HMC-Sim-2.0: A Simulation Platform for Exploring Custom Memory Cube Operations

John D. Leidel
Department of Computer Science
Texas Tech University
Lubbock, Texas, US
john.leidel@ttu.edu

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, US
yong.chen@ttu.edu

***Abstract*—The recent advent of stacked memory devices has led to a resurgence of research associated with the fundamental memory hierarchy and associated memory pipeline. The bandwidth advantages provided by stacked logic and DRAM devices have inspired research associated with eliminating the bandwidth bottlenecks associated with many applications in high performance computing. Further, recent efforts have focused on utilizing stacked memory devices as last-level caches.**

In addition to the two aforementioned focus areas, a third area of research is emerging to explore augmenting the stacked memory logic layer with additional operations. This first generation of Hybrid Memory Cube (HMC) devices provided rudimentary atomic memory operations. The Gen2 Hybrid Memory Cube devices provide more expressive atomic memory operations that include primitive integer arithmetic operations. Despite the inclusion of more expressive arithmetic operations, many users have expressed interest in more complex and potentially orthogonal *custom memory cube*, or CMC, operations in future revisions of the Hybrid Memory Cube specification.

This work presents recent development associated with the HMC-Sim Hybrid Memory Cube simulation framework that provides users a powerful infrastructure to experiment and research augmented custom memory cube, or CMC, operations within the current Gen2 Hybrid Memory Cube device infrastructure. We provide an overview of extending the original HMC-Sim simulation infrastructure to include support for CMC operations with requiring users to modify the core simulation code base. We also present three examples of building and utilizing custom, user-defined CMC operations in sample simulations to exhibit potential application speedup with future HMC device specifications. In doing so, we present a model to replace traditional thread mutexes with custom HMC mutex commands.

***Keywords*-Hybrid Memory Cube; Custom Memory Cube; Simulation**

I. Introduction

Recent advancements in memory manufacturing technology have inspired a new class of memory devices based upon the notion of stacking multiple DRAM and logic layers. This has permitted the memory architects to re-craft the physical connectivity to host processors and significantly improve the available bandwidth. Technologies such as High Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) show significant potential to improve the performance for applications that are traditionally memory bound [1].

In addition to the potential performance improvement realized by changing the fundamental physical memory organization and memory interconnect, we also see a significant resurgence in research associated with augmenting the logic portions of the memory device in order to perform more expressive operations. This processing in or near memory, often referred to as *PIM* operations, has the potential to reduce the memory bandwidth to perform common operations such atomic arithmetic operations, atomic boolean operations and mutexes by performing the operations in-situ within the resident memory device.

One of the major difficulties in research associated with PIM operations is deciding which operations are most advantageous for eventual production devices. Logic space within the memory device package may come at a premium cost and, as such, adding all potential PIM or atomic operations may be infeasible. In this manner, a significant portion of the research associated with placing additional operations in the logic layer will be performing a cost-benefit analysis of the various potential operations.

In this work we present an extension to the existing Hybrid Memory Cube simulation infrastructure, HMC-Sim, that permits users to craft custom memory cube, or *CMC*, operations and examine their efficacy using varying user applications or pathological algorithms/kernels. In addition to adding core support for the operations and packet formats in the HMC 2.0/2.1 specification, the CMC infrastructure provides users the ability to construct arbitrarily complex operations and simulate their efficacy residing within the current HMC command infrastructure. The CMC infrastructure has the ability to load up to seventy disparate operations concurrently and simulate their efficacy and performance from user applications without modifying the HMC-Sim core library. All the necessary code required to implement the new CMC operations is implemented in externally compiled and loaded shared library objects, thus simplifying the respective implementation complexity for the user. The goal is to provide a simple and stable platform to perform advanced research on arbitrarily complex or expressive memory operations.

The remainder of this paper is organized as follows. Section II introduces previous work associated with HMC-Sim and other stack memory simulation infrastructures. Section III presents the introductory architecture and implementation details of HMC-Sim as well as the relative differences with previous versions. Section IV provides a detailed description of the driving requirements, implementation and user framework behind the Custom Memory Cube architecture within the HMC-Sim 2.0 simulation infrastructure. Section V provides an examples of building three CMC operations that could potentially replace traditional mutex operations commonly found in Linux/UNIX systems. We conclude with a summary of this work and potential future development.

II. Previous Work

Given the resurgence of research associated with high performance memory architectures and orthogonal memory device technologies, we find several relevant works that also present simulated results of Hybrid Memory Cube devices. Rosenfeld (et. al.) has a very complete implementation of a Hybrid Memory Cube simulation infrastructure built upon traditional DRAM simulation techniques [6]. This simulation environment contains discrete device information beyond what is currently enabled in HMC-Sim such as DRAM timing information and vault controller timing. Further, the authors integrate the simulation infrastructure into the MARSSx86 full system simulation [8] and presents additional data regarding random memory access kernel performance [7].

In addition to basic simulation research, several research efforts have focused on adapting the capability found in current HMC devices for more inclusive system architectural tasks. One such effort was detailed by Kim (et. al.) to utilize the HMC physical layer, packet format and routing infrastructure as the basis for a system interconnect [9]. They propose the use of *memory-centric networks*, or *MCNs*, to serve as a generic replacement for traditional system interconnects such as Intel's QPI and AMD's HyperTransport. While the work does not utilize any inherent features present in the Gen2 HMC specification, it does detail a very novel use of the inherent routing capabilities of the HMC specification.

One of the more recent works associated with the application of HMC devices on actual workloads detailed the use of the HMC 2.0 specification's atomic memory operations to accelerate graph traversals [10]. That work presents results and future research work on utilizing the 2.0 spec's *compare and swap* operations to accelerate a model of the traditional breadth first search of a graph. The authors utilize the CAS operations to replace check-and-update portion of the kernel, thus using the HMC device as a makeshift processor in memory (PIM). The end result depicts a potentially significant savings in overall kernel bandwidth utilization.

