2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

MeSSMArch – A Memory System Simulator for Hardware Multithreading Architectures

A Study Exploring the Application of the Transaction Level Modeling Methodology in the development of a Memory System Simulator at the System-Level

Sushil Menon, NVIDIA, Bangalore, Karnataka, India (sushil.menon.1988@gmail.com)

***Abstract*— The trend of scheduling multiple software-applications (or software-threads) with ever-increasing memory-footprints, for concurrent execution on multiple, increasingly accelerated hardware processing-cores, has necessitated the simultaneous compliance of two orthogonal requirements of computer memory-systems: higher speed and higher storage capacity. This has resulted in the adoption of a memory-hierarchy, which takes advantage of locality to optimize system performance and cost, where smaller, faster memory is placed in close proximity to the processing-cores, with larger, slower memory being further away. The organization of such a hierarchy is characterized by multiple parameters, thereby obligating system-architects to conduct a thorough exploration of the design-space as a pre-requisite to defining the system-architecture. The process of exploration requires the employment of simulators (typically trace-based) which enable experimentation and analysis by providing insights into estimated system-performance. The scarcity of free, easily extensible and fast, memory-hierarchy simulators has inspired us to develop MeSSMArch – a Memory-System Performance Simulator for Hardware Multithreading Architectures.**

In this paper, we present our experience of applying the Transaction Level Modeling (TLM) methodology in the process of developing MeSSMArch at the system-level. We first motivate the necessity for developing a generic memory-system performance simulator at the system-level, a process resulting in the logical inference of design-requirements. We then present a brief description of the TLM methodology, attempting to capture the salient features of the design-philosophy. Next, we showcase the application of the TLM methodology, in the process of modeling each major unit of the memory-system (Abstract Hardware-Thread, Serializing Interconnect, Generic Cache and Memory Controller), and integrating them in the context of a hardware multithreaded system. We then showcase the process of validating the simulator, by verifying selected use-cases that portray the fundamental tenets of memory-hierarchies. We finally conclude by enunciating our learning from this experience.

Keywords—Memory Systems, Memory Hierarchy, Cache, Memory Controller, Interconnect, Hardware Multithreading, Transaction Level Modeling (TLM), SystemC, Performance Estimation

I. Introduction

Moore's law of exponential growth in the capacity of integrated-circuits has greatly promoted the advancement of computing infrastructure. While processors are typically designed to channelize these resources towards maximizing speed, ever-increasing memory-footprints have coerced designers to channelize these resources towards increasing the storage-capacity of memories. As memories expand in capacity, it becomes intractable to accommodate them with processors, both on the same die, thereby constraining designers to place them off-chip, resulting in increased memory latency, thus reducing system performance. Designers, are thus, compelled to construct a hierarchy of memories, in order to simultaneously increase storage-capacity whilst preventing degradation of system performance.

The foundation of a memory-hierarchy is based on the fundamental principles of locality. While temporal locality suggests that recently accessed data may-be accessed again in the near future, thereby favoring high-speed memories, spatial locality suggests that data in close-proximity to recently accessed data may-be accessed again in the near future, thereby favoring higher-capacity memories. These orthogonal requirements are simultaneously satisfied by organizing a hierarchy where smaller, high-speed memories are placed closer towards the processors (typically on the same die), and larger, slower memories are placed further away (typically off-chip), resulting in a vast design-space, of which an extensive exploration must be performed prior to defining the system-architecture.

The existence and interaction of multiple parameters in the design-space impels system-architects to conduct a cost-performance analysis, a process requiring the employment of simulators (typically trace-based) that enable architectural-exploration. The design of such simulators involves a process of maintaining an appropriate equilibrium between result-accuracy and simulation-speed, thus emphasizing the importance of modeling the system at an appropriate level of abstraction, as deemed necessary, for a given use-case. The combinatorically explosive nature of the design-space, coupled with stringent time-to-market requirements, requires system-architects to utilize efficient simulators, which provide results that signify an estimation of system performance, in a minimal turnaround time, implying trading-off result-accuracy for simulation-speed. The scarcity of free, easily extensible, fast memory-hierarchy simulators that provide course-grained data on memory-system performance has served as our inspiration to develop MeSSMArch.

II. Simulator Design Requirements

Our intended use-case is to develop a simulator that aids in the architectural-exploration of memory-system performance at the system-level. Based on the targeted use-case, we reason the following requirements:

1. The simulator must be *functionally generic* – implying the avoidance of modeling functionality for implementation-specific features, and instead capturing its effects on system performance. This enables exploration at the system-level, offering system-architects a wider array of architectural options, as opposed to exploration at the micro-architecture level, a process more constraining due to increased design-complexity. For example, instead of modeling the functionality of comparators to perform a parallel tag-lookup along all the ways of a set-associative cache, we model a parallel tag-lookup by performing a sequential tag-lookup, and then dividing the time taken to perform the lookup by the number of comparators, thus capturing the effects of parallel tag-lookup on system performance.
2. The simulator must provide an *estimate into system performance* – implying the sufficiency of coarse-grained result-accuracy. This permits the modeling of transaction-accurate components, which ignore temporal effects *within* a transaction, and instead capture temporal effects *between* transactions, resulting in a fast simulator with an abstract notion of system performance.
3. The simulator must be *extensible* – implying the usage of parameterized components that communicate over a unified interface. This permits the easy exploration of multifaceted memory-systems in a plug-and-play environment.

Section IV discusses the development of each component of the simulator in the context of these design-requirements.

III. Transaction Level Modeling (TLM) Methodology – An Overview

A. *Brief Description of the TLM Methodology*

Transaction Level Modeling (TLM) refers to the approach of modeling systems such that the details of module computation are separated from the details of inter-module communication. Module computations are modeled as *processes* which exchange data via *interface method call*s. An *interface* declares communication access methods which are implemented by *channels*. Modules are bound using *ports* that export an interface. A *transaction* is the action of sending data from a *master/initiator* process to a *slave/target* process. Figure 1 showcases a graphical representation of the structural semantics of the TLM methodology.

The dissociation of communication from computation implies that each may be modeled arbitrarily as either timed or untimed, resulting in a spectrum of acceptable transaction-accurate models. Figure 2 showcases the design-space for widely used system-models [2]. Typical timed transaction-level models are of two types:

1. Approximately-timed computations exchanging data over an untimed communication channel, referred to as a *component-assembly model* in figure 2, which resembles the Loosely-timed TLM 2.0 coding style [4].

2. Approximately-timed computations exchanging data over an approximately-timed communication channel, referred to as a *bus-arbitration model* in figure 2, which resembles the Approximately-timed TLM 2.0 coding style [4].

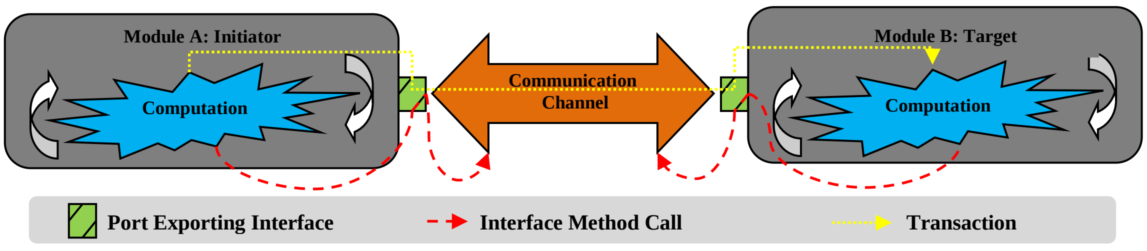


Figure 1: Graphical Representation of the Structural Semantics of the TLM Methodology

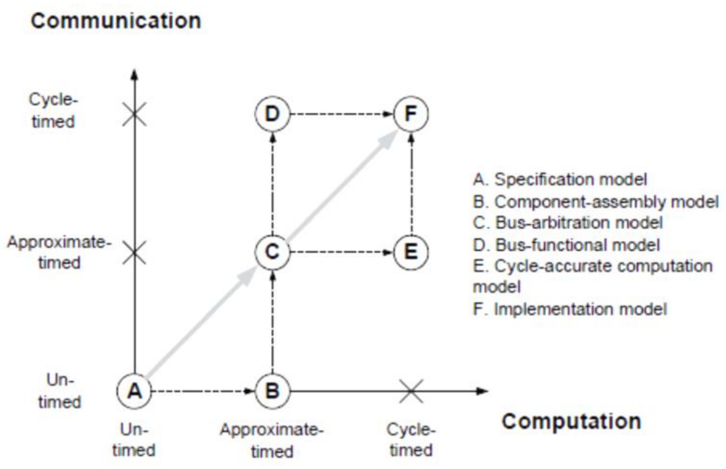


Figure 2: Design-space of widely used System Models [2]

Furthermore, the separation of module computation from inter-module communication results in effortless architectural-extensibility. Modules can be designed to be easily multiply instantiated and connected via a simple binding of the ports to communication channels, regardless of the computation performed and the communication protocol adopted, provided that the ports export a common interface.

