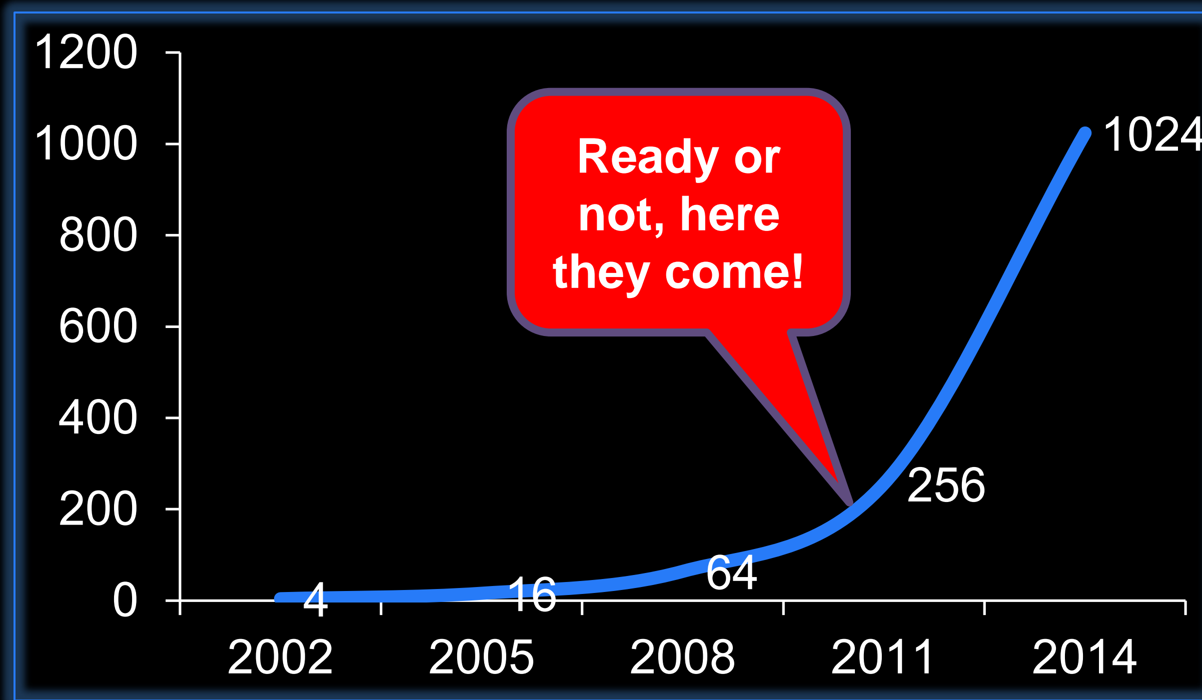
A reminder from the first talk …



These lectures are intended to raise more questions than they answer

The Core is the Logic Gate of the 21st Century Anant Agarwal, MIT

Agarwal proposes a corollary to Moore�означ's law:
The # of cores will double every 18 months



The Tail Wagging the Dog

In 2000, Intel transitioned from the Pentium 3 to the Pentium 4. The transistor count increased by 50%, but the performance only increased by 15%

10s to 100s of cores/chip; the memory wall; the ILP complexity and performance wall, the power and thermal wall; the education wall

Increasing complexity in a parallel world: development of algorithms; programming languages appropriate for the algorithm abstractions; compiler technology

Increasing complexity of operating system support for a wide variety of system architectures using multi-/many- core chips; differing run-time support for a variety of tool chains and architectures; testing parallel programs, recovering from errors.

How many is “many”?

Multicore: the number of cores is such that conventional operating system techniques and programming approaches are still applicable

Manycore: the number of cores is such that either conventional operating system techniques or programming approaches no longer apply, i.e., they do not scale and performance degrades

Multicore Impact

multicore processors are increasingly being used in a wide range of systems and are having a significant impact on system design in multiple industries

- Cellphones, electronic game devices, automobiles, trains, planes, display walls, medical devices, TVs, movies, digital cameras, tablets, laptops, desktops, workstations, servers, network switches/routers, datacenters, clouds, supercomputers

multicore changes much about software specification, development, testing, performance tuning, system packaging, deployment and maintenance

Some Short Term Reactions to Multicore I

Some companies currently using ASICs, DSPs and FPGA are exploring replacing them using multicores

- Will multicore improve time-to-market, ease of upgrades, extension to new services?
- Will embedded device development become more software or more hardware focused?
- Will modeling, prototyping & evaluation methodologies and tools to determine how to exploit multicore technology be available?

Some Short Term Reaction to Multicore II

Determine which key applications can benefit from multicore execution

- Task level parallelism and data level parallelism

Determine how to go about parallelizing them with the least amount of effort to increase performance and reduce power consumption

Using the least amount of resources

- Recode? Redesign? New algorithms?
- Use of threads dominates current approaches. Does it scale? Is it the best approach? Testing parallel programs?
- What languages, compilers, standards to use? Tool sets?

Manycore is a Disruptive Technology

Multicore, up to a certain number of cores, allows for traditional responses to accommodate the required changes in systems design, implementation, test, etc.

Manycore, however, is a completely disruptive technology. Most contemporary operating systems have limited scalability and the tool chains for parallel program development are woefully inadequate

Will Manycore Succeed when so Many Previous Attempts at Parallel Computers Failed?

Goodyear MPP

Convex

Ardent

Burroughs D825

KSI

nmos

Kendel Square Research

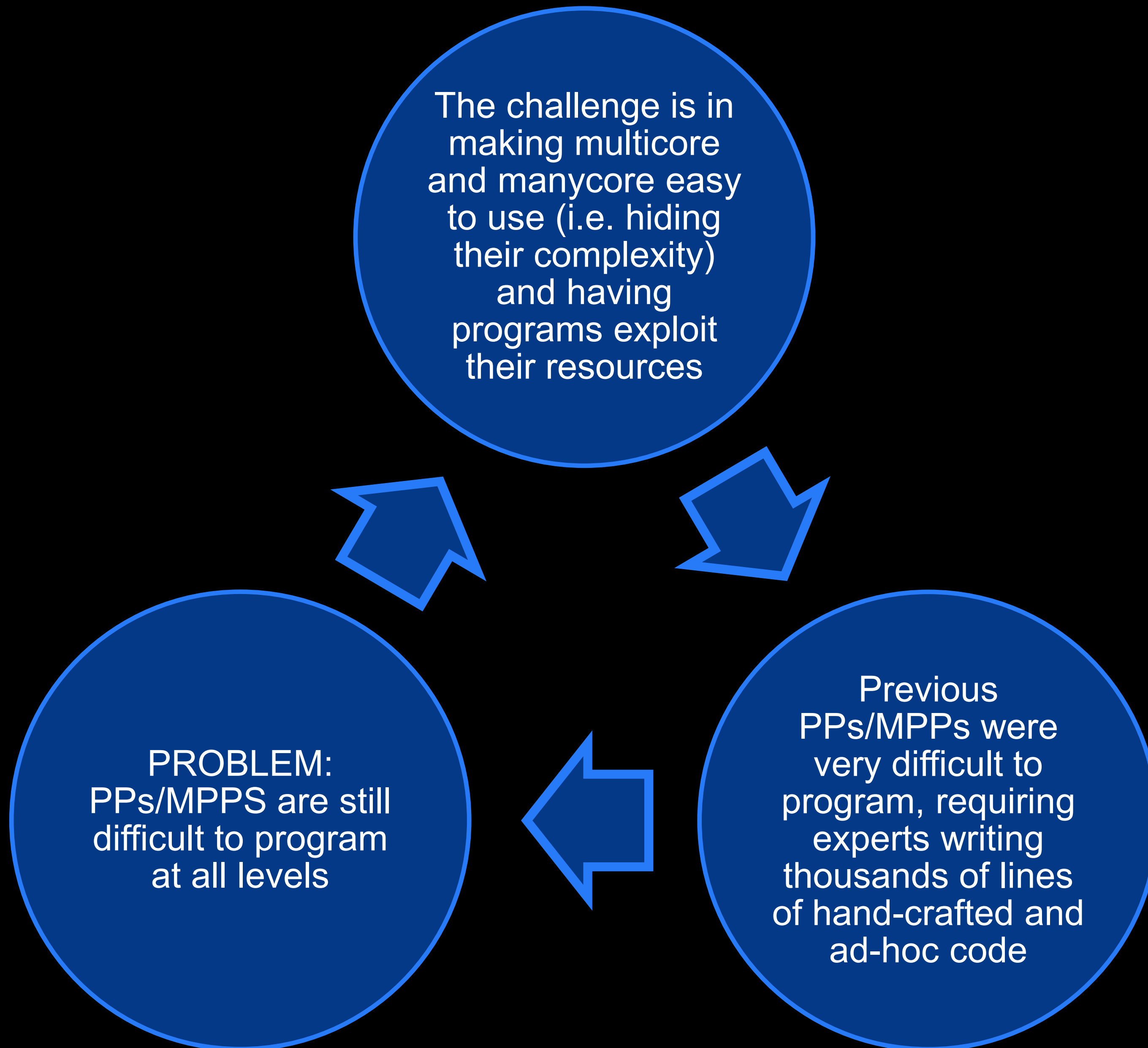
They didn"t fail.
They were just not commercial successes

In fact, there is a good deal to learn from studying the algorithm, software and hardware and software insights gained with these systems.

