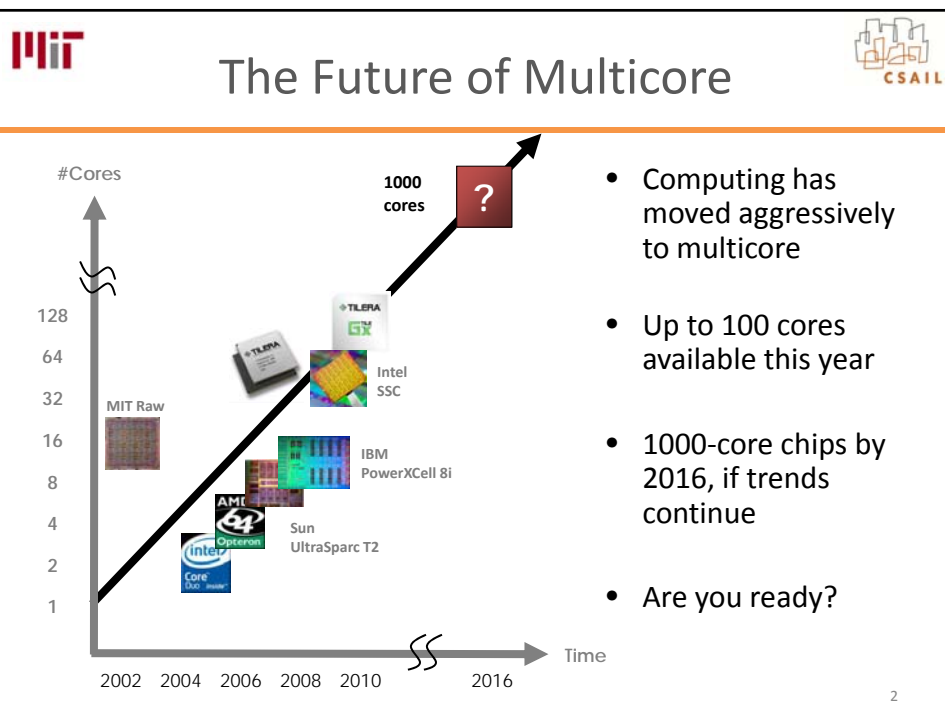


Graphite Overview

Goals, Architecture, and Performance





Simulation in Multicore Research

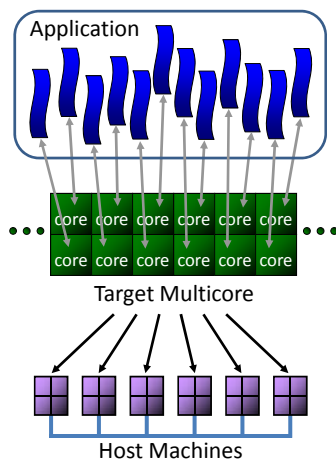


- Simulation is vital for exploring future architectures
 - Experiment with new designs/technologies
 - Abstract away details and focus on key elements
 - Rapid exploration of design space
 - Early software development for upcoming architectures
- The future of multicore simulation:
 - Need to simulate 100's to 1000's of cores
 - Massive quantities of computation
 - High-level architecture is becoming more important than microarchitecture
 - On-chip networks, Memory hierarchies, DRAM access, Cache coherence

3



Graphite At-a-Glance



- A fast, high-level simulator for large-scale multicores
- Application-level simulation where threads are mapped to target cores
- Runs in parallel on multicore host machines
- Multi-machine distribution
 - Invisible to application
 - Runs off-the-shelf pthread apps
- Relaxed synchronization scheme
 - Trades some timing accuracy for performance
 - Guarantees functional correctness

4



Graphite Performance



Graphite slowdown on 8 host machines (64 cores total)
versus native execution on one 8-core machine

| | |
|--------|-------|
| Min | 41x |
| Max | 3930x |
| Median | 616x |

- Typical slowdown for existing sequential simulators:
10,000x – 100,000x
- Results from SPLASH2 benchmarks on a 32-core
target processor

5





Graphite Trades Accuracy for Performance



- Simulator performance is a major limiting factor
 - Limits depth and breath of studies, size of benchmarks
 - Too much detail slows simulation
 - Cannot simulate 1000's of cores
 - Most simulators are sequential, Graphite is parallel
 - Typical performance: 10,000x – 100,000x slowdown
 - Our target performance: 100x
- Performance vs. accuracy
 - Cycle-accurate: very accurate but slow
 - High-level: trade some accuracy for performance
 - For next year's chips, you need cycle-accuracy
 - For chips 5-10 years out, you need performance



6



Outline

- Introduction
- Graphite Architecture
 - Overview
 - Multi-machine distribution
 - Clock Synchronization
- Results
- Conclusions

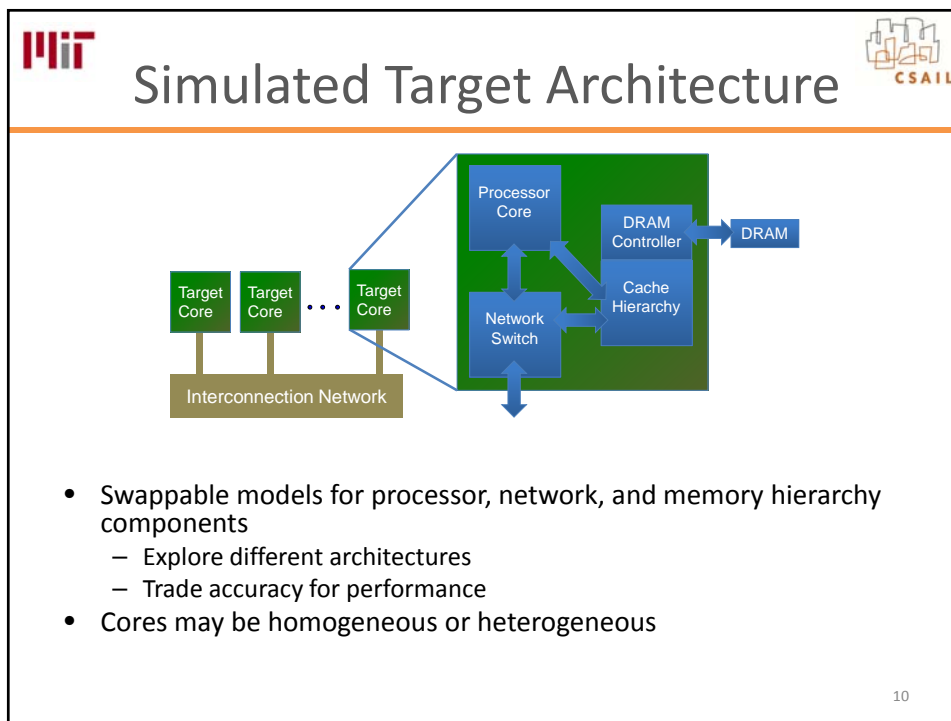
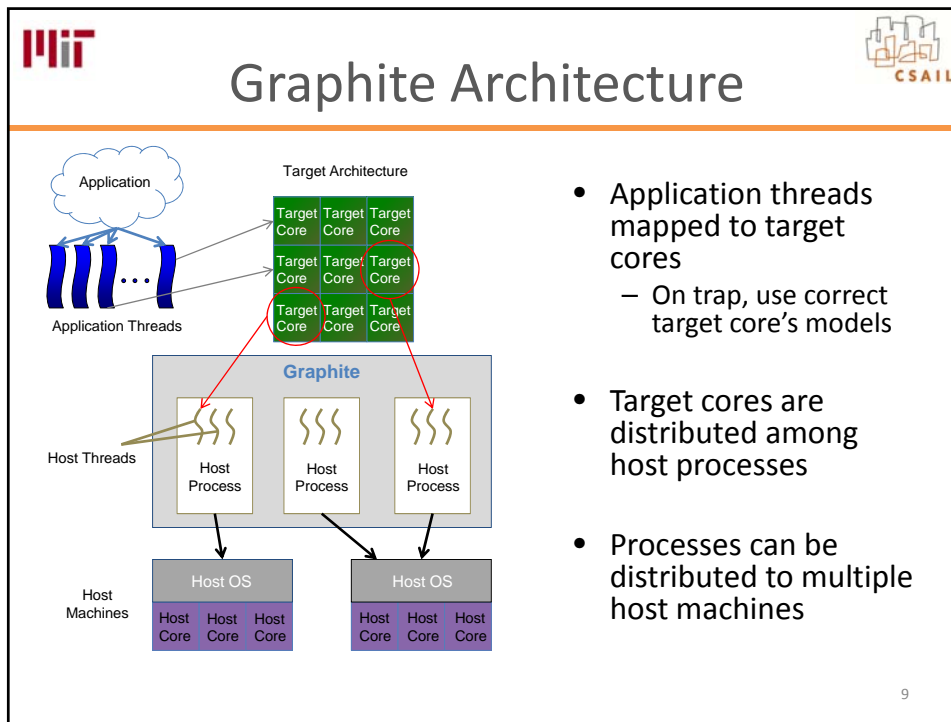
7