B. Advantages of the TLM Methodology

[3] lists the following advantages of the TLM methodology:

1. *Early software development* – given a system-architecture specification, a functionally-accurate TLM platform that permits the execution of embedded software can be constructed, thereby aiding pre-silicon software development. Additionally, the functionally-accurate TLM models can be annotated with transaction-level timing information, thereby enabling the early optimization of embedded software.

2. *Architectural Analysis* – timed TLM platforms which are comprised of parameterized components may be used for architectural-exploration of a given system-architecture specification. Rather than implementing them, the effects of micro-architectural features can be captured by specifying timing information, thereby reducing the turnaround time for such an exploration.

3. *Functional Verification* – TLM platforms, and the models comprising them, represent an executable functional specification, whose output maybe used for comparison with RTL, during the process of functional verification.

C. *Salient Features of the TLM Methodology*

The following are the salient features of the TLM methodology, and are intended to be used as guidelines while developing models which compose TLM platforms:

1. *Separate Module Computation from Inter-module Communication* – by modeling communication related details inside the channel. This permits the modeling of computation and communication at different levels of abstraction, thus providing flexibility, and also eases architectural-extensibility.
2. *Avoid modeling functionality of micro-architectural features and instead capture only their effects* – by modeling their effects through the specification of timing information. This results in simpler models, and thus, faster simulation.
3. *Simulate data-exchange at the Transaction-Level* – by raising the level of timing-abstraction from cycle-accuracy to timing-accuracy, by accounting for multiple clock-cycles as a lumped-delay, ensuring to retain any synchronization present between clock-cycles that occur within the boundary of the specified transaction. This results in faster simulation.

IV. Using the TLM Methodology to Model a Generic Memory-System at the Transaction-Level

In this section, we showcase the process of modeling each of the components of a memory-system, at the transaction-level. We describe an appropriate model for each component, which we derive based on the guidelines of the TLM methodology. We then showcase the process of constructing a memory-hierarchy for a hardware multithreaded system. Note that the sufficiency of coarse-grained result-accuracy implies that we model all components to be loosely-timed by implementing only the TLM 2.0 b_transport( ) interface [4].

A. *Modeling a Generic Cache at the Transaction-Level*

A cache is a small, high-speed memory located in close proximity to the processor, which is designed to store frequently accessed data, in an attempt to reduce memory-access latency. In the event that the requested data is not located in the cache, the processor is stalled for additional penalty clock-cycles until the data is fetched from a lower-level of memory and brought into the cache [6].

Figure 3 showcases the block diagram of the derived generic cache model. It mainly consists of:

- Tag RAM structures which store the tag-address, dirty-bit, valid-bit and age-counter of a way in the set
- A Cache controller which implements state-machines that capture the generic functionality of a cache
- A Bus Interface Unit which implements the interface for inter-module communication

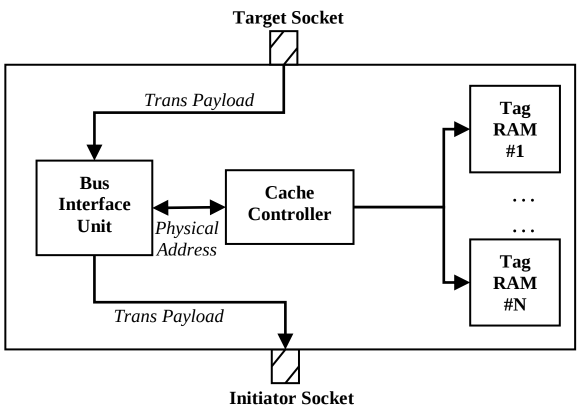


Figure 3: Block Diagram of the derived Generic Cache Model

The model is parameterized via support for arguments in the module constructor. Configurable module parameters include:

- *Cache-size* – specified in kilobytes

- *Cache Line-size* – specified in bytes
- *Cache Associativity* –the number of ways in a set
- *Number of Comparators* – which are utilized during a parallel tag-lookup (a micro-architectural feature)
- *Write Allocate* – controls the allocation of a way-entry in a set, in the event of a write-miss
- *Write Through* – controls the generation of a write-transaction to the next-level of memory, in the event of a write-hit (thus maintaining a consistent clean-state along the entire hierarchy)
- *Way Prediction* – controls the early-setting of the multiplexor to select the LRU-block during a tag-lookup, in order to reduce lookup time (a micro-architectural feature)
- *Clock Period* – the time-period of a clock-cycle (internally, time for all operations are tracked as clock-cycles)

In accordance with the guidelines of the TLM methodology:

1. Micro-architectural features are modeled as follows:
    a. Only the effects of the number of comparators during the process of a tag-lookup are captured, via the following mathematical equation

$$\# \, of clock \; ticks = \left\lceil \frac{Associativity}{\# \; of \; Comparators} \right\rceil$$

    b. When enabled, only the effects of way prediction during the process of a tag-lookup are captured, via the conditional execution of the following mathematical equation

$$if \; \big((WayPrediction \; is \; Enabled) \;\&\&(Hit \; Way \; \# == LRU \; Way \; \#)\big) \quad then \quad \# \; of \; clock \; ticks = 1$$

2. Data exchange is modeled at the transaction-level by:
    a. Tracking the cumulative number of clock-cycles during execution of the state-machines at the current level without context-switching processes
    b. Tracking the total time at the current level by adding a lumped time-delay received from the lower level to the product of the number of clock-cycles and the clock-period at the current level

Model performance counters and execution statistics reported include:

- Number of cache-hits and cache-misses (further classified into reads/writes)
- Number of overhead-writes generated at current-level due to write policy (write-though/write-back)
- If enabled, classification of cache-misses into compulsory/capacity/conflict misses
- Cache hit-rate and miss-rate
- Cache hit-time and miss-time (average and total)
- If way-prediction enabled,
    - Number of correct and incorrect predictions
    - Non-mispredict and mispredict rate
- Total memory-bandwidth, memory-bandwidth wasted due to overhead-writes generated at current level due to write-policy (write-through/write-back), effective memory-bandwidth

B. Modeling a Memory Controller at the Transaction-Level

DRAMs are organized into banks, rows and columns, thus introducing a format of addresses called *raw-addresses.* A memory controller is a device that manages data-flow between the upper levels of memory and

DRAM by translating physical-addresses to raw-addresses and generating a sequence of commands that perform the data-transfer to/from the DRAM.

Figure 4 showcases the block diagram of the derived memory controller model. It mainly consists of:

- A Memory Mapping unit which translates an incoming physical-address to raw-address format
- A Command Generation unit which generates a sequence of commands (Row Activate, Column Activate, Row Precharge) which are routed to each bank's command-queue
- Command-Queue structures for each bank, which contain a list of commands to be performed on the respective DRAM bank
- Row-Buffer structures for each bank, which contain the row-address of the currently opened row in the row-buffer, for the respective bank
- A Bus Interface Unit which implements the interface for inter-module communication

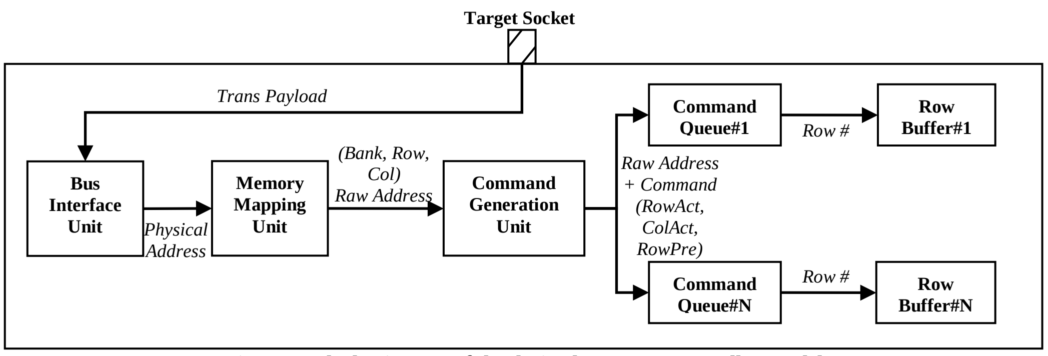


Figure 4: Block Diagram of the derived Memory Controller Model

The model is parameterized via support for arguments in the module constructor. Configurable module parameters include:

- *DRAM Page-size* – specified in bytes
- *Cache Line-size* – specified in bytes
- *Number of DRAM banks* – to capture the effects of a multi-banked DRAM (a micro-architectural feature)
- *Memory Data-bus Size* – size of the data-bus connecting the memory controller to the DRAMs, specified in bits (a micro-architectural feature)
- *Technology dependent Memory Timing Parameters* – including tRCD, tCL, tRP, specified in clock-cycles
- *DRAM Memory Type* – whether the DRAM is synchronous or asynchronous, affecting the derived tRAS memory-timing parameter
- *Physical-Address to Raw-Address mapping* – options include byte-interleaved, bank-sequential, and row-sequential. These affecting the decoding of a physical-address into the raw-address tuple (Bank, Row, Col)
- *Clock Period* - the time-period of a clock-cycle (internally, time for all operations are tracked as clock-cycles)

