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

John D. Leidel, Yong Chen

May 23, 2016

AsHES 2016
Overview

- Introduction & Overview
- CMC Simulation
- Sample CMC Mutexes
- Future Research
INTRODUCTION & OVERVIEW

Hybrid Memory Cube Device Simulation
GC64 Driving Research

• Driving force behind the GC64 architecture research is the ability to find and exploit memory bandwidth

• Exhaustive search on forthcoming memory technologies
  • Traditional DDR/GDDR devices did not provide sufficient accessibility and bandwidth

• Hybrid Memory Cube devices were chosen

http://gc64.org
Intro to Hybrid Memory Cube

• **Technology**
  - Through-silicon-via [TSV] design that combines logic layer and DRAM layers
  - Packetized interface specification the behaves similar to a network device
  - Routing capabilities built into the device logic layer
    • *Device-to-device routing*

• **Hybrid Memory Cube Consortium**
  - Standards body to drive the public HMC specification.
  - Similar in function to JEDEC for DDR memory
HMC TSV Technology

- **Substrate**
  - Contains the physical pin-out for data, power and ground
  - SERDES

- **Logic Layer**
  - Contains the logic necessary to perform:
    - Routing
    - Arbitration (weakly ordered)
    - Addressing
    - AMO

- **DRAM Layers**
  - Contains the DRAM arrays

Our architecture research required access to a configurable HMC simulation platform
  • None existed that were: 1) open source and/or 2) available without an NDA

We exhaustively studied the HMC specification and developed HMC-Sim based upon the spec
  • ...as opposed to a individual device SKU

**HMC-Sim Design Requirements**
  • Configurable for different host CPUs (link connectivity, clock frequency, packet configuration, etc)
  • Configuration for different device SKU’s
  • Support for device-to-device routing
  • Simulation of all the internal queuing arbitration stages as defined by the spec
  • Cycle-based simulation
  • Discrete logging capabilities
  • Packaged as a library (can be integrated into other high-level simulators)
HMC-Sim 1.0

- Developed the first open source HMC simulation platform
  - Designed to explore how different applications affect memory throughput & latency
  - Becoming the standard for HMC modeling and simulation
- Permits us to model different concurrency mechanisms to determine the best mixture of parallelism and bandwidth across different algorithms and applications
HMC-Sim 2.0

• Several users of HMC-Sim requested a number of new features in future revisions:
  • Support for Gen2 HMC specification
  • Gen2 specification’s inclusive support for atomic memory operations
  • Gen2 packet specification
  • *Custom Memory Cube (CMC) exploration*

• CMC Exploration
  • What if we could implement new operations in the HMC logic layer?
  • What if these operations were *NOT* just simple memory operations?
  • *Additional Atomic operations, transactional operations, arithmetic reductions, logical reductions, processing near memory, etc*
  • *If we could have any operation embedded in the HMC logic layer, what would it be?*
Custom Memory Cube Operation Simulation

CMC SIMULATION
CMC Support Requirements

- **API Compatibility:**
  - Existing integration with other simulators shouldn’t be broken (Sandia SST)
- **External Implementation:**
  - CMC implementer should focus on CMC, not learning HMC-Sim internals
- **Creative Experimentation**
  - No limitation to the user’s creativity in implementing CMC ops
- **Utilize Existing HMC Packet Formatting**
  - Existing crack/decode logic should be maintained

- **Discrete Tracing**
  - HMC-Sim 1.0 had extensive support for logging, CMC ops will need this as well
- **Separable Implementation**
  - Current HMC-Sim is BSD licensed. We want to make sure users can develop/distribute their CMC ideas separate from the simulator
- **No Simulation Perturbation**
  - No perturbation to existing simulation results!
CMC Support Architecture

