MeSSMArch – A Memory System Simulator for Hardware Multithreading Architectures

Sushil Menon

NVIDIA
Agenda

• Motivation (“Why build this simulator?”)
• TLM to the Rescue – Mapping Simulator Design Requirements to TLM Guidelines
• Modeling a Generic Memory-System at the Transaction-Level using TLM
• Simulator Validation
• Future Enhancements
• Conclusion
• Appendix
Why Memory Hierarchies? – Problem

- Increasing Software Footprint
- High Performance Processors
- Large Memories
- Integrate on same die
- Bad Performance!

Memory too far from processor!

Let's put the memory off-chip...

System on Board

Computer Architect

© Accellera Systems Initiative
Why Memory Hierarchies? – Solution

Caches (Smaller memories)

RAM (Larger memory)

Processor Die

System on Board

On-chip

Processor

Level#1 $

Level#N $

Off-chip

DRAM

Hierarchy of Memories

Higher Speed (Temporal Locality)

Higher Capacity (Spatial Locality)
Problematic Solution?! – Need for Simulators

Too many configuration options...best one???

Time-to-market pressure

Fast Simulators – Trade-off result
accuracy for simulation-speed by
raising level of abstraction!

⇒ Enable faster design-space exploration!

Hierarchy of Memories

© Accellera Systems Initiative
# TLM to the Rescue – Mapping Simulator Design Requirements to TLM Guidelines

## Design Requirement

<table>
<thead>
<tr>
<th>Functionally Generic</th>
<th>Don’t model functionality of μ-architectural features</th>
</tr>
</thead>
<tbody>
<tr>
<td>No implementation-specific details, easy to model and explore</td>
<td>Capture effects through timing information</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Estimate System Performance Only</th>
<th>Model data exchange at Transaction-Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse-grained result accuracy, fast simulation</td>
<td>Timing-accuracy via lumped delays</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Easily Extensible</th>
<th>Separate Computation from Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug-n-play style architectural exploration</td>
<td>Model computation details inside the process and communication details inside the channel</td>
</tr>
</tbody>
</table>

© Accellera Systems Initiative
Modeling a Generic Memory-System at the Transaction-Level using TLM

• Memory-System Components:
  – Generic Cache (full-flow covered)
  – Memory Controller (only structure)*
  – Serializing Interconnect (only structure)*
  – Hardware-Thread (only structure)*

• Construct Memory-Hierarchy for a Hardware Multithreaded Architecture

• Coarse-grained accuracy → All components modeled as loosely-timed
  – Implement b_transport( ) only

*Refer the Appendix Section and the Paper for More Details
Generic Cache – Transaction-Level Model

- Tag RAM – stores tag-address, dirty-bit, valid-bit and age-counters of a way
- Cache Controller – implements state-machines that capture functionality of cache
- Bus Interface Unit – implements interface for inter-module communication
## Generic Cache – Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit/Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>Kilobytes</td>
<td>Size of the cache</td>
</tr>
<tr>
<td>Cache Line-Size</td>
<td>Bytes</td>
<td>Size of a cache-line</td>
</tr>
<tr>
<td>Associativity</td>
<td>&lt;NA&gt;</td>
<td># of ways in a set</td>
</tr>
<tr>
<td># of Comparators</td>
<td>&lt;NA&gt;</td>
<td># of comparators used during a lookup</td>
</tr>
<tr>
<td>Write-Allocate</td>
<td>Yes/No</td>
<td>Allocation of a way-entry on a cache-miss</td>
</tr>
<tr>
<td>Write-Through</td>
<td>Yes/No</td>
<td>Generate write-transaction to lower-level on write-hit</td>
</tr>
<tr>
<td>Way Prediction</td>
<td>Yes/No</td>
<td>Predict way of current access to reduce lookup-time</td>
</tr>
<tr>
<td>Clock Period</td>
<td>Nanosecs</td>
<td>Time-period of a clock-cycle</td>
</tr>
</tbody>
</table>

Green entries indicate micro-architectural features
Generic Cache - µArch Features

• # of comparators
  – Used in parallel tag-lookup – don’t do parallel lookup!
  – Perform sequential tag-lookup and then divide the time by # of comparators
  – Equation: \( \frac{\text{Associativity}}{\text{# of Comparators}} \)