In accordance with the guidelines of the TLM methodology:

1. Micro-architectural features as modeled as follows:

a. Only the effects of the number of DRAM banks during the process of a memory-transaction to/from DRAM are captured by modeling only data-structures that are essential to capture them. This is done by modeling the command-queue associated with each bank, and computing the delays contributed by the processing of commands in each queue. We do not model the entire memory-bank.

b. Only the effects of the memory data-bus size during the process of a data-transfer of a cache-line to/from DRAM are captured by splitting the transfer of a cache-line into multiple memory data-bus transactions, the size of each being represented via the following mathematical equation

$$\begin{aligned} &\#\ of\ bits\ transfered\ by\ current\ memory\ data\ bus\ transaction \\ &= \min\left(Memory\ Data\ Bus\ Size, \#of\ Bits\ Until\ End\ of\ Current\ DRAM\ Page, \#\ of\ Bits\ Until\ End\ of\ Cache\ Line\right) \end{aligned}$$

2. Data exchange is modeled at the transaction-level by tracking the cumulative number of clock-cycles during the processing of commands in each of the command-queues without context-switching processes, and returning the computed lumped time-delay on the transaction return-path.

Model performance counters and execution statistics reported include:

- Row-buffer hit-rate and miss-rate (per-bank and average)
- Average memory-transaction latency
- Average memory-bandwidth
- Per-bank and average DRAM utilization (%age of DRAM cycles spent on data-transfer vs. DRAM cycles spent on initiating transfer)
- Bank-specific statistics:
    - Number of row activates, column activates, and row precharges
    - Row-buffer hit-count and miss-count
    - Bank active cycles (cycles spent on data-transfer i.e. column activates) and inactive cycles (cycles spent on initiating data-transfer i.e. row activates and row precharges)
    - Total row precharge time

C. *Modeling a Serializing Interconnect at the Transaction-Level*

In a hardware multithreading architecture, multiple execution cores are organized as hardware-threads that share a unified memory-hierarchy [9], [11], implying the need for a serializing interconnect that manages arbitration. The serializing interconnect arbitrates access to the shared memory-hierarchy by buffering incoming transactions from all hardware-threads and injecting them downstream for processing.

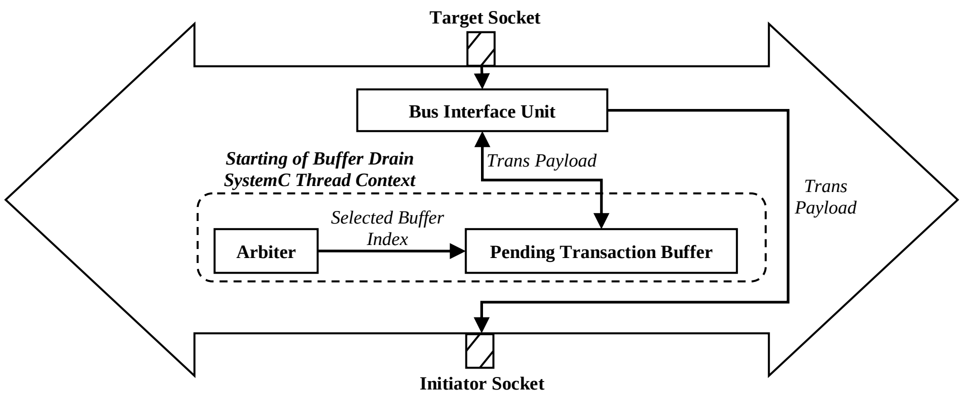


Figure 5: Block Diagram of the derived Serializing Interconnect Model

Figure 5 showcases the block diagram of the derived serializing interconnect model. It mainly consists of:

- A Pending Transaction Buffer that stores the payload of an incoming transaction
- An Arbiter that implements algorithms to select a transaction from the pending transaction buffer for injection downstream
- A Bus Interface Unit which implements the interface for inter-module communication

In order to arbitrate accesses by multiple hardware-threads to the shared memory-hierarchy, the following protocol is adopted:

1. The hardware-thread must first acquire access to the serializing interconnect to ensure the reservation of an entry in the pending transaction buffer for the corresponding transaction. If the serializing interconnect fails to reserve an entry in the pending transaction buffer, it blocks the hardware-thread.
2. Once access is acquired, the serializing interconnect must queue the transaction in the pending transaction buffer and block the hardware-thread until the transaction has completed.
3. The serializing interconnect must notify the hardware-thread upon the completion of the transaction, following which the hardware-thread must relinquish access by returning the reserved pending transaction buffer entry.

The serializing interconnect model implements a SystemC process to drain the pending transaction buffer. When the SystemC kernel schedules this thread for execution, the arbiter selects a transaction from the pending transaction buffer and injects it downstream for processing. The criterion for selection of a transaction is dictated by the configured arbitration algorithm.

The model is parameterized via support for arguments in the module constructor. Configurable module parameters include:

- *Number of outstanding transactions* – which denotes the size of the pending transaction buffer, specified in number of transactions
- *Arbitration algorithm* – choice between first-pending (the first pending transaction found in the buffer), first-come-first-served (the earliest transaction to be queued in the buffer), static priority (to prioritize transactions from specific threads), prioritize hits (test-inject each transaction in the buffer, and pick the one that hits at the highest-level of the memory system, implying the fastest transaction)

In accordance with the guidelines of the TLM methodology, data exchange is modeled at the transaction-level by:

1. Associating the completion of a transaction injected by a hardware-thread with a single event, instead of polling its completion status every clock-cycle
2. Modeling the total transaction delay returned from the lower level with a single wait( ), and hence only a single context-switch

D. *Modeling a Hardware-Thread at the Transaction-Level*

A typical hardware-thread encapsulates logic such as instruction-fetch units, instruction-decode units, instruction-execution units and ALUs, registers, branch-predictors, and load-store units, usually organized in a pipeline, which collectively enable the execution of an instruction-stream [10], [11]. In this context of this work, we model the hardware-thread to comprise only of a load-store unit, ignoring all other aspects that are not involved with the memory-system.

Figure 6 showcases the block diagram of the derived hardware-thread model. It mainly consists of:

- A Load/Store unit with a single-entry depth FIFO that given a load/store transaction, generates the corresponding transaction payload

- A Trace Parser that implements infrastructure to read and parse an input benchmark-file
- A Bus Interface Unit which implements the interface for inter-module communication

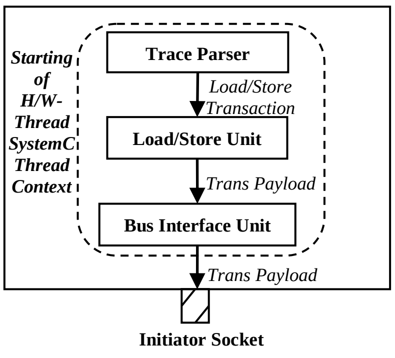


Figure 6: Block Diagram of the derived Hardware-Thread Model

The hardware-thread model implements a SystemC process to inject memory-transactions downstream. When the SystemC kernel schedules this thread for execution, the trace parser reads and parses an instruction from the benchmark-file, and forwards it over to the load/store unit which generates a transaction payload for injection downstream. This process executes until the benchmark-file is empty.

The model is parameterized via support for arguments in the module constructor. Configurable module parameters include:

- *Benchmark File* – path to the benchmark-file to read, parse and execute
- *Master Priority* – representing a static-priority for the thread (used only for the *static priority* serializing interconnect arbitration scheme)

In accordance with the guidelines of the TLM methodology, data exchange is modeled at the transaction-level by the avoidance of invoking wait( ) to model transaction-latency (this is modeled by the downstream serializing interconnect).

Model performance counters and execution statistics reported include:

- Number of instructions issued
- Total thread execution time
- Bus-contention time, effective execution time, bus-queuing time (%age and time-units)
- Average memory-operation execution time

E. Constructing a Memory-Hierarchy for a Hardware Multithreaded system

Constructing the memory-hierarchy involves instantiating each component and binding their ports. Figure 7 showcases the block diagram of a memory-hierarchy for a hardware multithreaded system. As shown in the figure, it is comprised of an arbitrary number of hardware-threads bound to the serializing interconnect, which further sinks into a hierarchy of an arbitrary number of caches, and finally terminates with a connection to the memory controller. Each instance of the hardware-thread spawns a SystemC thread-context which injects transactions downstream to the serializing interconnect, which are queued in the pending transaction buffer in the same thread-context. Once queued, the serializing interconnect suspends the hardware-thread's SystemC thread-context until the transaction has completed. When the SystemC kernel schedules the pending transaction buffer drain SystemC thread-context, an arbitrary transaction is injected downstream through the entire hierarchy, in the same thread-context. On completion of the transaction, the pending transaction buffer SystemC thread-context schedules an event that notifies the hardware-thread's SystemC thread-context associated with that transaction, of its completion, and then suspends itself. Upon receiving the event-notification, the hardware-thread's SystemC thread-context continues the cycle, injecting the next transaction until all have been processed.

