HMC-Sim: A Simulation Framework for Hybrid Memory Cube Devices

John D. Leidel, Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
{john.leidel,yong.chen}@ttu.edu

***Abstract*—The recent advent of stacked die memory and logic technologies has lead to a resurgence in research associated with fundamental architectural techniques. Many architecture research projects begin with ample simulation of the target theoretical functions and approach. However, the logical and physical nature three-dimensional stacked devices, such as the Hybrid Memory Cube (HMC) specification, fundamentally do not align with traditional memory simulation techniques. As such, there currently exists a chasm in the capabilities of modern architectural simulation frameworks. This work introduces a new simulation framework developed specifically for the Hybrid Memory Cube specification. We present a set of novel techniques implemented on an associated development framework that provide an infrastructure to flexibly simulate one or more Hybrid Memory Cube stacked die memory devices attached to an arbitrary core processor. The goal of this development infrastructure is to provide architectural simulation frameworks the ability to begin migrating current banked DRAM memory models to stacked HMC-based designs without a reduction in simulation fidelity or functionality. In addition to the core simulation architecture, this work also presents a series of memory workload test results using the infrastructure that elicit device, vault and bank utilization trace data from within a theoretical device. These evaluations have confirmed that HMC-Sim can provide insightful guidance in designing and developing highly efficient systems, algorithms, and applications, considering the next-generation three-dimensional stacked memory devices. HMC-Sim is currently open source, licensed under a BSD-style license and is freely available to the community.**

Keywords—Hybrid Memory Cube; computer simulation; memory architecture; memory management

I. INTRODUCTION

Historically, computer architects and researchers have relied heavily on a myriad of simulation techniques in order to test new functionality, improve threoretical designs, and reduce the costs associated with refabricating erroneous devices. Traditional simulation infrastructures have relied upon the memory organizations with simple row and column storage techniques. These *two dimensional memory models* do not take into account the forthcoming trend in constructing devices using through silicon via, or stacked, manufacturing process technologies. A prime example of three dimensional memory technology that currently exists as an industry standard is the Hybrid Memory Cube specification [1]. As a result, there exists a gap in the ability to simulate multidimensional memory devices that may contain layers dedicated to functionality outside the current scope of banked memory storage.

In this paper, we introduce three novel techniques that enable flexible simulation capabilities for three dimensional stacked memory devices based upon the Hybrid Memory Cube specification:

1) Multidimensional Memory Infrastructure: Unlike traditional row/column oriented DDR3 and DDR4 [2-3] main memory standards, the Hybrid Memory Cube specification has the addition of multiple die layers that provide parallel access to memory storage in three dimensions. This three dimensional layered approach permits memory transactions to exist in parallel not only across banks within a target storage array, but also across parallel storage arrays. As a result, current memory simulation techniques fall short of providing adequate support for the three dimensional block memory storage.

2) Coupled Logic and Memory Layered Packages: In addition to the stacked memory layers, the HMC specification contains a base logic die layer. This base layer provides flexible access to the three dimensional storage layers via a set of parallel logic blocks. Given that current memory simulation techniques rely upon discrete memory storage and memory controller (logic), they fall short of providing adequate functionality that has the ability to mimic such a tightly coupled, inherently parallel package.

3) Flexibility of Logic and Memory Layer Configurations: Unlike the rigid aforementioned main memory specifications, the HMC specification permits the flexible interpretation and implementation of the target device. There exist several device configurations with respect to capacity, bandwidth, connectivity and internal logic block functionality. The resulting design specification permits implementors from a disparate set of mobile, embedded and high performance device manufacturers. As such, any attempt at building a general purpose simulation infrastructure for an HMC device must also adopt an equivalently flexible infrastructure.

To address the shortcomings of the current memory simulations, we propose and leverage the aforementioned three novel techniques to build *HMC-Sim*, a flexible simulation

infrastructure that enables architectural simulation infrastructures to migrate from traditional banked main memory models to implementations based upon the HMC specification. We are currently utilizing HMC-Sim to develop a high bandwidth memory system to support the massively parallel Goblin-Core64 processor and system architecture project [30]. Furthermore, we enable those performing systems software research associated with addressing models and virtual to physical address translation techniques to accurately simulate the future of stacked memory and logic devices. HMC-Sim is currently open source and freely available to the community, licensed under a BSD-style license model [30][31].

The remainder of this work is organized as follows. Section II gives an overview of historical simulation techniques and why they fall short for three dimensional stacked die implementations. Section III provides a brief overview of the HMC specification and capabilities of compliant devices. Sections IV and V describe the novel technqiues, architectural features and API implementation details of the HMC-Sim infrastructure. Finally, Section VI describes a series of memory workload tests used to demonstrate the novel archtiecture and tracing capabilities present in the HMC-Sim implementation.

II. Related Work

Memory simulation techniques can be classified into two categories, hardware-based techniques and software-based techniques. Despite the recent progress on building fast, accurate and high performance FPGA-based simulation systems and techniques [27-28], the overall class of hardware-based simulation techniques require a high degree of understanding in both logic architecture and systems software architecture. Associated with these trends in hardware simulations, we also observe many advances in cross compilation techniques [4] in order to ease the burden of creating hardware logic. Even so, the burden of creating combinatorial logic that mimcs a multidimensonial logic die is both expensive (hardware costs) and time consuming. HMC-Sim currently focuses only on software simulation models.

Software simulation techniques can be further classfied across several multi-faceted dimensions. These techniques define the intersection of functionality between the following four method classes:

- Synchronous: Defines how the simulation model discretizes time into fixed increments. At individual time increments, all functional components are evaluated and their respective states are updated.
- Event-Driven: Components are only evaluated when their respective inputs are modified between discretized points in time. This is also analogous to message-driven models.
- Process-Model: Considers each component to be a holistic unit and defines processes or process models associated with each unit. Processes advance their own state based upon its interaction with other processes in the simulation system.
- Hybrid Model: Many modern simulation infrastructures utilize two or more of the aforementioned techniques during execution. Sandia's SST (Structural Simulation Toolkit) framework [9-11] is an excellent example of a simulation framework that utilizes both synchronous exectuion and event-driven models. The result is a highly scalable, yet flexible infrastructure.