• Way Prediction
  – Don’t model setting of multiplexor to channel data, etc.
  – Perform normal lookup and conditionally adjust time based on way accessed (prediction: LRU-way)
  – Equation:
    
    \[
    \text{if (WP enabled \& HitWay == LRUWay) then } \text{# of clock cycles} = 1
    \]
Generic Cache – Transaction-Level Data Exchange

• Track cumulative # of clock-cycles during execution of state-machines at current-level without context-switching

• Conditionally forward transaction to lower-level – lumped time-delay received on return-path

\[ T_{TOHEE} = O_{OCC} + C_{COC} \]

• Return Total Transaction Time upstream as lumped time-delay

Total Transaction Time at Current Level
= (# of cycles at current level x cycle time)
+ lower level lumped time delay
The Other Memory-System Components....

1. Memory Mapping Unit – Physical to Raw-address (Row, Col, Bank) Translation
2. Command Generation Unit – generate commands for DRAM access (stored in command queues)
3. Row Buffer – stores row-address of currently opened row

- **Memory Controller**
  - **Bus Interface Unit**: Physical Address
  - **Memory Mapping Unit**: (Bank, Row, Col)
  - **Command Generation Unit**: Raw Address + Command (RowAct, ColAct, RowPre)
- **Target Socket**
- **Row Buffer #1**
- **Row Buffer #N**

- **Serializing Interconnect**
  - **Bus Interface Unit**: Trans Payload
  - **Arbiter**: Selected Buffer Index
  - **Pending Transaction Buffer**: Trans Payload
- **Target Socket**
- **Initiator Socket**

- **Hardware Thread**
  - **Load/Store Unit**: single-entry depth FIFO that generates transaction payload from load/store instruction
  - **Trace Parser**: infrastructure to read and parse benchmark file
**Sequentially Consistency**
Memory-Hierarchy → correct functional execution of mutual-exclusion primitives (refer the Appendix and the Paper for details)
Memory-System – Transaction-Level Model

1. Hardware-Thread acquires access to Serializing Interconnect

2. Serializing Interconnect injects transaction through memory-hierarchy to compute lumped delay
   Cumulative Lumped-Delay = \( t_{\text{TagLookup}} + t_{\text{DRAM-access}} \)

3. Arbiter invokes \( \text{wait}() \) to update timestamp
   \( \text{wait} (t_{\text{TagLookup}} + t_{\text{DRAM-access}}) \)

4. After timestamp update, Hardware-Thread releases Serializing Interconnect

Timing Accurate – Lumped delay!
Fast Simulation – Single context-switch only!

**Refer Appendix for Animation of Tracing of a Memory Transaction**
Simulator Validation

- Verify use-cases that signify fundamental tenets of memory-hierarchy

**Test Scenario#1:** Cache Use-Case

**Test Scenario#2:** Memory-Controller Use-Case

**Test Scenario#3:** Multithreading Use-Case

*Refer Appendix for Information about Benchmarks used and Paper for more Use-Cases*
Test Scenario 1: Sweep Cache Line-Size

Larger Cache-Line $\rightarrow$ Higher probability of Cache-hit (Spatial Locality) $\rightarrow$ Reduced Miss-Rate

...but miss-penalty increases!

Larger Cache-Line $\rightarrow$ More Data Fetched during Cache-miss $\rightarrow$ Increased Miss-Penalty
Test-Scenario 2: Larger DRAM pages reduce the average DRAM access latency

Larger DRAM page → Higher Probability of Row-Buffer hit (Spatial Locality) → Lower DRAM access-latency

![Bar chart showing average DRAM access latency vs. DRAM page size](chart.png)

- **DRAM access latency reduces with larger pages!**
Test-Scenario 3: Prioritization of a thread is achieved at the cost of performance of other threads

Priorities for Static Priority case

T2 and T1 are prioritized over T0 so T0’s performance degrades
Future Enhancement – Support for Multicore Architectures

Enhancement: Arbitrated Coherent memory-system – Requires Implementing Coherency Protocols in Cache and Interconnect + Arbiter (with arbitration algorithms) in Memory Controller

Implement Coherent Interconnect Model

Implement Arbiter + Arbitration Algorithms

Implement Coherence Protocols!
Conclusion – Challenge – No Thread/Process Scope Guidelines in TLM?