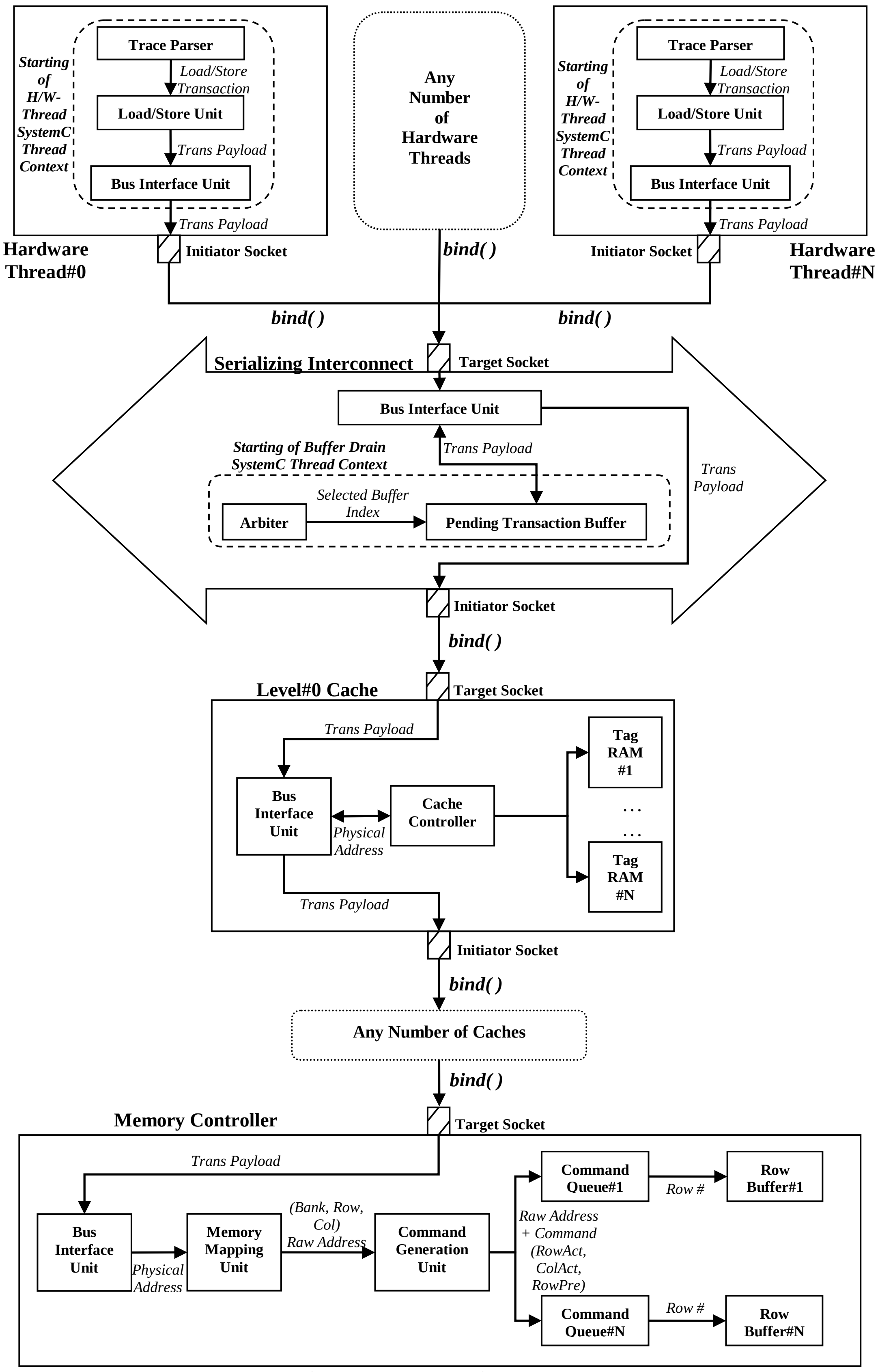


Figure 7: Block Diagram of a Memory-Hierarchy for a Hardware Multithreading System

Note that the memory-hierarchy constructed in figure 7, with the derived component-models, results in a sequentially consistent memory-system, implying the correct functional execution of programs implementing mutual-exclusion primitives (although this does not guarantee the avoidance of or recovery from a deadlock). From [12], [13], a multiprocessor memory-system is said to be sequentially consistent if:

- The result of an execution of memory-transactions issued from a single processor is the same as if the transactions were issued in program-order
- The execution of memory-transactions issued by multiple processors maybe arbitrarily interleaved

An in-depth observation of the derived component-models and the memory-hierarchy constructed in figure 7 indicates the following proof of confirmation:

- The load/store unit of the hardware-thread model is designed to have a single-depth entry FIFO. Since the Trace Parser reads the input benchmark-file in order, it issues transactions to the load/store unit in-order, which due to it's single-depth FIFO, cannot re-order transactions (as it cannot hold more than one transaction), thus making it impossible for the hardware-thread to issue memory-transactions out-of-order, implying that a stream of memory-transactions processed by a hardware-thread will always be in program-order
- The arbitration algorithm governing the arbiter of the serializing interconnect may interleave memory-transactions issued by multiple hardware-threads

V. Validation of the Generic Memory-System Simulator

In this section, we showcase the process of validating the simulator by verifying use-cases that portray the fundamental tenets of memory-hierarchies. For each use-case, we first provide a brief description of the test-scenario and the system-architecture on which the scenario is executed. We then present a plot of certain execution-statistics obtained from the simulator which illustrates the successful passing of the test.

For all test-scenarios, we use one or more of the SPEC CPU benchmarks specified in table 1. The benchmarks are composed of all memory-instructions (loads and stores) present in the first 100-million instructions of a dynamic execution-trace collected for each benchmark.

Table I. SPEC CPU Benchmarks used in test-scenarios

Benchmark Name	**# of Memory Transactions**	**Brief Description of Benchmark**
art-100M	19888117	Adaptive Resonance Theory – Image Recognition/Neural Networks [14]
mcf-100M	32362081	Single-Depot Vehicle Scheduling [15]
go-100M	35497321	Artificial Intelligence: Game of Go [16]

A. *Test-Scenario 1: Larger cache line-sizes reduce miss-rate but increase miss-penalty*

Increasing the size of a cache-line reduces the miss-rate because of spatial locality – a larger cache-line implies that a larger chunk of data in close-proximity to the address of the currently accessed cache-line is fetched on a miss, thereby increasing the probability of a cache-hit, if nearby addresses are accessed. However, a larger cache-line also implies that more data is fetched during a cache-miss, hence increasing the penalty of a miss.

In order to verify this test-scenario, we construct a system with configuration parameters as shown in figure 8. We sweep through cache line-sizes in the range [4-bytes, 1024-bytes] in powers of 2, plotting the miss-rate and miss-time for each data-point. Note that since we only require a single hardware-thread, the configuration parameters of the serializing interconnect are irrelevant.

Figures 9 and 10 showcase a plot of cache miss-rate and cache miss-penalty against cache line-size. As indicated by the plots, an increase in the cache line-size results in a reduction in cache miss-rate, but also increases the average miss-penalty, thus indicating the successful passing of test.

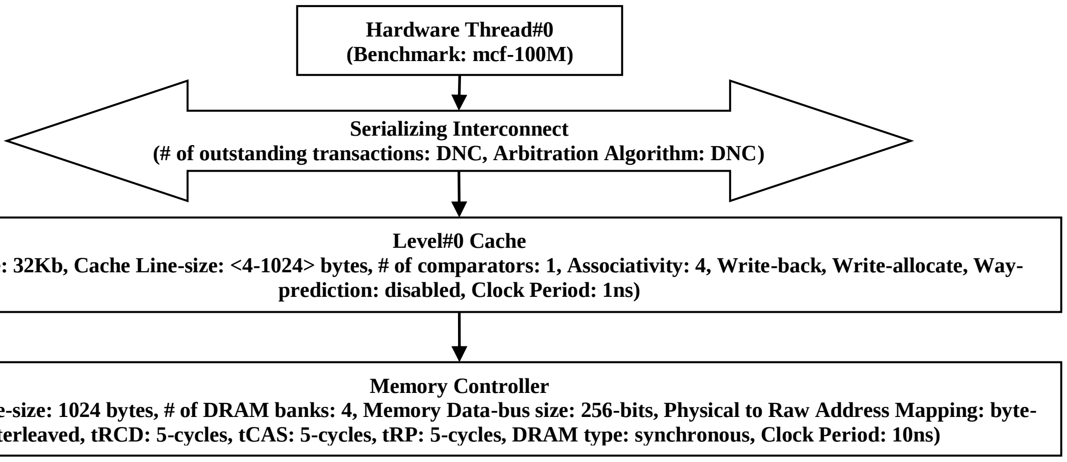

Hardware Thread#0
(Benchmark: mcf-100M)
Serializing Interconnect
(# of outstanding transactions: DNC, Arbitration Algorithm: DNC)
Level#0 Cache
(Cache-size: 32Kb, Cache Line-size: <4-1024> bytes, # of comparators: 1, Associativity: 4, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Memory Controller
(DRAM Page-size: 1024 bytes, # of DRAM banks: 4, Memory Data-bus size: 256-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 8: Block Diagram of a System-Architecture for Test-Scenario 1