Virtualization and emulation have become common topics and techniques present among several simulation infrastructures. In several implementations [7-8, 20, 23-24], this enables the infrastructure to abstract the notion of cycle sampling and operation execution from the microarchitectural model in order to provide flexibility across major classes of system architectures. The Zsim [6] notion of virtualization is also very lightweight in order to promote a high degree of parallelization and simulation performance. While this increased flexibility is convenient, this approach does not consider the tightly coupled nature of stacked die memory systems. Virtualization subsequently suffers from a lack of microarchitectural fidelity when modeling granular memory operations in three dimensions.

In summary, the lack of cross-cutting memory simulation techniques beyond the functional and cycle-driven DRAM models [12-15] has lead to a gap in the ability to accurately simulate multidimensional memories. Several attempts have been made to combine techniques from fixed memory models with queueing methods and DRAM cycle accuracy in order to improve the overall fidelity of the simulation [13]. However, these techniques rely upon separate simulations of fixed logic devices that serve the purpose of controlling memory transacations to and from the devices, thus proving inadequate for simulating stacked memory devices.

III. Overview of The Hybrid Memory Cube Specification

A. Device Hierarchy

The Hybrid Memory Cube, or *HMC*, specification [1] is a new high performance, volatile storage paradigm developed by a consortium of memory industry manufacturers and consumers. The guiding principle behind the new device specification is the ability to build through-silicon via, or *TSV*, devices. TSV manufacturing techniques enable adopters to interconnect multiple die layers in order to construct three-dimensional dies. This ability of interconnecting multiple die layers has enabled the specification authors to build a memory device with a combination of memory storage layers and one or more logic layers. In this manner, the device provides the physical memory storage and logical memory transaction processing in a single die package. The end result is a very compact, power efficient package with available bandwidth capacity of up to 320GB/s per device.

The HMC device specification is capable of such bandwidth via a hierarchical and parallel approach to the design. The device hierarchy occurs vertically across the logic layers and the hardware parallelism occurs across a given die layer. The base die layer is generally configured as the logic

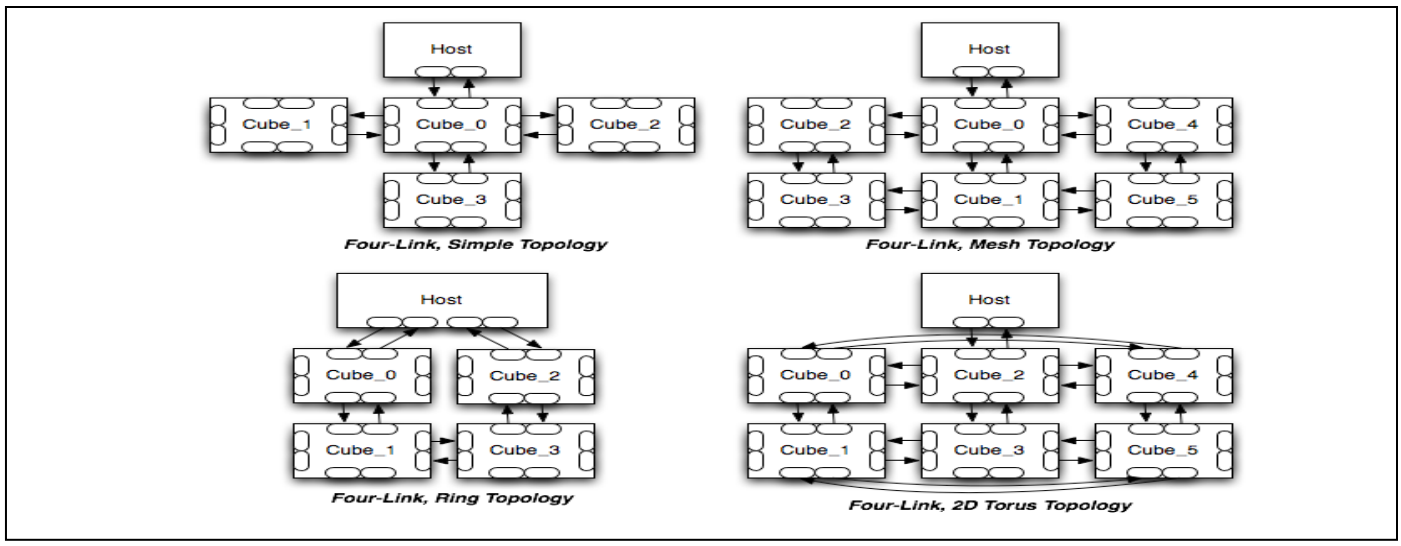

Fig. 1. Hybrid Memory Cube Device Topologies: Simple, Ring, Mesh and 2D Torus.

base, or *LoB*. The logic layer consists of multiple components that provide both external link access to the HMC device as well as internal routing and transaction logic. The external I/O links are provided by four or eight logical links. Each link is a group of sixteen or eight serial I/O, or SERDES, bidirectional links. Four link devices have the ability to operate at 10, 12.5 and 15Gbps. Eight link devices have the ability to operate at 10Gbps.

Internally, the links are attached to routing logic in order to direct transactions at logic devices that control each vertical memory storage unit. The specification does not currently define the routing logic. However, it suggests that a crossbar is a logical example of this interconnection logic.

The link structure in the HMC specification has a unique quality among high performance memory devices in that it supports the ability to attach devices to both hosts (processors) or other HMC devices. This concept of *chaining* permits the construction of memory subsystems that require capacities larger than a single device while not perturbing the link structure and packetized transaction protocols. Links can be configured as host device links or pass-through links in a multitude of topologies. Figure 1 presents four potential device topologies based upon the base, four-link HMC configuration. The HMC specification provides a novel ability to configure memory devices in a traditional network topology such as a mesh, torus or crossbar.

The final major component within the LoB is the vault logic block. Memory in the HMC specification is organized in a three dimensional manner. The major organization unit is referred to as a *vault*. Each vault vertically spans each of the memory layers within the die using the through-silicon vias. The vault logic block is analogous to a DIMM controller unit for each independent vault. These vault logic blocks and their respective vault storage units are organized into quad units. Each quad unit represents four vault units. Each quad unit is loosely associated with the closest physical link block. In this manner, host devices have the ability to minimize the latency through the logic layer of an HMC device by logically sending request packets to links whose associated quad unit is physically closest to the required vault.