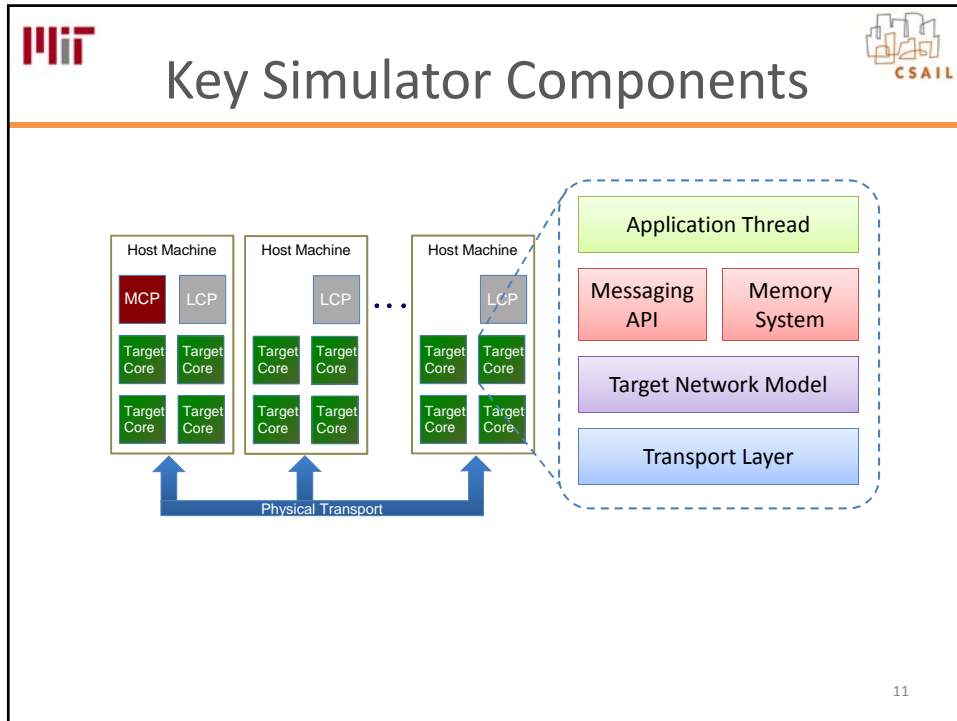


Graphite Overview

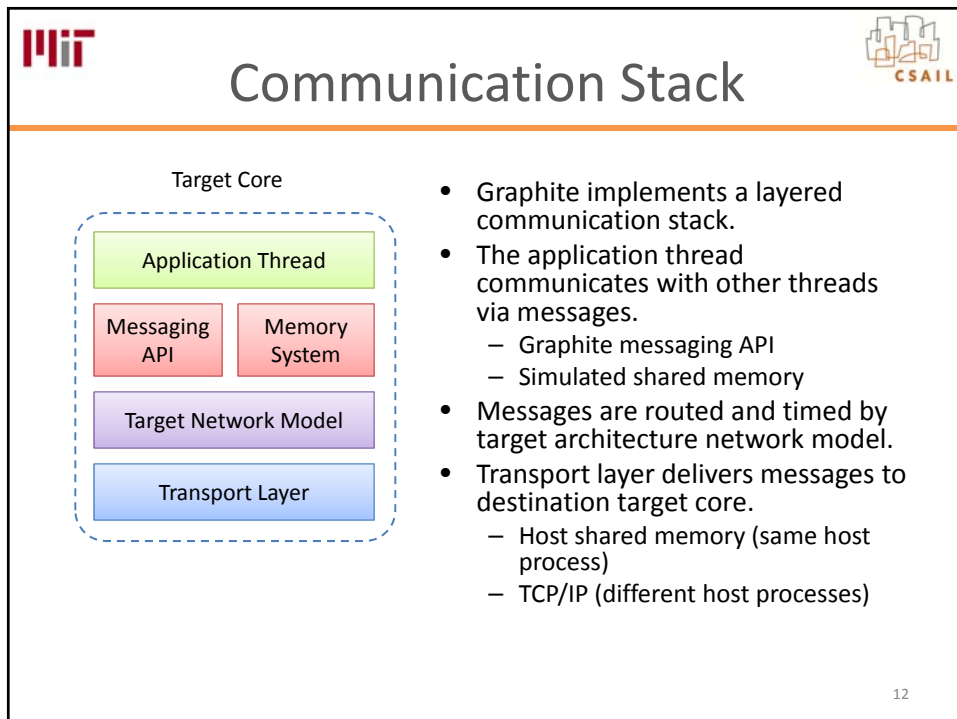
- Application-level simulator based on dynamic binary translation
 - Uses Intel's Pin
 - App runs natively except for new features and modeled events
 - On trap, model functionality and timing
- Simulation consists of running an application on a target architecture
 - Target specified by swappable models and runtime parameters
 - Different architectures
 - Accuracy vs. Performance
 - Result:
 - Application output
 - Simulated time to completion
 - Statistics about processor events

8







11





12



Outline

- Introduction
- Graphite Architecture
 - Overview
 - Multi-machine distribution
 - Clock Synchronization
- Results
- Conclusions

13



Parallel Distribution Challenges

- Wanted support for standard pthreads model
 - Allows use of off-the-shelf apps
 - Simulate coherent-shared-memory architectures
- Must provide the illusion that all threads are running in a single process on a single machine
 - Single shared address space
 - Thread spawning
 - System calls

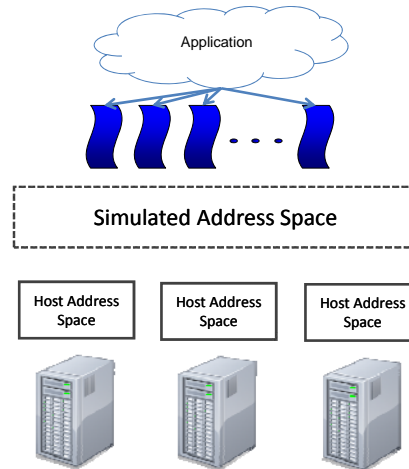
14



Single Shared Address Space



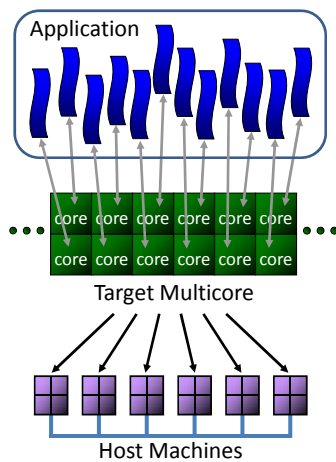
- All application threads run in a single simulated address space
- Memory subsystem provides modeling as well as functionality
- Functionality implemented as part of the target memory models
 - Eliminate redundant work
 - Test correctness of memory models



15



Thread Distribution



- Graphite runs application threads across several host machines
- Must initialize each host process correctly
- Threads are automatically distributed by trapping threading calls

16



System Calls



- Three kinds of system calls need to be handled specially
 - System calls that pass memory operands to the kernel
 - System calls that implement synchronization/communication between threads
 - System calls that deal with allocating and deallocating dynamic memory
- Other system calls can simply be allowed to fall through

17




Outline




- Introduction
- Graphite Architecture
 - Overview
 - Multi-machine distribution
 - Clock Synchronization
- Results
- Conclusions

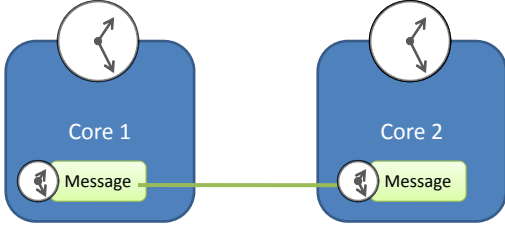
18




Clock Synchronization




- Cores *only* interact through messages
- Clocks are updated with message timestamps



