Parallel Computing 68 (2017) 77–88

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

HMC-Sim-2.0: A co-design infrastructure for exploring custom memory cube operations

John D. Leidel*, Yong Chen

Department of Computer Science, Texas Tech University, Box 43104, Lubbock, Texas, 79409-3104 USA

A R T I C L E I N F O

Article history:
Received 13 October 2016
Revised 20 April 2017
Accepted 24 July 2017
Available online 25 July 2017

Keywords:
Hybrid memory cube
Simulation
3D stacking
Barrier synchronization

A B S T R A C T

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. These augmented memory subsystems stand to change the landscape of high performance computing algorithm optimization.

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. The goal of this approach is to provide computer architects the ability to experiment with augmentations to future memory devices in the scope of co-designing the future of scalable high performance computing instruments. 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. In addition, we provide a sample series of CMC operations that implement near-memory mutexes and demonstrate their efficacy using central locking and barrier synchronization algorithms traditionally found in parallel programming models and runtime libraries.

1. 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

* Corresponding author.
E-mail addresses: john.leidel@ttu.edu, john.leidel@gmail.com (J.D. Leidel), yong.chen@ttu.edu (Y. Chen).

http://dx.doi.org/10.1016/j.parco.2017.07.008
0167-8191/© 2017 Elsevier B.V. All rights reserved.

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. The goal of this research is to dramatically improve the performance of traditionally performance sensitive operations in high performance parallel algorithms and applications.

One of the major difficulties in research associated with PIM operations is deciding which operations are most advantageous for eventual production devices. This co-design of the memory device often evolves to a balance of augmented hardware components and complex, user-accessible intrinsic operations. 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 2 introduces previous work associated with HMC-Sim and other stack memory simulation infrastructures. Section 3 presents the introductory architecture and implementation details of HMC-Sim as well as the relative differences with previous versions. Section 4 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 5 provides an examples of building three CMC operations that could potentially replace traditional mutex operations commonly found in Linux/UNIX systems. Further, we demonstrate traditionally performance sensitive central locking and barrier synchronization benchmarks scaled from 2 to 100 concurrent threads. We conclude with a summary of this work and potential future development.

2. 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 [2]. 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 [3] and presents additional data regarding random memory access kernel performance [4].

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 [5]. 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 [6]. 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 [7,8]. The initial HMC-Sim infrastructure contained support for the entire version 1.0 HMC Specification [9], 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

Table 1
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

order to elicit the efficacy of resolving data from the simulation infrastructure [7]. The second set of results presented actual kernel execution data resolved from executing a STREAM Triad kernel [10] and an HPCC RandomAccess kernel [11] 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). Finally, we have formerly described the seminal work behind the HMC-Sim 2.0 infrastructure, including a full description of the supported commands and potential for future experimentation [12].

3. 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 [13]. 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.

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 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.

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.

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 1). 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.

4. Custom memory cube architecture

4.1. 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 an additional 70 commands via 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. While users can define an unlimited number of CMC operations, only 70 can be in use at any given time.
- *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.

4.2. 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 Fig. 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
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

The separate structures are interconnected via a series of explicitly crafted function interfaces. These function interfaces are identical in each respective implementation of a CMC operation. In order to separate the implementations for each potential CMC operation, we utilize function pointers in the HMC-Sim core library initially set them to NULL. When a user application or driver requests to load a CMC operation, the HMC-Sim core library utilizes the Linux/UNIX dynamic loader (libdl) to resolve the correct addresses within the target CMC library implementations for the necessary functions. At which time, the function pointers are initialized and available for use during the normal processing of HMC commands. In this manner, we enable users to load up to seventy concurrent CMC operations without collisions between the individual operations. The separation of core library functionality and user-provided CMC libraries is depicted in Fig. 1.