Yes, for One Primary Reason

Previous parallel and massively parallel processors were enormously expensive. Furthermore, they drew huge amounts of power, and required significant space, special cooling and complex programming

Multicore and manycore processors are *commodity processors* at *commodity prices*


The challenge is in making multicore and manycore easy to use (i.e. hiding their complexity) and having programs exploit their resources
Previous PPs/MPPs were very difficult to program, requiring experts writing thousands of lines of hand-crafted and ad-hoc code
PROBLEM: PPs/MPPS are still difficult to program at all levels

Important OS Issues

Diversity at All Levels

- How to manage the resources of a set of ... chip & off-chip interconnects, topologoes, interfaces and ...

Performance

- How to effectively use 10s, 100s and 1000s of heterogeneous ...

Power and Thermal

- How to use the least amount of power and generate the leas... ...nt of heat while achieving the highest possible performance

Reliability and Availability

- How to meet reliability and availabili... ...of hundreds of billions of transistors

Security and Privacy

- How to meet increasingly dema...

Application Developer

Architecture and Microarchitecture

Computer Science and Engineering Education

And, don't forget about dynamic reconfigurability; i.e. self-monitoring and fault tolerant systems

Some Manycore Chip Observations We'll Revisit Later

- Very high neighborhood bandwidth
- Bandwidth quickly decreases beyond the neighborhood
- Neighborhood protection issues
- Neighborhood isolation
- Proximity to I/O impacts performance & power consumption

Common denominator of these observations

They are Spatial Issues

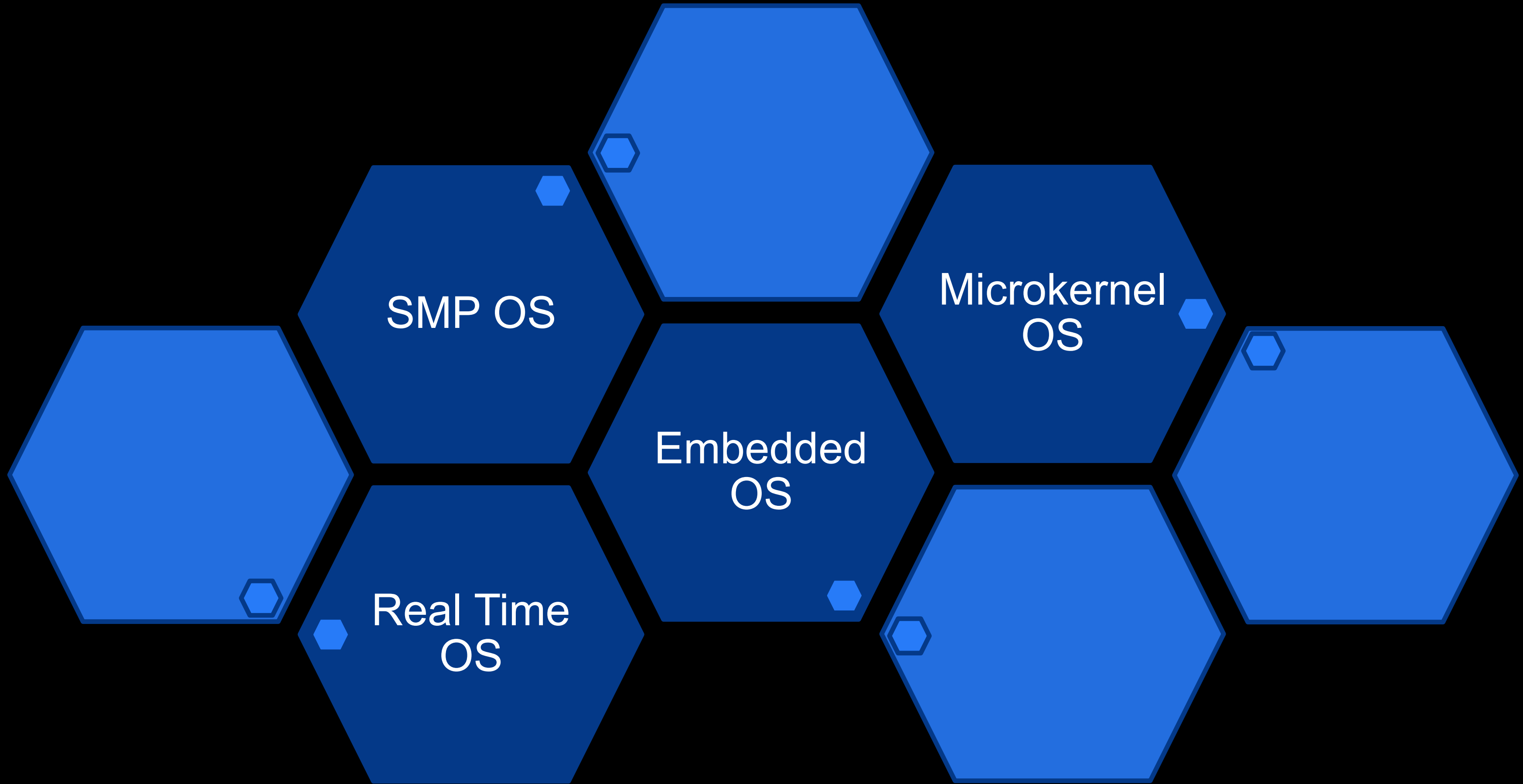
Typical SMP OS Structure

Shared-memory kernel on every processor (monolithic)

OS required data structures protected by locks, semaphores, monitors, etc.

The OS and the applications share the same memory hierarchy -- caches, TLBs, etc.

This Structure Leads to Limited Scalability



More Questions

What is a “good” mix of various types of cores for a multi-/manycore chip for workloads with specific characteristics?

- How many CPU cores, GPU cores, FPGA cores, DPS cores, etc.

What different types of interconnects and topologies should co-exist on-chip?

- Shared bus, point-to-point, crossbar, mesh, etc. Consider, for example, the differences between the 8-socket Opteron, the 8-socket Nehalem, the NVidia Fermi, and the Tilera Gx

What resources should be allocated to and used by applications and OS services? When should they be allocated?

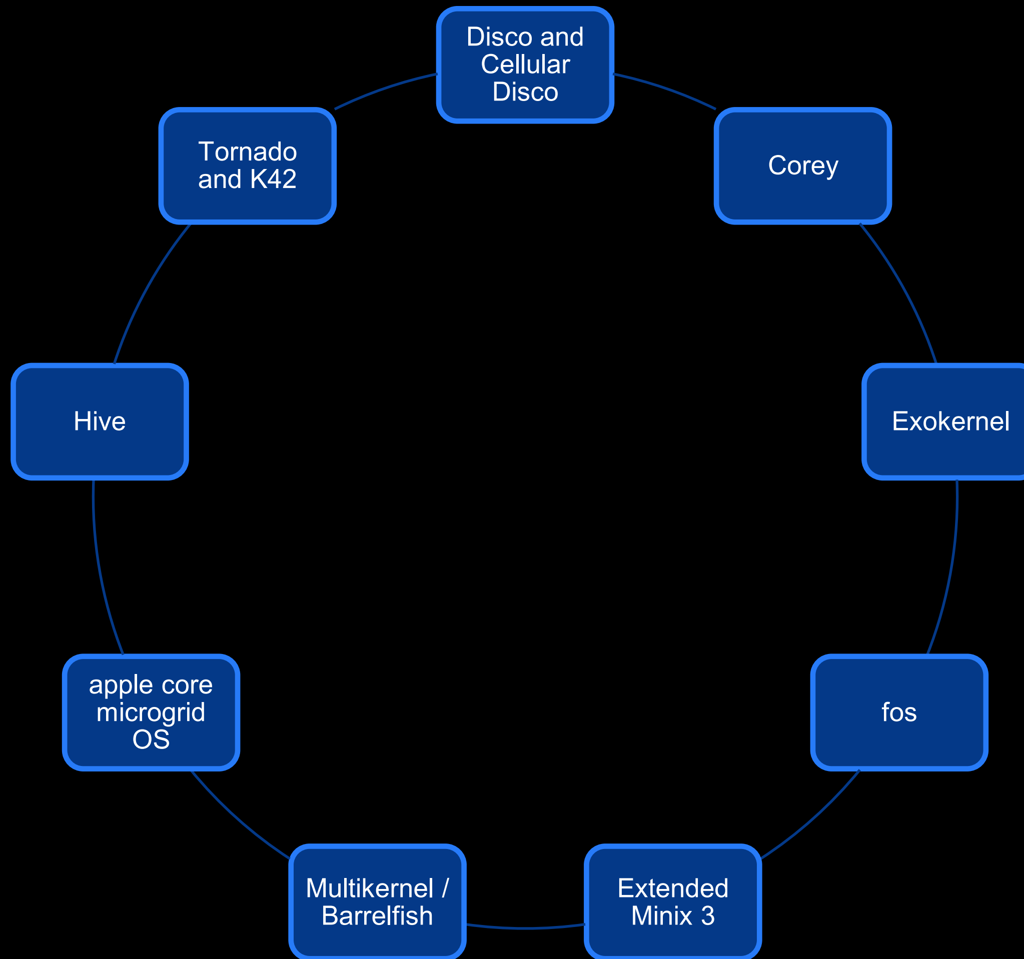
Mutual In

Do the answers change based on the targeted domain of the OS – for example, real-time or embedded or conventional SMP processing?

