Analyzing Consistency Issues in HMC Atomics

Pranith Kumar
Georgia Institute of Technology
pranith@gatech.edu

Lifeng Nai
Georgia Institute of Technology
lnai3@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
hyesoon@cc.gatech.edu

ABSTRACT

As 3D stacked technology gets popular, Processing-in-memory (PIM) is gaining momentum. HMC 2.0 specification offers a fine-grained, instruction granularity offloading capability to the host processor. The current work studies the potential consistency issues which arise from offloading the atomic instructions from CPU to HMC as present in the current specification.

1. INTRODUCTION

Utilizing 3D-stacked memory technology, high-performance memory systems are in active development. This not only increases the memory bandwidth and performance, but also includes computing capabilities. HMC is one example which has introduced the capability to offload certain computations from the host processor to the memory system [1]. These HMC instructions can atomically read-modify-write inside the memory system. The question arises about how HMC atomic instructions will affect the processor consistency. In current systems, the processor ensures the consistency guarantees and the memory system is not involved. However, when memory starts to perform computations directly, it is likely to violates these guarantees if not carefully designed.

CPU[1] atomic operations have high overhead to provide processors' consistency semantics[3]. However, if utilizing HMCâĂŹs atomic operations has similar overhead as processor atomic instructions, the motivation to use them will be reduced. There has been no discussion on the processor side implementation of HMC's atomic operations. Since with instruction offloading, loads and stores are all performed in the memory directly (by either operating on a non-cached memory location or invalidating the data in cache), at a first glance, it seems like HMC atomic instructions are free from any consistency violations. However, there are no studies supporting this.

[1]Here, CPU means an architecture that implements x86 ISA.

MEMSYS 2016 October 3–6, 2016, Washington, DC, USA

DOI: 10.1145/ 2989081.2989104

In this paper, we show the consistency violations that will happen by considering HMC atomic instructions as a regular store instruction on the processor side. The paper will also present the scenarios where the consistency violations are not likely to occur.

2. BACKGROUND

HMC is composed of vaults, each of which contains multiple banks and a single logic layer which is managed by a vault controller. Each vault controller may have a queue that is used to buffer references to that vault's memory. The vault controller can execute references in that queue in any order. The only ordering guarantee provided is that references from a single serial I/O link to the same vault are executed in arrival order. Requests from different serial links to the same vault/bank address are not guaranteed to be executed in a specific order and must be managed by the host controller.

Let us consider two different architectures, in each of which we discuss how the host controller needs to handle consistency.

Architecture 1: There is no coherence support between HMC and the host. HMC atomics operate only on non-cached memory regions. We need to allocate memory in non-cacheable memory region and execute PIM atomic instructions to access that memory address.

Architecture 2: Coherence support for HMC memory regions. Bringing the cache line into local cache is useful in applications with high locality of reference. Updates to the memory still happen only using PIM commands. The CPU should broadcast an invalidate message for the cache line before sending the HMC command.

2.1 CPU Atomic Operations

Executing atomic instructions on the CPU involves costly operations [3]. On x86 processors, some instructions will be atomic when using the *lock* prefix. All such instructions also guarantee to ensure sequential consistency (SC) of the memory accesses. Employing techniques like draining the write buffer, disabling ILP and locking the target cache line on scheduling an atomic instruction help in achieving this consistency guarantee. Most of the x86 atomic instructions can functionally map directly to the available HMC atomic operations.

ARMv7 processors use LL/SC instructions to implement their atomic operations. The main difference with x86 is that these instructions do not guarantee any consistency. We need to use additional memory barrier instructions to ensure consistency. Mapping such atomic instructions from

ARMv7 to HMC commands is not straightforward.

