Architectural Models and Software APIs’ in Graphite

What’s available and how to extend
Architectural Models Overview

• Architectural Models
  – List of available models in Graphite
  – Configuration Parameters
  – Base Class / Main Interface Functions
• Compute core, memory subsystem, messaging API and network
Architectural Models - Outline

- Memory Subsystem
- Core Model
- Network Model
- Contention Model
- Heterogeneity
Architectural Models - Outline

• Memory Subsystem
• Core Model
• Network Model
• Contention Model
• Heterogeneity
Memory Subsystem

• Handles Load/Store requests from cores
• Both a functional and performance model
  – Verify correctness of cache coherence protocols
• Enable application to share same address space across multiple machines
• Consists of the on-chip cache hierarchy, directories, memory controllers and off-chip DRAM
## Memory Subsystem

### Cache Hierarchy Models

<table>
<thead>
<tr>
<th>Private L1</th>
<th>Private L1, private L2 hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>MSI</td>
</tr>
<tr>
<td>MSOI</td>
<td>MSI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared L2</th>
<th>Private L1, shared L2 hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI</td>
<td>MSI</td>
</tr>
</tbody>
</table>
- Private L1-I, L1-D and L2 cache models
- Directory distributed across all the cores
- Directories communicate with memory controllers if data present off-chip
Memory Subsystem

Private-L1 Private-L2 Cache Hierarchy

- Directory size auto-adjusted according to L2 size
  - *Sparse Directories* (with configurable associativity)
- Two variants
  - MSI (Modified, Shared, Invalid)
  - MOSI (Modified, Owned, Shared, Invalid)
Memory Subsystem

Private-L1 Shared-L2 Cache Hierarchy

- Private L1-I and L1-D caches
- Distributed shared L2 cache (NUCA-style)
- Directory co-located with L2 cache
  - Placed in L2 cache tags (*in-cache directory*)
• Building block for private/shared caches
• Set-associative caches with configurable
  – Cache Size, Cache Block Size, Associativity, Replacement Policy, Access Time
• Used to implement L1-I, L1-D and L2 caches
Memory Subsystem
Caches: Configuration & Interface

• Configuration Parameters
  - [l1_icache/T1]
    cache_size = 32 # In KB
    cache_line_size = 64
    associativity = 4
    replacement_policy = LRU # Least Recently Used
    data_access_time = 1 # In cycles

• Base Class / Interface Functions
  - class Cache (common/tile/memory_subsystem/cache/cache.h)
  - accessCacheLine()
  - insertCacheLine()
  - getCacheLineInfo()
Memory Subsystem
Directory

• Directory Organizations for N cores
  – Full-Map Directory
    • N bits

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Sharer List (N bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Shared</td>
<td>0 1 1 1 ... 0 1</td>
</tr>
</tbody>
</table>
• Directory Organizations for N cores
  – Full-Map Directory
    • Hardware tracking for all possible sharers
    • N bits
  – Limited Directory \( (\text{Dir}_k \text{NB}, \text{Dir}_k \text{B}, \text{ACKwise}) \)
    • Hardware only tracks k sharers
    • \( k \times \log_2(N) \) bits

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Sharer List (k hardware pointers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Shared</td>
<td>( \text{Core}<em>{s1} ) ( \text{Core}</em>{s2} ) ... ( \text{Core}_{sk} )</td>
</tr>
</tbody>
</table>
• Configuration Parameters
  – [directory]
    type = limited_broadcast # DirkB
    max_hw_sharers = 6

• Base Class / Interface Functions
  – class DirectoryEntry
    (common/tile/memory_subsystem/directory_schemes/directory_entry.h)
  – add(remove)Sharer()
    getSharersList()
  – get(set)Owner()
  – getNumSharers()
Memory Subsystem
Instruction Fetch Buffer

- Improves performance and reduces energy by reducing # accesses to L1 Instruction Cache
- Logically placed at Instruction-Fetch stage
- Consists of a single cache line of instructions
- Present in all current memory subsystem implementations
Memory Subsystem
Memory Controller

• Controller for off-chip memory
• Has contention model for accessing DRAM
• Memory Requests served in FIFO order
• Assumes no row buffer locality, infinite number of banks