How should **an OS** be structured for multicore systems so that it is scalable to manycores and accommodate heterogeneity and hardware diversity?

What are the implications of this structure for the underlying manycore architecture and microarchitecture as well as that of the individual cores?

Some OS Research Projects Addressing These Questions



FOS and the Multikernel

We‟ll talk about some of what has been learned in two of these research projects but, before we do, we‟ll talk about Amdahl's Law and threads for a few minutes

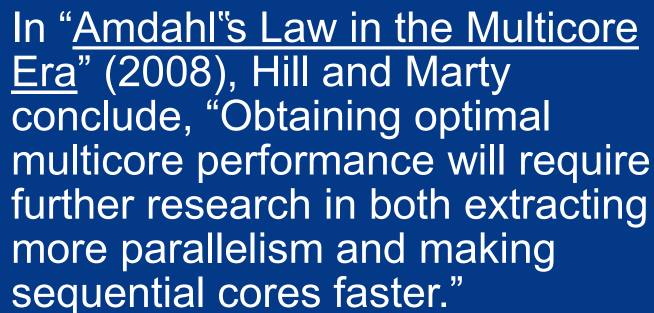
Amdahl's Law for manycore Processors

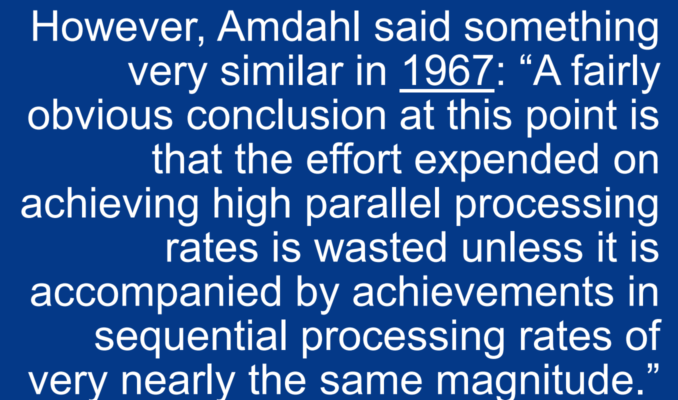
Amdahl's Law

The speedup of a program using multiple processors in parallel is limited by the time needed to execute the “sequential portion” of the program (i.e., the portion of the code that cannot be parallelized).

Example, if a program requires 10 hours to execute using one processor and the sequential portion of the code requires 1 hour to execute, then no matter how many processors are devoted to the parallelized execution of the program, the minimum execution time cannot be less than the 1 hour devoted to the sequential code.

Hill's and Marty's Results





Simple as Possible, but no Simpler …

"Amdahl‟s law and the corollary we offer for multicore hardware seek to provide insight to stimulate discussion and future work. Nevertheless, our specific quantitative results are suspect because the real world is much more complex. Currently, hardware designers can‟t build cores that achieve arbitrary high performance by adding more resources, nor do they know how to dynamically harness many cores for sequential use without undue performance and hardware resource overhead. Moreover, our models ignore important effects of dynamic and static power, as well as on- and off-chip memory system and interconnect design. Software is not just infinitely parallel and sequential. Software tasks and data movements add overhead. It‟s more costly to develop parallel software than sequential software. Furthermore, scheduling software tasks on asymmetric and dynamic multicore chips could be difficult and add overhead." (Hill and Marty)

Sun & Chen Took Up the Charge

"Reevaluating Amdahl"s law in the multicore era" (2010)

"Our study shows that multicore architectures are fundamentally scalable and not limited by Amdahl's law. In addition to reevaluating the future of multicore scalability, we identify what we believe will ultimately limit the performance of multicore systems: the memory wall."

"We have only studied symmetric multicore architectures where all the cores are identical. The reason is that asymmetric systems are much more complex than their symmetric counterparts. They are worth exploring only if their symmetric counterparts cannot deliver satisfactory performance."

Threads and Manycore Processors

Threads

GPUs are already running 1000s of threads in parallel!

GPUs are manycore processors well suited to data-parallel algorithms

The data-parallel portions of an application execute on the GPU as kernels running many cooperative threads

GPU threads are very lightweight compared to CPU threads

GPU threads run and exit (non-persistent)

Feedback-Driven Threading

Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs by M. Suleman, Qureshi & Patt

They challenge setting the

# of threads =

# of cores

They develop a run-time method to estimate the best number of threads

Their Intuition

Assign as many threads as there are cores scalable applications **only**

And not for applications that don"t

- Performance may max out earlier wasting cores
- Adding more threads may increase power consumption and heat
- Adding more threads may actually increase execution time

Two Workload Models

Synchronization-Limited Workloads

- Example: use of critical sections to synchronize access to shared data structures

Bandwidth Limited Workloads

- Example: use of an off-chip bus to access shared memory or a co-processor

Critical Sections

Code that accesses a shared resource which must not be concurrently accessed by more than one thread of execution

A synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, e.g., a lock or a semaphore

Critical sections are used: (1) To ensure a shared resource can only be accessed by one process at a time and (2) When a multithreaded program must update multiple related variables without other threads making conflicting changes

Synchronization-Limited Workloads

The execution time inside the critical section increases with the number of threads

Bandwidth-Limited Workloads

More threads, execution time decreases but the bandwidth demands increase

Who/What Selects & Schedules Threads?

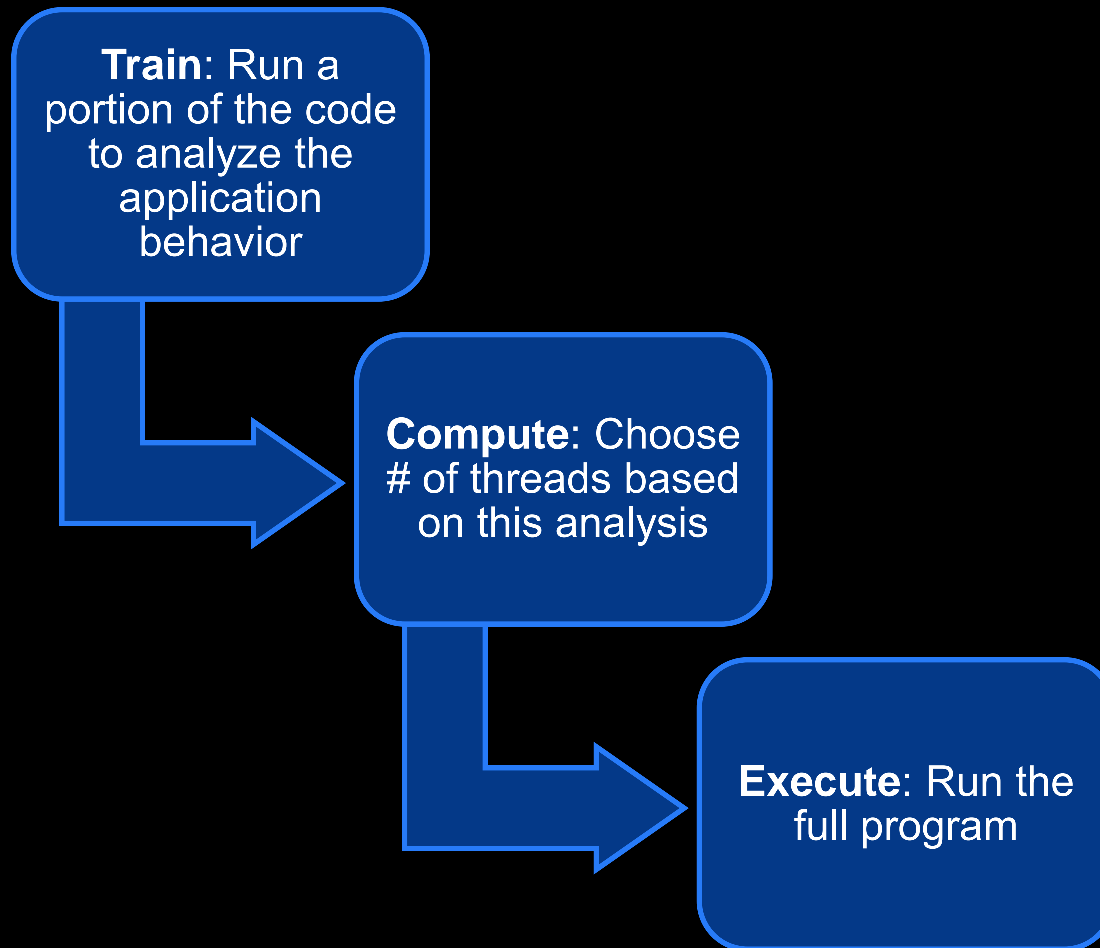
User of application

Programmer who writes the application

Compiler that generates code to execute the application (static and/or dynamic)