In our previous work, we have detailed the internal architecture and efficacy of utilizing the HMC-Sim infrastructure to simulate application kernels executing against a memory system comprised of one or more Hybrid Memory Cube devices [4] [5]. The initial HMC-Sim infrastructure contained support for the entire version 1.0 HMC Specification [3], including the ability to *chain* multiple HMC devices together in a multitude of different topologies. The initial HMC-Sim release included support for all normal I/O packets, register read packets as well as internal access to the device via a simulated JTAG API. The initial set of results presented the simple application of random memory requests against varying device configurations in order to elicit the efficacy of resolving data from the simulation infrastructure [4]. The second set of results presented actual kernel execution data resolved from executing a STREAM Triad kernel [11] and an HPCC RandomAccess kernel [12] against varying device configurations. The end result was a simple analysis of the efficacy and potential bottlenecks of different device configurations based upon the pathological kernel memory access pattern (stride-1 versus random).

III. HMC-Sim-2.0 Overview

In order to preserve the efficacy and current user interface, we have limited the number of user-facing changes made to the HMC-Sim infrastructure for the 2.0 release. However, we have gone to great lengths to adapt the initial infrastructure for the changes to the 2.0/2.1 version of the HMC Specification [2]. The updates to the infrastructure were focused on four main areas. First, we received patches back from several users that found bugs in the packet handling and device layout structure. These bug fixes were integrated into the original 1.0 version of the source and brought forward to the current 2.0 version of the source as well.

Second, we updated the internal packet handlers within the HMC-Sim infrastructure to take into account the differences in packet format moving to the 2.0/2.1 specification. The biggest changes were related to the command fields. The HMC Consortium elected to expand the command fields such that more commands could be supported in future releases while limiting the use of other data fields that proved less useful. The end result was the request and response packets (specifically, the head and tail data payloads) changed format.

The vast majority of the work to update the initial HMC-Sim infrastructure to the latest specification was to support the vast number of additional commands supported in the 2.0/2.1 specification. The initial specification had support for basic memory read and write operations in granularities of 16 bytes. Memory requests could be made for 16 bytes and up to 128 bytes for both reads and writes. In addition to the basic write operations, the initial specification also defined a

Table I
HMC-SIM 2.0 GEN2 ADDITIONAL COMMAND SUPPORT

Read Commands	*Command Enum*	*Request Flits*	*Response Flits*	*Write Commands*	*Command Enum*	*Request Flits*	*Response Flits*
256 Byte Read	RD256	1	17	256 Byte Write	WR256	17	1
*	*	*	*	Posted 256 Byte Write	P_WR256	17	0

Atomic Commands	*Command Enum*	*Request Flits*	*Response Flits*	*Atomic Commands*	*Command Enum*	*Request Flits*	*Response Flits*
Dual 8 Byte Signed Add Imm	TWOADD8	2	1	Single 16 Byte Signed Add Imm	ADD16	2	1
Posted Dual 8 Byte Signed Add Imm	P_2ADD8	2	0	Posted 16 Byte Signed Add Imm	P_ADD16	2	0
Dual 8 Byte Signed Add Imm and Rtn	TWOADDS8R	2	2	Single 16 Byte Signed Add Imm and Rtn	ADDS16R	2	2
8 Byte Increment	INC8	1	1	Posted 8 Byte Increment	P_INC8	1	0
16 Byte XOR	XOR16	2	2	16 Byte OR	OR16	2	2
16 Byte NOR	NOR16	2	2	16 Byte AND	AND16	2	2
16 Byte NAND	NAND16	2	2	*	*	*	*
8 Byte CAS If Greater Than	CASGT8	2	2	16 Byte CAS If Greater Than	CASGT16	2	2
8 Byte CAS If Less Than	CASLT8	2	2	16 Byte CAS If Less Than	CASLT16	2	2
8 Byte CAS If Equal	CASEQ8	2	2	16 Byte CAS If Zero	CASZERO16	2	2
8 Byte Equal	EQ8	2	1	16 Byte Equal	EQ16	2	1
8 Byte Bit Write	BWR	2	1	Posted 8 Byte Bit Write	P_BWR	2	0
8 Byte Bit Write with Rtn	BWR8R	2	2	16 Byte Swap or Exch	SWAP16	2	2

set of *posted* write operations that did not return a response packet. With the advent of the 2.0/2.1 specification, we added support for the new 256 byte read, write and posted write operations beyond what was carried forth from the initial implementation. We present the full set of commands added in the HMC-Sim 2.0 infrastructure in Table I.

In addition to the extension of the basic read/write commands to 256 bytes, the Gen2 specification added a number of atomic memory operations, or AMOs. Each atomic memory operation is designed to utilize additional logic in the logic layer of the device to perform *read-modify-write* operations in-situ. If an application kernel relies upon these new AMO commands, the potential result is a significant increase of available bandwidth.

Consider the following example. A user desires to perform an atomic increment of a single 8 byte value. This is commonly used for shared counters in multi-threaded applications. The user has the choice of a traditional cache-based atomic increment or the atomic increment present in the HMC 2.0/2.1 specification. The cache-based atomic increment functions by fetching a cache line from memory, performing the atomic increment on the respective data value in the cache and flushing the value back to memory. This requires a full *read-modify-write* cycle on the cache line. While this may appear somewhat optimal if multiple CPU cores share the same cache hierarchy, any lack of cache locality will induce significant coherency traffic.