2.2 HMC Atomic Operations

HMC atomic commands execute in three steps: reading data from DRAM, performing an operation on the data in the logic die, and then writing back the result to the same DRAM location. These steps occur atomically; the corresponding DRAM bank is locked during the atomic request so that no other requests to the same bank can be interleaved. Besides, all commands include only one memory operand. Memory requests that are operating on different banks can execute in any order to maximize the memory bank parallelism. Furthermore, if memory requests to the same address are issued through different serial I/O links that connect between HMC and processors, the order between these two memory requests is also not preserved [1].

3. HMC OPERATIONS FROM CPUS

The question that we address in this paper is when HMC atomic instructions are executed from a processor, should the processor treat them as just like other atomic instructions in CPUs or regular store or even load instructions? Treating them as atomic instructions means that the processor has to drain the write buffer before executing HMC atomic instructions. However, if the processor treats them as regular stores, multiple HMC atomic instructions will be issued to the memory system in-order regardless of whether the previous store instructions completed. So multiple stores and HMC atomics can be concurrently updating the memory.

3.1 Example of a Consistency Violation

If the processor treats the stores as regular stores even when the processor sends HMC atomic commands in-order, since HMC can execute commands to different addresses in any order, there is a possibility of a consistency violation. Consider the independent-reads of independent-writes (IRIW) scenario where initially A and B are 0. Using lock instructions on CPU ensures that the increment to A is ordered before the increment to B. If the increment to A and B from CPU 0 are performed using HMC atomics, and CPU 1 is able to read values of 0 and 1 for A and B respectively, it violates SC.

Table 1: CPU atomic instructions offloaded to HMC

CPU 0	CPU 1	CPU 0	CPU 1
lock inc A, 1;	read B;	HINC8(A);	HRead(B);
lock inc B, 1;	read A;	HINC8(B);	HRead(A);

In order to offload such atomic instructions without consistency violations, we need the following: First, the CPU should receive a response from the HMC controller when an HMC atomic operation is completed such as HINC8(A) in this example. Second, The CPU should drain the write buffer and wait without scheduling any further instructions until responses for all the previous issued HMC commands are received. The drawback in this approach is that the CPU needs to wait for the write buffer drain and completion of all HMC atomic instructions. If there are series of HMC atomic instructions in the code, these will all be serialized and on top of that, they will have a long latency waiting to get a response from the memory controller.

3.2 Atomicity vs. Consistency

Table 2: Operations of typical graph applications

Program Phases	Operation
loop:	
foreach vertex in task queue:	
read property	/* HMC-Read */
fetch neighbor list	
foreach neighbor:	
update neighbor property	/* HMC-atomic */
update next-iter task queue	
barrier	

There are also applications which only require atomicity, not strict SC when using atomic operations. For example, most graph computing applications need to perform both regular read and atomic operations on graph property in an iterative way. However, as shown in Table 2, the read commands and atomic operations happen at different execution phases. These phases are separate and hence this naturally avoids consistency issues. Here, we assume that the barrier operation guarantees that all previous HMC atomic instructions are complete. Besides, graph computing applications, as well as most machine learning applications, are iterative convergent and can converge to the same result even if the consistency of intermediate values is relaxed [2]. Therefore, the consistency concerns of HMC atomic operations are less applicable for such applications.

4. FUTURE WORK

Hence, to reduce the HMC atomic serialization execution overhead, it is important to identify situations when both consistency and atomicity are required and when only atomicity is sufficient. Also, hardware support to provide consistency with low overhead is important. An explicit serialization operation similar to the memory fence operation or enforcing a wait for responses for all issued HMC atomic instructions are examples of such support. From a programmer's perspective, the serialization instruction is similar to existing memory barrier instructions which ensures ordering. The programmer or compiler can utilize this serialization instruction in cases where ordering needs to be explicitly ensured. Our future work will investigate such software and hardware optimizations.

5. ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of National Science Foundation (NSF) XPS-1533767.

6. REFERENCES

[1] H. M. C. Consortium. Hybrid memory cube specification 2.0. 2014.

[2] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS '13*. 2013.

[3] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. ACM, Oct. 2015. Proceedings of the 24th International Conference on Parallel Architectures and Compilation (PACT'15).