**Fully Partitioned Mapping**

*Any Number of Hardware-Threads*

![Diagram showing Fully Partitioned Mapping](image)

*Is this good?*

*Any Number of Caches*
Conclusion – Challenge – No Thread/Process Scope Guidelines in TLM?

Semi Partitioned Mapping

Any Number of Hardware-Threads

How about this??

© Accellera Systems Initiative
Conclusion – Our Experience using TLM

- Easy to transform an architecture-specification to an executable-model
- Separation of computation from communication enables flexible simulator design and architectural-exploration
- Modeling at Transaction-Level enables fast simulation with reasonable accuracy for exploration
- But, need a guideline to define thread/process scope!
- And, if I ever get down to improving it:
  - MeSSMArch v2.0 – A Memory System Simulator for Multicore Hardware Architectures?
References

15. MCF Benchmark. URL: https://www.spec.org/cpu2006/Docs/429.mcf.html
Thank You! ☺

Questions/Thoughts/Comments?
Design-space of Widely used System Models

Graph Showing the Design-space of Widely used System Models [2]
TLM - Overview

- Separate Computation from Communication
- TLM 2.0 LT – {Timed Computation} + {Untimed Communication}
- TLM 2.0 AT – Timed {Computation + Communication}
Advantages of the TLM Methodology

• Early Software Development
  – Functional TLM platform can be constructed from system-architecture specification – aids pre-silicon software development

• Architectural Analysis
  – Timed TLM platforms comprising of parameterized components can be used for swift architectural exploration

• Functional Verification
  – TLM platforms represent an executable specification, functional o/p can be compared with RTL for verification

This Information is Borrowed from [3]
Timing-Accuracy vs. Cycle-Accuracy

Total Transaction Time = t

Clock

Cycle-Accurate
(Slower Simulation)
(Finer Result Accuracy)

Transaction Start
Thread Processes Event
Multiple Context-Switches
Transaction End

Timing-Accurate
(Faster Simulation)
(Coarser Result Accuracy)

Transaction Start
Thread Misses Event
Single Context-Switch
Transaction End

wait (t)
Hardware Multithreading Architectures

- Multiple Threads share a Unified Memory-Hierarchy
  - Thread scheduling may be coarse-grained, fine-grained or simultaneous-multithreaded (SMT)
- Replicate “software state” for each thread (PC, registers)
- Share “hardware state” (caches, branch predictors etc.)
- Reason: improve exploitation of ILP
  - Hardware may provide many execution resources
  - Single instruction stream cannot fully utilize those resources
  - Share resources between multiple threads to increase utilization
- No memory-coherence issues since hierarchy is shared!
Multicore Architectures

- Multiple “cores” share a memory hierarchy
- “Cores” contain memory hierarchies
  - May also contain multiple threads
- Reason: improve exploitation of TLP
  - Replicate hardware “cores” to enable true parallelism by providing a “private” memory-hierarchy
- Memory-coherence issues arise when private hierarchies do not present the same view of memory
  - Need coherence protocols
Tracing a Memory Transaction

**Step 1:** Hardware-Thread acquires access to Serializing Interconnect (If Arbiter cannot grant access, Hardware-Threads is Blocked)

Green Arrows Indicate Hardware-Thread SystemC Thread Context
Tracing a Memory Transaction

Step 2: Serializing Interconnect Queues Current Transaction and Blocks Hardware-Thread

Green Arrows Indicate Hardware-Thread SystemC Thread Context

© Accellera Systems Initiative
Tracing a Memory Transaction

Step 3: Arbiter Picks Transaction and Injects it downstream

Red Arrows Indicate Serializing Interconnect Buffer Drain SystemC Thread Context
Step 4: L0$ Performs Tag-Lookup (assume miss) and returns time
Cumulative Lumped-Delay = t_{TagLookup}
Tracing a Memory Transaction