Table II
HMC GEN2 ATOMIC MEMORY OPERATION EFFICIENCY

AMO Type	*Request Structure*	*128 Byte FLITS Required*	*Total Bytes*
Cache-Based	Read 64 Bytes + Write 64 Bytes	(1FLIT + 5FLITS) + (5FLITS + 1FLIT)	1536
HMC-Based	INC8 Command	1FLIT + 1FLIT	256

Conversely, consider the potential to utilize the HMC's atomic increment. Regardless of the shared caching hierarchy, multiple cores could effectively have equivalent access to the target data member via the *INC8* HMC command. Each core's memory pipeline would dispatch an *INC8* request for the same block, utilizing only one FLIT (128 bytes) of data for the request and the response. This, compared to the twelve total FLITs required for full *read-modify-write* cycle required for 64 byte cache lines (Table II). As such, it becomes clear how the HMC device's expressive memory operations provide the potential to significantly improve application throughput.

Finally, after several requests from the HMC-Sim user and research community, we added support for users to craft their own custom memory cube, or CMC, operations within the confines of the current HMC specification. The remainder of this work is focused on detailing how this was integrated into the HMC-Sim infrastructure, how new design and development are made to support users to craft their own CMC implementations and providing sample results of performing CMC operations using application kernels.

IV. Custom Memory Cube Architecture

A. Requirements

Given that the original HMC-Sim 1.0 implementation had an established user base, we went to great lengths to architect and define the architectural requirements behind the CMC implementation such that the impact to users was minimized. In order to focus on the inherent implementation requirements, we made several critical decisions regarding platform support that permitted us to decouple the HMC core and the support for CMC operations. Specifically, we made a critical decision to only continue support Linux and UNIX-style platforms and discontinue any potential support for Windows. This enabled us to rely upon the existing and consistent dynamic loading capabilities and API interfaces present in Linux and UNIX-style system libraries. We define the fundamental requirements as follows:

- *API Compatibility*: Every effort was made to continue the support for the existing HMC-Sim 1.0 API implementation. Given that several high-level simulation infrastructures have utilized HMC-Sim 1.0 as the basis for their stacked memory support, we were very cognizant to avoid perturbing the existing API.
- *External Implementation*: Every effort was made to avoid forcing a potential CMC implementor from learning the inner mechanism of the existing HMC-Sim implementation. The existing HMC-Sim implementation is very explicitly designed to mimic the hardware structure of the respective HMC generation in order to provide users the ability to discretely model the effects of different memory request patterns on device latency and bandwidth. As such, the internal implementation relies upon a deep knowledge of the respective HMC specification. Furthermore, we have utilized several techniques in order to improve the overall performance of the simulation infrastructure that may not be obvious to outside developers.
- *Creative Experimentation*: One of the primary requirements for live simulation is the ability to support many disparate combinations of user-defined CMC operations. In this manner, we did not want to limit the user's creativity to combine multiple CMC operations in different simulation environments.
- *Utilize HMC Packet Formats*: In addition to supporting many disparate combinations of CMC operations, we wanted to make explicit use of all the current unused HMC Gen2 command codes. The Gen2 architecture has sufficient command code space to support all the normal read/write, posted read/write, atomic and flow control operations while leaving room for an additional 70 unused command codes. As we describe in later sections, we make use of each of these 70 command codes in order to remain absolutely compliant with the current Gen2 HMC packet formats.
- *Discrete Tracing*: The HMC-Sim 1.0 infrastructure had powerful tracing capability that permitted users to see exactly how and where memory operations progressed through the device or devices. Rather than implementing simple, opaque tracing of user-defined CMC operations, we sought to enable more expressive tracing for CMC operations. This would allow users to create simulation traces where their distinct CMC operations would be resolved in the trace file just as any normal HMC command.
- *Separable Implementation*: The current HMC-Sim infrastructure is licensed under a BSD-like license. However, many users may elect to license their CMC operation implementations under other license structures or potentially make the implementations proprietary. We did not want to prohibit any of these potential paths for the user community. In order to do so, we required that the implementation of the CMC functionality be split such that a portion resides in the BSD-licensed HMC-Sim core and a portion reside outside the core code base. At which time, the user community would have the ability to discretely separate their implementations from the HMC-Sim core.
- *No Simulation Perturbation*: The final, and arguably, the most pertinent requirement is the ability for us to integrate all the aforementioned changes into the simulation infrastructure without perturbing the inherent functionality already present. We do not want to disturb the stability of the current codebase or risk perturbing the ability to replicate existing results in user simulations and previous publications.

B. Design and Implementation Overview

Given the aforementioned requirements, we chose to architect the CMC implementation for the 2.0 version of HMC-Sim such that only a limited portion of the implementation was embedded in the core library. In this manner, we fundamentally decouple the design and implementation of any potential CMC operations from the HMC-Sim library core. As shown in Figure 1, we break the design and implementation into two structures:

1) *Internal Structure*: The internal HMC-Sim library changes necessary to operate a simulation with or without CMC operations loaded