19




Clock Synchronization




- Threads may run at different speeds, causing clocks to deviate
 - Clocks are only used for timing, functional correctness is always preserved
 - Must be synchronized on explicit interaction
 - Clocks may differ on implicit interaction → timing inaccuracy
- Define synchronization as *managing the skew of different target core clocks*.
 - This is *not* application synchronization!
- Graphite supports three synchronization schemes with different accuracy and performance tradeoffs

20



Synchronization Schemes

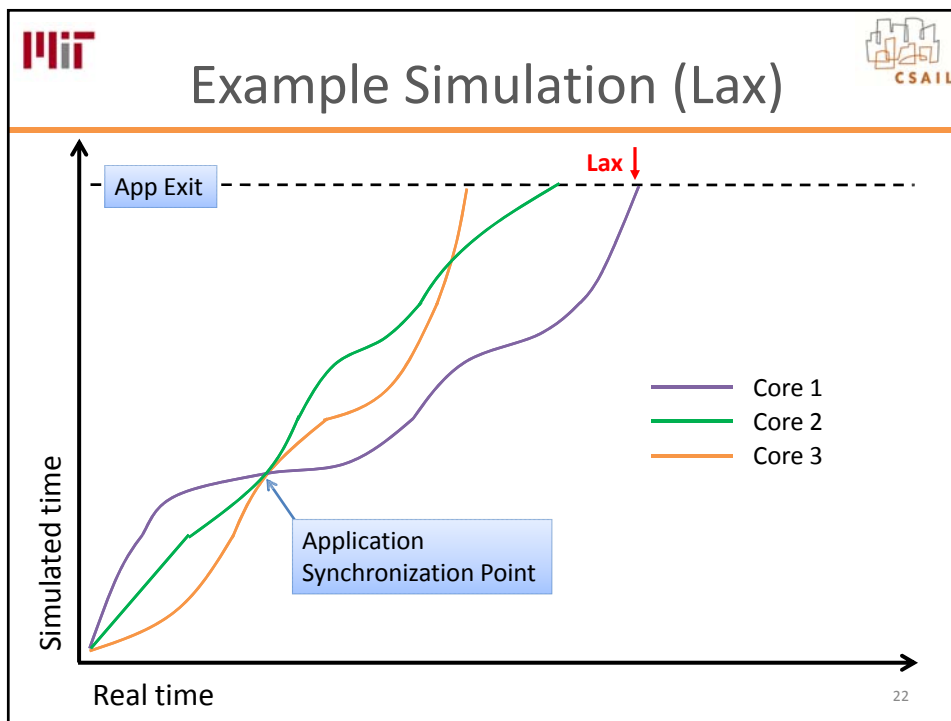


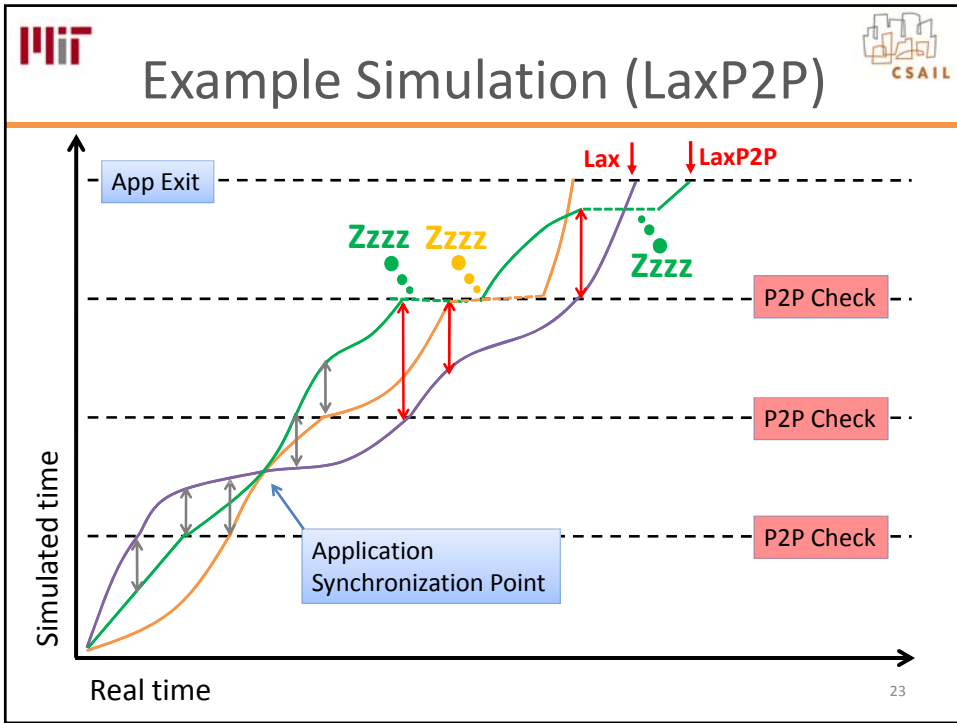
- Lax
 - Relies *exclusively* on application synchronization events to synchronize tiles' local clocks
 - Functionally, events may occur out-of-order w.r.t. simulated time
 - Best performance; worst accuracy

- LaxP2P
 - Observation: Timing inaccuracy is due to a few outliers
 - Every N cycles, each target core randomly pairs with another
 - If cycles differ by too much, 'future core' goes to sleep
 - Good performance; good accuracy

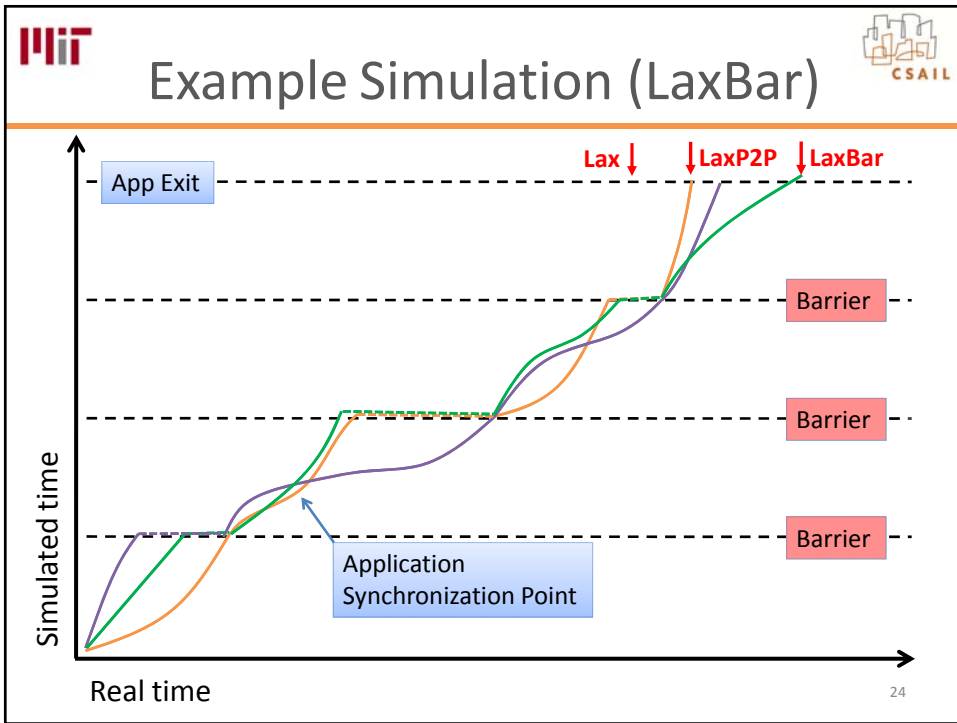
- LaxBar
 - Every N cycles, all target cores wait on a barrier
 - Keeps cores tightly synchronized, imitates cycle-accuracy
 - Worst performance; best accuracy