Step 5: L0$ Forwards Transaction Downstream

Red Arrows Indicate Serializing Interconnect Buffer Drain SystemC Thread Context

© Accellera Systems Initiative 36
Step 6: Memory Controller translates Physical Address to Raw Address, accesses DRAM and returns time
Cumulative Lumped-Delay = $t_{TagLookup} + t_{DRAM-access}$
Step 7: Transaction is returned to the Serializing Interconnect on the backward-path. Arbiter invokes `wait()` to update time and then notifies hardware-thread of transaction completion. Cumulative Lumped-Delay Waited = $t_{TagLookup} + t_{DRAM-access}$

Red Arrows Indicate Serializing Interconnect Buffer Drain SystemC Thread Context
Step 8: Control is returned back to Hardware-Thread after timestamp update

Green Arrows Indicate Hardware-Thread SystemC Thread Context
Sequential Consistency - Theory

**Requirement R1:** Each processor issues memory requests in the order specified by its program.

Requirement R1 is not sufficient to guarantee correct execution. To see this, suppose that each memory module has several ports, and each port services one processor (or I/O channel). Let the values of “a” and “b” be stored in separate memory modules, and consider the following sequence of events:

1. Processor 1 sends the “a := 1” request to its port in memory module 1. The module is currently busy executing an operation for some other processor (or I/O channel).
2. Processor 1 sends the “fetch b” request to its port in memory module 2. The module is free, and execution is begun.
3. Processor 2 sends its “b := 1” request to memory module 2. This request will be executed after processor 1’s “fetch b” request is completed.
4. Processor 2 sends its “fetch a” request to its port in memory module 1. The module is still busy.

There are now two operations waiting to be performed by memory module 1. If processor 2’s “fetch a” operation is performed first, then both processes can enter their critical sections at the same time, and the protocol fails. This could happen if the memory module uses a round robin scheduling discipline in servicing its ports.

In this situation, an error occurs only if the two requests to memory module 1 are not executed in the same order in which they were received. This suggests the following requirement.

**Requirement R2:** Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

1. Transactions from single-processor are in-order

2. Transactions from different processors may be interleaved

Snapshots of actual text in paper borrowed from [13]
Sequential Consistency – Illustration

Assume that each Processor needs to send its transactions in program-order to the Shared Buffer.

What combinations of transaction ordering are possible in the Shared Buffer?
Sequential Consistency – Illustration

All of these transaction orderings are sequentially consistent!

1. Transactions from each processor are in program-order
2. Transactions between processors can be interleaved
Sequential Consistency – Illustration

Not Sequentially Consistent! – From the perspective of each processor, its transactions have been re-ordered!
Why is MeSSMArch Sequentially Consistently?

Load/Store Unit of Hardware-Thread has FIFO of single-entry depth only → Since Parser reads benchmark in-order, impossible for Load/Store unit to re-order them!
This satisfies requirement 1

Arbiter in Serializing Interconnect may re-order transactions between hardware-threads, but cannot re-order transactions from the same hardware-thread, because the hardware-thread always issues them in-order!
This satisfies requirement 2
## Memory Controller – Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit/Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM Page-size</td>
<td>Bytes</td>
<td>Size of a DRAM page – effectively Row-buffer size</td>
</tr>
<tr>
<td>Cache Line-Size</td>
<td>Bytes</td>
<td>Size of a cache-line</td>
</tr>
<tr>
<td># of DRAM Banks</td>
<td>&lt;NA&gt;</td>
<td># of banks in a multi-banked DRAM</td>
</tr>
<tr>
<td>Memory Data-bus Size</td>
<td>Bits</td>
<td>Size of the data-bus connecting memory controller and DRAM</td>
</tr>
<tr>
<td>Memory Timing Parameters</td>
<td>Cycles</td>
<td>tRCD, tCL, tRP</td>
</tr>
<tr>
<td>DRAM Memory Type</td>
<td>Sync/Async</td>
<td>Affects derived tRAS memory-timing parameter</td>
</tr>
<tr>
<td>Physical Address to Raw Address Mapping</td>
<td>Byte Interleaved/Bank Seq/Row Seq</td>
<td>Affects decoding of Physical Address to Raw Address</td>
</tr>
<tr>
<td>Clock Period</td>
<td>Nanosecs</td>
<td>Time-period of a clock-cycle</td>
</tr>
</tbody>
</table>