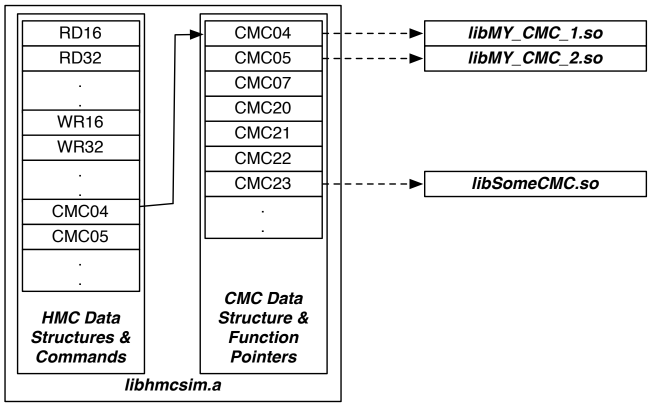


Figure 1. CMC Functional Overview

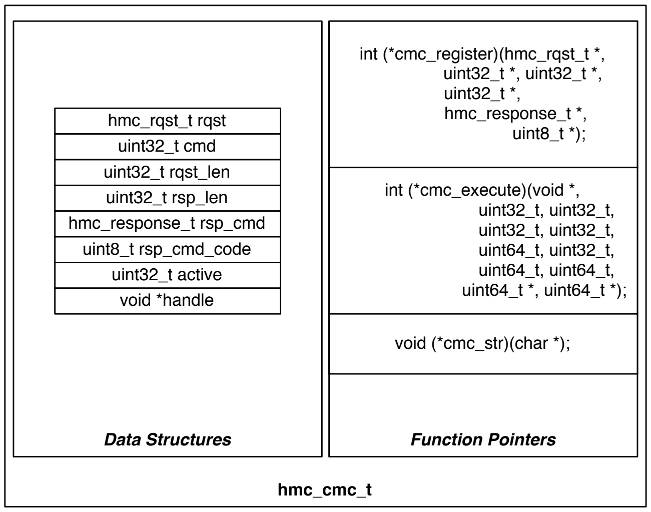
2) *User Library Structure*: The external CMC shared library created by a user that implements the data structures and processing behind a single CMC operation

C. Internal Structure

The internal structure of the HMC-Sim 2.0 core library infrastructure requires several updates and additions in order to accommodate the user-defined CMC functionality. These updates were concentrated on two main areas: the data structures necessary to enumerate an arbitrary set of CMC operations and the logic necessary to handle the registration and processing of the external CMC operations. Each of the aforementioned updates were patched into the existing HMC-Sim source code in a manner that did not perturb the existing packet processing methodology.

1) CMC Data Structures: The first area of concern within the core HMC-Sim library that required attention was the core data structures necessary to express both traditional HMC operations and the extended CMC operations. The existing HMC-Sim codebase makes use of a set of enumerated types to describe the request packet types and the response packet types. This made executing normal HMC requests much more convenient than manually constructing the packets using their true binary encodings. For example, the 64 byte write request is enumerated as *WR64*. This feature proved useful for adding support for the CMC operations. Each of the seventy unused command codes was added to the *hmc_rqst_t* enumerated type table as *CMCnn*, where *nn* describes the corresponding decimal command code. In this manner, all possible unused command codes are now enumerated as potential CMC operations.

Alongside the basic enumerated request types, we also add a single additional enumerated response type to the *hmc_response_t* enumerated types. The response command code field is eight bits in width, so there exists sufficient encoding space for non-traditional response codes. We add



hmc_rqst_t rqst
uint32_t cmd
uint32_t rqst_len
uint32_t rsp_len
hmc_response_t rsp_cmd
uint8_t rsp_cmd_code
uint32_t active
void *handle
int (*cmc_register)(hmc_rqst_t *, uint32_t *, uint32_t *, uint32_t *, hmc_response_t *, uint8_t *);
int (*cmc_execute)(void *, uint32_t, uint32_t, uint32_t, uint32_t, uint64_t, uint32_t, uint64_t, uint64_t, uint64_t *, uint64_t *);
void (*cmc_str)(char *);
Data Structures
Function Pointers
hmc_cmc_t

Figure 2. CMC Internal Structure

a single additional enumerated type, *RSP_CMC*, that permits the external CMC library implementation to define a non-traditional response packet command code. CMC implementors have the ability to define entirely custom response commands that are arbitrarily loaded at runtime.

In addition to the basic enumerated request and response support, we constructed a separate data structure to store all the necessary information associated with a single CMC operation. As depicted in Figure 2, the *hmc_cmc_t* structure is designed as a generic data structure that is manipulated when a user requests to load a new CMC operation into an HMC-Sim simulation context. Each potential CMC operation is allocated a unique *hmc_cmc_t* structure.

Each structure instance contains the respective CMC request enum (*rqst*) and its associated decimal command code (*cmd*). In addition to the request enum and code, the structure also contains a value (*rqst_len*) that describes the length of the incoming request packet in *FLITS*. A single HMC FLIT represents 128 bits of packet data. The request length recorded in the CMC structure describes the total packet length, which includes the packet header and tail. As such, the minimum packet request size is one FLIT. The maximum request packet is 17 flits, analogous to a 256 byte write request.

The *hmc_cmc_t* structure also contains data relevant to the respective CMC response data. The first field, *rsp_len*, contains the total length of the response packet sent back to a host processor when a packet request is complete. The response packet is optional as the CMC operation may describe the request as being *posted*. The maximum response packet size is 17 flits, analogous to a 256 byte read request.

Alongside the packet response size, the structure also records the response command (*rsp_cmd*) if necessary. As stated above, we permit the CMC library implementation to define a non-standard response command code that is encoded into the response packet. This value can be described in the *rsp_cmd_code* field if the response command is set to *RSP_CMC*.

The final block of data stored in the *hmc_cmc_t* structure is critical to the processing stages of the CMC functionality. This final block stores three function pointers that are resolved from a CMC implementation's shared library instance when the library is loaded into the HMC-Sim context. These function pointers are resolved via calls to the dynamic loader *dlsym* API. We store three function pointers as follows:

- *cmc_register*: The CMC registration function that resolves all the internal data items as described above.
- *cmc_execute*: The CMC execution function that is instantiated to execute a single CMC operation.
- *cmc_str*: The CMC string handler that is utilized to resolve a human-readable name for the CMC operation that is printed in the trace logs.