Once within a target memory vault, memory storage is again broken into the traditional concept of banks and DRAMs. Vertical access through the stacked memory layers is analogous to choosing the appropriate memory bank. Lower banks are configured in lower die layers while vertical ascension selects subsequent banks. Once within a bank layer, the DRAM is organized traditionally using rows and columns. The vault controller breaks the DRAM into 1Mb blocks each addressing 16-bytes. Read or write requests to a target bank are always performed in 32-bytes for each column fetch.

B. *Interleave and Addressing Methods*

Considering the hierarchical nature of the physical memory storage, the HMC device specification defines a different physical addressing and interleave model than traditional banked DRAM devices. Physical addresses for HMC devices are encoded into a 34-bit field that contain the vault, bank and address bits. The current specification defines four link devices to utilize the lower 32-bits of the field and eight link devices to utilize the lower 33-bits of the field.

Rather than relying on a single addressing structure, the specification permits the implementer and user to define an address mapping scheme that is most optimized for the target memory access characteristics. It also provides a series of default address map modes that marry the physical vault and bank structure to the desired maximum block request size. The default map schemas implement a *low interleave* model by mapping the less significant address bits to the vault address, followed immediately by the bank address bits. This method forces sequential address to first interleave across vaults then across banks within vault in order to avoid bank conflicts.

A sample physical configuration is provided in Figure 2 with the associated HMC-Sim equivalent software structures.

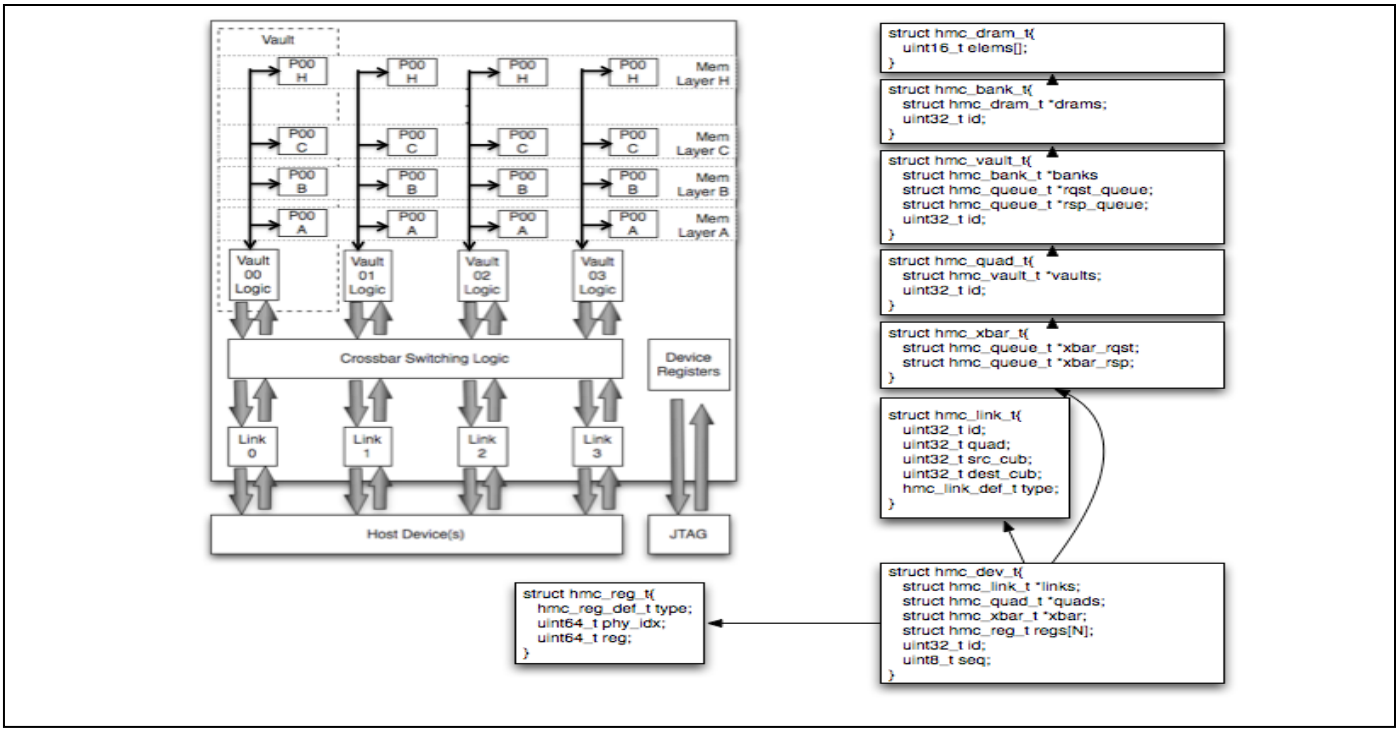

Fig. 2. Hybrid Memory Cube Physical Structure and HMC-Sim memory representation.

The figure contains the host link structure, internal crossbar structure and a single quadrant with the associated four vault units.

C. Packet Structure

All in-band communication between host devices and HMC devices are performed via a packetized format. This includes all three major packet classifications: request packets, response packets and flow control packets. All packets are configured as a multiple of a single 16-byte flow unit, or FLIT. The maximum packet size contains 9 FLITs, or 144-bytes. The minimum 16-byte (one FLIT) packet contains a packet header and packet tail.

Memory read request packets for all memory payload sizes only require the packet header, tail and the respective physical memory address. As such, read requests are always configured using a single FLIT. Write request and atomic request packets, however, must also contain the required input data for write and read-modify-write operations, respectively. As such, these request types have packet widths of 2-9 FLITs.

The HMC device specification defines a weak-ordering model between packets. As such, there may exist multiple packet reordering points present within a target implementation. Arriving packets that are destined for ancillary devices may pass those waiting for local vault access. Local vaults may also reorder queued packets in order to make most efficient use of bandwidth to and from the respective vault banks. However, the specification also states that all reordering points present in a given HMC implementation must maintain the order of a stream of packets from a specific link to a specific bank within a vault. This ensures that memory write requests followed by memory read requests deliver correct and deterministic behavior.