Operating system that provides resources for the running application

Feedback-Driven Threading: Basic Strategy



Three Approaches

Synchronization-Aware Threading

Measure time inside and outside critical section using the cycle counter

Reduces both power and execution time

Bandwidth-Aware Threading

Measure bandwidth usage using performance counters

Reduces power without increasing execution time

Combination of Both

Train for both SAT and BAT

SAT + BAT reduces both power and execution time

Scope of Results

- Assumes only one thread/core, i.e. no SMT on a core
- Bandwidth assumptions ignore cache contention and data sharing
- Single program in execution model
- Dynamic nature of the workload in systems not accounted for

Extending the Scope

How could <u>application heartbeats</u> (or a similar technology) be used to extend the scope of these results?

The Factored Operating System

The Case for a Factored Operating System

Wentzlaff and Agarwal, in their 2008 MIT report are motivated to propose FOS are driven by the usual issues

µP performance is no longer on an exponential growth path

- Design complexity of contemporary µPs
- Inability to detect and exploit additional parallelism that has a substantive performance impact
- Power and thermal considerations limit increasing clock frequencies

SMPs are not scalable due to structural issues

Current SMP OSs Rely On

- Fine grain locks
- Efficient cache coherence for shared data structures and locks
- Execute the OS across the entire machine (monolithic)
- Each processor contains the working set of the applications and the SMP

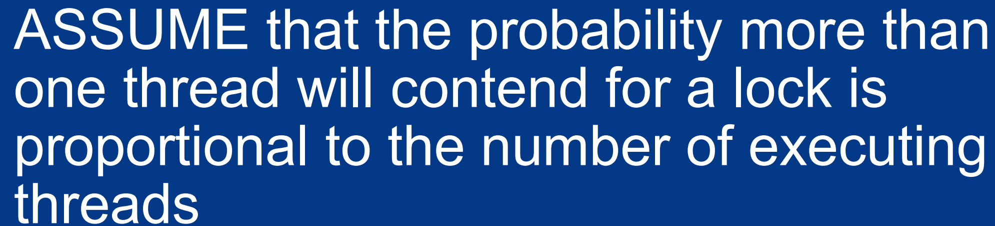
Current SMPs Support a Small # of Cores

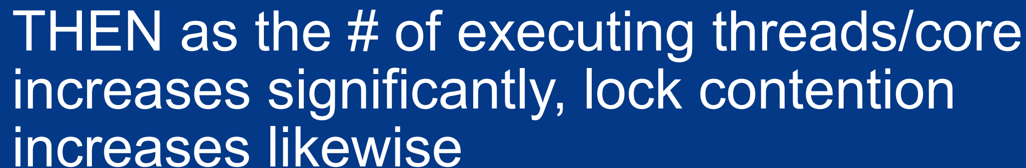
Minimize the portions of the code that require fine grain locking

As the number of cores grows, 2 to 4 to 6 to 8 to etc., incorporating fine grain locking is a challenging and error prone process

These code portions are shared with large numbers of cores and 100s/1000s of threads in manycore systems: **50 is quite different from 1000.**

An Ever Increasing # of Fine-Grained Locks





THIS IMPLIES the number of locks must increase proportionately to maintain performance

Wentzlaff's and Agarwal's Data

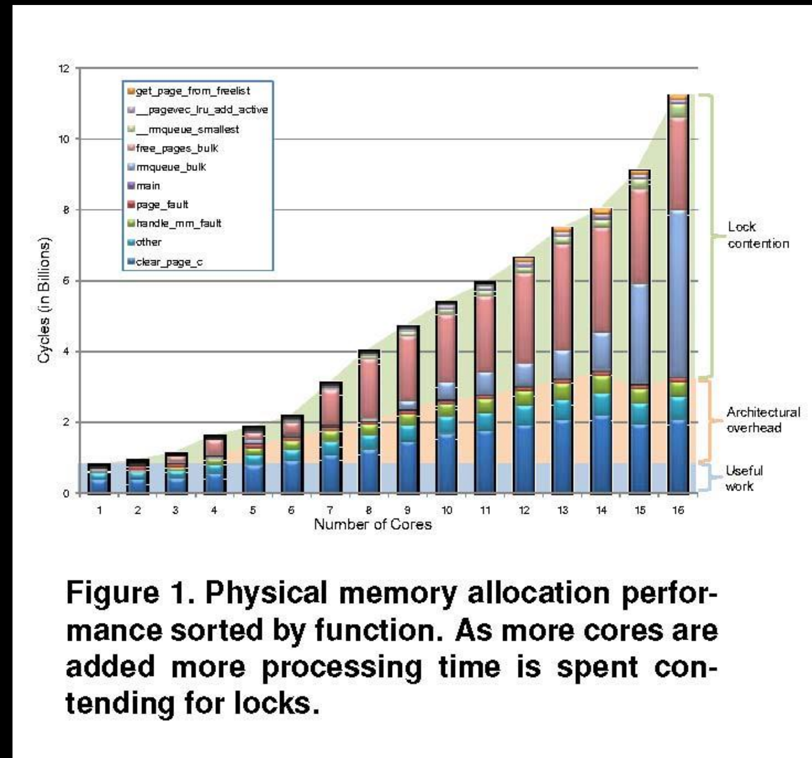


Figure 1. Physical memory allocation performance sorted by function. As more cores are added more processing time is spent contending for locks.

This figure is taken from 2008, The Case for a Factored Operating system (fos), MIT Report, Wentzlaff and Agarwal

Programmer Productivity Does Not Scale with Transistor Count

Increasing the # of locks is time consuming and error prone

Locks can cause deadlocks via difficult to identify circular dependencies

There is a limit to the granularity. A lock for each word of shared data?

Shared Memory and Locality

Executing OS code & application code on the same core

Implies the cache system on each core must contain the shared working set of the OS and the set of executing applications

Reduces hit rate for applications and, subsequently, single stream performance

Cache System Performance Degradation

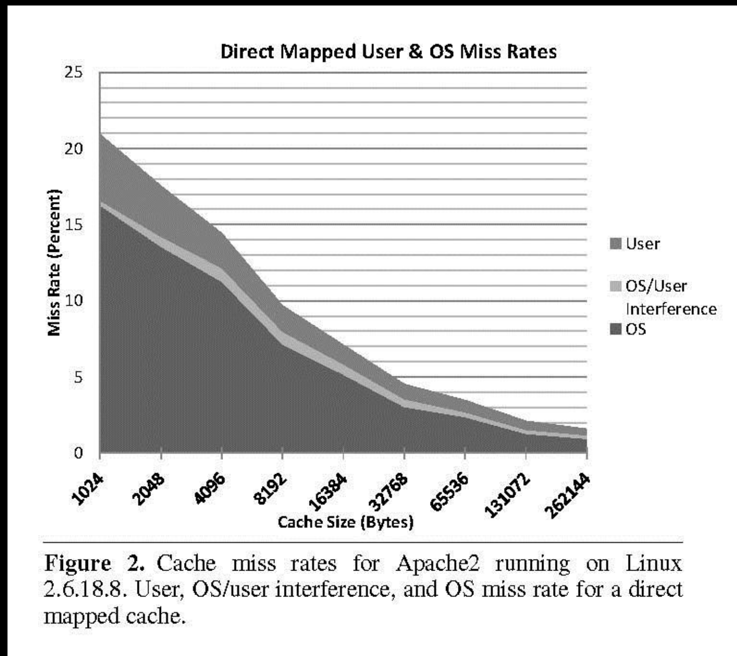


Figure 2. Cache miss rates for Apache2 running on Linux 2.6.18.8. User, OS/user interference, and OS miss rate for a direct mapped cache.