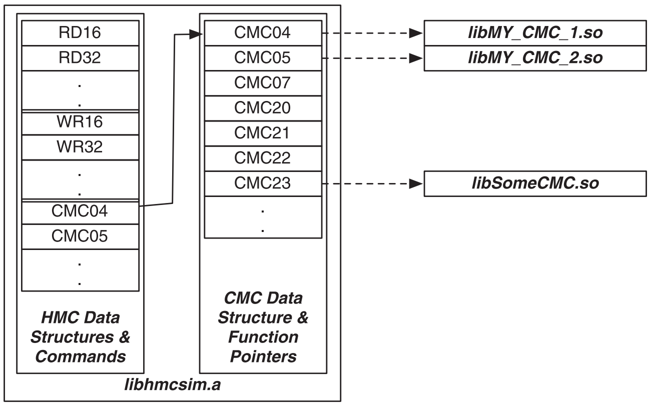


Fig. 1. CMC functional overview.

4.3. 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.

4.3.1. 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.

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. 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.

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.

4.3.2. 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 4.3.1. 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 4.3.1.

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, 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 [7,8]. 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()*.

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.

4.4. 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 [12]. This includes a set of static global variables and basic functions that implement the necessary data and logic behind each unique CMC operation. We provide templates for users to quickly and easily adapt their own CMC operation implementations.

In addition to the core CMC operation templates, we provide several addition CMC operations in the HMC-Sim 2.0 source tree. These sample CMC operations are designed to showcase the heterogeneous or *processing near memory* exploration capabilities of the aforementioned infrastructure. We describe the additional operations as follows:

- *amo_popcount*: Performs an atomic population count of an 8-byte memory location.
- *ClrXX*: Performs a full-empty style memory and tag clear [14].
- *IncFF*: Performs a full-empty style increment if full and remain full [14].
- *ReadEF*: Performs a full-empty style memory read if empty and set to full [14].
- *ReadFE*: Performs a full-empty style memory read if full and set to empty [14].
- *ReadFF*: Performs a full-empty style memory read if full and remain full [14].
- *ReadXX*: Performs a full-empty style memory exclusive read [14].
- *WriteEF*: Performs a full-empty style memory write if empty and set to full [14].
- *WriteFF*: Performs a full-empty style memory write if full and remain full [14].
- *WriteXE*: Performs a full-empty style memory write exclusive and set to empty [14].
- *WriteXF*: Performs a full-empty style memory write exclusive and set to full [14].

5. CMC simulation

5.1. 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.

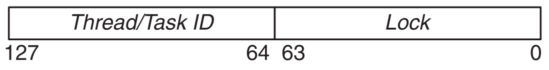


Fig. 2. HMC mutex data structure.

Table 2
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

- *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 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 (Fig. 2). 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 2.

5.2. 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:

The second algorithm utilized in our testing is design to simulate a traditional barrier synchronization method utilized in parallel programming models and runtime libraries. The algorithm utilizes a two-stage, binary sensed central barrier approach that includes an arrival and departure phase [15]. The initial state, or *SENSE*, is detected by each thread upon its arrival and the sense is updated prior to the full departure. Updating the sense includes changing its binary value to the complement. This alleviates the traditional *ABA* problem in barrier synchronization. The arrival phase includes obtaining the central lock using the HMC_LOCK CMC operation and updating the number of arrived threads. The departure phase is centered upon the last arriving thread updating the sense value for the entire team. The goal of this algorithm is to elicit the potential performance of the mutex CMC operations when executing a barrier synchronization operation for a multi-core host CPU. We further describe the barrier synchronization algorithm in Algorithm 2.

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
```

Algorithm 2 CMC Mutex Sensed Central Barrier Algorithm.