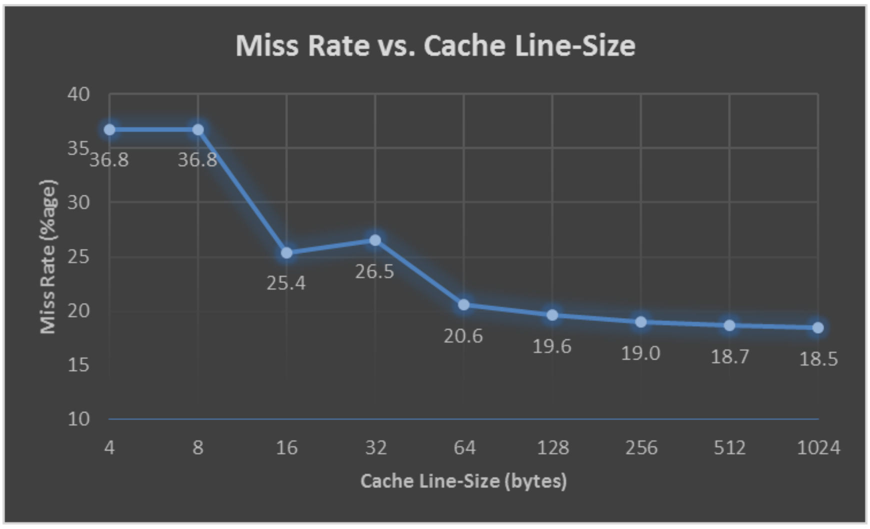


Figure 9: Plot of Cache Miss-Rate vs. Cache Line-Size for Test-Scenario 1

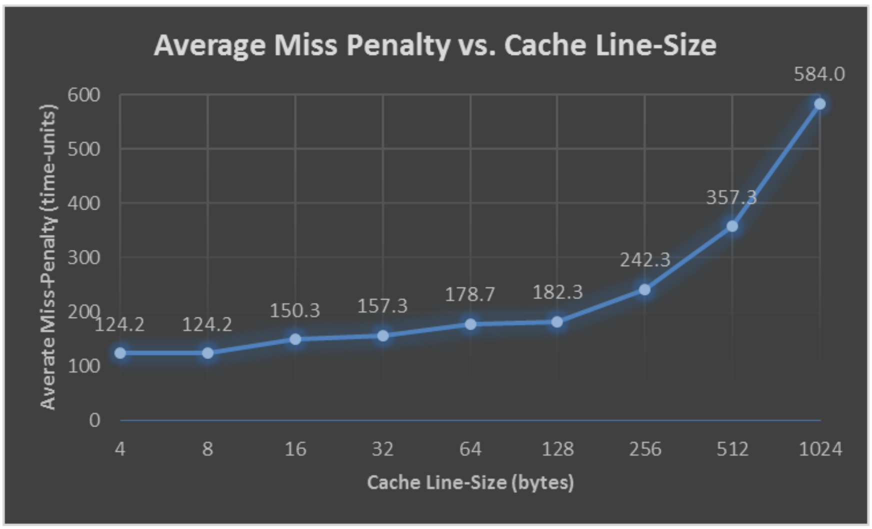
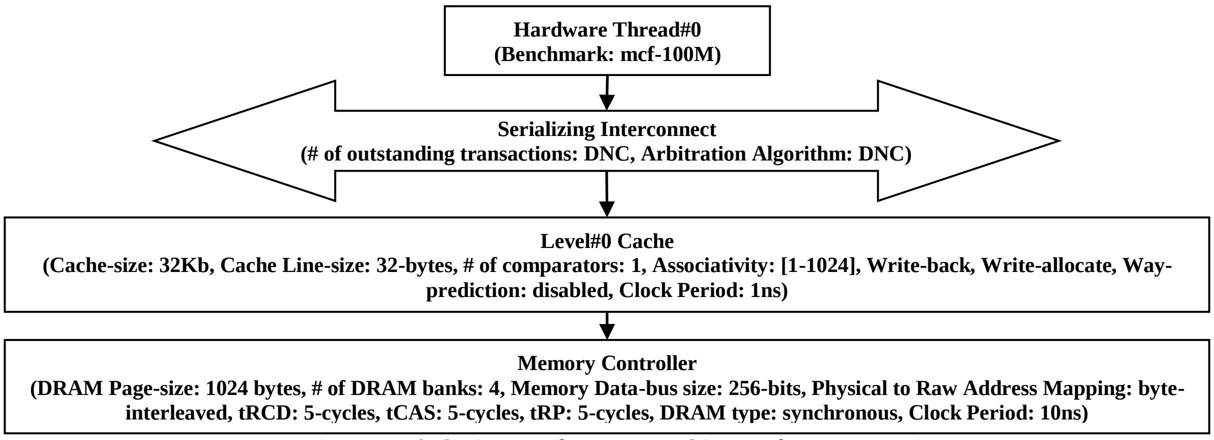


Figure 10: Plot of Cache Miss-Penalty vs. Cache Line-Size for Test-Scenario 1

B. Test-Scenario 2: Higher associativity reduces cache miss-rate but increases cache hit-time

For a fixed cache-size, a bounded increase in associativity reduces the miss-rate because it increases the spatial distribution of addresses that may be mapped to the corresponding set. However, it also increases hit-time because more ways need to be searched during a tag-lookup.

In order to verify this test-scenario, we construct a system with configuration parameters as shown in figure 11. We sweep through cache-associativity in the range [1-way, 1024-way] in powers of 2, plotting the miss-rate and hit-time for each data-point. Note that since we only require a single hardware-thread, the configuration parameters of the serializing interconnect are irrelevant.



Hardware Thread#0
(Benchmark: mcf-100M)

Serializing Interconnect
(# of outstanding transactions: DNC, Arbitration Algorithm: DNC)

Level#0 Cache
(Cache-size: 32Kb, Cache Line-size: 32-bytes, # of comparators: 1, Associativity: [1-1024], Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)

Memory Controller
(DRAM Page-size: 1024 bytes, # of DRAM banks: 4, Memory Data-bus size: 256-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 11: Block Diagram of a System-Architecture for Test-Scenario 2