IV. HMC-Sim Architecture

Given the high degree of flexibility and the compound logical and physical structures of the Hybrid Memory Cube specification, it is quite difficult to theorize all possible combinations of device configurations, queuing mechanisms and link structures. As such, we adopted a relatively narrow set of initial implementation goals and requirements in order to promote correctness and flexibility at all levels of the implementation. These requirements are outlined as follows:

1) Maintain Flexibility: As of the writing of this paper there currently did not exist any publicly accessible implementations of the HMC specification. The specification and corresponding HMC-Sim implementation was interpreted holistically in order to support any possible configuration option or topological device organization.

2) Topologically Agnostic: Given the link structures defined in the initial HMC specification, there exists a myriad of correct and incorrect device topologies. Rather than supporting only correct topologies, a concious decision was made to support all possible topologies in the event that the target user and application deliberately misconfigured the devices. In this manner, users and applications can receive response packets with error structures due to incorrect topologies.

3) Flexible Queuing: The device specification explicitly calls out the existence of queueing in two physical logic blocks of an HMC device. Both the crossbar queueing and

vault queueing mechanisms are deliberately defined in an ambigous manner such that implementers may taylor the device to specific requirements. The HMC-Sim implementation follows this paradigm by requiring users to specify the depth of both queueing layers at initialization time.

4) Rudamentary Clock Domains: Many modern architecture simulation infrastructures can be executed in a functional mode, cycle accurate mode or a hybrid of both. The HMC-Sim implements a rundamentary clock domain mechansism such that users or target applications have the ability to instantitate a clock signal to the simulated devices and thus simulate latent packet delivery.

5) Support for all HMC Packet Variations: In order to provide support for a wide array of simulation scenarios, including functional simulation, error simulation and performance simulation; HMC-Sim implements all possible device packet variations using all combinations of FLITs. The internal packet handlers strictly adhere to the packet format specifications and lengths set forth in the specification.

A. Structure Hierarchy

Given the logical and physical hierarchy present in the HMC device specification, we utilize a similar approach in constructing the internal software representation of an HMC device. This structure was chosen in order to easily track packet source and destination correctness throughout the life of a device object. This was also chosen in order to provide both the ability to recognize and trace latency throughout each packet's respective lifetime.

For example, we have a device that is configured with four total links attached to a host device. Each link is physically closest to the respectively number quad unit, which contains a block of four vaults. In this manner, this device contains four quad units and sixteen vaults. The bank layout is not required for this example. The host device transmits a 64-byte write request packet over link zero with the required data payload. The packet is queued in the top-level crossbar queue that is physically closest to the combination of quad unit and vault zero. However, the base destination physical address physically falls on a DRAM bank held within vault five within the second quad unit. In order to successfully complete the write request, the packet must be forwarded across the device to vault five. HMC-Sim will perform this operation correctly. It will also recognize and trace the notion that this operation may be cause for higher latency than simply transmitting the initial write request packet through the second physical link.

As stated above, the internal representation of an HMC device is logically ordered in a hierarchical manner. Each instantiation of an HMC-Sim object adheres to this hierarchy. An application may contain more than one HMC-Sim object in order to simulate architectural characteristics such as non-uniform memory access. Within each object, the structure hierarchy adheres to the following, organized from the highest level to the lowest:

- HMC Devices: Devices are analogous to a single Hybrid Memory Cube device package. The notion of device chaining is supported. Each device structure contains three sub-structures: Links, Crossbar Units and Quad Units. The device structure also contains any device specific configuration registers that are typically accessed via MODE_READ requests, MODE_WRITE requests or the Joint Test Action Group (JTAG) IEEE 1149.1 hardware interface.
- Links: Links are analogous to an HMC physical device link. Per the current specification, device links may connect a host and an HMC device or two HMC devices (chaining). There exists four or eight links per device. HMC-Sim does not currently support mixing devices with different link counts. Each link contains a reference to its closest quad unit and the source and destination device identifiers (including host devices).
- Crossbar Units: Crossbar units are analogous to the first-level logic layer present in an HMC device. They simulate the queuing mechanisms present in the crossbar unit between device links and device vault controllers. Crossbar units contain the request and response queues for the respective device that are accesible from the host.
- Quad Units: Quad units map directly to the notion of a quadrant, or locality domain on an HMC device. Each quad unit is closely related to four vaults in both four and eight link configurations. Each quad unit also contains a pointer to the closest vault unit structures.
- Vaults: The vault structure map directly to the notion of a vertically stacked vault unit within the HMC device specification. Each vault contains response and request queues whose respective depths are configured at initialization time in order to mimic the prescense of a vault controller. Each vault also contains a reference to a block of memory bank structures.
- Banks: The bank structures map directly to the notion of a physical memory bank in the device hierarchy. Each bank is physically nested within its respective vault such that I/O operations do not occur outside the respective vault queue structure. Each bank contains a reference to a block of DRAMs.
- DRAMs: The DRAM structure maps directly to the a physical memory DRAM in the device specification. The DRAM contains the designated data storage for all I/O operations.
- Queue Structure: All the queuing structures present in the HMC-Sim structure hierarchy share the same software representation. Each queue contains one or more queue slots. There must exist at least one queue slot for each logical queue representation in order to act as a registered input or output logic stage. Each queue slot contains a valid designator that describes whether the respective is in use or not. It also contains the storage for a single packet. Each packet is configured to contain sufficient storage for the largest possible packet with nine FLITs.

Despite being configured using a hierarchy of software structures, each HMC-Sim device objects performs well-

aligned internal memory allocation at initialization time. Each respective structure type is allocated as a single block, while hierarchical pointers are initialized to point within this well-aligned allocation. In this manner, we make a best effort to promote good memory utilization and large-page allocation on operating systems and architectures that support these features.