```
for Nthreads do
   ORIG = SENSE
   HMC_LOCK(ADDR)
   while !LOCK_SUCCESS do
      HMC_LOCK(ADDR)
   end while
   UPDATE NUM_ARRIVED
   HMC_UNLOCK(ADDR)
   if NUM_ARRIVED == Nthreads then
      UPDATE_SENSE
   else
      SPINWAIT(SENSE != ORIG)
      LOCAL = SENSE
      while LOCAL == ORIG do
         LOCAL == SENSE
      end while
   end if
end for
```

We implemented and executed the aforementioned algorithms 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 each of the respective configurations. From each simulation, we recorded three data values (in addition to the core HMC-Sim tracing). The *MIN_CYCLE* describes the minimum number of cycles required for any of the threads to perform the algorithm. The *MAX_CYCLE* describes the maximum number of cycles required for any of the threads to perform the algorithm and the *AVG_CYCLE* metric describes the average number of cycles required for all the respective threads for the given simulation to perform the algorithm.

5.3. Simulation results

5.3.1. Central lock algorithm

After executing the aforementioned simulations, we find that the 4Link and 8Link HMC devices delivered very similar performance for the central locking algorithm. 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 queuing structures begin to induce slightly lower minimum cycle timings. The larger number of injection queue slots in the 8 link device configurations permit a larger number of outstanding requests before stalling the host, hence the lower minimum

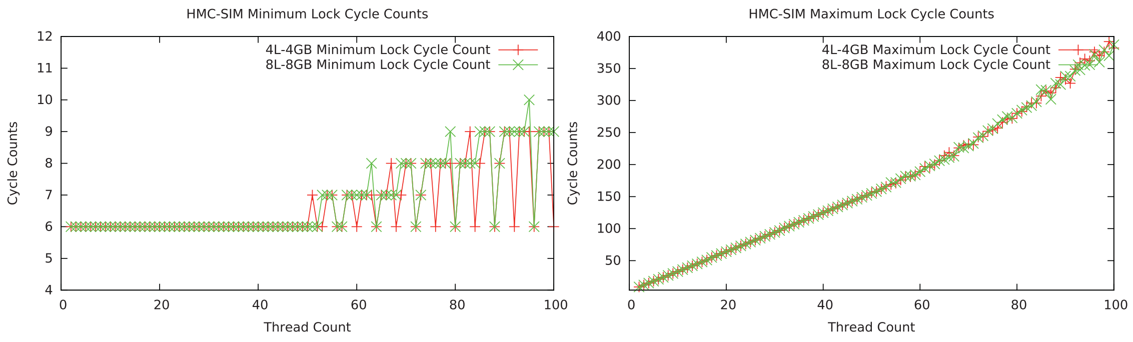


Fig. 3. Minimum and maximum lock cycles.

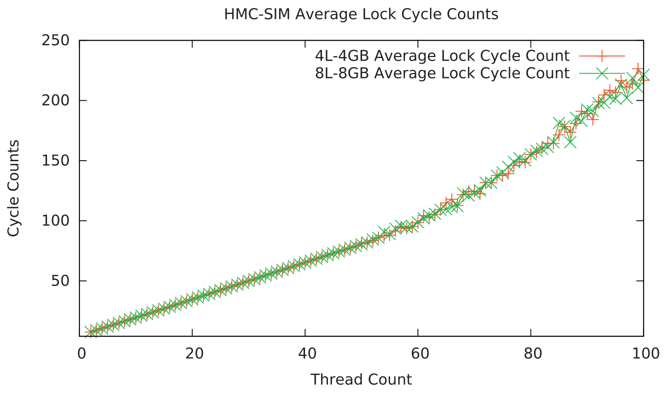


Fig. 4. Average lock cycles.

Table 3
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

lock cycles beyond fifty threads. Fig. 3 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 Fig. 3. 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 Fig. 4. 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 3.

5.3.2. Central barrier algorithm

In a similar manner elicited in the previous central locking results, the central barrier results indicate that 4Link and 8Link devices deliver rather similar performance. This can again be attributed to the similarity in queuing structure for both configurations as well as the natural hot-spotting induced during the lock acquisition entry phase.

However, unlike the central locking approach, we see very little perturbation in results for nearly all the tests. As we see in Fig. 5, the 4Link and 8Link results are nearly identical for all thread counts from 2 to 98 threads. The respective configurations share identical minimum cycle counts of 38 cycles and their respective average cycle counts only differ by 1.492%. Table 4 contains a full reference of the minimum, maximum and average cycle timings across the full suite of barrier simulations.

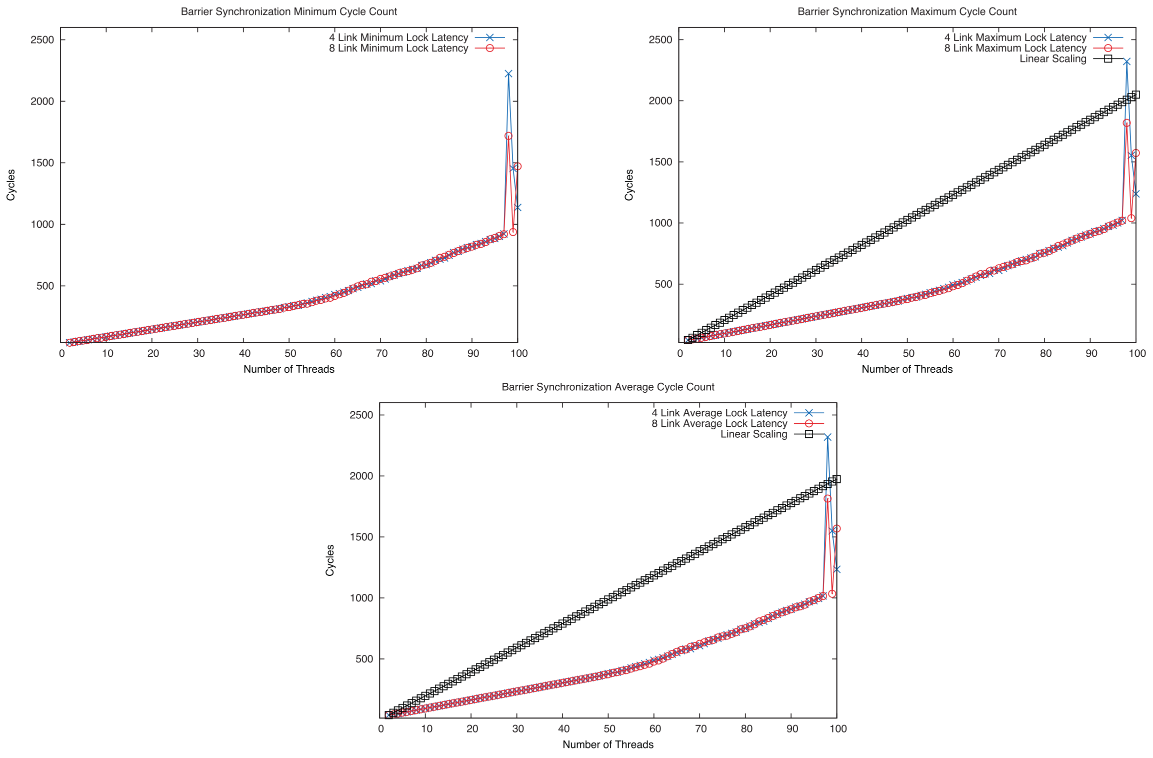


Fig. 5. Minimum, maximum and average barrier cycles.