21

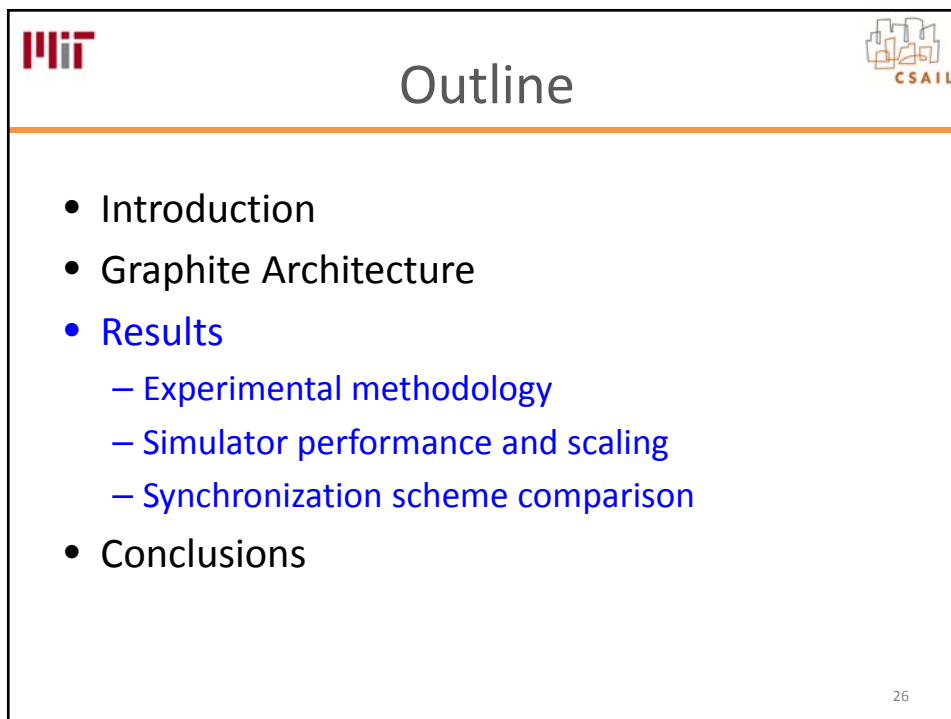
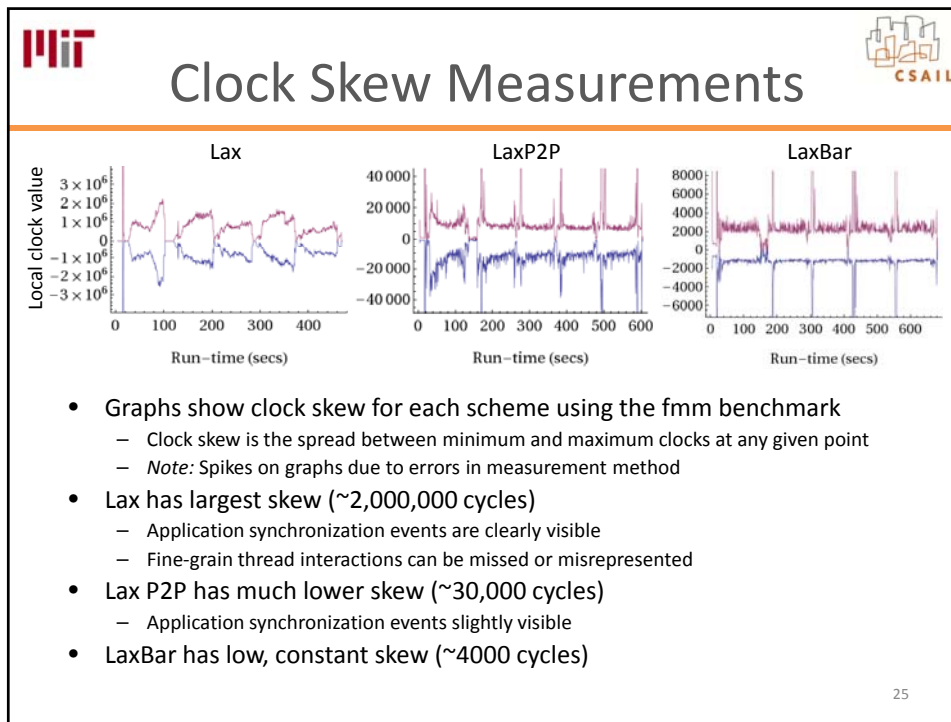


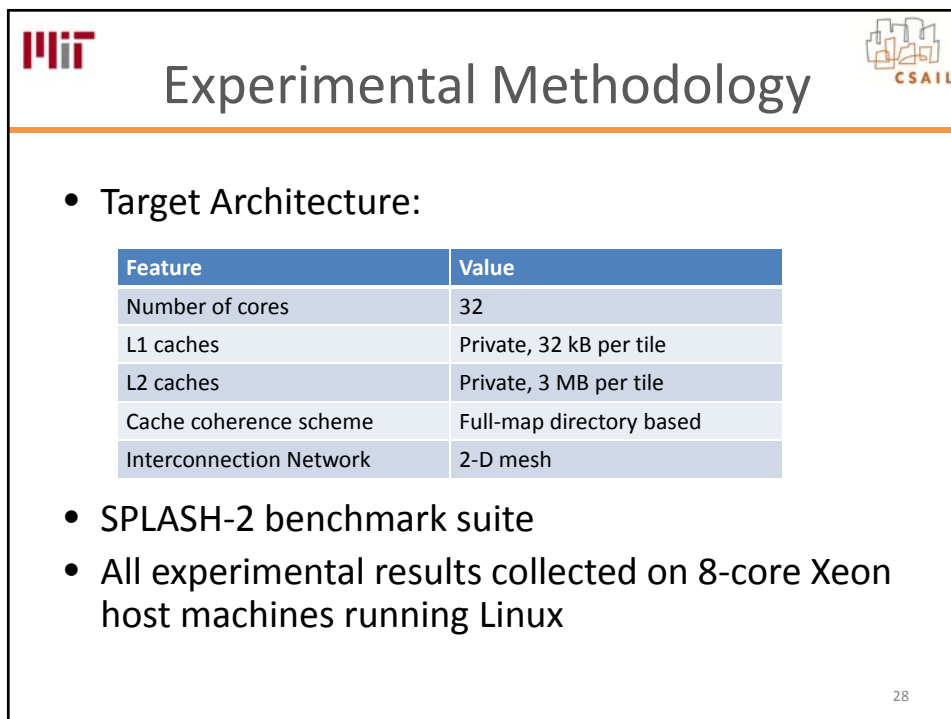
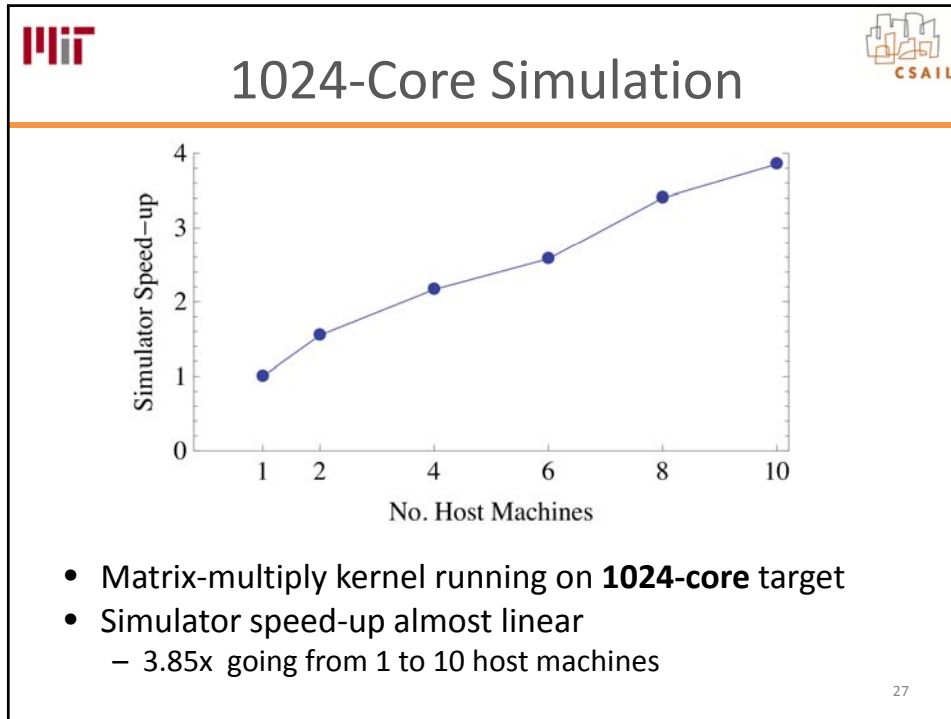


