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
Graphite Architecture

- Application threads mapped to target tiles
  - On trap, use correct target tile’s models

- Target tiles are distributed among host processes

- Processes can be distributed to multiple host machines
Parallel Distribution Benefits

• Accelerate slow simulations
  – Additional compute/cache resources
  – Reduces latency of simulation for quick turn-around
  – Some efficiency lost to communication overhead

• Enable huge simulations
  – Can easily exhaust OS or memory resources of one machine
  – Additional DRAM, memory bandwidth
  – Additional thread contexts
  – No other way to run these experiments
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
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
• Simulated address space distributed among hosts
• Graphite manages the simulated address space
  – Follows the System V ABI
Managing the address space

<table>
<thead>
<tr>
<th>Code Segment</th>
<th>Static Data</th>
<th>Program Heap</th>
<th>Stack Segment</th>
<th>Dynamically Allocated Segments</th>
<th>Kernel Reserved Space</th>
</tr>
</thead>
</table>

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
Memory Bootstrapping

<table>
<thead>
<tr>
<th>Code Segment</th>
<th>Static Data</th>
<th>Program Heap</th>
<th>Stack Segment</th>
<th>Dynamically Allocated Segments</th>
<th>Kernel Reserved Space</th>
</tr>
</thead>
</table>

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
Rewriting memory operands

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
Rewriting memory operands (contd.)

ADD \textit{addr}, rax

\textbf{Solution: scratchpads!}
Atomic memory operations

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

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

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

File Management:
- open, access, read, write

Memory Management:
- mmap, munmap, brk

Synchronization/Communication:
- kill, waitpid, futex

Signal Management:
- sigprocmask, sigsuspend, sigaction

Other syscalls:
- getrlimit, nanosleep, gettid
System Calls

- **File Management**
  - open, access, read, write

- **Memory Management**
  - mmap, munmap, brk

- **Synchronization/Communication**
  - kill, waitpid, futex

- **Signal Management**
  - sigprocmask, sigsuspend, sigaction

- **Other syscalls**
  - getrlimit, nanosleep, gettid

**Handled locally**
System Calls

File Management
- open, access, read, write

Memory Management
- mmap, munmap, brk

Synchronization/Communication
- kill, waitpid, futex

Signal Management
- sigprocmask, sigsuspend, sigaction

Other syscalls
- getrlimit, nanosleep, gettid

Handled at the MCP
Syscalls Handled Locally

**Mechanism - Syscalls Handled Locally**

- **Core**
  
  ...  
  ... 
  mov eax, 1
  int $0x80  
  ...  
  ...

- **Syscall Executed**
  
  Arguments are copied into a local buffer (if needed)

  On Syscall Entry

  Arguments copied back into simulated memory (if needed)

  On Syscall Exit
Syscalls Handled Centrally

Mechanism - Syscalls Handled Centrally at the MCP

1) Arguments are copied from simulated memory into a local buffer
2) Syscall is changed to “NOP” (getpid)

1) Syscall return value received from MCP
2) Arguments copied back to simulated memory

On Syscall Entry

On Syscall Exit

Sent to MCP

MCP

Syscall Executed

... mov eax, 1
int $0x80
...
Emulated Syscalls

• Some system calls have to be emulated and not simply executed locally/centrally

• Synchronization (e.g., futex)
  – Ensures global thread co-ordination and correct target time updates

• Memory management (brk, mmap, munmap)
  – Ensures global management of virtual memory

• Syscalls with modeled state dependence
  – E.g., clock_gettime (needs target time)
Application Synchronization

• Normal futex / atomic instructions
  – Useful for pthread style programs
  – Falls through to mechanisms previously described
  – Implemented via memory system

• Application function calls (e.g., Barrier())
  – Gets replaced by a simulated version
  – Allows exploration of architectural support for synchronization mechanisms
  – Does not depend on the memory system
Outline

• Multi-machine distribution
  – Single shared address space
  – Thread distribution
  – System calls

• Component Models
  – Overview
  – Core
  – Memory Hierarchy
  – Network
  – Contention
  – Power
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
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
• Energy
  – Estimated on-line using events from modeled components
• Each tile actually has two threads
  – App thread is the original application thread instrumented by Pin
  – Sim thread executes most models (including memory and network)
Interaction between Models

Front End
(application thread running on Pin)

- Core Model
- Cache Model
- Memory Controllers/DRAM
- Network Model
- Contention Model

Inputs from all models

Power Model
McPAT/DSENT
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
  – Basic pipeline with configurable latency for different classes of instructions
  – Allows multiple outstanding memory operations
• “Special instructions” used to model aspects such as message passing
Memory Modeling

1) Private L1, private L2 cache hierarchy
   - Directory-based coherence scheme for L2
   - Directory distributed across all tiles

2) Private L1, shared L2 cache hierarchy
   - L2 cache distributed across all tiles
   - Directory co-located with L2 tags

- 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
Network models

• Functional and timing components
  – Message type (unicast, multicast, broadcast)
  – Routing algorithms
  – Network traversal latencies (queuing, serialization, zero-load)

• Uses Physical Transport layer to send messages to other cores’ network models
• Opportunity for performance/accuracy trade-off
  – Timing may be analytical, fully detailed or a combination
Contestion 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

• Activity counters track events during simulation
  – E.g., cache access, network link traversal
  – Energy calculated online from static and dynamic components

• Models available for following components:
  – Network (using DSENT)
  – Caches (using McPAT)
  – Cores (using McPAT)
Summary

• Special techniques used for distributed simulation:
  – Single, distributed shared address space
  – Thread spawning and distribution
  – Syscall interception and proxying

• Graphite provides performance and power models for core, memory, and network subsystems
  – Modular architecture makes it easy to create new models
Examples of Alternate Models

- Ghent U., Intel Exascience Lab
  - Replaced core model with “interval simulation”
- Univ. of New Mexico
  - Modified cache models
- CSG group @ MIT
  - Modified cache protocols
  - Added special-purpose multi-threading to core
- MIT, Intel PAR group
  - Implemented hierarchy of shared caches
  - New cache coherence protocol
  - Formally verified with Murphi checker
- Carbon group (in-house)
  - Multiple network models (accuracy/performance, on-chip optics)
  - Multiple cache coherence protocols