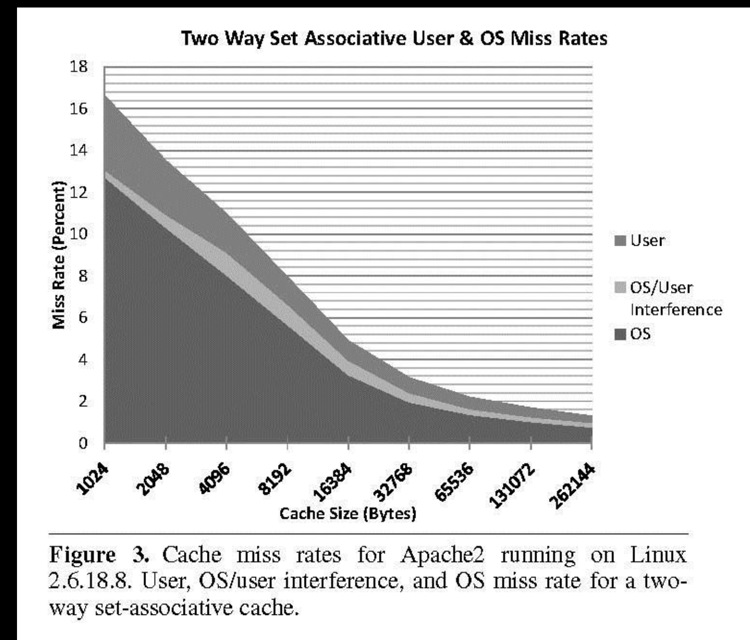


Figure 3. Cache miss rates for Apache2 running on Linux 2.6.18.8. User, OS/user interference, and OS miss rate for a two-way set-associative cache.

Both of these figures are taken from a 2009 article, "Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores," by Wentzlaff and Agarwal, a 2009 article which is an enhanced version of the original 2008 MIT report.

Cache Coherence

"It is doubtful that future multicore processors will have efficient full-machine cache coherence as the abstraction of a global shared memory space is inherently a global shared structure." (Wentzlaff and Agarwal)

"While coherent shared memory may be inherently unscalable in the large, in a small application, it can be quite useful. This is why fos provides the ability for applications to have shared memory if the underlying hardware supports it." (Wentzlaff and Agarwal)

Their Recommendations

- Avoid the use of hardware locks
- Separate the operating system resources from the application execution resources
- Avoid global cache coherent shared memory

FOS Design Principles

Factored Operating System
Space Sharing Replaces Time Sharing

- Inspired by distributed Internet services model
- Each OS service is designed like a distributed internet server
- Each OS service is composed of multiple server processes which are spatially distributed across a multi-manycore chip
- Each server process is allocated to a specific core eliminating time-multiplexing cores
- The server processes collaborate and exchange information via message passing to provide the overall OS service

Distributes High & Low Level Services

As noted, each OS system service consists of collaborating servers

OS kernel services **also** use this approach

For example, physical page allocation, scheduling, memory management, naming, and hardware multiplexing

Therefore, all system services and kernel services run on top of a microkernel

OS code is not executed on the same cores that are executing applications code

The “Thin” FOS Microkernel

- Platform **dependent**
- A portion of the microkernel executes on each processor core
- Implements a machine dependent communication infrastructure (API); message passing based
- Controls access to resources (provides protection mechanisms)
- Maintains a name cache to determine the location (physical core number) of the destination of messages

Distributing Data as Well as Services

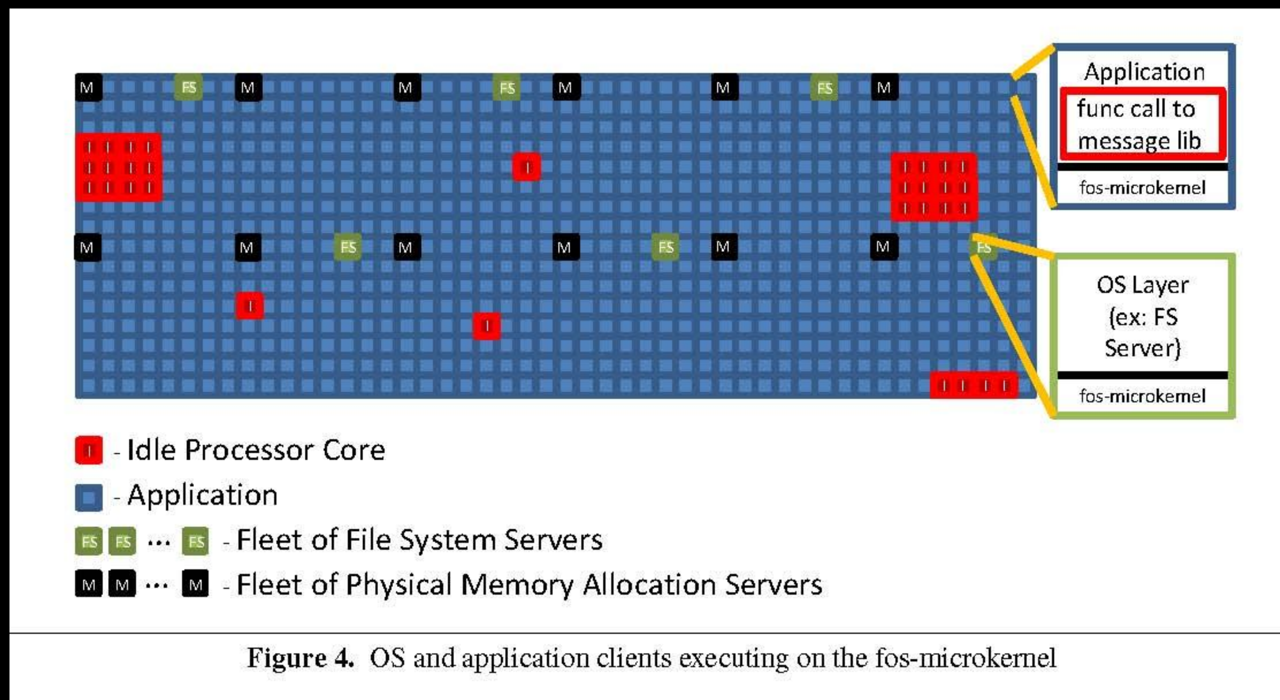


Figure 4. OS and application clients executing on the fos-microkernel

The applications and the OS system services operate on separate cores on top of the microkernel

Distributed ILP

Combining multiple cores to behave like a more powerful core

The “cluster” is a “core”

Project Angstrom

Algorithms, programming models, compilers, operating systems and computer architectures and microarchitectures have no concept of space

Underlying uniform access assumption: a wire provides an instantaneous connections between points on an integrated circuit

Assumption is no longer valid: the energy spent in driving the wires and the latency (the time to get from here to there) must now be taken into consideration

2008 Angstrom Presentation

Project Angstrom: A Cross-System Effort

Theory

- **Abstractions**: define space related abstractions between system layers
- **Theory**: Define SPRAM (Spatial PRAM model for algorithmic complexity)

Architecture

- **Architecture**: Create the Angstrom multicore processor scalable to thousands of cores
- **Prototype**: At-scale 256 core system

Applications

- Video
- Data center on chip
- Autonomous vehicles
- Speech
- Computational photo and video
- Web

Tools

- **Compiler**: Spatially aware JIT multicore compiler
- **OS, Networking and Databases**: libOS uses space multiplexing instead of time multiplexing
- **Language**: Expose spatial attributes to compiler

CSAIL is seeking major industrial partner in this effort

1

The Multikernel Operating System

The Multikernel: A new OS architecture for scalable multicore systems, Baumann et al

"Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures."

"We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems."