Figure 2 provides a visual comparison between the physical HMC device configuration and the complementary HMC-Sim software structure. The HMC physical device in the diagram contains the host-to-device link structure, internal link crossbar and a single quadrant with associated vault units. Each vault unit contains eight vertical layers of memory storage. The software structures associated with each hardware component are shown in the respective nested format.

B. Link Structure

The previous subsection examines the internal hierarchical software structure present within HMC-Sim. This includes the definition of the link structure that includes the closest physical quad unit as well as the source and destination endpoint identifiers. Associated with each of these source and destination identifiers is an enumerated type that specifies the physical endpoint configuration of the respective link. If the device link is connected to a host device (a non-HMC device), the source link is always configured as the host-side connection. Internally, hosts are represented using non zero HMC Cube ID's of one greater than the total number of devices (*num_devices + 1*). In this manner, hosts are uniquely identified from pure memory devices but are permitted to send and receive request and response packets in a seamless manner.

C. Simulation Clocking

The HMC-Sim infrastructure provides support for rudimentary clock domains between internal and external memory operations. In this manner, the host processor may operate asynchronously from the target memory devices in the event that core processor frequencies, SERDES link frequencies and the target memory device operation frequencies exist across asynchronous clock boundaries. This also promotes the ability to connect multiple HMC-Sim device objects to single host and operate them completely independently. This is analogous to the current system on chip methodology of utilizing multiple memory channels per socket.

Internal memory operations are classified as follows:

- Packet transfers between link crossbar queues and vault queues
- Memory read or write request processing for each respective vault queue.
- Bank conflict recognition within each vault request queue
- Response packet generation following a successful read or write operation
- Response packet generation following a failed read or write operation [error response packets]

External memory operations are classified as follows:

- Sending read or write request packets to a target HMC-Sim device.
- Receiving response or error response packets from a target HMC-Sim device.
- Sending flow-control packets to a target HMC-Sim device
- Any out of band communication via the JTAG interface

Clock signals are instantiated against a single HMC-Sim object using a single API call. Without this call, external memory operations may progress until appropriate stall signals are recognized. However, internal device operations will not progress until an appropriate call to the clock function has been instantiated. One call to this function progresses the internal memory operations and device clock by a single leading and trailing clock edge, or one clock cycle.

The internal clock cycle handlers are progressed in a very explicit order. The goal of this ordering is to promote the reasonable accuracy of internal operations based upon priority and relative latency within the device given the flexibility of the specification. The internal ordering is currently broken into six sub-cycle operations. Request and response packets are only progressed by a single internal stage per sub-cycle operation. For example, it is not possible for an individual memory read or memory write packet to progress from the device crossbar interface directly to the memory bank in a single sub-cycle operation. Figure 3 presents an example simulation flow diagram that describes the sub-cycle stages for single and multiple device configurations.

The sub-cycle operations are organized and performed in the following order:

1) Process Child Device Link Crossbar Transacations: The first sub-cycle stage processes the crossbar packets from each of the configured child HMC devices. These are devices that are not connected directly to a host device. The simulation infrastructure walks each link's respective queue structure and determine which appropriate vault devices or remote HMC devices are candidate destinations for the respective packet request. This stage registers trace messages if packets are misrouted, stalled due to queue congestion or higher latencies are detected due to the physical locality of the queue versus the destination vault.

2) Process Root Device Link Crossbar Request Transactions: The second stage of sub-cycle processes the crossbar packets from the device or devices connected directly to a host interface. The simulation infrastructure walks each link's respective queue structure and determine which appropriate vault devices or remote HMC devices are candidate destinations for the respective packet request. This stage also registers trace messages if packets are misrouted, stalled due to queue congestion or higher latencies are

detected due to the physical locality of the queue versus the destination vault.

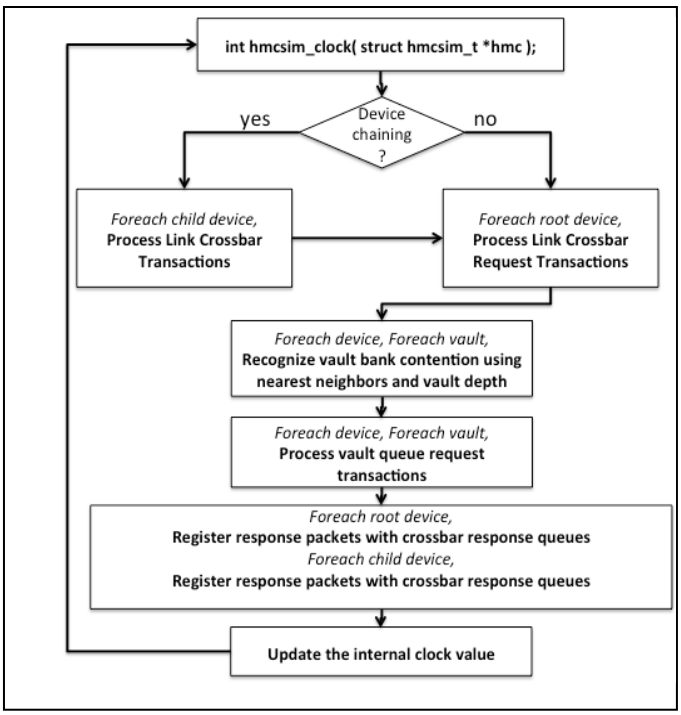

Fig. 3. HMC-Sim sub-cycle clock domain state diagram

3) Recognize Bank Conflicts on Vault Request Queues: This sub-cycle stage does not modify any internal data representations. Rather, it focuses on recognizing potential bank conflicts on each vault based upon the current state of the request queues. This stage performs this operation by decoding the physical memory addresses present in the request packets and determining whether there exists conflicting packets within a spatial window of the queue. If so, a trace message is generated with the physical locality and clock value of the respective bank conflict event.

4) Process Vault Queue Memory Request Transactions: Once appropriate bank conflicts have been identified, each vault queue on all devices is traversed in FIFO order and each request packet is appropriately processed. This includes write packets, read packets and atomic operation (read-modify-write) packets. All packets are currently processed in equivalent and constant time as long as their bank addressing does not conflict. All responses to the aforementioned packet transactions are registered in the appropriate vault response queues.