2) CMC Registration and Processing: The second area of concern in implementing the CMC functionality was the ability to quickly and easily inject registration and processing capabilities for an arbitrary set of CMC operations into the infrastructure without perturbing the associated API. A single function, *hmc_load_cmc()* is added to the user API to perform this process.

The registration process first ensures that the HMC-Sim context has been successfully initialized. If this is true, the CMC shared library object is loaded into the current process's address space using the Linux/UNIX dynamic loader API (*dlopen*). Once we have verified that the library was successfully loaded, we begin resolving the three function pointers as described in Section IV-C1. Each of the three function pointers are resolved and stored in the corresponding *hmc_cmc_t* structure instance. If any of the functions (symbols) fail to resolve, then the registration function returns an error.

The final stage of the registration process resolves the data members of the respective CMC operation. It is at this stage that the *cmc_register* function in the CMC shared library object is executed via its function pointer. The data returned is written to the convenience data members also described in Section IV-C1.

Now that at least one CMC operation is loaded and registered in the respective HMC-Sim simulation context, we can begin accepting packets for the initialized commands. At this stage of the operation, as depicted in Figure 3, much of the functionality occurs internal to static function members within the core HMC-Sim library. More details regarding how packets are processed is described in previous works [4] [5]. Once a request is pushed into a vault queue instance, it begins working through the actual processing stages. The internal function that performs much of this is referred to as *hmcsim_process_rqst()*.

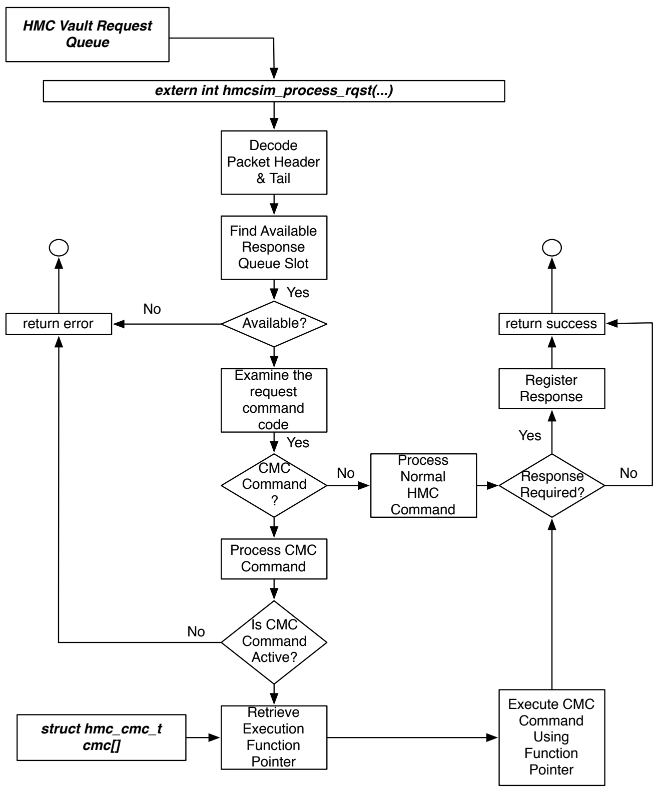

HMC Vault Request Queue
extern int hmcsim_process_rqst(...)
Decode Packet Header & Tail
Find Available Response Queue Slot
Yes
No
return error
Available?
Examine the request command code
Yes
CMC Command ?
No
Process Normal HMC Command
Response Required?
Yes
No
Register Response
return success
Process CMC Command
Is CMC Command Active?
No
struct hmc_cmc_t cmc[]
Retrieve Execution Function Pointer
Execute CMC Command Using Function Pointer

Figure 3. CMC Processing

This packet processing step first examines the respective CMC command code and ensures that it has been marked as *active*. If the command is not marked as active, an error is returned. If the command is marked as active, the processing logic retrieves the respective function pointers for the *cmc_execute* and *cmc_str* functions. The next step in the packet processing is the actual execution of the CMC operation via a call to the *cmc_execute* function pointer. If the return status of the execute function successful, then a trace entry is inserted using the string value for the respective CMC command retrieved via a call to the *cmc_str* function. At this point in the CMC processing, the execution function has returned any necessary response packet data and normal response packet construction is resumed.

D. User Library Structure

The user library structure utilized for a CMC operation implementation is designed to support access to all the necessary data structures to and from the HMC-Sim core library infrastructure without requiring the user to manually implement a large amount of source code. Each CMC library implementation is written in C using a well-defined set of static global values and a set of externally accessible symbols. First, the static global values are designed to describe the various command values, request/response

Table III
REQUIRED CMC STATIC GLOBAL VARIABLES

Variable	*Type*	*Description*
__op_name	static char *	Character string uniquely identifying the operation in the trace files
__rqst	static hmc_rqst_t	Defines the enumerated command type
__cmd	static uint32_t	Defines the command code. Must match the __rqst field
__rqst_len	static uint32_t	Defines the length of the request packet in FLITS
__rsp_len	static uint32_t	Defines the length of the response packet in FLITS
__rsp_cmd	static hmc_response_t	Defines the response packet type.
__rsp_cmd_code	static uint8_t	Defines the command code value if the response type has been set to RSP_CMC

structures and other values that uniquely identify the CMC operation from other, adjacent operations. Table III describes the entire set of static global values required in a unique CMC implementation.

In addition to the global static variables utilized to describe the general functionality of the CMC operation, each implementation is also required to expose at least three functions that map to the respective *cmc_execute*, *cmc_register* and *cmc_str* functions. The latter two functions were described in detail in previous sections. However, the *cmc_execute* function is of special interest. This is the only function whose implementation is not provided by the CMC template source within the HMC-Sim 2.0 source tree. This is also the function that performs the actual CMC operation and; as such, the user is required to implement.