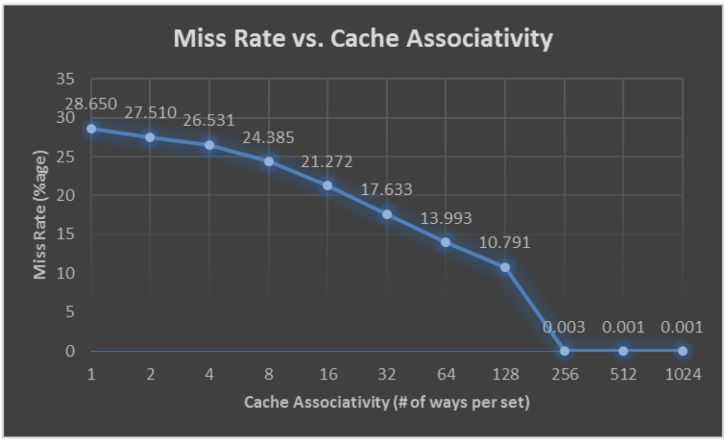


Figure 12: Plot of Cache Miss-Rate vs. Associativity for Test-Scenario 2

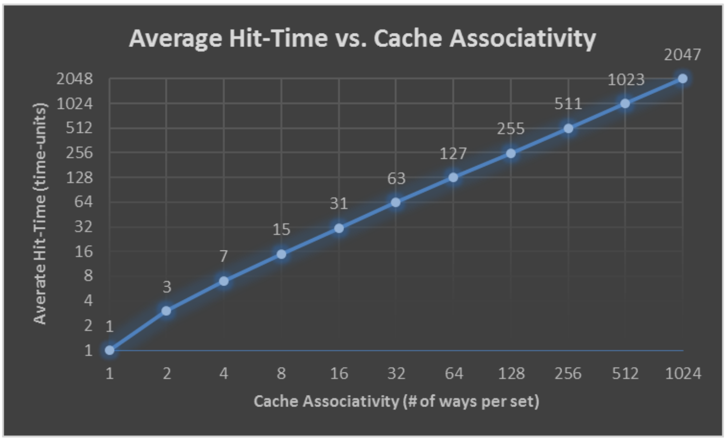


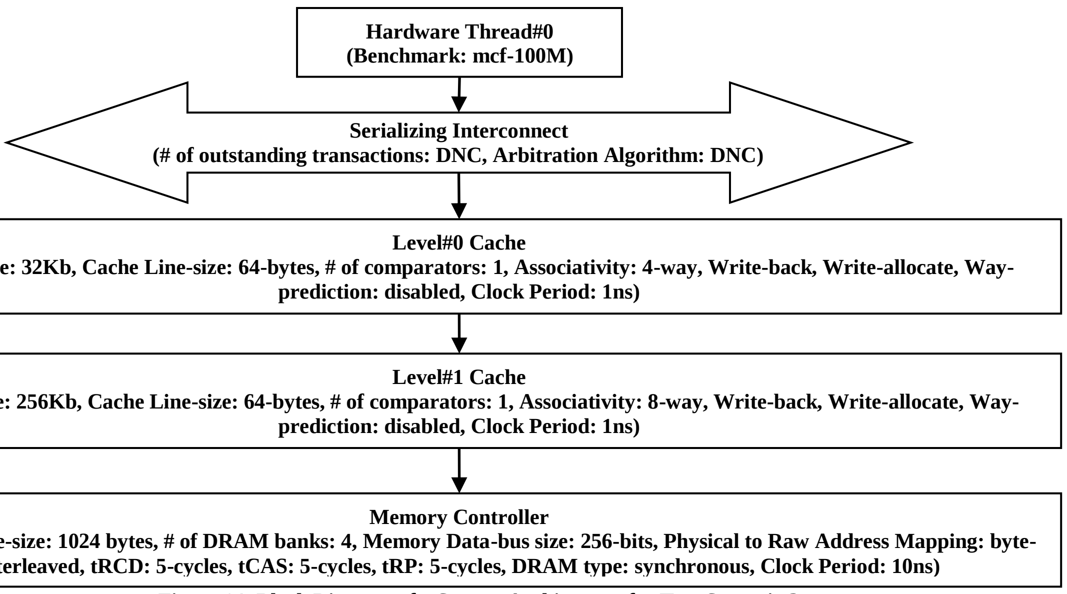
Figure 13: Plot of Cache Hit-Time vs. Associativity for Test-Scenario 2

Figures 12 and 13 showcase a plot of cache miss-rate and cache hit-time against associativity. As indicated by the plots, an increase in associativity reduces the miss-rate and increases hit-time, indicating the successful passing of the test.

C. Test-Scenario 3: Multilevel caches reduce miss-penalty

A hierarchy of caches can be constructed to simultaneously optimize hit-time and miss-penalty. Placing a smaller, less-associative cache closer to the processor reduces hit-time. Placing a larger, more-associative cache after the smaller cache, increases the probability of encapsulating misses from the previous level, in the current level, hence reducing the effective miss-penalty of the smaller cache at the previous level.

In order to verify this test-scenario, we construct a system with configuration parameters as shown in figure 14. We plot the miss-penalty for the Level#0 cache in the presence and absence of the Level#1 cache. Note that since we only require a single hardware-thread, the configuration parameters of the serializing interconnect are irrelevant.



Hardware Thread#0
(Benchmark: mcf-100M)
Serializing Interconnect
(# of outstanding transactions: DNC, Arbitration Algorithm: DNC)
Level#0 Cache
(Cache-size: 32Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 4-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Level#1 Cache
(Cache-size: 256Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 8-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Memory Controller
(DRAM Page-size: 1024 bytes, # of DRAM banks: 4, Memory Data-bus size: 256-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 14: Block Diagram of a System-Architecture for Test-Scenario 3

Figure 15 showcases the plot for miss-penalty of the Level#0 cache in the presence and absence of the Level#1 cache. As indicated by the plot, in the presence of the Level#1 cache, the miss-penalty for the Level#0 cache is significantly reduced, indicating the successful passing of the test.

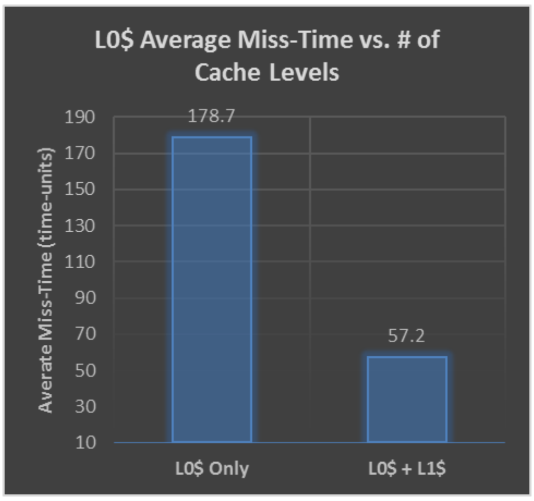


Figure 15: Plot of Level#0 Cache Miss-Time vs. # of Cache-Levels for Test-Scenario 3

D. Test-Scenario 4: Way-prediction reduces cache hit-time

Due to the larger number of ways to be searched within a set, caches with high-associativity suffer from increased hit-times. Way-predictors can help alleviate this inefficiency by predicting the way of the next access, and preloading the multiplexor to select the predicted way to access data in advance. An incorrect prediction would result in the searching of all ways in the next clock-cycle.

In order to verify this test-scenario, we construct a system with configuration parameters as shown in figure 16. We sweep through cache-associativity in the range [4-way, 1024-way] in powers of 4, plotting the average hit-time in the presence and absence of way-prediction, for each data-point. Note that since we only require a single hardware-thread, the configuration parameters of the serializing interconnect are irrelevant.

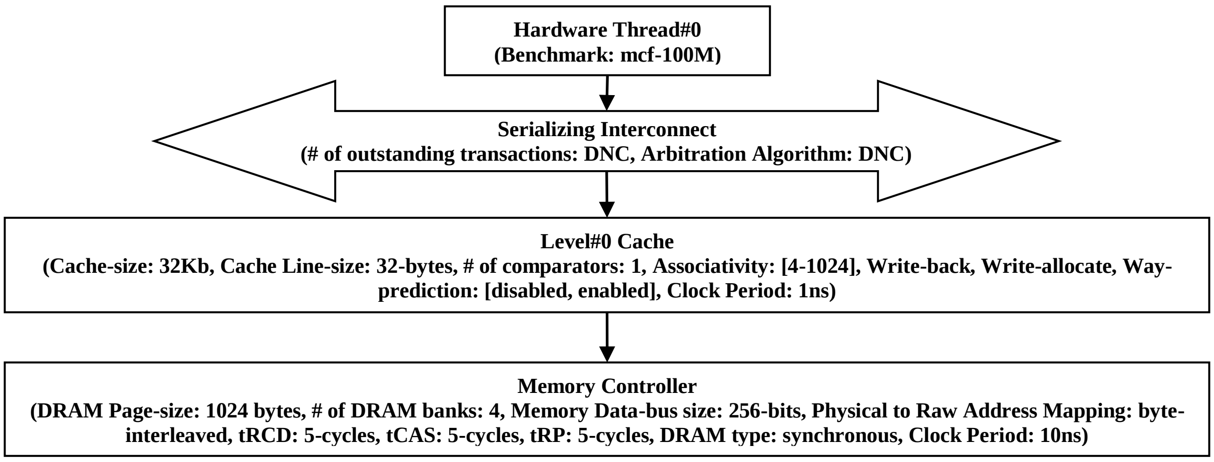

Hardware Thread#0
(Benchmark: mcf-100M)
Serializing Interconnect
(# of outstanding transactions: DNC, Arbitration Algorithm: DNC)
Level#0 Cache
(Cache-size: 32Kb, Cache Line-size: 32-bytes, # of comparators: 1, Associativity: [4-1024], Write-back, Write-allocate, Way-prediction: [disabled, enabled], Clock Period: 1ns)
Memory Controller
(DRAM Page-size: 1024 bytes, # of DRAM banks: 4, Memory Data-bus size: 256-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 16: Block Diagram of a System-Architecture for Test-Scenario 4

Figure 17 showcases the plot for average hit-time vs. associativity, in the presence and absence of way-prediction. As indicated by the plot, way-prediction reduces hit-time, indicating the successful passing of the test.