• We explicitly map all the unused HMC opcodes to CMC* ops
  • 70 potential CMC opcodes
• We provide a template infrastructure to construct a single CMC operation mapped to a single opcode in a shared library
• We provide one additional API interface to load the CMC shared library at runtime
• Runtime processing is otherwise the same for CMC operations!
The CMC library requires the user to define structure of the CMC operation:

- **CMC Name (string)**: used for logging
- **Request command enum** (from the list of 70)
- **Request & Response packet lengths**
- **Response command enum** (can be custom response)

One function must be implemented by the user:

- **hmcsim_execute_cmc()**

**Everything else is provided in our example CMC implementation**

```c
#include <stdio.h>

typedef struct {
    char* cmc_name;
    int request_len;
    int response_len;
    int request_cmd;
    int response_cmd;
    int request_active;
    int response_active;
    void (*execute)(void*, uint32_t, uint32_t, uint32_t, uint32_t, uint32_t, uint32_t);
} cmc_t;

int main() {
    cmc_t cmc = {"CMC01", 100, 200, 1, 2, 1, hmcsim_execute_cmc};
    return 0;
}
```

**CMC Tutorial:**

http://gc64.org/?page_id=140
extern int hmcsim_load_cmc( struct hmcsim_t *hmc, char *cmc );

Is HMC-Sim Initialized?

Yes

Begin Registering CMC Library

Initiate Dynamic Loader
dlopen( char *cmc, RTLD_NOW)

Shared Lib Loaded?

No

Register CMC Function Pointers
dlsym(handle,FUNC)

Execute Registration Function
int (*cmc_register)(hmc_rqst_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 *);

return error

return success

Save Data to hmc_cmc_t Structure
CMC Processing

HMC Vault Request Queue

extern int hmcsim_process_rqst(...)

Decode Packet Header & Tail

Find Available Response Queue Slot

Available?

Yes

Examine the request command code

Yes

CMC Command?

Yes

Process CMC Command

No

Is CMC Command Active?

No

Retrieve Execution Function Pointer

struct hmc_cmc_t cmc[]

Process CMC Command

No

Is CMC Command Active?

Yes

Response Required?

Yes

Register Response

No

Process Normal HMC Command

Execute CMC Command Using Function Pointer

return success

return error
Locking Primitives as CMC Operations

CMC MUTEXES
CMC Mutexes

- We implemented several CMC commands as initial tests
- What if we could accelerate traditional mutex operations?
  - `HMC_LOCK`
  - `HMC_TRYLOCK`
  - `HMC_UNLOCK`
- Designed to perform pthread-style mutex operations
  - **does not block on HMC_LOCK**

<table>
<thead>
<tr>
<th>Thread/Task ID</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>63</td>
</tr>
<tr>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

- Each HMC mutex payload is a 16-byte memory location
- Lower 8 bytes: LOCK region
- Upper 8 bytes: Thread/Task ID
  - “Owner” of the LOCK region
  - Relative to the user’s process space
- 16-bytes is wasteful... but
  - 16-bytes in the minimum request size for normal HMC RD/WR requests
  - Minimal logic overhead required to implement our mutexes
CMC Mutex Implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pseudocode</th>
<th>Command Enum</th>
<th>Request Command</th>
<th>Request Length</th>
<th>Response Command</th>
<th>Response Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>hmc_lock</td>
<td>IF (ADDR[63:0] == 0) { ADDR[127:64] = TID; ADDR[63:0] = 1; RET 1 } ELSE { RET 0 }</td>
<td>CMC125</td>
<td>125</td>
<td>2 FLITS</td>
<td>WR_RS</td>
<td>2</td>
</tr>
</tbody>
</table>
| hmc_trylock| IF ( ADDR[63:0] == 0) { ADDR[127:64] = TID; ADDR[63:0] = 1; RET }
|            | ELSE { 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               |