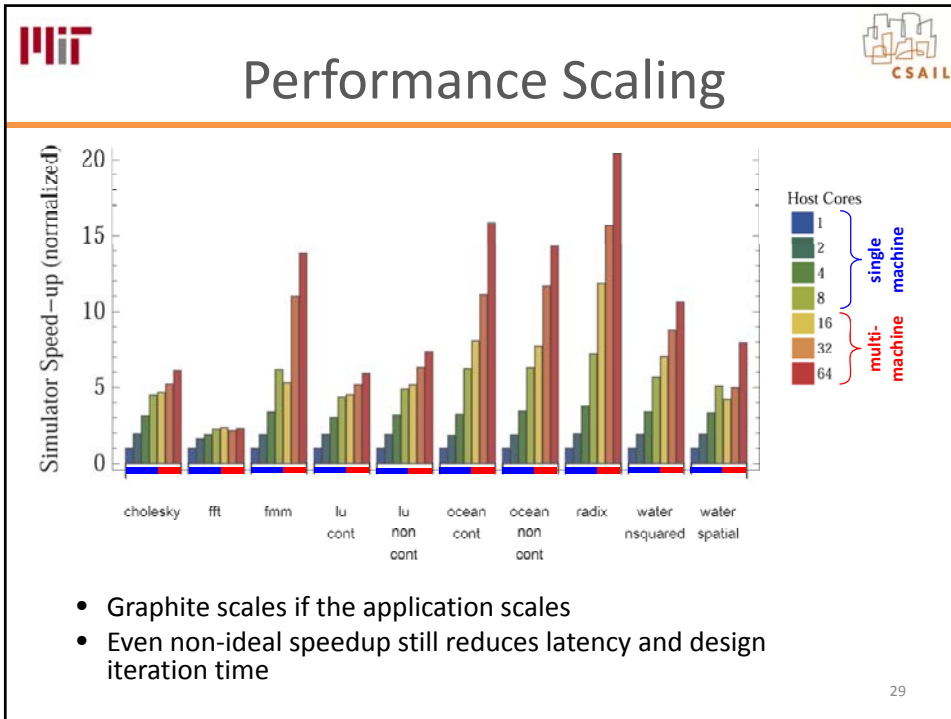
23



24







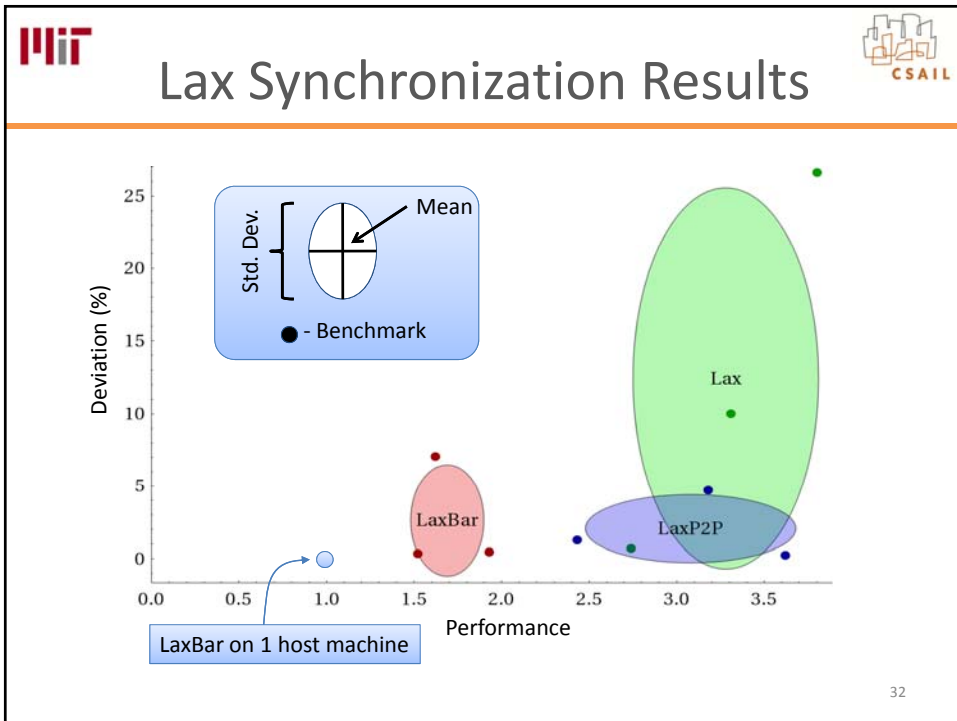
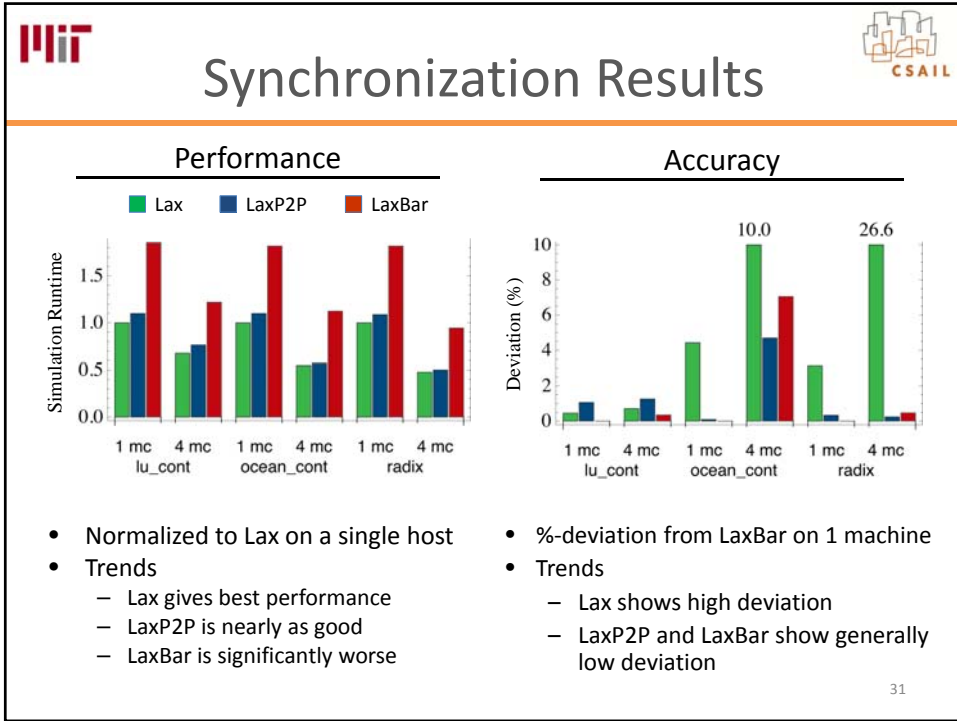
Performance Summary

| | Slowdown over native execution on 8 cores | | |
|---------------|-------------------------------------------|-------------------|---------------------|
| | Sequential (1 core) | 1 host* (8 cores) | 8 hosts* (64 cores) |
| Min | 580x | 94x | 41x |
| Max | 17,459x | 4007x | 3930x |
| Mean | 8,027x | 1751x | 1213x |
| Median | 6,940x | 1307x | 616x |

* Host machines are 8-core servers

- Sequential simulator slowdown is unacceptable
- Slowdown versus native execution as low as 41x
 - Would continue to drop with larger targets and more hosts
- Simulator overhead depends heavily on application characteristics
- Still more room for optimization

30





Summary

- Graphite accelerates multicore simulation using multi-machine parallel distribution
 - Enables simulation of 1000's of cores
 - Invisible to application, runs off-the-shelf pthread apps
- Graphite provides fast, scalable performance
 - As little as 41x slowdown vs. native execution
 - Up to 20x speedup on 64 host cores (across 8 machines)
- LaxP2P synchronization provides a good balance between performance and accuracy