Multikernel Design Principals

- Organize the OS as a distributed system
- Implement the OS in a hardware-neutral way
- View “state” as replicated

Their Vision

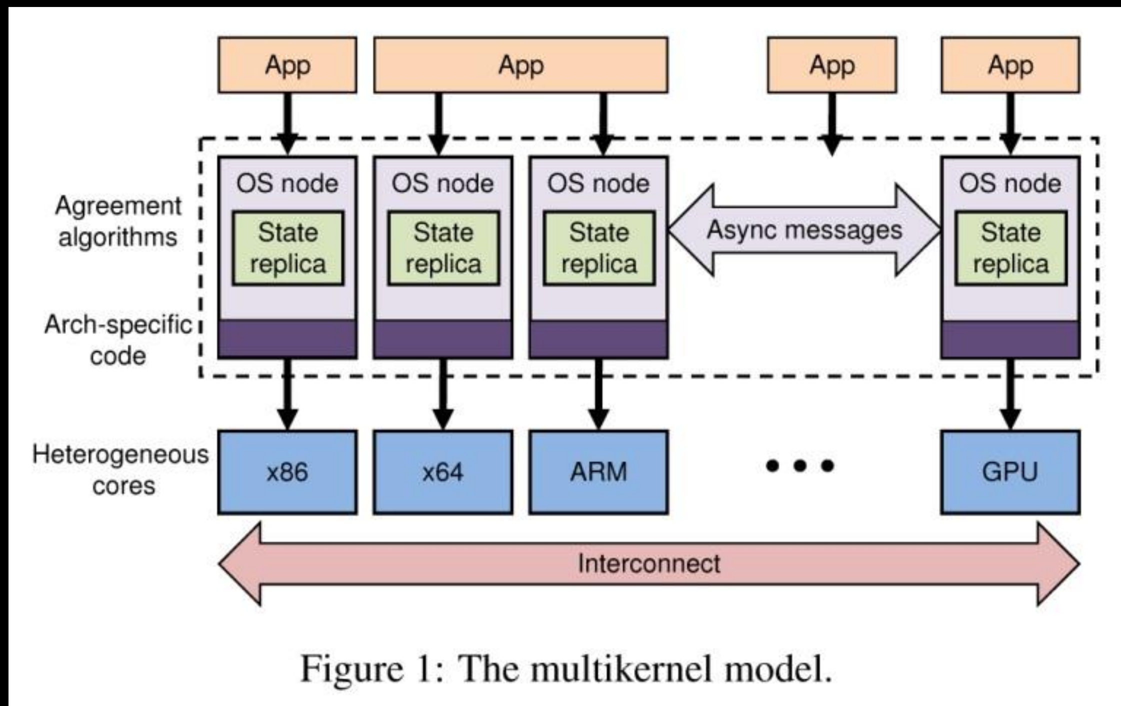


Figure 1: The multikernel model.

Replicated State and Agreement Protocols

"The principal impact on clients is that they now invoke an agreement protocol (propose a change to system state, and later receive agreement or failure notification) rather than modifying data under a lock or transaction. The change of model is important because it provides a uniform way to synchronize state across heterogeneous processors that may not coherently share memory."

From Baumann er al, "Your computer is already a distributed system. Why isn"t your OS?"

All Inter-Core Communications are via Messages

Separation of “method” and “mechanism”

- Messages decouple OS communication structure from the hardware inter-core communications mechanisms

Transparently supports

- Heterogeneous cores
- Non-coherent interconnects
- Split-phase operations by decoupling requests from responses and thus aids concurrency
- System diversity (e.g., Tile-Gx and the Intel 80-core)

Message Passing vs. Shared Memory Experiments

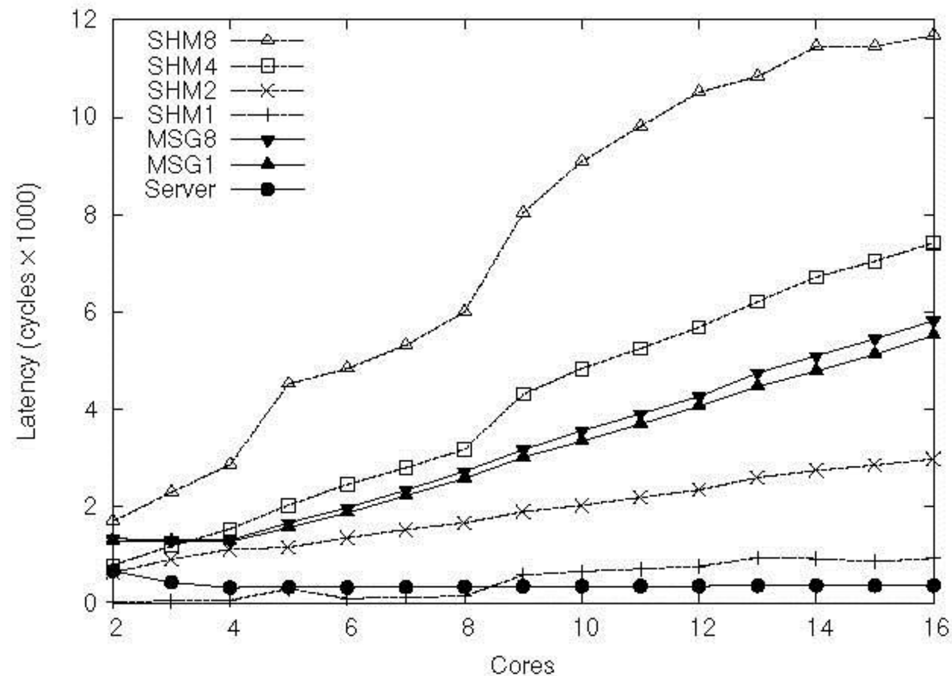


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

Conclude: Messages Cost Less than Memory

Barrelfish: A Sample Implementation

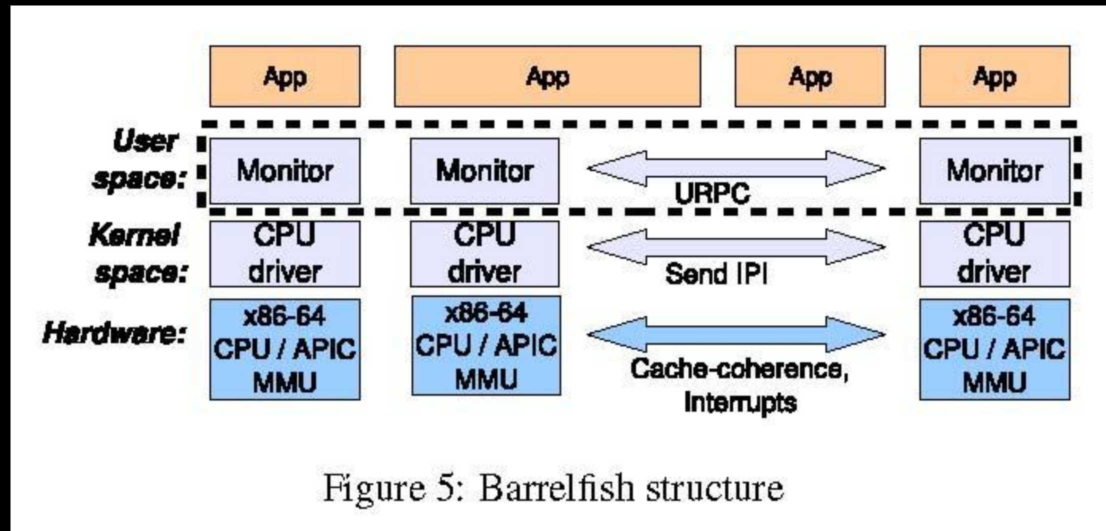


Figure 5: Barrelfish structure

Cache Coherency

"A separate question concerns whether future multicore designs will remain cache-coherent, or opt instead for a different communication model (such as that used in the Cell processor). A multikernel seems to oer the best options here. As in some HPC designs, we may come to view scalable cache-coherency hardware as an unnecessary luxury with better alternatives in software"

"On current commodity hardware, the cache coherence protocol is ultimately our message transport."

From Baumann er al, "Your computer is already a distributed system. Why isn"t your OS?"

Challenging FOS and the Multikernel?

A Word About Linux Scalability

Linux Scalability

In "An Analysis of Linux Scalability to Many Cores" (2010), Boyd-Wickizer et al study the scaling of Linux using a number of web service applications that are:

- Designed for parallel execution
- Stress the Linux core
- MOSBENCH = Exim mail server, memcached (a high-performance distributed caching system), Apache (an HTTP server), serving static files, PostageSQL (an object-relational database system), gmake, the Psearchy file indexer, and a multicore MapReduce library (Google"s framework for distributed computing on large data sets)