5) Register Response Packets with Crossbar Response Queues: Once all the initial memory read and write transaction operations are performed, the fifth sub-cycle stage processes all the response packets from the respective operations. This includes routing the packets from child devices to root devices in order to deliver them back to the appropriate host device. Response queues are first processed on the root devices, then the attached child devices. Otherwise, the root device response queues may appear unnecessarily congested and produce false positive stall signals back to the child devices.

6) Update the Internal Clock Value: The final sub-cycle stage updates the unsigned sixty four bit clock value by one. In this manner, all trace messages reported by the first four stages are registered within the *current* clock domain. Once the clock value is updated, the clock signal is considered to be at the leading edge of the next clock domain.

D. Register Structure

In order to provide functionality similar to its physical counterparts, the HMC-Sim device representation contains storage for all internal device configuration, read and status registers found within the HMC device specification. The specification defines that registers are grouped into three classes. There are registers that can be read and written (RW), registers that are read-only (RO) and registers that are self-clearing after being written to (RWS). Each structure contains the configuration class and the register's respective storage. Register indexing on physical HMC devices is not purely linear and does not begin at zero. As such, we have implemented a series of macros that translate HMC device register index formats to a linear format in order to promote efficient memory utilization.

E. Simulation Tracing

As previously stated, the HMC-Sim internal infrastructure is designed to provide cycle by cycle and sub-cycle simulation tracing abilities. Users have the ability to designate the tracing verbosity as well as the target output file buffers. Trace granularity can be set such that each internal sub-cycle operation is recorded to the trace file. As such, entire application memory traces can be revisited and analyzed for accuracy, latency characteristics, bandwidth utilization and overall transaction efficiency. Each trace event is marked with its physical locality as well as the respective internal clock tick when the respective trace event was raised.

V. HMC-Sim Application Programming Interface

The HMC-Sim infrastructure is implemented in ANSI-style C and packaged as a single library object. It can be easily utilized in existing infrastructures without modification. The API consists of four major function classes: *device initialization*, *topology initialization*, *packet handlers* and *register interface* functions. Figure 4 provides a sample API calling sequence in *C*.

A. Device and API Initialization

The HMC-Sim API interface provides a single master initialization function. This function provides the user or parent application the ability to initialize one or more simulated HMC devices as well as instantiate the devices into a reset state. This initialization function requires the user to specify the physical details associated with one more target HMC devices. Currently, HMC-Sim requires that the devices within a single object must be physically homogeneous. In this manner, each device will be initially configured and reset

to an identical state. The entire HMC-Sim API function descriptions can be found in the HMC-Sim online documentation [31].

```
/* Section A. Init the devices */
struct hmcsim_t hmc;
ret = hmcsim_init( &hmc, num_devs, num_links, num_vaults,
                         queue_depth, num_banks, num_drams,
                         capacity, xbar_depth );

/* Section B. Config the link topology */
for( i=0; i<num_links; i++ ){
        ret = hmcsim_link_config( &hmc, (num_devs+1), 0, i,
                                  i, HMC_LINK_HOST_DEV );
}

/* Section C. Build a request packet */
uint64_t payload[8];
uint64_t head, tail;
hmcsim_build_memrequest( &hmc, 0, phy_address, tag,
                                 RD_64, link, &(payload[0]),
                                 &head, &tail );

/* Section C. Send the request */
ret = hmcsim_send( &hmc, &(payload[0]) );

/* Clock the sim */
hmcsim_clock( &hmc );

/* Section A. Free the devices */
hmcsim_free( &hmc );
```

Fig. 4. HMC-Sim sample API sequencing

B. Link and Topology Initialization

Given the chaining ability present in the HMC device links, the HMC-Sim infrastructure provides the ability for the user to define the memory topology between multiple devices. This includes the ability to define the host-to-device relationships as well as the device-to-device relationships.

There are, however, several constraints induced by the simulation infrastructure. First, devices that link to one another must exist within the same HMC-Sim object structure. The infrastructure does not currently support linking devices and their inherent queuing structures between disparate HMC-Sim objects. Next, the infrastructure does not permit users to configure links as loopbacks. Loopbacks in the link topology have a high probability of inducing zombie response requests that never reach a reasonable destination. Finally, the user must configure at least one device that connects to a host link. Otherwise, the host will have no access to main memory.

C. Request and Response Packet Handlers

The HMC-Sim API interface provides two sets of interface functions to assist with packet request and response handling. The first set of functions provide users or host applications the ability to execute simple *send* or *receive* operations of request or response packets, respectively. These send and receive operations interact with the request and response queues of the target HMC devices directly. The send function requires the application to have a preformatted, fully formed, compliant packet structure. This packet structure determines the candidate link and device combination as well as the desired memory transaction. The receive operation requires the user to specify the HMC device and link combination to poll for candidate response packets. Response packets are delivered as fully formed, compliant packet structures and may arrive out of order. It is up to the calling application to decode and correlate the response packet information to the correct memory transaction request. The API provides two functions to assist with encoding and decoding request and response packets, respectively.

D. Modifying and Querying HMC Device Registers

The current HMC device specification provides two interfaces for hosts to query and modify device registers. The packet interface utilizes MODE_READ and MODE_WRITE packet types to read and write device registers using the attached memory links. The benefit to this interface is its familiar packet structure and the ability to query or modify registers on devices that are chained or not directly connected to the host. These packet types will route to the destination cube ID as would any other packet type. However, the downside to this method is the use of available memory bandwidth. HMC-Sim supports the use of MODE_READ and MODE_WRITE packets to query and modify register state, but warns that performing these operations may have negative performance implications.

The second access method provided in the current specification is via a Joint Test Action Group IEEE 1149.1 (JTAG) or Inter-Integrated Circuit (I^2C) bus infrastructure. The benefit to this access method is the side-band nature of the bus. It does not interrupt main memory traffic to and from the HMC devices. HMC-Sim provides a rudimentary function interface to both read and write device registers using this JTAG interface. This interface exists external to the normal HMC-Sim notion of clock domains. As such, this interface appears to exist out of band to the user or application simulation infrastructure.