Table 4
CMC mutex sensed central barrier operations.

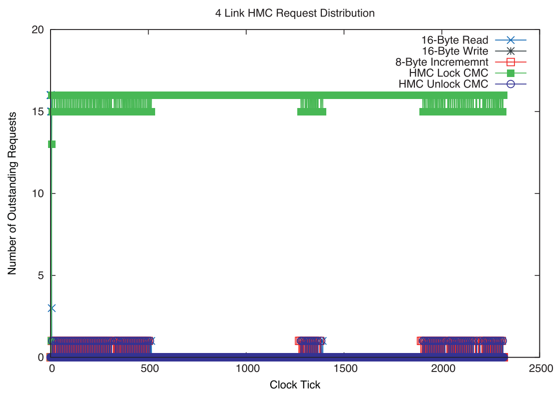
Device	*Min Cycle Count*	*Max Cycle Count*	*Avg Cycle Count*
4Link-4GB	38	2322	476.978
8Link-8GB	38	1819	469.857

As we see in Fig. 5, the perturbation in performance is only realized when examining the final two simulation runs. Both the 4Link and 8Link simulations experience a significant reduction in performance when executing with 98 and 99 threads, respectively. We see in the Maximum Cycle Count and Average Cycle Count graphs from Fig. 5 that the 4Link performance is sub-linear. The 4Link performance is actually 16.537% slower than the equivalent linear performance at 98 threads. Conversely, the 8Link performance is 6.219% faster than the equivalent linear performance at 98 threads. The resulting maximum cycle count, as shown in Table 4, shows us that the 4Link devices are 21.66% slower at their worst or maximum cycle timings.

This idiosyncratic behavior for the 4Link and 8Link devices at larger thread counts warranted additional investigation. The utilized in the embedded logging features in HMC-Sim [7,8] to perform a full analysis of the status of each device configuration for each cycle of execution for our 98 thread simulation. When enabled, the logging features recorded the status of each outstanding request and each internal component of the HMC device.

The first value that we examined was the number of bank conflicts present at each cycle. Given the homogeneous nature of our simulations, we experienced no bank conflicts at any cycle count for each of the 4Link and 8Link configurations. We further examined the number of stall signals produced when routing requests across the crossbar to its destination vault. We found that the crossbar request stalls were minimal for both the 4Link and 8Link devices. We did experience some degree of crossbar latency, or latency induced by routing requests across the device to a distant vault. However, this was somewhat expected given that the simulated HMC physical layer injects requests in a round-robin manner across all active links. As a result, we experience good injection bandwidth at the expense of higher on-package routing latency. This latency is well within the means of traditional latency hiding techniques found in high performance computing application development.

Finally, we examined a full cross section of all the injected requests for each cycle of the simulation. Fig. 6 depicts these results organized by request type. Notice that we encapsulate standard HMC I/O requests with our custom HMC Lock and Unlock requests in the results. However, we can clearly see that the HMC Lock requests dominate the outstanding memory traffic by a significant margin. When we map these results back to the original algorithm, we find that a significant number of threads will require multiple attempts in order to acquire the initial arrival lock. Each thread will only require a single