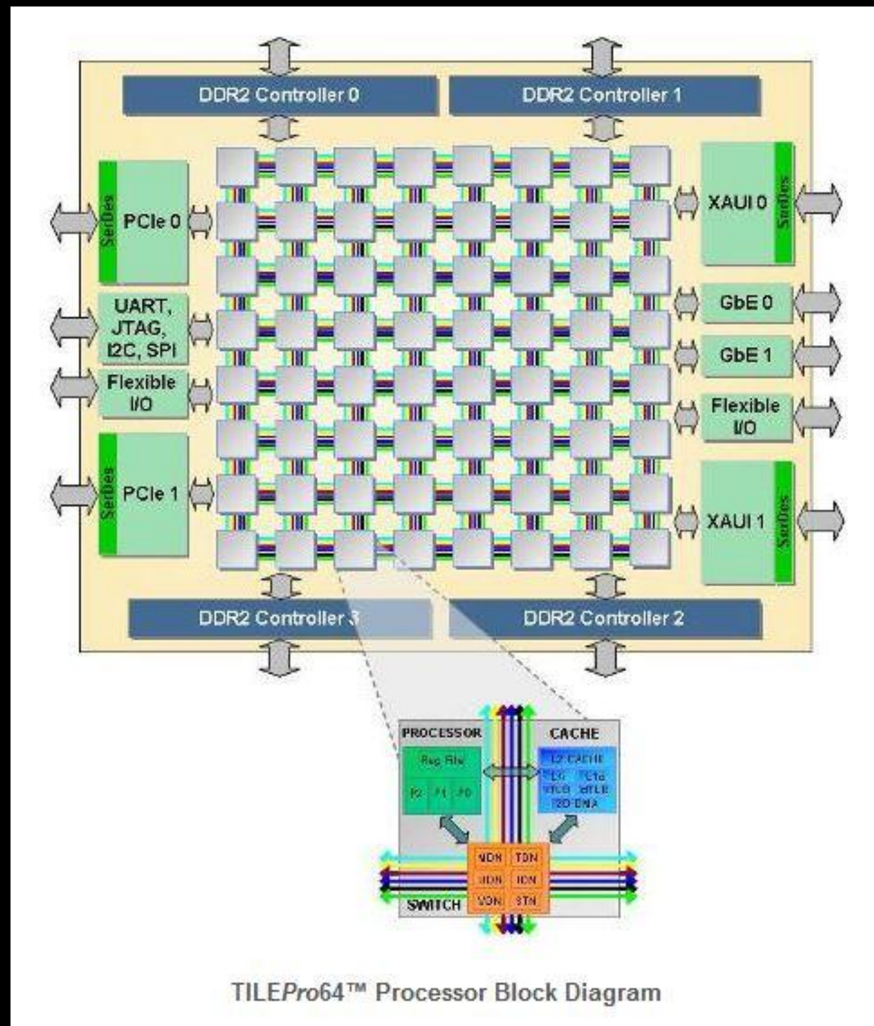
Some of Their Conclusions

MOSBENCH applications can scale well to 48 cores with *modest* changes to the applications and to the Linux core

"The cost of thread and process creation seem likely to grow with more cores"

"If future processors don"t provide high-performance cache coherence, Linux"s shared-memory intensive design may be an impediment to performance."

Tilera's TILE*Pro*64



TILE*Pro*64™ Processor Block Diagram

Dynamic Distributed Cache

DDC is a fully coherent shared cache system across an arbitrarily-sized array of tiles

Does not use (large) centralized L2 or L3 caches to avoid power consumption and system bottlenecks

DDC‟s distributed L2 caches can be coherently shared among other tiles to evenly distributing the cache system load

iMesh™

- Instead of a bus, the TILE64 uses a non-blocking, cut-through switch on each processor core
- The switch connects the core to a two dimensional on-chip mesh network called the “Intelligent Mesh” - iMesh™
- The combination of a switch and a core is called a 'tile„
- iMesh provides each tile with more than a terabit/sec of interconnect bandwidth
- Multiple parallel meshes separate different transaction types and provide more deterministic interconnect throughput

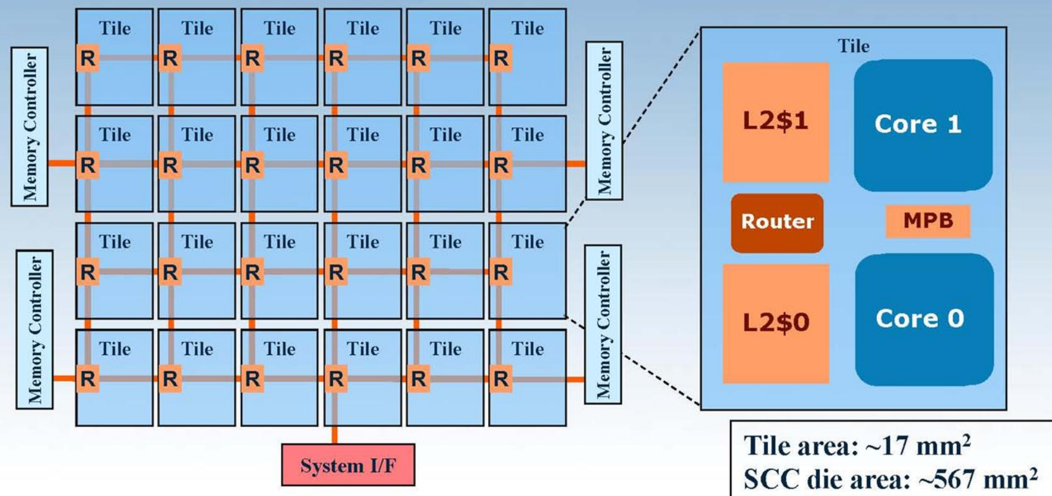
Tilera Multicore Developemnt Environment

- I"ll let MDE speak for itself

Intel's Single-chip Cloud Computer (SCC)

Top Level Hardware Architecture

- 6x4 mesh 2 Pentium™ P54c cores per tile
- 256KB L2 Cache, 16KB shared MPB per tile
- 4 iMCs, 16-64 GB total memory



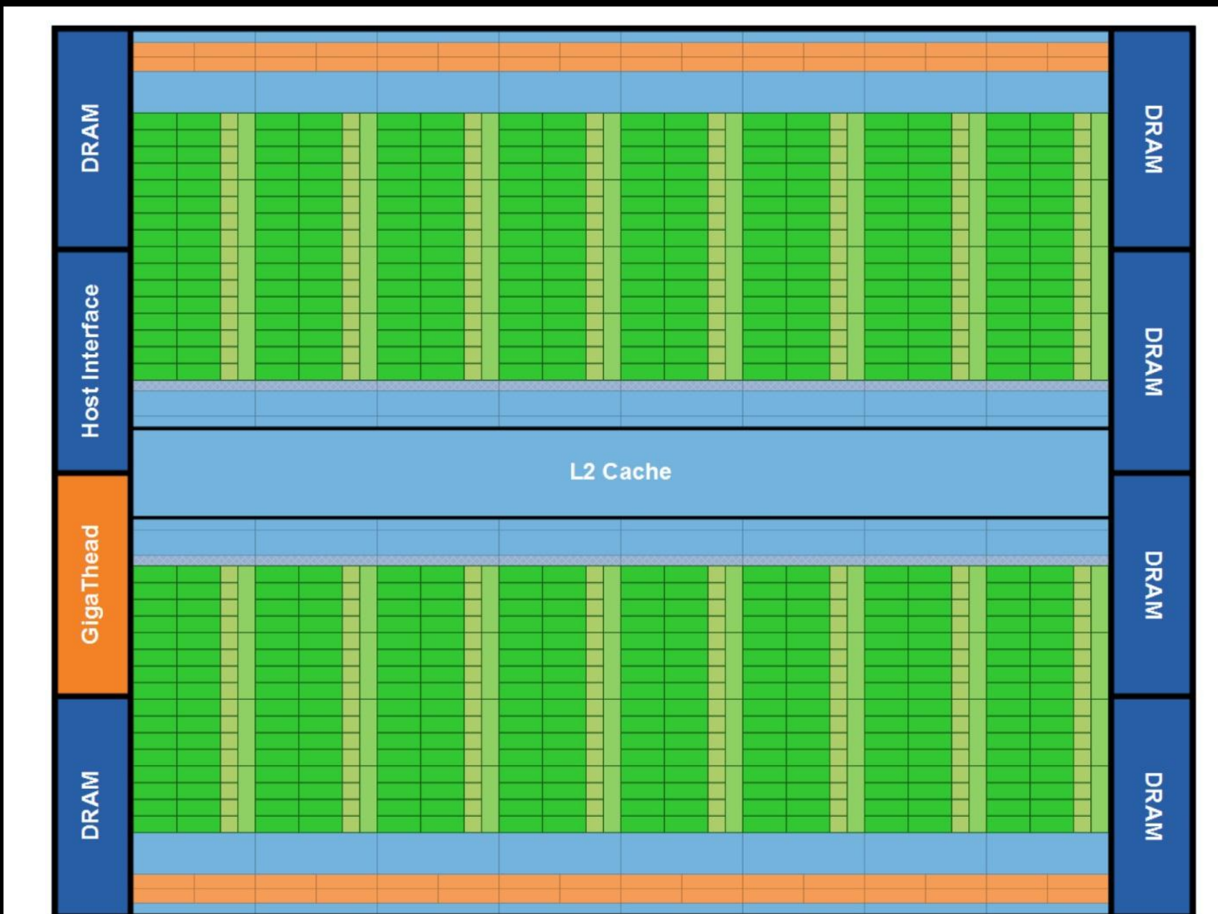
R = router, iMC = integrated Memory Controller, MPB = message passing buffer