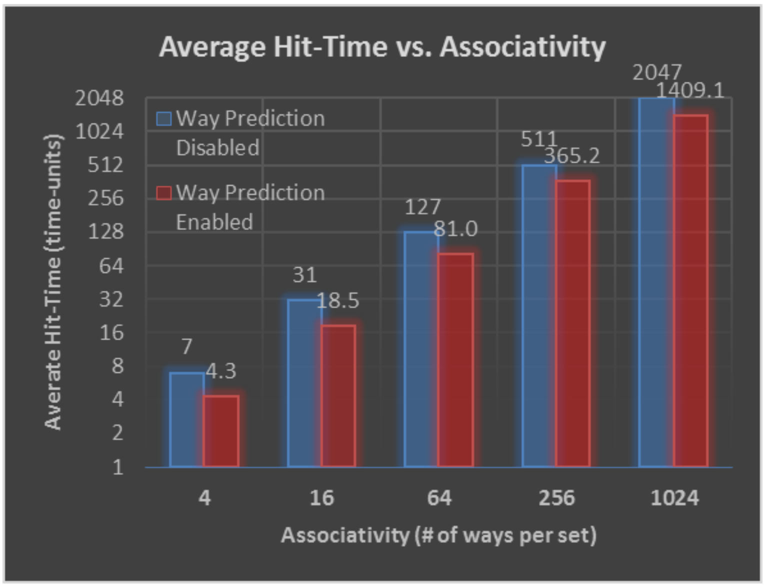


Figure 17: Plot of Average hit-time vs. Associativity for Test-Scenario 4

E. Test-Scenario 5: Larger DRAM pages reduce the average DRAM access latency

Increasing the size of a DRAM page reduces its access latency due to spatial locality – a larger DRAM page implies that a larger chunk of data in close-proximity to the address of the currently accessed page is fetched on a row-buffer miss, thereby increasing the probability of a row-buffer hit, if nearby addresses are accessed.

In order to verify this test-scenario, we construct a system with configuration parameters as shown in figure 18. We sweep through DRAM page-size in the range [64-bytes, 8192-bytes] in powers of 2, plotting the average DRAM access latency for each data-point. Note that since we only require a single hardware-thread, the configuration parameters of the serializing interconnect are irrelevant.

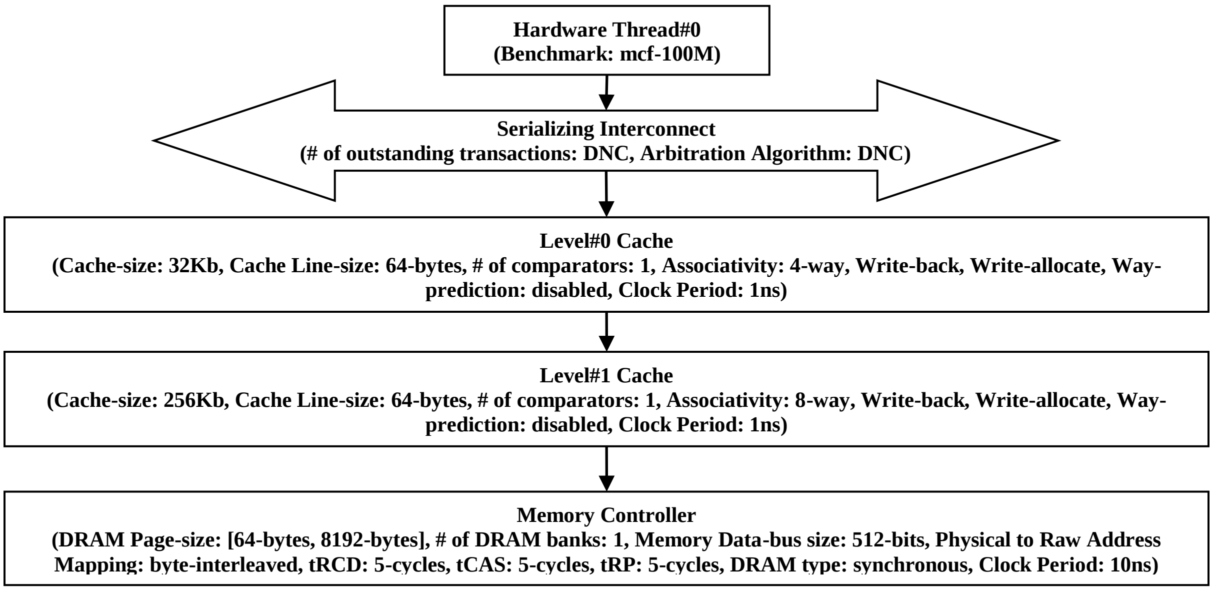

Hardware Thread#0
(Benchmark: mcf-100M)
Serializing Interconnect
(# of outstanding transactions: DNC, Arbitration Algorithm: DNC)
Level#0 Cache
(Cache-size: 32Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 4-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Level#1 Cache
(Cache-size: 256Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 8-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Memory Controller
(DRAM Page-size: [64-bytes, 8192-bytes], # of DRAM banks: 1, Memory Data-bus size: 512-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 18: Block Diagram of a System-Architecture for Test-Scenario 5

Figure 19 showcases the plot for average DRAM access latency vs. DRAM page-size. As indicated by the plot, a larger page reduces access latency, indicating the successful passing of the test.

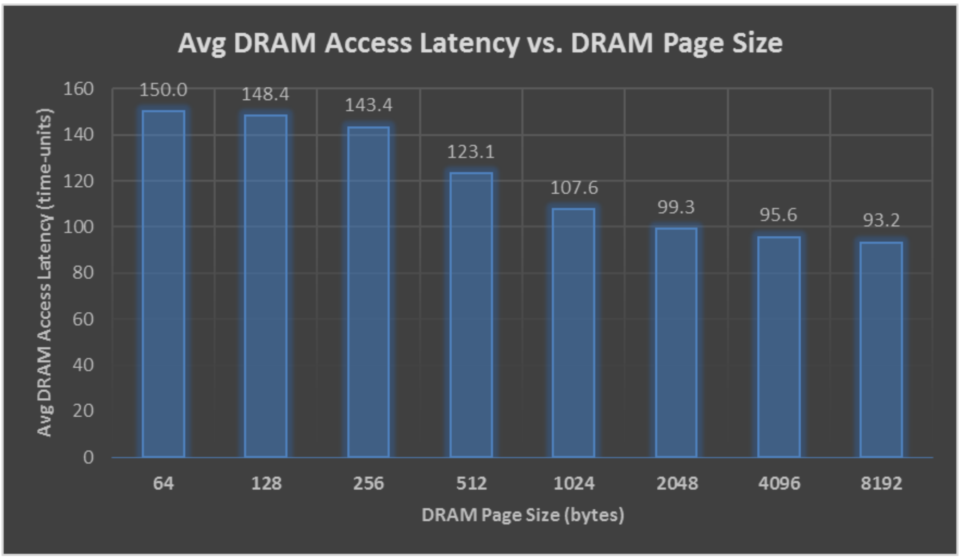


Figure 19: Plot of Average DRAM Access Latency vs. DRAM Page-size for Test-Scenario 5

F. *Test-Scenario 6: Prioritization of a thread is achieved at the cost of the performance of other threads*

In a system where multiple requestors contend for a unified shared-resource, access to the shared-resource must be meticulously arbitrated to ensure fairness to all requestors. In such a system, prioritization of a thread is achieved at the cost of a performance-loss on other threads, resulting in an overall lower system performance.

In order to verify the observability of this phenomenon in this test-scenario, we construct a system with configuration parameters as shown in figure 20. We study the effects of varying serializing interconnect arbitration algorithms on the effect of performance of each thread by plotting the average memory-access latency for each thread. For the static prioritization case, we assign the following priorities: {Thread#0: Priority 3, lowest}, {Thread#1: Priority 2, medium}, {Thread#2: Priority 1, highest}.