• Configuration Parameters
  - [dram]
    latency = 100 # In ns
    per_controller_bandwidth = 5.0 # In GBps
    num_controllers = 4
Architectural Models - Outline

- Memory Subsystem
- **Core Model**
- Network Model
- Contention Model
- Heterogeneity
Core Model

- Models the instruction fetch/decode, execution units and load/store units
- Input: Instruction stream from Pin + Dynamic information from the network and memory
- “Special instructions” used to model message passing and synchronization
- Purely a performance model
Core Model
Existing Models & Limitations

• Models assume constant instruction costs except for memory accesses and branch prediction

• Existing Models in Graphite
  – *simple*
    – In-order core model with in-order memory completion
  – *iocoom*
    • In-order core model with out-of-order memory completion
    • Register scoreboard tracks dependencies
Core Model
Configuration & Interface

• Configuration Parameters (General & Model-Specific)
  - [core/static_instruction_costs]
    add = 1 # In cycles
    mul = 3 # In cycles
  - [core/iocoom]
    num_outstanding_loads = 8
    num_store_buffer_entries = 8

• Base Class / Interface Functions
  - class CoreModel
    (common/tile/core/core_model.h)
  - handleInstruction(Instruction* ins)
Architectural Models - Outline

• Memory Subsystem
• Core Model
• **Network Model**
• Contention Model
• Heterogeneity
Network Model

• Models the on-chip interconnection network
• Graphite supports 3 networks
  – User-level messages
  – Shared memory messages
  – System messages
Network Model

Existing Models in Graphite

• Two models for electrical mesh networks
  – Hop Counter
  – Hop-By-Hop (Support broadcast tree)

• Opto-electronic network model (ATAC)

• Magic network model
  – unicast/multicast/broadcast takes 1 cycle
Network Model
Performance and Accuracy Trade-offs

- Electrical mesh networks

<table>
<thead>
<tr>
<th></th>
<th>Hop Counter</th>
<th>Hop By Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Routing</td>
<td>Sent directly to destination core</td>
<td>Stops at intermediate routers (X-Y routing)</td>
</tr>
<tr>
<td>Contention Modeling</td>
<td>No</td>
<td>Models output link contention</td>
</tr>
<tr>
<td>Performance</td>
<td>High (+)</td>
<td>Low (-)</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Low (-)</td>
<td>High (+)</td>
</tr>
</tbody>
</table>
Parameterized router and link model enable easy implementation of complex network topologies

- **RouterModel** (frequency, num_input_ports, num_output_ports, etc.)
- **LinkModel** (frequency, link_length, link_width)

Model designer takes care of only topology and routing

Power modeling, contention, packet time updates all handled under the covers

Could be used to implement electrical or optical networks (Examples: *emesh_hop_by_hop*, *atac*)
Network Model
Implementing Indirect Networks

• How are intermediate routers & links processed?
  – Routers in a fat tree
  – Nodes in a butterfly/clos network

• Solution
  – Assign processing of intermediate elements to a “handler” tile
  – Create router & link models within NetworkModel object of “handler” tile

• Example: atac network model
Network Model

Assumptions

• Only link contention modeled
• Infinite output buffering
• Will show comparison against a network with finite input buffers and wormhole flow control
• Configuration parameters
  
  – [network]
    user_model = emesh_hop_counter
  
  – Network model-specific configuration params
    • [network/emesh_hop_counter/router]
      delay = 1

• Base Class / Interface Functions
  
  – class NetworkModel (common/network/network_model.h)
  
  – routePacket()
    processReceivedPacket()
Architectural Models - Outline

• Memory Subsystem
• Core Model
• Network Model
• Contention Model
• Heterogeneity
Contention Models

• Compute the queuing delay when accessing a shared object (server)

• Examples of shared objects
  – Network Links
  – Off-chip Memory
  – Shared Caches
Contention Models

A hard problem in Graphite!

- NOT easy as in a cycle-accurate simulator
- Each core has independent clock
- Packets arrive with out-of-order timestamps
Contention Models
A hard problem in Graphite!

- NOT easy as in a cycle-accurate simulator
- Each core has independent clock
- Packets arrive with out-of-order timestamps
Contention Models
A hard problem in Graphite!