We will focus the remainder of this section on describing the arguments to the execution function. The symbol name of the function is required to be set to *hmcsim_execute_cmc* such that the dynamic symbol resolution API (*dlsym*) can locate the symbol by name given an arbitrary shared library object. The arguments to this function were carefully selected to provide the implementor with a sufficient amount of information and data to perform arbitrarily complex operations using the standard HMC Gen2 packet formats. However, it is up to the implementor to manage any internal data structures discretely and in a thread-safe manner. We caution the potential implementers to review the requirements of the incoming and outgoing request and response payloads, respectively, in order to avoid inducing buffer overflow states and corrupting their simulation data. We describe the arguments further in Table IV.

V. CMC SIMULATION

A. CMC Operation Overview

In order to showcase the potential utility of the aforementioned CMC functionality, we constructed a set of three CMC operations that provide fundamental atomic locking or mutex capabilities within a target HMC device or devices, when bundled together. The proposed set of three CMC operations are modeled after the traditional *pthread_lock*, *pthread_trylock* and *pthread_unlock* system API functions, respectively. The goal of these coupled CMC operations is to provide basic atomic mutex capabilities that exist entirely in user execution space without the requirement to induce a context switch into kernel execution space. The end result being a strong method to control access to critical sections that exist entirely in user space. Before we describe our respective implementation, we must first define a set of governing assumptions as follows:

- *User Allocation*: The implementation assumes that the user or application has allocated at least one 16 byte (one FLIT) data block on an HMC device that is visible to all encountering threads.
- *Initial State*: The mutex values are initialized to a known state that signifies that no locks are present and no threads own the lock.
- *CPU Access*: The CMC operations assume that the host CPU or CPU's contain memory pipelines and associated memory controllers that have the ability to induce the aforementioned three CMC operations.
- *User API*: The CMC operations assume that there exists

Table IV
CMC EXECUTION FUNCTION ARGUMENTS

Argument	*Type*	*Description*
hmc	void *	A pointer to the HMC-Sim core context structure that can be cast to an hmc_sim_t
dev	uint32_t	The respective device where the op is occurring
quad	uint32_t	The respective quad where the op is occurring
vault	uint32_t	The respective vault where the op is occurring
bank	uint32_t	The respective bank where the op is occurring
addr	uint64_t	The target base address of the incoming request
length	uint32_t	The length of the incoming request in FLITS
head	uint64_t	The raw packet header
tail	uint64_t	The raw packet tail
rqst_payload	uint64_t *	A pointer to the raw request payload containing any potential incoming data payloads. It is up to the implementor to discern which portions of the request payload are header, data and tail
rsp_payload	uint64_t *	A pointer to the raw response payload that may contain any potential outgoing data. It is up to the implementor to discern which portions of the response payload are header, data and tail

Table V
CMC MUTEX OPERATIONS

Operation	*Pseudocode*	*Command Enum*	*Request Command*	*Request Length*	*Response Command*	*Response Length*
hmc_lock	IF (ADDR[63:0] == 0){ ADDR[127:64 = TID; ADDR[63:0]=1; RET 1}ELSE{ RET 0 }	CMC125	125	2 FLITS	WR_RS	2
hmc_trylock	IF (ADDR[63:0] == 0){ADDR[127:64 = TID; ADDR[63:0]=1; RET ADDR[127:64]}ELSE{ RET ADDR[127:64] }	CMC126	126	2 FLITS	RD_RS	2
hmc_unlock	IF (ADDR[127:64] == TID && ADDR[63:0] == 1){ ADDR[63:0] = 0; RET 1}ELSE{ RET 0 }	CMC127	127	2 FLITS	WR_RS	2

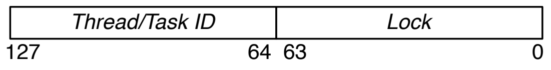


Figure 4. HMC Mutex Data Structure

a well-defined user API and/or compiler intrinsic to induce the aforementioned CMC operations from high-level user code.

As mentioned above, the CMC mutex operations are loosely modeled after the traditional pthread locking API's present in traditional Linux/UNIX systems. We also note that the target memory space for the locking operations is a 16 byte data block, as opposed to a single mutex variable (Figure 4). Given that the minimum granularity of DRAM request is 16 bytes, there are no inherent performance advantages for reducing the lock data structure to less than 16 bytes. Furthermore, the additional data structure capacity permits us to encode additional information that may become useful for some algorithms or applications. In doing so, we define the encoding space to be split into two 64 bit payloads. The least significant 64 bits encodes the lock value. Any nonzero value indicates that the lock has been set. We reserve the ability to encode more expressive locks (such as *soft* locks) in this space in the future. The most significant 64 bits encode a thread or task ID (visible to the user application) of the parallel unit that currently owns the lock. If the lock in the least significant 64 bits is not set, then the state of the most significant 64 bits is assumed to be undefined.

In order to operate on the aforementioned mutex data structure, we define three CMC operations that permit the user to *lock*, *try-and-lock* and *unlock* the data structure, respectively. Despite the common naming convention, our locking primitives differ in their request and return arguments from the traditional pthread mutex equivalents. The locking primitive, *hmc_lock*, sends a single block of data as an argument to the request that contains the thread or task ID for the respective unit of parallelism that dispatches the operation. The *hmc_lock* operation will attempt to lock the target data structure and write its respective thread ID in to the most significant 64 bits of the lock struct. If the operation is successful, the return payload will contain a positive value. If it is not successful, the operation will not modify the data and will return zero in its response payload. Conversely, the *hmc_trylock* operation will perform a similar task. However, rather than return the success or failure of the operation, the response payload will contain the thread or task ID of the unit of parallelism that currently holds the lock. It is up to the encountering thread to check the response payload against its respective thread ID. Finally, the *hmc_unlock* command requires the same thread ID as a request argument and attempts to unlock the target data structure only if the thread ID currently resident in memory is equivalent to the value sent in the request payload. The response for the *hmc_unlock* operation follows the same response convention as the *hmc_lock* operation. We summarize our CMC mutex operations with respect to the HMC 2.0/2.1 specification in Table V.