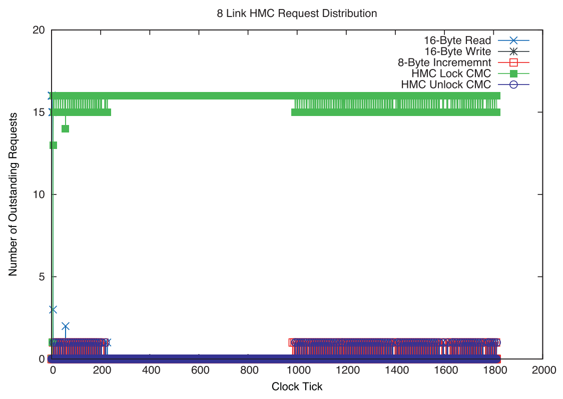


Fig. 6. 4 and 8 Link HMC request distribution.

update to the *NUM_ARRIVED* global value and a small number of traditional read requests on the final sense value. This coalescing of requests to a single lock value at scale begins to induce a significant serialization effect in the overall runtime of the algorithm. The end result being that we have sufficient device queue state and bandwidth to service the number of outstanding requests, but due to the inherent singularity of the lock, we effectively serialize the algorithm at scale. In effect, the algorithm becomes bound by the initial arrival lock. We can therefore conclude that executing scalable barrier synchronization methods on HMC devices with a large number of threads will require an approach with distributed locking such as the static tree barrier technique [14].

6. 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 CMC functionality, HMC-Sim has the ability to simulate user-defined processing near memory, or *PIM*, operations alongside the augmented set of Gen2atomic 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 executing a simple central locking algorithm as well as a scalable barrier synchronization algorithm. The goal being to model traditional performance sensitive memory operations commonly found in parallel programming models and runtime libraries.

Despite the inherent hot spotting that occurs with a central 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. Further, we find that for scalable barrier synchronization approaches, we must adopt distributed locking techniques in order to promote better scalability beyond 98 threads of execution. However, for both algorithms, we find that our mutex CMC operations provide total algorithmic performance in excess of the equivalent linear scale.

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. Further, these CMC operations can be evaluated in the scope of traditional performance or latency sensitive algorithms and operations found in traditional high performance computing applications and runtime libraries such that we may promote co-design and development for the future HPC memory system architectures.

7. 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.

In addition to the aforementioned power modeling additions, we would also like to begin experimenting with more complex HPC-related CMC operations. We would like to begin experimenting with CMC operations that stand to significantly increase the performance characteristics of certain pathological HPC operations, such as communication primitives [5], reductions and partitioned global memory address mapping.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.parco.2017.07.008.

References