Graphite Internals

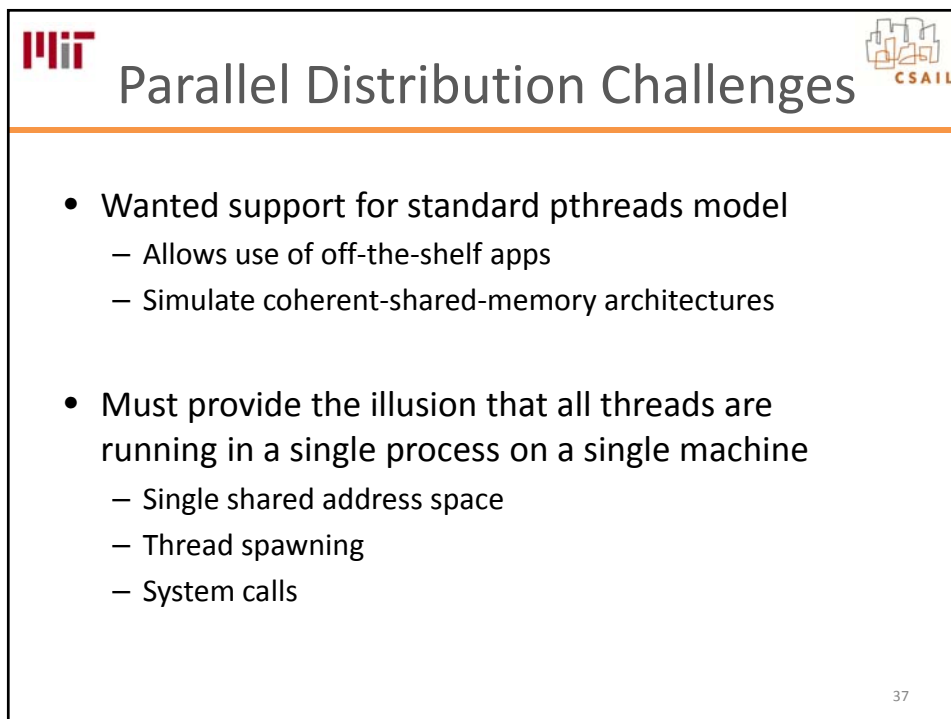
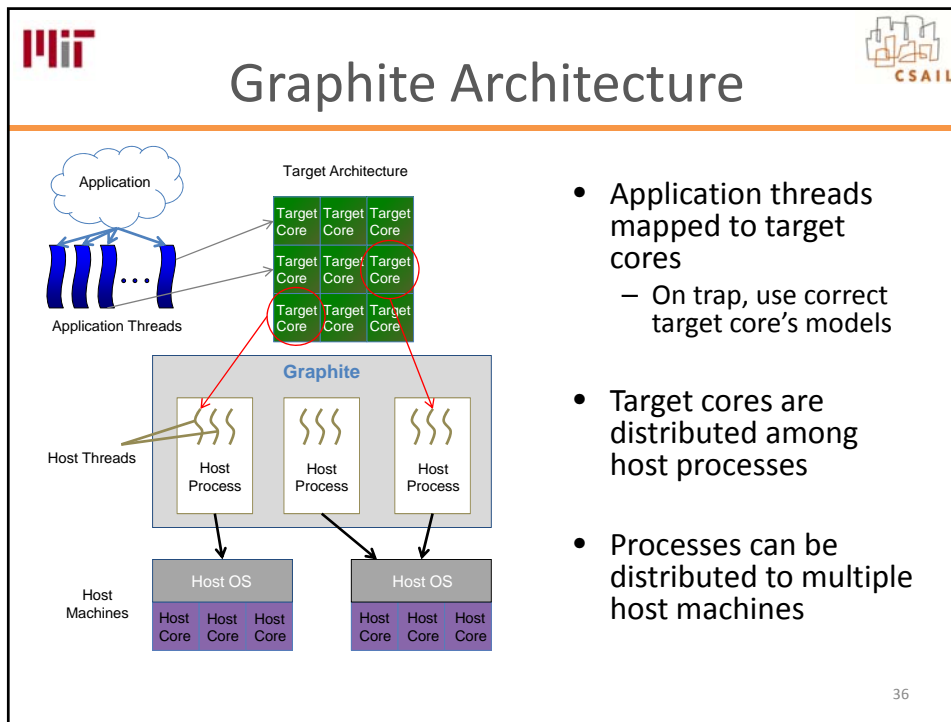
Additional Details of the Architecture
and Operation of the Simulator




Outline




- Multi-machine distribution
 - Single shared address space
 - Thread distribution
 - System calls
- Component Models
 - Overview
 - Core
 - Memory Hierarchy
 - Network
 - Contention
 - Power

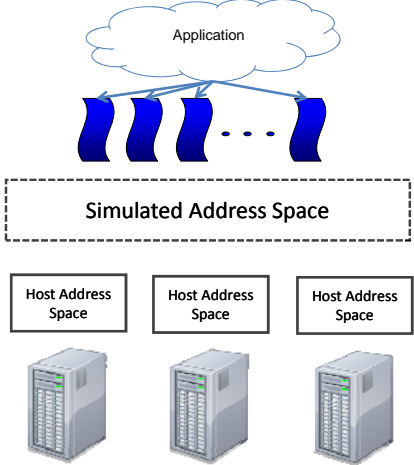





Single Shared Address Space




- All application threads run in a single simulated address space
- Memory subsystem provides modeling as well as functionality
- Functionality implemented as part of the target memory models
 - Eliminate redundant work
 - Test correctness of memory models




38

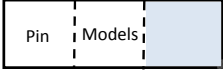


Simulated Address Space






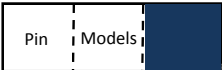
Simulated Address Space



Host Address Space





Host Address Space



Host Address Space

- Simulated address space distributed among hosts
- Graphite manages the simulated address space
 - Follows the System V ABI

39



Managing the address space

| | | | | | |
|--------------|-------------|--------------|---------------|--------------------------------|-----------------------|
| Code Segment | Static Data | Program Heap | Stack Segment | Dynamically Allocated Segments | Kernel Reserved Space |
|--------------|-------------|--------------|---------------|--------------------------------|-----------------------|

Simulated Address Space

- Stack space is allocated at thread start
- Appropriate syscalls are intercepted and handled by Graphite
 - mmap and munmap use dynamically allocated segments
 - brk allocates from program heap
- Memory accesses corresponding to instruction fetch not redirected
 - These accesses are still modeled
 - Don't support self modifying or dynamically linked code at the moment

40



Memory Bootstrapping

| | | | | | |
|----------------|---------------|--------------|---------------|--------------------------------|-----------------------|
| ✓ Code Segment | ✓ Static Data | Program Heap | Stack Segment | Dynamically Allocated Segments | Kernel Reserved Space |
|----------------|---------------|--------------|---------------|--------------------------------|-----------------------|

Simulated Address Space

- Need to bootstrap the simulated address space
 - Copy over code and data from the application binary
 - Copy over arguments and environment variables from the stack

41



Rewriting memory operands

Diagram illustrating the rewriting of memory operands for the instruction `AND addr, rax`.

The instruction `AND addr, rax` is rewritten into `Emulation code`. The emulation code interacts with the Target Memory System (L1 Cache, Cache Hierarchy, DRAM) via `read(addr, 8)` and `write(addr, 8)` operations.

- Graphite uses Pin API calls to rewrite memory accesses
- Data resides somewhere in the modeled memory system
 - May be on a different machine!
- Data access may span multiple cache lines

42

Rewriting memory operands (contd.)

Diagram illustrating the rewriting of memory operands for the instruction `ADD addr, rax`.


The instruction `ADD addr, rax` is rewritten into `Emulation Code`. The emulation code interacts with the Target Memory System (L1 Cache, Cache Hierarchy, DRAM) via `read(addr, 8)` and `write(addr, 8)` operations. The diagram highlights the execution flow with numbered steps:

- ① `read(addr, 8)` operation
- ② `Emulation Code` execution
- ③ `write(addr, 8)` operation


The word **Execute** is written in green next to the emulation code.