Intel's 80-Core Tiled Processor

Lessons Learned from the 80-core Tera-Scale Research Processor, by Dighe et all

1. The network consumes almost a third of the total power, clearly indicating the need for a new approach
2. Fine-grained power management and low-power design power techniques enable peak energy of 19.4 GFLOPS/Watt and a 2X reduction in standby leakage power, and
3. The tiled design methodology quadruples design productivity without compromising design quality.

nVidia's Fermi



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Project Apple-CORE

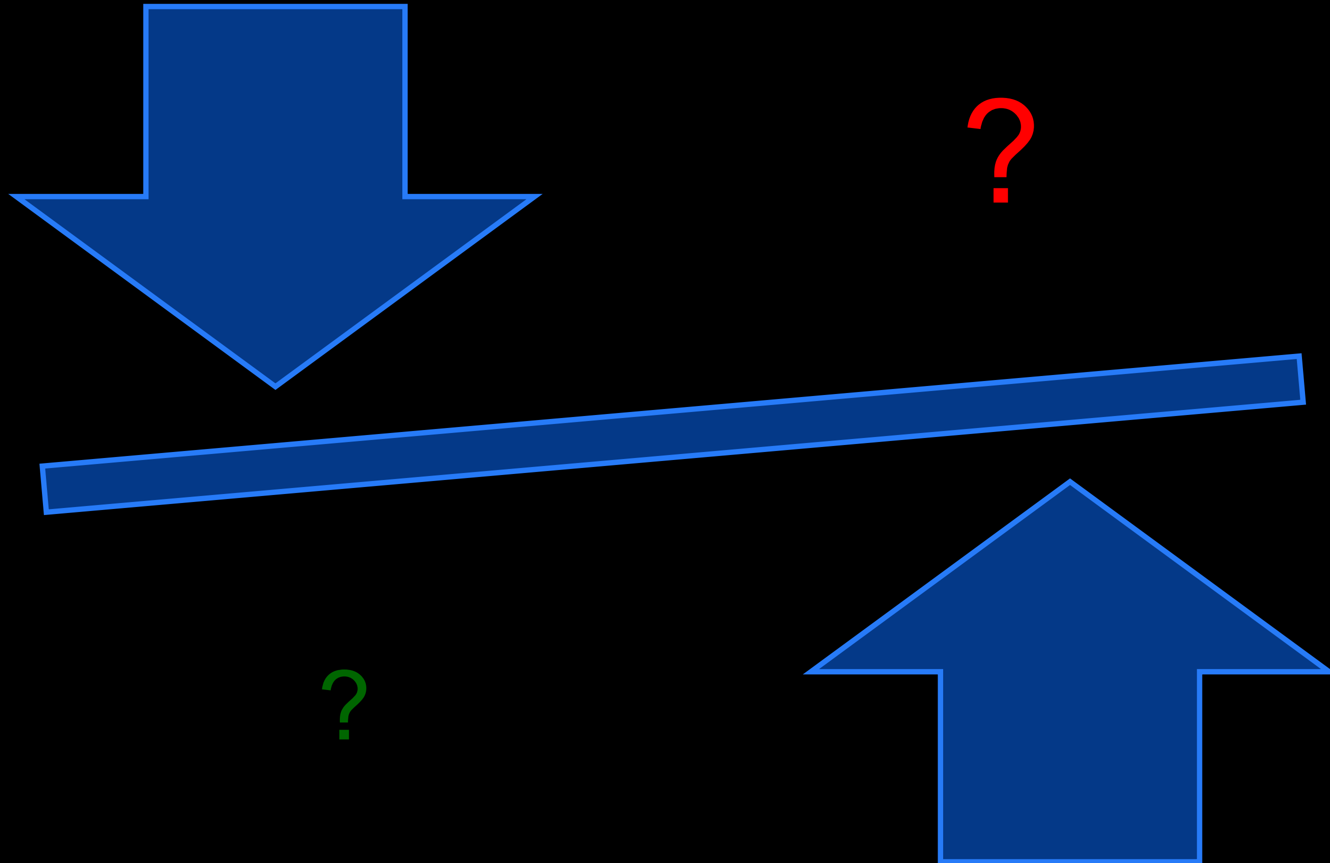
Architecture paradigms and programming languages for efficient programming of multiple CORES

EU Funded

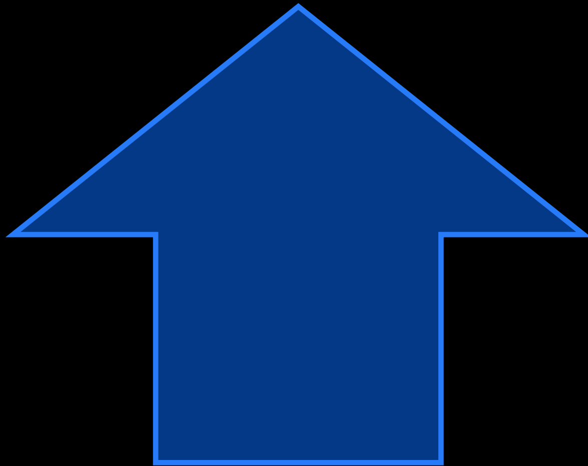
Self-adaptive Virtual Processor (SVP) execution model

“The cluster is the processor” –the concept of place (a cluster) allocated for the exclusive use of a thread (space sharing)

What Should Go on the Chip

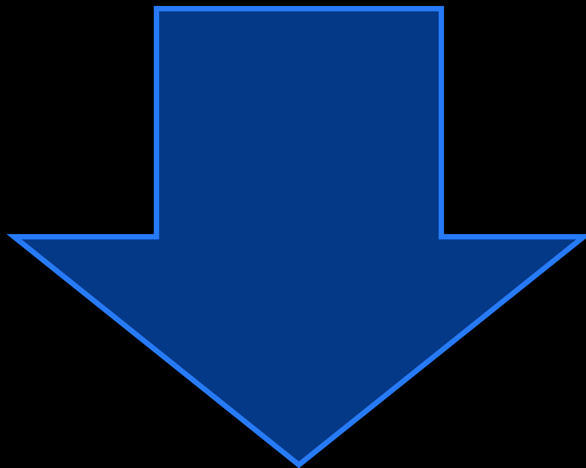


Anat Agarwal's Approach to Sizing Core Resources: The KILL Rule



Increase the resource size (chip area) only if for every 1% increase in core area there is at least a 1% increase in core performance, i.e., Kill (the resource growth) If Less than Linear (performance improvement is realized)

- The KILL Rule applies to all multicore resources, e.g., issue-width, cache size, on chip levels of memory, etc.



KILL Rule implies many caches have been sized "well beyond diminishing returns"

Communications Centric Computational Models

Communication requires less cycles & energy than cache (10X) or memory accesses (100X)

Develop algorithms that are communication centric rather than memory centric

- Stream algorithms: read values, compute, deliver results
- Dataflow: arrival of all required data triggers computation, deliver results

Use frameworks that allow the expression of parallelism at all levels of abstraction

Some Simple & Complex Questions

Do existing complex cores make “good” cores for multi-/manycore?

When do bigger L1, L2 and L3 caches increase performance? Minimize power consumption?

What % of interconnect latency is due to wire delay?

What programming models are appropriate for developing multi-/manycore applications?

Communications Latency

Latency arises from coherency protocols and software overhead

Ways to reduce the latency to a few cycles

- Minimize memory accesses
- Support direct access to the core-to-core interconnect (bus, ring, mesh, etc.)
- Eliminate or greatly simplify protocols

Questions

What programming models can we used for specific hybrid organizations?

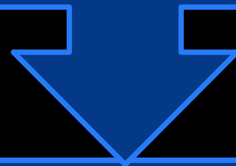
What should a library of “build block” programs look like for specific hybrid organizations?

Should you be able to run various operating systems of different “clusters” of cores – i.e., when and where does virtualization make sense in a manycore environment?

How can you determine if your “difficult to parallelize” application will consume less power running on many small cores versus running on a couple of small cores?

Parallelizable Applications

Some applications -- large scale simulations, genome sequencing, search and data mining, and image rendering and editing - can scale to 100s of processors



They can be	
Decomposed into independent tasks	Structured to operate on independent sets of data

By and large, however, the set of easily parallelizable applications is small.

Questions

Parallelism

Data parallelism is when several processors in a mutiprocessor system execute the same code, in parallel, on different parts of the data. This is sometimes referred to as SIMD processing.

Task parallelism is achieved when several processors in a multiprocessor system execute a different thread (or process) on the same or different data. Different execution threads communicate with one another as they execute to pass data from one thread to the another as part of the overall program execution. In the general case, this is called MIMD processing.

When multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep on different data that data parallelism requires it is referred to as SPMD processing.

Applications often employ multiple types of parallelism.