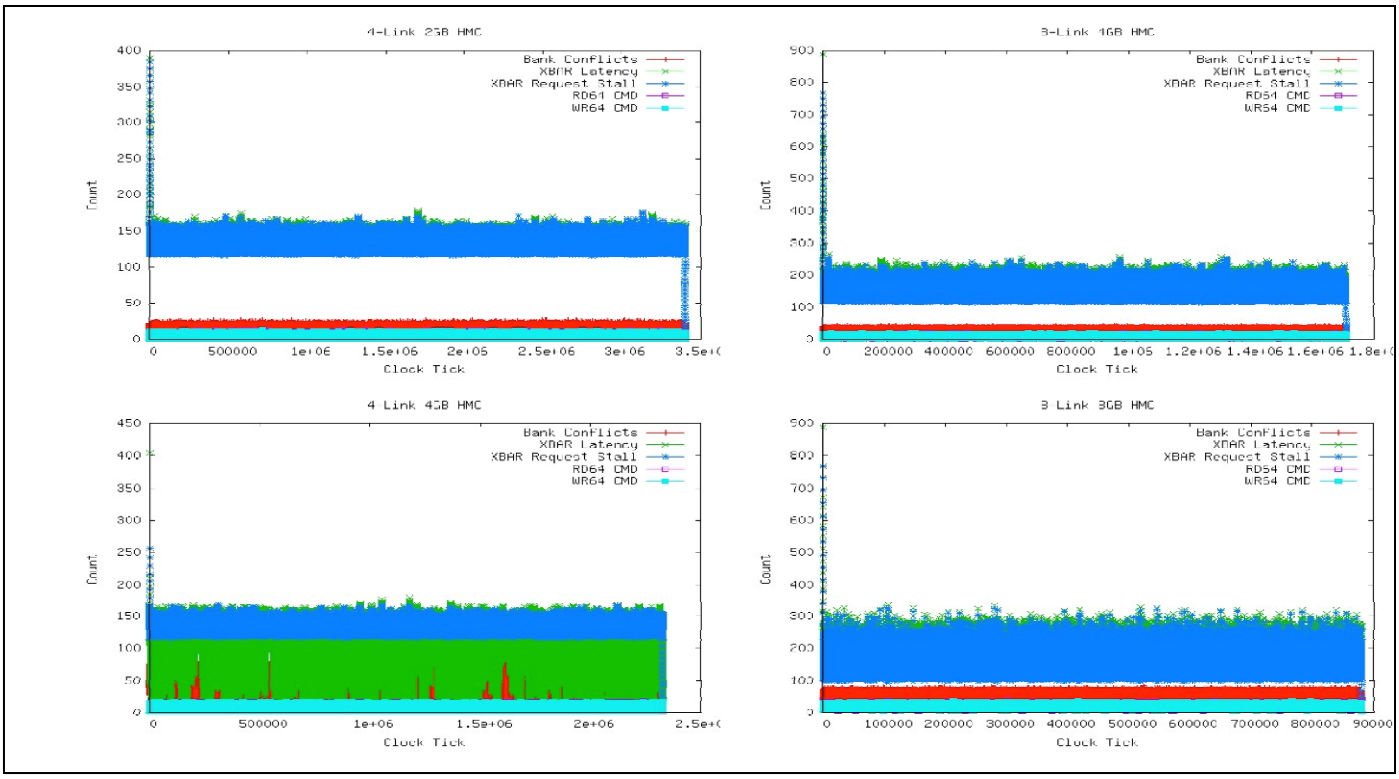

Fig. 5. HMC-Sim Random Access Simulation Results

VI. Simulation Results

A. Simulation Overview

For the purposes of both testing the simulation infrastructure and learning more about the fundamental characteristics of utilizing HMC devices, we have constructed a random access memory test harness. The test application has the ability to generate a randomized stream of mixed reads and writes of varying block sizes against a specified HMC device configuration. The randomness is driven via a simple linear congruential method provided by the GNU libc library. The application will send as many memory requests as possible to the target device or devices until an appropriate stall is received indicating that the crossbar arbitration queues are full. The application selects appropriate HMC links in a simple round-robin fashion in order to naively balance the traffic across all possible injection points. The tests were executed using 33,554,432 64-byte memory requests where the read/write mixture was 50/50. The resulting memory pattern is similar to a parallel random number sort of 2GB of data.

The HMC test configurations consisted of devices with 128 bi-directional arbitration queue slots for each crossbar link and 64 bi-directional arbitration queue slots for each vault unit. We tested devices with both 4 and 8-links configured in both 8 and 16 banks per vault for each link configuration. The resulting set of four device configurations provided ample opportunity to observe the trace data from within the simulator model.

B. Simulation Analysis

The simulations were configured to enable all the possible internal tracing outputs in order to capture every potential unit of data. The resulting trace files ranged in capacity from 16GB for the 8-link/8GB device to 40GB for the 4-link/2GB device. The graphical results provided in Figure 5 are direct representations of five pertinent values. First, the graphs project the number of bank conflicts, read requests and write requests that occurred within each vault at each respective cycle. The graph also plots the number of crossbar request stalls observed internal to the device and the number of events raised due to the potential routed latency penalties at each simulated clock cycle. The crossbar request stalls occur when a request cannot be routed from a crossbar arbiter to the target vault due to inadequate open vault queue slots. The latency penalties are raised when a request is received on a link that is not co-located with the destination quadrant and vault.

From the data that we collected in the aforementioned tests, we observe several interesting phenomena. First, the number of clock cycles that the simulator required to complete the same number of operations predictably scaled with the number of links and the number of banks within a vault. Table 1 provides the raw number of required clock cycles for each device type as well as the appropriate speedup values. We observe an average speedup of 1.7X by using the same number of links, but increasing the number of banks. We also observe an average speedup of 2.319X by using the name number of banks, but doubling the link count from 4 to 8.

TABLE I. SIMULATION RUNTIME IN CLOCK CYCLES

Simulation Runtime in Clock Cycles	
Device Configuration	***Simulated Runtime in Cycles***
4-Link; 8-Bank; 2GB	3,404,553
4-Link; 16-Bank; 4GB	2,327,858
8-Link; 8-Bank; 4GB	1,708,918
8-Link; 16-Bank; 8GB	879,183

In addition to this observation, we observe that proper host-side link routing plays an important factor in minimizing latency and maximizing throughput. The number of crossbar link stalls and the number raised latency degradation events are similar in all four tested configurations. This simple corollary implies that locality-aware host devices have the potential to reduce memory latency and reduce internal memory device contention in order to make most efficient use of the available bandwidth.