- Solution: scratchpads!**

43



Atomic memory operations



`LOCK CMPXCHG addr, rax`

Rewrite →

Emulation Code


Execute

Target Memory System


L1 Cache
 Cache Hierarchy
 DRAM

- Need to prevent other cores from modifying data
 - Lock the private L1 cache during execution
 - This together with the cache coherence protocol ensures atomicity

44





Thread Distribution

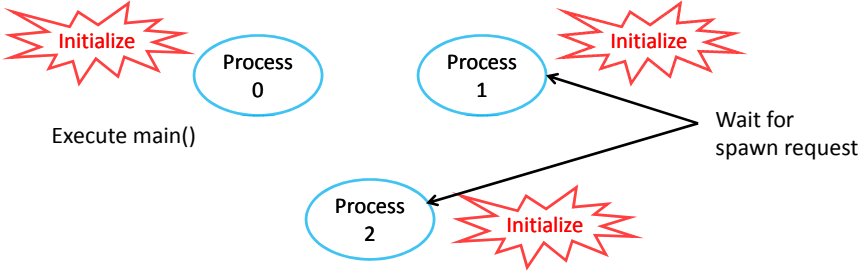


- Graphite runs application threads across several host machines
- Must initialize each host process correctly
- Threads are automatically distributed by trapping threading calls

45






Process Initialization



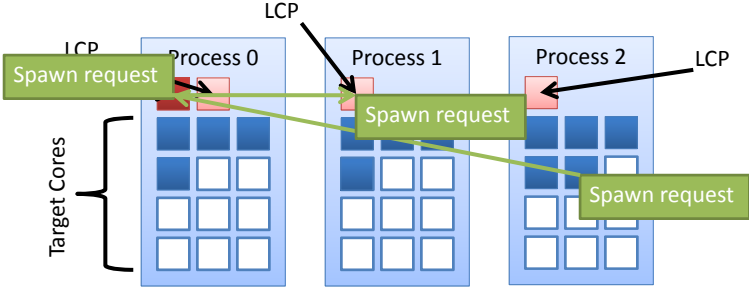
- Need to initialize state correctly in each process (glibc initialization, TLS setup)
- Execute initialization routines serially in each process
- Process 0 executes main()

46






Thread Spawning

- Thread distribution managed through MCP/LCPs
 - MCP and LCPs not part of target architecture
 - Perform management tasks (thread spawning, syscalls, etc.)





47

Thread Management

- MCP keeps table of thread state
- Performs simple load balancing on spawns
 - Target cores striped across host processes
 - Future work: better scheduling/load balancing
- Implements *pthread* API by intercepting calls
 - *Pthread_create()* initiates a spawn request to MCP
 - *Pthread_join()* messages MCP and waits for a reply when thread exits

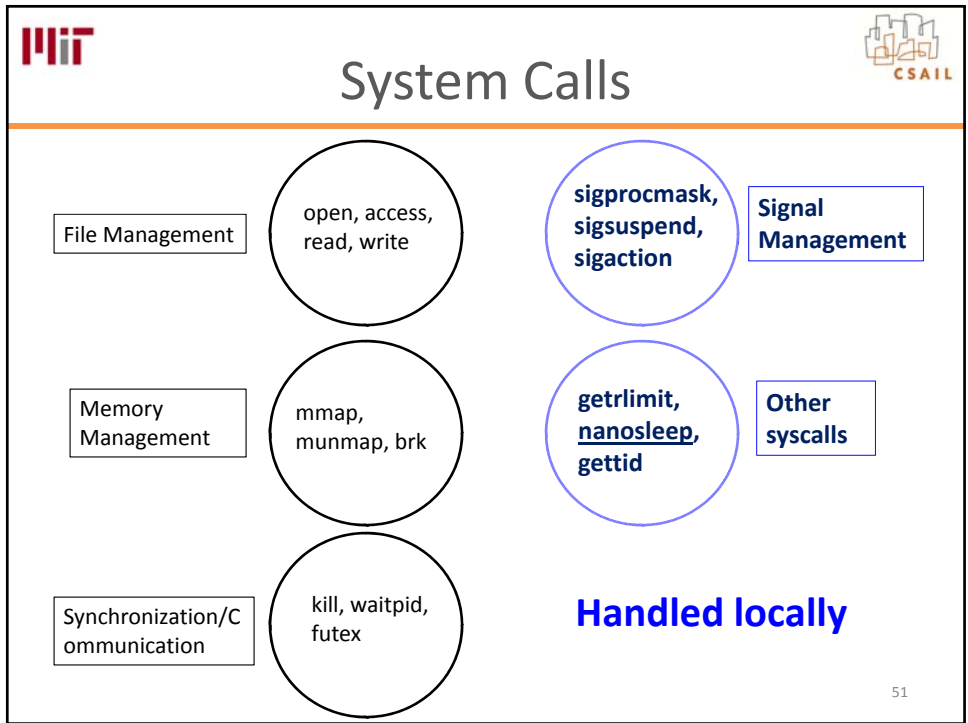
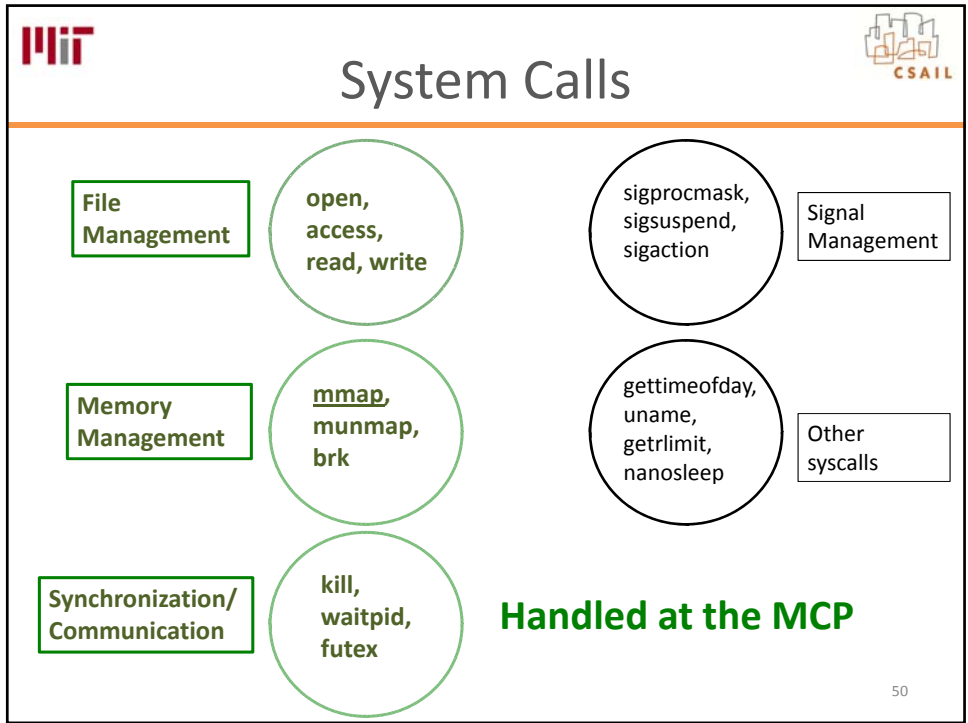
48

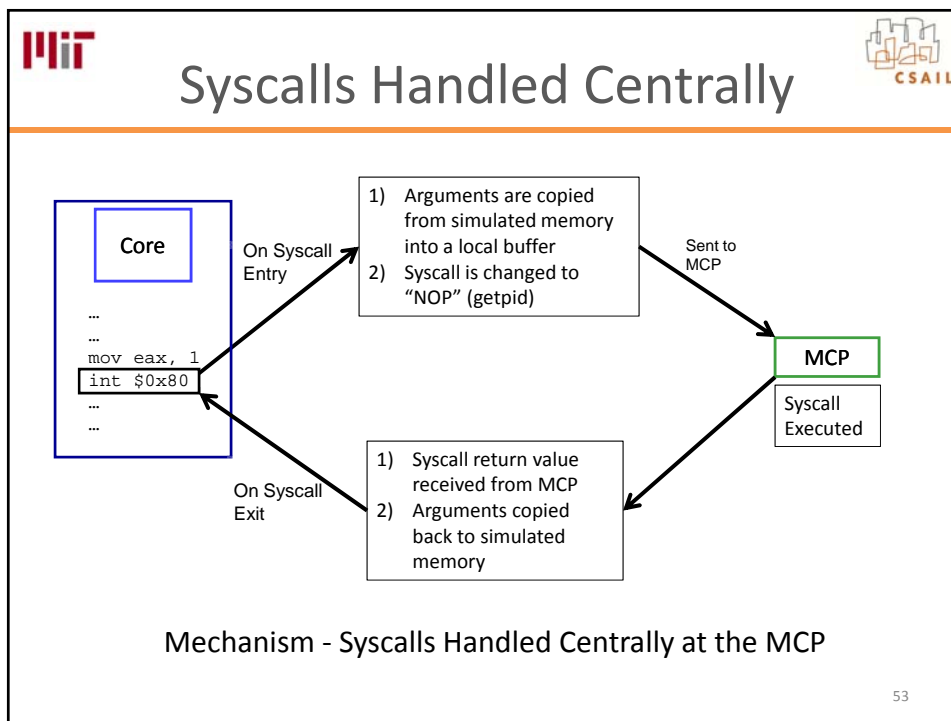
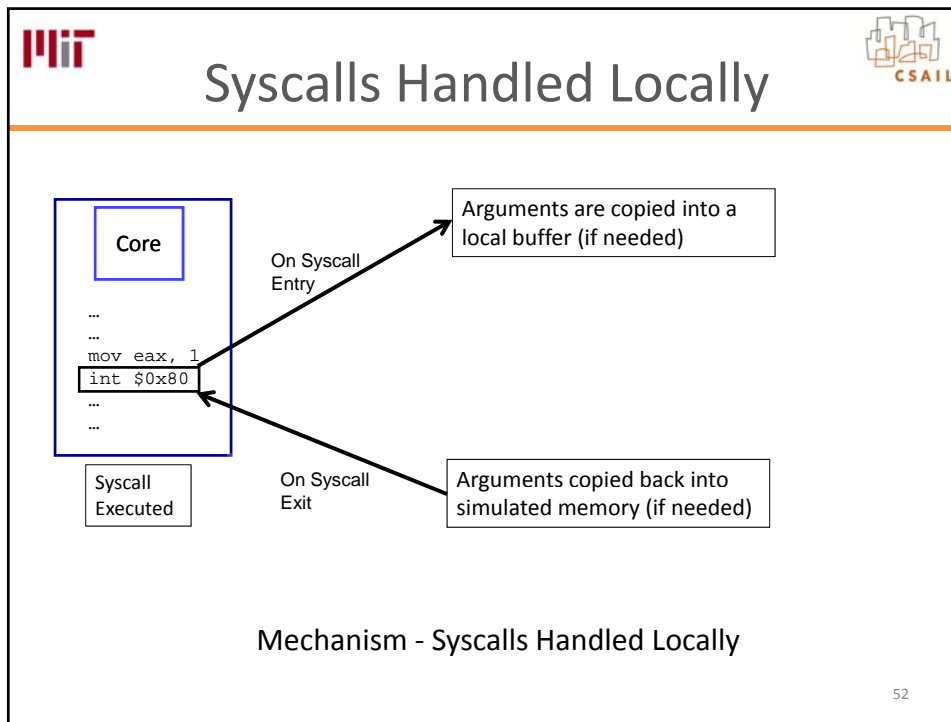