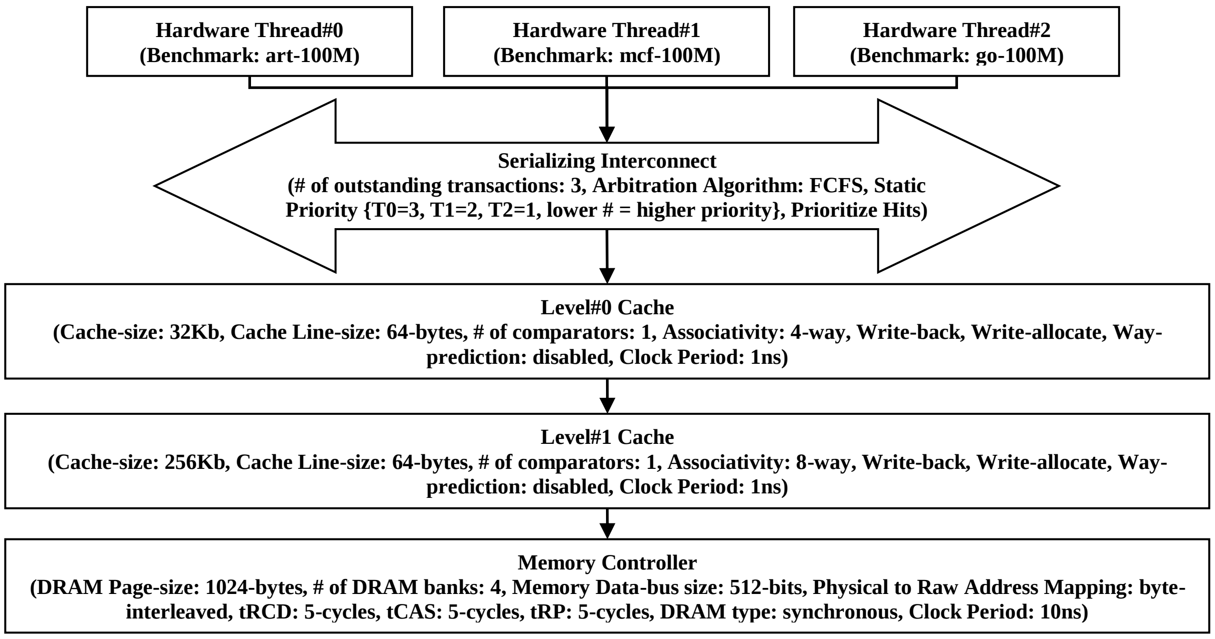

Hardware Thread#0 (Benchmark: art-100M)
Hardware Thread#1 (Benchmark: mcf-100M)
Hardware Thread#2 (Benchmark: go-100M)
Serializing Interconnect (# of outstanding transactions: 3, Arbitration Algorithm: FCFS, Static Priority {T0=3, T1=2, T2=1, lower # = higher priority}, Prioritize Hits)
Level#0 Cache (Cache-size: 32Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 4-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Level#1 Cache (Cache-size: 256Kb, Cache Line-size: 64-bytes, # of comparators: 1, Associativity: 8-way, Write-back, Write-allocate, Way-prediction: disabled, Clock Period: 1ns)
Memory Controller (DRAM Page-size: 1024-bytes, # of DRAM banks: 4, Memory Data-bus size: 512-bits, Physical to Raw Address Mapping: byte-interleaved, tRCD: 5-cycles, tCAS: 5-cycles, tRP: 5-cycles, DRAM type: synchronous, Clock Period: 10ns)

Figure 20: Block Diagram of a System-Architecture for Test-Scenario 6

Figure 21 showcases the plot for average memory-access latency vs. the selected arbitration algorithm. In the first-come-first-served case, all threads share the loss in performance more uniformly. However, in the static-prioritization case, thread#2 (go-100M) and thread#1 (mcf-100M) are prioritized over thread#0 (art-100M), resulting in the degradation of performance of thread#0. The observability of the effects of this phenomenon indicates the successful passing of the test.

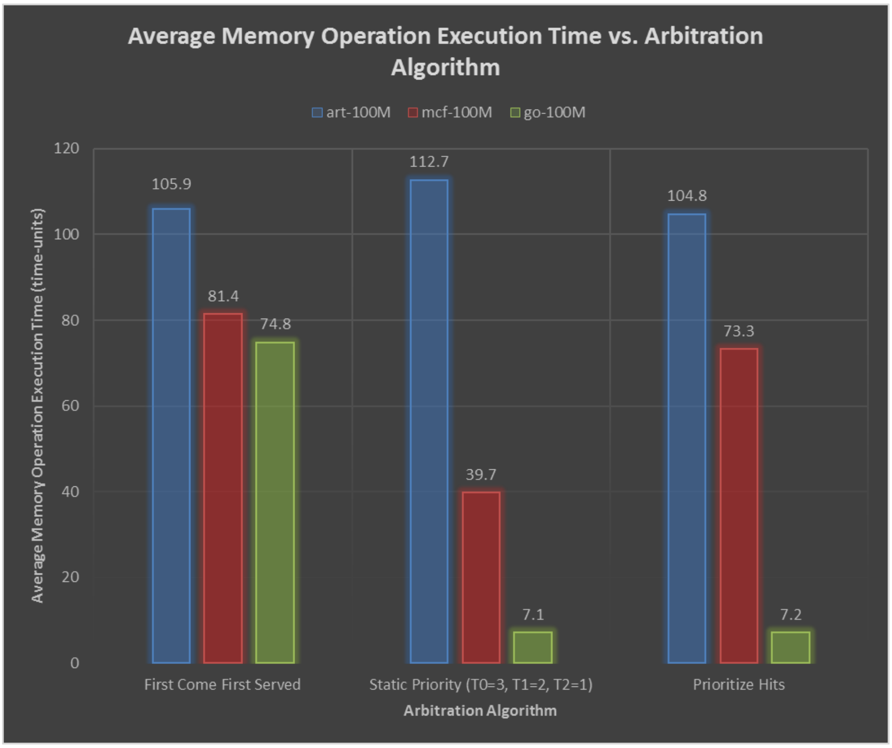


Figure 21: Plot of Average Memory-Access Latency vs. Arbitration Algorithm for Test-Scenario 6

VI. Current Limitations – Scope for Future Enhancements

Although MeSSMArch successfully provides an approximation of memory-system performance, in its current state, it suffers from the following limitations:

1. The support for only a fixed cache-line size throughout the entire hierarchy restricts the exploration of architectures where line-sizes differ at cache-levels. In a typical system, caches closer to the processor are configured to have smaller line-sizes to minimize data-flow to/from the next-level, in order to improve hit-times, and caches further away from the processor are configured to have larger line-sizes to improve miss-rates by exploiting spatial-locality.

2. No support for non-blocking caches restricts the exploration of out-of-order execution hardware-threads because, when the hardware-thread is stalled on a long-latency miss, it cannot issue another memory-transaction (that may result in a hit) because the cache is busy.

3. No support for coherence protocols in the cache and interconnect restricts the exploration of multicore architectures. Although it may be structurally possible to construct a multicore architecture by connecting multiple memory-hierarchies to the same memory controller, the lack of coherence protocols implies that the invalidation-messages required to maintain coherence between the hierarchies will not be communicated due to their non-existence, resulting in functional incorrectness.

4. No support for arbitration in the memory controller restricts the exploration of arbitrated multicore memory-hierarchies. Even if coherence protocols are supported in the cache and interconnect, the lack of an arbiter in the memory controller implies the lack of choice between transactions injected from different hierarchies, thus making it impossible to model behavior such as core-unfairness [17].

VII. Conclusion

As systems grow increasingly complex, in order to cope with constraining time-to-market requirements, it is imperative that designers adopt the usage of efficient tools, which enable the rapid exploration of massive design-spaces in a minimal turnaround time. This necessitates tackling the system-design problem by adopting flows that enable the co-design of hardware and software. The development of such flows is feasible only when designers adopt methodologies that enable the design of systems at higher-levels of abstraction.

Although one of its challenges is the lack of a guideline for the definition of the scope of processes in a transaction-level platform, in our experience, the TLM methodology has been instrumental in enabling the easy transformation of an architecture-specification to an executable-model of the system at the transaction-level, thus enabling swift architectural-exploration.

References

[1] John L. Hennessy and David A. Patterson. 2011. Computer Architecture, Fifth Edition: A Quantitative Approach (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[2] Lukai Cai and Daniel Gajski. 2003. Transaction level modeling: an overview. In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03). ACM, New York, NY, USA, 19-24

[3] Frank Ghenassia. 2006. Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4] "IEEE Standard for Standard SystemC Language Reference Manual," IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1,638, Jan. 9 2012

[5] Ye Lu; Sezer, S.; McCanny, J., "TLM2.0 based timing accurate modeling method for complex NoC systems," Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on , vol., no., pp.2900,2903, May 30 2010-June 2 2010

[6] Menon, S.; Suryaprasad, J., "A pattern based methodology for the design and implementation of multiplexed Master-Slave devices at the system-level use-case: Modeling a Level-2 Cache IP module at transaction level," Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on , vol., no., pp.1,6, 25-26 Nov. 2010

[7] Benny Akesson. An introduction to SDRAM and memory controllers. URL: http://www.es.ele.tue.nl/premadona/files/akesson01.pdf

[8] Onur Mutlu. Computer Architecture, Spring 2015, Lecture 21: Main Memory, Carnegie Mellon University. URL: http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=onur-447-spring15-lecture21-main-memory-afterlecture.pdf

[9] Milo M. K. Martin. Introduction to Computer Architecture, Fall 2010, Unit 10: Hardware Multithreading, University of Pennsylvania. URL: https://www.cis.upenn.edu/~milom/cis501-Fall10/lectures/10_multithreading.pdf

[10] Amir Roth, "A High-Bandwidth Load-Store Unit for Single- and Multi-Threaded Processors", . January 2004

[11] Intel® Hyper-Threading Technology: Technical User's Guide. January 2003. URL: http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf

[12] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence (1st ed.). Morgan & Claypool Publishers.

[13] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput. 28, 9 (September 1979), 690-691. DOI=10.1109/TC.1979.1675439 http://dx.doi.org/10.1109/TC.1979.1675439

[14] Adaptive Resonance Theory 2 (ART 2) Benchmark. URL: https://www.spec.org/cpu2000/CFP2000/179.art/docs/179.art.html

[15] MCF Benchmark. URL: https://www.spec.org/cpu2006/Docs/429.mcf.html

[16] Game of Go (Go) Benchmark. URL: https://www.spec.org/cpu2006/Docs/445.gobmk.html

[17] Mutlu, O.; Moscibroda, T., "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers," Micro, IEEE , vol.29, no.1, pp.22,32, Jan.-Feb. 2009, doi: 10.1109/MM.2009.12