B. Simulation Overview

We construct a series of simulations in order to examine the efficacy of our aforementioned mutex operations. In order to do so, we construct a basic parallel algorithm that is commonly used to protect critical sections in parallel code. The algorithm utilizes three CMC operations in order to obtain the critical lock. The same lock structure is utilized for all threads, regardless of the three types of parallelism. While this will undoubtedly induce a memory hot spot once the degree of parallelism reaches a sufficient level, our test serves to elicit the efficacy of the CMC infrastructure and the scalability of the HMC queuing structures. We further describe the algorithm implemented in our test as follows:

We implemented and executed the aforementioned algorithm using two different HMC Gen2 configurations that include a 4Link-4GB device and an 8Link-8GB device. Both configurations were initialized to contain a maximum block size of 64 bytes (which subsequently does not affect our respective simulation), a request queue depth of 64 slots and a logic-layer crossbar queue depth of 128 slots. We varied the number of threads from two to one hundred threads for

Algorithm 1 CMC Mutex Algorithm

```
for Nthreads do
    HMC_LOCK(ADDR)
    if LOCK_SUCCESS then
        HMC_UNLOCK(ADDR)
    else
        HMC_TRYLOCK(ADDR)
        while LOCK_FAILED do
            HMC_TRYLOCK(ADDR)
        end while
        HMC_UNLOCK(ADDR)
    end if
end for
```

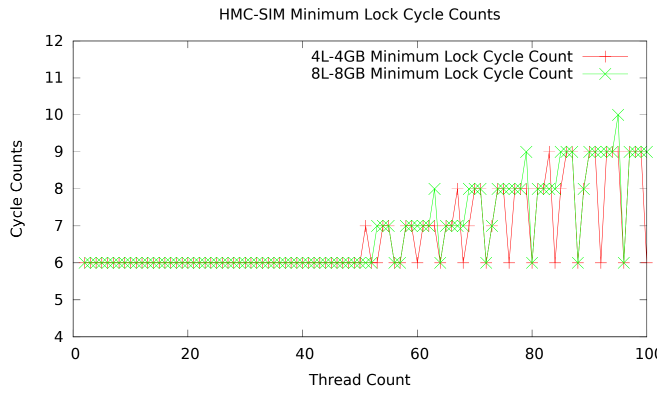


Figure 5. Minimum Lock Cycles

each of the respective configurations. From each simulation, we recorded the following data values (in addition to the core HMC-Sim tracing):

- *MIN_CYCLE*: The minimum number of cycles required for any of the threads to perform the algorithm.
- *MAX_CYCLE*: The maximum number of cycles required for any of the threads to perform the algorithm.
- *AVG_CYCLE*: The average number of cycles required for all the respective threads for the given simulation to perform the algorithm.

C. Simulation Results

After executing the aforementioned simulations, we find that the 4Link and 8Link HMC devices delivered very similar performance. The minimum, maximum and average HMC-Sim cycle counts are actually identical between both the 4Link and 8Link device configurations for thread counts from two to fifty. We attribute this similarity to the identical queueing structure for both configurations and the hot spotting induced from utilizing a single lock structure.

However, when the simulations grew beyond fifty threads, we begin to see perturbations in the results. As the thread count grows, the distributions of requests across the additional 8 links and their associated request and crossbar

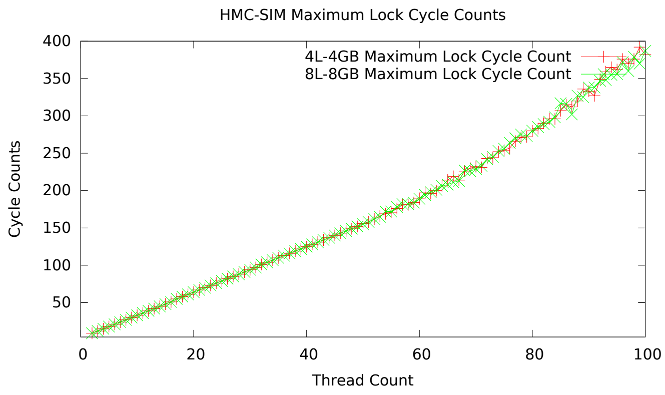


Figure 6. Maximum Lock Cycles

Table VI
CMC MUTEX OPERATIONS

Device	*Min Cycle Count*	*Max Cycle Count*	*Avg Cycle Count*
4Link-4GB	6	392	226.48
8Link-8GB	6	387	221.48

queuing structures begin to induce slightly lower minimum cycle timings. Figure 5 clearly identifies these slightly lower cycle counts beyond fifty threads for the 8 link devices. We also see slightly larger maximum cycle timings in the 4 link device as well. While it is more difficult to visually depict the maximum cycle timings due to the likeness of values, we provide the timings in Figure 6. The worst case maximum cycle count recorded by the 4 link device occurred when using 99 threads and required 392 cycles. Conversely, the 8 link device exhibited its maximum cycle count at 100 threads with 387 cycles. In this manner, the 4 link device clearly becomes overwhelmed with requests faster, thus inducing more stall conditions, than the complementary 8 link device configuration.

