Architectural Models and Software APIs’ in Graphite

What’s available and how to extend
• 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></th>
<th>Private L1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Private L2</td>
<td></td>
<td>MSI MOSI</td>
</tr>
<tr>
<td>Shared L2</td>
<td></td>
<td>MSI</td>
</tr>
</tbody>
</table>

- Private L1, private L2 hierarchy
- Private L1, shared L2 hierarchy
Memory Subsystem

Private-L1 Private-L2 Cache Hierarchy

- Private L1-I, L1-D and L2 cache models
- Directory distributed across all the cores
- Directories communicate with memory controllers
Memory Subsystem

Private-L1 Private-L2 Cache Hierarchy

- Directory size adjusted according to L2 cache size
  - Sparse directories
- 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
Memory Subsystem
Caches

• 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 1 ... 0 1</td>
</tr>
</tbody>
</table>
• Directory Organizations for N cores
  – Full-Map Directory
    • N bits
  – Limited Directory \((\text{Dir}_{k\text{NB}}, \text{Dir}_{k\text{B}}, \text{ACKwise})\)
    • \(k\log_2(N)\) 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>Core_{s1} Core_{s2} ... Core_{sk}</td>
</tr>
</tbody>
</table>
• Configuration Parameters
  - [directory]
    type = limited_broadcast # Dir_kB
    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()
• Controller for off-chip memory
• Has contention model for accessing DRAM
• Memory Requests served in FIFO order

• 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
- Dynamic Frequency Scaling (DFS)
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 and branch prediction
• Existing Models in Graphite
  – iocoom
    • In-order core model with out-of-order memory completion
Core Model
Configuration + Interface

• Configuration Parameters (General + Model-Specific)
  – [core/static_instruction_costs]
    add = 1
  – [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
• Dynamic Frequency Scaling (DFS)
Network Model

• Models the on-chip interconnection network

• Graphite supports 5 networks
  – 2 for user-level messages
  – 2 for shared memory messages
  – 1 for system messages
Network Model
Existing Models in Graphite

• Two models for electrical mesh networks
  – Hop Counter
  – Hop-By-Hop (Support broadcast tree)
• Electrical Clos network model
• Opto-electronic network model
• 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>
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_1 = 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()
Network Model
Router & Link Models

• Parameterized router and link model to model more complex network architectures
  - RouterModel(float frequency, SInt32 num_input_ports, SInt32 num_output_ports, SInt32 num_flits_per_port_buffer, UInt64 delay, SInt32 flit_width, bool contention_model_enabled, string& contention_model_type)
  - LinkModel(float frequency, double link_length, UInt32 link_width)

• Model designer takes care of only routing
• Power modeling, contention, packet time updates all handled under the covers
Architectural Models - Outline

- Memory Subsystem
- Core Model
- Network Model
- Contention Model
- Heterogeneity
- Dynamic Frequency Scaling (DFS)
Contention Models

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

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

Core 1
Core 2

Memory Controller

DRAM

Queuing Delay?

Time = 20
Length = 5
Contention 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**
Contetion Models – History Tree

• Utilization History stored as a set of free intervals
• Each free interval corresponds to a time interval when server is free
• Free intervals organized as a self-balancing binary tree
  – Tree Rotations used for height balancing
• Queuing Delays calculated by searching the history tree
  – Usually $O(\log_2 N)$ complexity for search
• Size of history tree adjusted according to maximum permissible clock skew
• 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
• Compute Queuing Delay (time = 52, length = 5)
• Compute Queuing Delay (time = 52, length = 5)
• Compute Queuing Delay (time = 52, length = 5)

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

- Can be used for uniform random traffic
- Stores the following 3 variables across entire history
  - Net Utilization ($\rho$)
  - Average Packet Length ($\mu_{len}$)
  - Standard Deviation of Packet Length ($\sigma_{len}^2$)

$$\text{QueueingDelay} = \frac{\rho}{2(1-\rho)} \left(\mu_{len} + \frac{\sigma_{len}^2}{\mu_{len}}\right)$$
# 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_basic/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 (magic/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
    < number of cores, frequency, core model,
      L1-I Cache Config, L1-D Cache Config, L2 Cache Config >
  – [core]
    model_list = "<30,1.0,magic,T1,T1,T1>,
                 <2 ,2.5,iocoom,T2,T2,T2>"
Software APIs’ - Outline

- Dynamic Frequency Scaling (DFS)
- Temporal Multithreading
Dynamic Frequency Scaling (DFS)

• Core Dynamic Frequency Scaling
  – Core Frequency can be increased/decreased at runtime by inserting a call in the application
  – CarbonGetCoreFrequency()
  – CarbonSetCoreFrequency()
  – See common/user/dvfs.h for APIs

• Network operates at constant frequency
• Changing core frequency changes cache frequency as well
Temporal Multithreading

- Thread scheduler time multiplexes different threads on the same core
- Configurable time quantum

```
[thread_scheduler]
quantum = 1000
```
Temporal Multithreading
Thread Placement and Migration

• Spawning function specify thread placement
  - CarbonSpawnThreadOnCore(core_id_t core_id, thread_func_t thread_func, void* arg)

• Software can declare core affinities
  - bool CarbonSchedSetAffinity(thread_id_t thread_id, Uint32 cpusetsize, cpu_set_t* set)
  - bool CarbonSchedGetAffinity(thread_id_t thread_id, Uint32 cpusetsize, cpu_set_t* set)