- NOT easy as in a cycle-accurate simulator
- Each core has independent clock
- Packets arrive with out-of-order timestamps

Queuing Delay?
Contestation Models
Existing Models in Graphite

• History Tree
  – Model traffic burstiness, hotspots, etc
  – Queuing delay calculated from Server Utilization History

• Analytical M/G/1 Queuing Model
  – Queuing delay calculated from Average Server Utilization, Packet Length Statistics
• Utilization history stored as a set of free intervals
• Each interval corresponds to a time interval when server is unoccupied
• Queuing delays calculated by searching the history
• Size of history adjusted according to maximum permissible clock skew

• History organized as a self-balancing binary tree of intervals
  – Tree rotations used for height balancing
  – $O(\log_2 N)$ average complexity for search
• Packet with length 5 arrives at time 52
• History Tree Structure shown below
  – (70 – 74), (100 – 110) are free intervals
  – (15 – 35), (110 – 167) are utilized intervals
Contention Models

History Tree Example (cont’d)

• Compute Queuing Delay (time = 52, length = 5)
• Compute Queuing Delay (time = 52, length = 5)
• Compute Queuing Delay (time = 52, length = 5)
Contention Models
History Tree Example (cont’d)

• Compute Queuing Delay (time = 52, length = 5) → 7
ContenEon Models
History Tree Example (cont’d)

- Compute Queuing Delay (time = 52, length = 5) \(\rightarrow\) 7
- History Tree adjusted according to reflect server utilization
Contestion Models
Analytical M/G/1

• Approximates contention delay using statistical parameters
• Stores the following parameters across entire history
  – Net Utilization ($\rho$)
  – Average Packet Length ($\mu_{len}$)
  – Standard Deviation of Packet Length ($\sigma_{len}^2$)

\[
\text{Queueing Delay} = \frac{\rho}{2(1 - \rho)} \left( \mu_{len} + \frac{\sigma_{len}^2}{\mu_{len}} \right)
\]
## Contention Models

### Performance Accuracy Trade-off

<table>
<thead>
<tr>
<th></th>
<th>History Tree</th>
<th>Analytical M/G/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Adaption to Network Traffic</td>
<td>Yes</td>
<td>No (Averages across entire history)</td>
</tr>
<tr>
<td>Performance</td>
<td>Low (-)</td>
<td>High (+)</td>
</tr>
<tr>
<td>Accuracy</td>
<td>High (+)</td>
<td>Low (-)</td>
</tr>
</tbody>
</table>
• Configuration Parameters
  – [network/emesh_hop_by_hop/queue_model]
    type = history_tree
  – Model-specific configuration params
    • [queue_model/history_tree]
      max_list_size = 100

• Base Class / Interface Functions
  – class QueueModel
    (common/shared_models/queue_model.h)
  – computeQueueDelay(uint64_t time, uint64_t length)
• Cycle-Level mode simulates all architectural models on a cycle-by-cycle basis
  – Network has finite input buffers, wormhole flow control
• LaxP2P is only 6.4% off from cycle-level
Architectural Models - Outline

• Memory Subsystem
• Core Model
• Network Model
• Contention Model
• Heterogeneity
Heterogeneity

• Cores can be configured with the following performance asymmetries:
  – Frequency
  – Different Core Models (simple/iocoom)
  – Private L1/L2 Cache Sizes and Organizations
• All cores follow the same ISA
Heterogeneity

Uses

• Uses
  – Modeling Helper Core
    • Core – very low frequency and simple core model
  – Modeling 1 Big Core + Many small cores
    • Big core – high frequency and sophisticated core model
Heterogeneity Configuration

• Configuration Parameters
  Format of each tuple
  < # Cores, Frequency, Core Model Type, L1-I Config, L1-D Config, L2 Config >
  
  [core]
  model_list = “<30, 1.0 , simple, T1, T1, T1>,
  <2 , 2.5 , iocoom, T2, T2, T2>”

  [core/iocoom]
  num_store_buffer_entries = 8
  num_outstanding_loads = 8

  [l1_icache/T1]
  cache_line_size = 64 # In Bytes
  cache_size = 32 # In KB
  associativity = 4
  replacement_policy = lru

  [l1_dcache/T2]
  cache_line_size = 64 # In Bytes
  cache_size = 32 # In KB
  associativity = 4
  replacement_policy = lru