[1] M. Radulovic, D. Zivanovic, D. Ruiz, B.R. de Supinski, S.A. McKee, P. Radojković, E. Ayguadé, Another trip to the wall: How much will stacked dram benefit hpc? in: Proceedings of the 2015 International Symposium on Memory Systems, in: MEMSYS '15, ACM, New York, NY, USA, 2015, pp. 31–36, doi:10.1145/2818950.2818955.

[2] P. Rosenfeld, Performance Exploration of the Hybrid Memory Cube, Dept. of Electrical Engineering, Univ. of Maryland, 2014 Ph.D. thesis.

[3] A. Patel, F. Afram, S. Chen, K. Ghose, Marss: A full system simulator for multicore x86 cpus, in: Proceedings of the 48th Design Automation Conference, in: DAC '11, ACM, New York, NY, USA, 2011, pp. 1050–1055, doi:10.1145/2024724.2024954.

[4] P. Rosenfeld, E. Cooper-Balis, T. Farrell, D. Resnick, B. Jacob, Peering over the memory wall: design space and performance analysis of the Hybrid Memory Cube, Technical Report, University of Maryland, 2012.

[5] G. Kim, J. Kim, J.H. Ahn, J. Kim, Memory-centric system interconnect design with hybrid memory cubes, in: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, in: PACT '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 145–156. http://dl.acm.org/citation.cfm?id=2523721.2523744.

[6] L. Nai, H. Kim, Instruction offloading with hmc 2.0 standard: a case study for graph traversals, in: Proceedings of the 2015 International Symposium on Memory Systems, in: MEMSYS '15, ACM, New York, NY, USA, 2015, pp. 258–261, doi:10.1145/2818950.2818982.

[7] J.D. Leidel, Y. Chen, Hmc-sim: a simulation framework for hybrid memory cube devices, in: Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, in: IPDPSW '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 1465–1474, doi:10.1109/IPDPSW.2014.164.

[8] J.D. Leidel, Y. Chen, Hmc-sim: a simulation framework for hybrid memory cube devices, Parallel Process. Lett. 24 (04) (2014) 1442002, doi:10.1142/S012962641442002X.

[9] H.M.C. Consortium, Hybrid memory cube specification 1.0, 2013.

[10] J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, IEEE Tech. Committee Comput. Archit. (TCCA) Newslett. (1995).

[11] P. Luszczek, J.J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi, Introduction to the HPC Challenge Benchmark Suite, Technical Report, 2005.

[12] J. Leidel, Y. Chen, Hmc-sim-2.0: a simulation platform for exploring custom memory cube operations, the proceedings of the Sixth International Workshop on Accelerators and Hybrid Exascale Systems (ASHES-2016), 2016.

[13] H.M.C. Consortium, Hybrid memory cube specification 2.1, 2015.

[14] J.D. Leidel, Y. Chen, Exploring tag-bit memory operations in hybrid memory cubes, in: Proceedings of the Second Internaitonal Symposium on Memory Systems, in: MemSys '16, ACM, Alexandria, VA, USA, 2016, pp. 153–163. http://dx.doi.org/10.1145/2989081.2989105.

[15] O. Villa, G. Palermo, C. Silvano, Efficiency and scalability of barrier synchronization on noc based many-core architectures, in: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, in: CASES '08, ACM, New York, NY, USA, 2008, pp. 81–90, doi:10.1145/1450095.1450110.