While actual empirical results on physical HMC devices were not available prior to submitting this work for acceptance, these simulation tests have successfully confirmed the functionality of the HMC-Sim simulation infrastructure as well as the HMC packet specification. It can provide insightful guidance in designing and developing highly efficient systems, algorithms and applications for the next generation three-dimensional stacked memory devices.

VII. CONCLUSIONS AND FUTURE WORK

In conclusion, through our research and development of HMC-Sim, we introduce three novel simulation techniques in this paper specific to constructing flexible and functionally accurate interface models for emerging three-dimensional stacked memory devices, the future high-bandwidth low-latency memory devices. Our technique of building multidimensional memory representations will enable others to perform research on the next generation of high bandwidth microarchitecture and system on chip techniques. Finally, we hope the flexibility present in HMC-Sim will encourage industry and research groups to begin experimenting and utilizing the Hybrid Memory Cube specification. HMC-Sim can have a broad impact in early algorithm, system and application design and development that consider the distinct features of three-dimensional stacked memory devices HMC-Sim is currently developed as a part of the Goblin-Core64 architecture research project [30][31].

REFERENCES

[1] *Hybrid Memory Cube Specification Version 1.0*, Hybrid Memory Cube Consortium, January 2013.

[2] *DDR3 SDRAM Standard*, JEDEC JESD79-3F, July 2012.

[3] *DDR4 SDRAM Standard*, JEDEC JESD79-4, September 2012.

[4] J. Bachrach, V. Huy, B. Richards, L. Yunsup, et al. "Chisel: Constructing hardware in a Scala embedded language." *DAC*, 49:1212-1221, June 2012.

[5] Computer architecture simulation and modeling. *IEEE Micro Special Issue*, 26(4), 2006.

[6] D. Sanchez, C. Kozyrakis. "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[7] T. Carlson, W. Heirman, and L. Eeckhout. "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Supercomputing*, 2011.

[8] S. Herrod, "Using complete machine simulation to understand computer system behavior," Doctor of Philosophy thesis, Department of Computer Science, Stanford University, California, USA, 1998.

[9] G. Hendry, and A. Rodrigues. "SST: A simulator for exascale co-design," in *ASCR/ASC Exascale Research Conference*, April 2012.

[10] C. Janssen, H. Adalsteinsson, S. Cranford, J. Kenny, et al. "A simulator for large-scale parallel architectures, " *International Journal of Parallel and Distributed Systems*, 1(2):57-73, 2010.

[11] R. Murphy, K. Underwood, A. Rodrigues, and P. Kogge. "The structural simulation toolkit: A tool for exploring parallel architectures and applications," Internal SAND Report, SAND2007-0044C, 2007.

[12] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAM-Sim2: A cycle accurate memory system simulator, " *CAL*, 10(1), 2011.

[13] S. Srinivasan, L. Zhao, B. Ganesh, et al. "CMP Memory Modeling: How much does accuracy matter?," in *MoBS-5*, 2009.

[14] P. Magnusson, and B. Werner. "Some efficient techniques for simulating memory," in *SICS*, 1994.

[15] R. Uhlig, and T. Mudge. "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, 29:128-170, 2004.

[16] M. Chidester and A. George. "Parallel simulation of chip-multiprocessor architectures," *TOMACS*, 12(3), 2002.

[17] R. Fujimoto. "Parallel discrete event simulation," *CACM*, 33-10, 1990.

[18] S. Chandrasekaran and M. D. Hill. "Optimistic simulation of parallel architectures using program executables," In *PADS*, 1996.

[19] C. J. Mauer, M. D. Hill, and D. A. Wood. "Full-system timing-first simulation," in *SIGMETRICS* conf., 2002.

[20] S. K. Reinhardt, M. D. Hill, J. R. Larus, et.al. "The Wisconsin Wind Tunnel: virtual prototyping of parallel computers," in *SIGMETRICS conf.*, 1993.

[21] A. Jaleel, R. Cohn, C. Luk, and B. Jacob. "CMPSim: A pin-based on-the-fly multi-core cache simulator." in *MoBS-4*, 2008.

[22] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind. "Fast cycle-accurate modeling of a multicore processor," in *ISPASS*, 2012.

[23] M. Martin, D. Sorin, B. Beckmann, et al. "Multifacet's general execution driven mulitprocessor simulator (gems) toolset," *Comp. Arch. News*, 33-4, 2005.

[24] T. Wenisch, R. Wunderlich, M. Ferdman, et al. "Simflex: statistical sampling of computer system simulation," *IEEE Micro*, 26(4), 1996.

[25] E. Witchel and M. Rosenblum. "Embra: Fast and flexible machine simulation," in *SIGMETRICS Perf. Eval. Review*, volume 24, 1996.

[26] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ISCA-30*, 2003.

[27] Z. Tan, A. Waterman, R. Avizienis, et al. "RAMP Gold: An FPGA-based architecture simulator for multiprocessors," in *DAC-47*, 2010.

[28] D. Chiou, D. Sunwoo, J. Kim, et al. "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle accurate simulators," in *MICRO-40*, 2007.

[29] P. Koopman, T. Chakravarty, "Cyclic redundancy code (CRC) polynomial selection for embedded networks," DSN-2004, June 2004.

[30] J. Leidel, *Goblin*-Core64. Goblin-Core64 Processor Architecture Research Project, June 2013. [Online]. Available: Gobline-Core64 Processor Architecture Research Project, www.gc64.org [Accessed: December 2013].

[31] J. Leidel, *HMC_SIM 1.0 API Documentation*. Goblin-Core64 Processor Architecture Research Project, June 2013. [Online]. Available: Goblin-Core64 Processor Architecture Research Project, https://goblin-core.googlecode.com/svn/doc/hmc-sim/html/index.html [Accessed: December 2013].