**HMC_LOCK**
if( LOCK == 0 ){
    TID = MY_TID;
    LOCK = 1;
    return 1;
}else{
    return 0;
}

**HMC_TRYLOCK**
if( LOCK == 0 ){
    TID = MY_TID;
    LOCK = 1;
    return TID;
}else{
    return TID;
}

**HMC_UNLOCK**
if( TID == MY_TID && LOCK == 1 ){
    LOCK = 0;
    return 1;
}else{
    return 0;
}
CMC Mutex Experimentation

- Attempt to perform naïve spin-wait locks on a single mutex location
- Deliberate hot-spotting
- Scale the number of parallel threads/tasks from 2-100
- Execute the tests for different HMC configurations
  - 4LINK-4GB
  - 8LINK-8GB
- Record:
  - \textit{Min\_Cycle}: Minimum number of cycles for any thread to obtain the lock
  - \textit{Max\_Cycle}: Maximum number of cycles for any thread to obtain the lock
  - \textit{Avg\_Cycle}: Average number of cycles for all threads to obtain the lock

\begin{algorithm}
\begin{algorithmic}
\For{$N_{\text{threads}}$}
  \State HMC\_LOCK(ADDR)
  \If{LOCK\_SUCCESS}
    \State HMC\_UNLOCK(ADDR)
  \Else
    \State HMC\_TRYLOCK(ADDR)
    \While{LOCK\_FAILED}
      \State HMC\_TRYLOCK(ADDR)
    \EndWhile
    \State HMC\_UNLOCK(ADDR)
  \EndIf
\EndFor
\end{algorithmic}
\caption{CMC Mutex Algorithm}
\end{algorithm}

Figure 5. Minimum Lock Cycles

Table VI

<table>
<thead>
<tr>
<th>Device</th>
<th>Min Cycle Count</th>
<th>Max Cycle Count</th>
<th>Avg Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4Link-4GB</td>
<td>6</td>
<td>392</td>
<td>226.48</td>
</tr>
<tr>
<td>8Link-8GB</td>
<td>6</td>
<td>387</td>
<td>221.48</td>
</tr>
</tbody>
</table>

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 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.
CMC Mutex Min and Max Cycle Results

- Cycle counts are in HMC logic cycles (not host cycles)
- 4LINK-4GB device has slightly higher maximum latency
- Identical minimum latencies

### Table VI

<table>
<thead>
<tr>
<th>Device</th>
<th>Min Cycle Count</th>
<th>Max Cycle Count</th>
<th>Avg Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4Link-4GB</td>
<td>6</td>
<td>392</td>
<td>226.48</td>
</tr>
<tr>
<td>8Link-8GB</td>
<td>6</td>
<td>387</td>
<td>221.48</td>
</tr>
</tbody>
</table>

**HMC-SIM Minimum Lock Cycle Counts**

**HMC-SIM Maximum Lock Cycle Counts**


CMC Mutex Average Cycle Results

- 8LINK-8GB device has slightly lower average and maximum latencies
- For latency-sensitive applications dependent upon primitive locking operations (embedded applications), the additional queuing capacity with more links is helpful
- The weak ordering of the HMC device promotes sub-linear scaling for both device configurations!
Additional Possibilities in CMC Exploration

FUTURE RESEARCH
What other common operations would be interesting to simulate as CMC operations?

Currently packaged with HMC-Sim:

- Atomic Popcount
- HMC Lock
- HMC Trylock
- HMC Unlock
- HMC Full Empty Bit Ops**

Other Interesting Operations:

- Reductions
- Sorting
- Bitwise Atomics
- Processing Near Memory
Simulating Fine-Grained Locking Primitives:

- Similar to MTA/XMT style full-empty (tag) bit operations
- Performs **read-modify-write** on lock bits and data payloads with a single command
- Splits the storage in the HMC array into tag bit vectors and data payloads for better concurrency
- Supports full complement of tag-bit operations
- *Publication accepted for MemSys 2016*
Questions

John Leidel
john.leidel@ttu.edu

Yong Chen
yong.chen@ttu.edu

HMC-Sim Development and Tutorials:
http://gc64.org