System Calls

| | | | |
|-----------------------------------|------------------------------|------------------------------------------|----------------------|
| File Management | open, access, read, write | sigprocmask, sigsuspend, sigaction | Signal Management |
| Memory Management | mmap, munmap, brk | getrlimit, nanosleep, gettid | Other syscalls |
| Synchronization/C ommunication | kill, waitpid, futex | | |

49







Application Synchronization



- Normal futex / atomic instructions
 - Useful for pthread style programs
 - Falls through to mechanisms previously described
 - Implemented via memory system
- Application function calls (*i.e.*, Barrier())
 - Gets replaced by a simulated version
 - Allows exploration of architectural support for synchronization mechanisms
 - Does not depend on the memory system

54




Outline




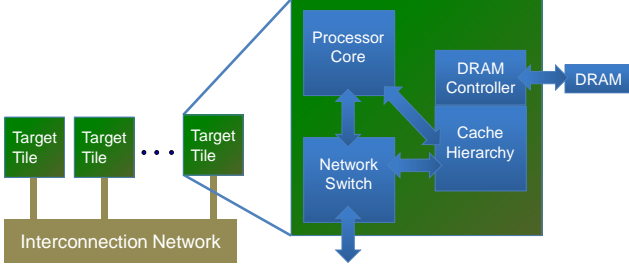
- Multi-machine distribution
 - Single shared address space
 - Thread distribution
 - System calls
- Component Models
 - Overview
 - Core
 - Memory Hierarchy
 - Network
 - Contention
 - Power

55




Simulated Target Architecture






- Swappable models for processor, network, and memory hierarchy components
 - Explore different architectures
 - Trade accuracy for performance
- Cores may be homogeneous or heterogeneous

56

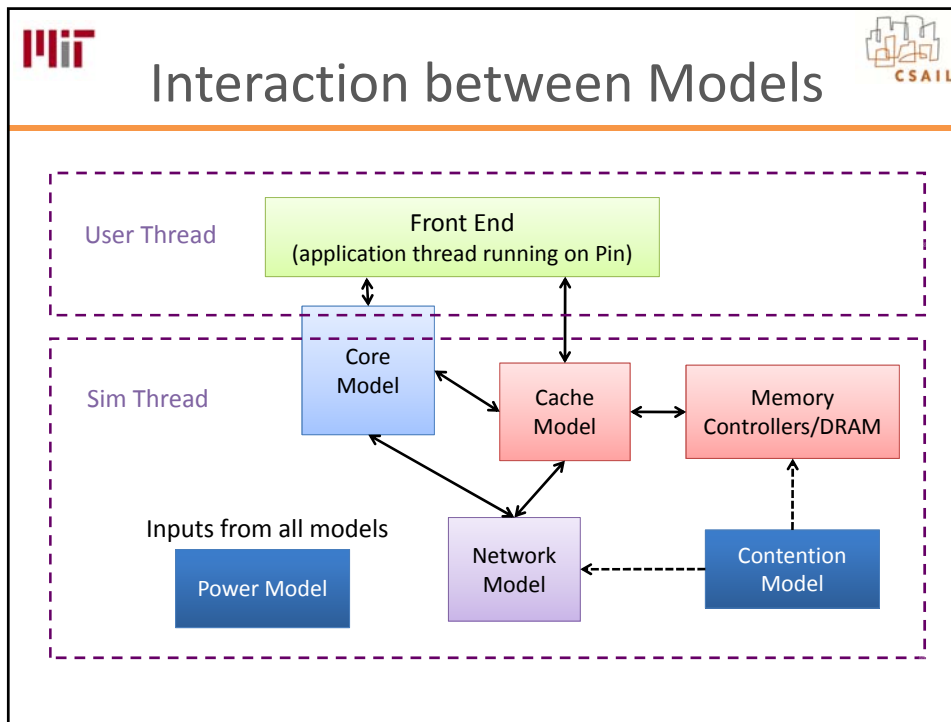


Modeling Overview



- Functional and timing components are separate where possible
 - Exceptions made for performance reasons
- Functionality
 - Direct-execution of as many instructions as possible
 - Trap into simulator for new behaviors
- Timing (performance)
 - Inputs from front end and functional components used to update simulated clock
- Each tile actually has two threads
 - User thread is the original application thread instrumented by Pin
 - Sim thread executes most models (including memory and network)

57



Core Modeling

- Performance model completely separate from functional component
 - Application executes natively
 - Stream of events fed into timing model
- Inputs from Pin as well as dynamic information from the network and memory components
 - Instruction stream
 - Latency of memory and network operations
- The current model is a simple in-order model
 - Fixed number of cycles for different classes of instructions
 - Allows multiple outstanding memory operations
- “Special instructions” used to model aspects such as message passing

59



Memory Modeling



- Private L1, L2 caches in each tile
- Directory-based coherence scheme for L2
 - Directory in DRAM, directory caches in each tile
 - Directory caches communicate with DRAM controllers via network messages
- Configurable number of controllers/DRAM channels
- Memory models are both functional and timing
 - Target coherence scheme used to maintain coherence across machines
 - Messages are used both to communicate data/update state and to compute latencies
- DRAM contention modeled by queuing models

60



Network models



- Functional and timing components
 - Functional: Determines routing algorithms
 - Timing: Calculates latencies
- Uses Physical Transport layer to send messages to other cores' network models
- Calculates queuing and delivery latencies for packets
- Opportunity for performance/accuracy trade-off
 - Timing may be analytical, fully detailed or a combination

61



Contention Models

- Used by network and DRAM to calculate queuing delay
- Analytical Model
 - Using an M/G/1 Queuing Model
 - Inputs are link utilization, average packet size
- History of Free Intervals
 - Captures history of network utilization
 - More accurately handles burstiness and clock skew



Power Models

- Work in progress
- Activity counters track events during simulation
 - E.g., cache access, network link traversal
 - Energy calculated from static and dynamic components
- Models available for following components:
 - Network (using Orion)
 - Caches (using CACTI)
- Currently under development:
 - Cores (using Wattch)
 - DRAM



Summary

- Special techniques used for distributed simulation:
 - Single, distributed shared address space
 - Thread spawning and distribution
 - Syscall interception and proxying
- Graphite provides models for core, memory, and network subsystems
- Contention and power models are used to support the other models