Green entries indicate micro-architectural features
Serializing Interconnect – Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit/Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Outstanding Transactions</td>
<td>Transactions</td>
<td>Denotes the size of the Pending Transaction Buffer</td>
</tr>
<tr>
<td>Arbitration Algorithm</td>
<td>First-pending/FCFS/Static Priority/Prioritize Hits</td>
<td>Algorithm determining the transaction picked from the Pending Transaction Buffer for injection downstream</td>
</tr>
</tbody>
</table>
## Hardware-Thread – Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit/Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark File</td>
<td>Path to file</td>
<td>Benchmark file to read, parse and execute</td>
</tr>
<tr>
<td>Master Priority</td>
<td>Integer (lower number implies higher priority)</td>
<td>Static-priority for the thread (used only for the static-prioritization arbitration algorithm)</td>
</tr>
</tbody>
</table>
# Generic Cache – Performance Counters

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td># of caches-hits and cache-misses (further classified into reads/writes)</td>
<td># of transactions</td>
</tr>
<tr>
<td># of overhead-writes generated by write-policy (write-through/write-back)</td>
<td># of transactions</td>
</tr>
<tr>
<td>Miss-classification into capacity/compulsory/conflict</td>
<td># of transactions and %ages</td>
</tr>
<tr>
<td>Way-prediction accuracy and inaccuracy</td>
<td># of transactions and %ages</td>
</tr>
<tr>
<td>Cache-bandwidth (effective and wasted)</td>
<td>Bytes per time unit</td>
</tr>
</tbody>
</table>
Memory Controller – Performance Counters

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row-buffer hit-rate and miss-rate (per-bank and average)</td>
<td>%ages and # of transactions</td>
</tr>
<tr>
<td>Average memory-transaction latency</td>
<td>time units</td>
</tr>
<tr>
<td>Average memory-bandwidth</td>
<td>Bytes per time unit</td>
</tr>
<tr>
<td># of row-activates, col-activates, row-precharges per bank</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>
### Hardware-Thread – Performance Counters

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td># of memory-transactions issued</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>Total Thread Execution Time</td>
<td>time-units</td>
</tr>
<tr>
<td>Bus-contention time, Bus-queuing time, Effective-execution time</td>
<td>time-units and %ages</td>
</tr>
<tr>
<td>Average memory-transaction execution time</td>
<td>time-units</td>
</tr>
</tbody>
</table>
Benchmarks used for Use-Case Verification

• Collect dynamic execution-trace for each SPEC CPU benchmark
• Pick first 100-million instructions
• Simulate all memory-transactions present in the first 100-million instructions

<table>
<thead>
<tr>
<th>SPEC CPU Benchmark</th>
<th># of Memory Transactions</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>art-100M</td>
<td>19888117</td>
<td>Adaptive Resonance Theory – Image Recognition/Neural Networks [14]</td>
</tr>
<tr>
<td>mcf-100M</td>
<td>32362081</td>
<td>Single-Depot Vehicle Scheduling [15]</td>
</tr>
<tr>
<td>go-100M</td>
<td>35497321</td>
<td>Artificial Intelligence: Game of Go [16]</td>
</tr>
</tbody>
</table>
Current Limitations – Future Enhancement #1

Limitation: Only Same Cache Line-Size Allowed!

Enhancement: Variable Cache Line-Size -- Requires implementing logic to split/merge transactions that differ in line-sizes
Current Limitations – Future Enhancement #2

Limitation: Blocking Caches!

Enhancement: Non-Blocking Caches - Requires implementing logic to pipeline cache-transactions and allow reordering of cache-transactions.

Cannot issue out-of-order transactions to hide miss-latency by issuing hit-after-miss because Cache is Blocked!
Current Limitations – Future Enhancement #3

Limitation: Cannot Explore Typical “Multicore” Architectures!

Enhancement: Coherent memory-system - Requires implementing coherency protocols in cache and interconnect

Model does not exist!!

No Coherence Protocols!

Copyright © Accellera Systems Initiative 2015
Current Limitations – Future Enhancement #4

Enhancement: Arbitrated Coherency memory-system - Requires implementing an Arbiter and Arbitration Algorithms in Memory Controller

Limitation: Cannot Model Core Unfairness by re-ordering transactions from multiple cores! [17]