Finally, the average resulting cycle counts continue to exhibit similar behavior as shown in Figure 7. The 4 link device recorded a slightly higher maximum average cycle timing of 226.48 cycles at 99 threads. The 8 link device recorded its highest average cycle count at 100 threads using 221.48 cycles. Despite the existence of twice the theoretical queueing capacity, the 8 link device only delivered a worst case maximum cycle timing that was 1.2% better than the complementary 4 link device. In addition, the maximum average cycle timing of the 8 link device was only 2.2% better than the 4 link device. We summarize our results in Table VI.

VI. CONCLUSION

In this work, we present a novel approach to simulating arbitrarily complex custom memory cube, or *CMC*, operations within the confines of the HMC-Sim Hybrid Memory Cube simulation infrastructure. With the addition of this

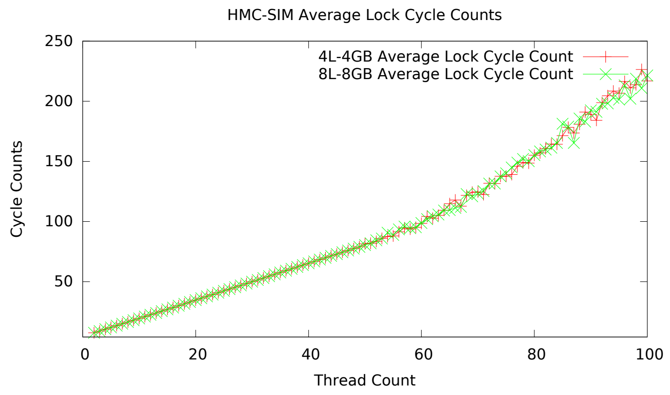


Figure 7. Average Lock Cycles

CMC functionality, HMC-Sim has the ability to simulate user-defined processing near memory, or *PIM*, operations alongside the augmented set of Gen2 atomic memory operations present in the 2.0/2.1 version of the HMC specification. The CMC implementation infrastructure is architected to remain partially decoupled from the core HMC-Sim library in order to prevent the CMC operations from perturbing normal HMC commands and to provide users the ability to quickly develop complex operations without modifying the HMC-Sim core.

Further, we showcase the aforementioned CMC functionality using a set of three user-defined operations modeled after traditional pthread mutex locking functions. These functions provide the ability to lock, test a lock and unlock an atomic mutex embedded in any 16 byte region of HMC memory. We provide simulation results scaling from 2 to 100 threads using a 4link-4GB and 8link-8GB HMC configurations. Despite the inherent hot spotting that occurs with a single lock region, we see that the 8link device configurations provide better performance than the 4link devices due to the enhanced queueing capacity with the additional 4 links. With this research and development, the HMC-Sim CMC functionality provides users the ability to quickly and easily prototype complex operations that, despite their orthogonality, can be simulated seamlessly alongside the operations present in the current HMC specification.

VII. Future Work

Following discussions with the current HMC-Sim user community, there is significant interest in the community to augment the simulation capabilities current present in HMC-Sim with more accurate timing and power resolution data. Given the agnostic nature of the HMC-Sim implementation, we have deliberately avoided including performance, timing and power data in the core infrastructure from one individual HMC implementation. However, given that the second generation of the device specification is publicly available and several versions of HMC devices are available to the public, we may be able to distill the necessary data down to the point where we can reasonably model the timing and power characteristics of an arbitrary HMC device.

References

[1] M. Radulovic, D. Zivanovic, D. Ruiz, B. de Supinski, et. al., "Another Trip to the Wall: How Much Will Stacked DRAM Benefit HPC?," In *Proceedings of the 2015 International Symposium on Memory Systems* (MEMSYS '15). ACM, New York, NY, USA, 31-36.

[2] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 2.1". Technical Report, http://www.hybridmemorycube.org/files/SiteDownloads/. December, 2015.

[3] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification Version 1.0". Technical Report, http://www.hybridmemorycube.org/specification-download/. January, 2013.

[4] J. Leidel and Y. Chen, "HMC-Sim: A Simulation Framework for Hybrid Memory Cube Devices," in *Proc. Workshop on Large Scale Parallel Processing*. IEEE International Symposium on Parallel & Distributed processing, Workshops and PhD Forum, 2014.

[5] J. Leidel and Y. Chen, "HMC-SIM: A Simulation Framework for Hybrid Memory Cube Devices," In *Journal of Parallel Processing Letters*, December 2014.

[6] P. Rosenfeld, "Performance Exploration of the Hybrid Memory Cube," Ph.D. dissertation, Dept. of Electrical Engineering, Univ. of Maryland, MD, 2014.

[7] P. Rosenfeld, E. Cooper-Balis, T. Farrell, D. Resnick and B. Jacob, "Peering over the memory wall: Design space and performance analysis of the Hybrid Memory Cube." Technical Report. University of Maryland Systems and Computer Architecture Group Technical Report UMD-SCA-2012-10-01. October 2012.

[8] A. Patel, F. Afram, S. Chen and K. Ghose, "MARSS: A Full System Simulator for Multicore x86 CPUS," In *Proceedings of the 48th Design Automation Conferece* (DAC '11). ACM, New York, NY, USA, 1050-1055.

[9] G. Kim, J. Kim, J.H. Ahn and J. Kim, "Memory-centric System Interconnect Design with Hybrid Memory Cubes," In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 145-156.

[10] K. Nai and H. Kim, "Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversal," In *Proceedings of the 2015 International Symposium on Memory Systems* (MEMSYS '15). CAM, New York, NY, USA. 258-261.

[11] J. McCalpin, "Memory Bandwidth and Machine Balance in High Performance Computers," In *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995, pages 19-25.

[12] P. Luszczek, J. Dongarra, D. Koester, et. al. "Introduction to the HPC Challenge Benchmark Suite," Technical Report, 2005.