Exploring Tag-Bit Memory Operations in Hybrid Memory Cubes

John D. Leidel
Department of Computer Science
Whitacre College of Engineering
Box 43104
Lubbock, Texas 79409-3104
john.leidel@ttu.edu

Yong Chen
Department of Computer Science
Whitacre College of Engineering
Box 43104
Lubbock, Texas 79409-3104
yong.chen@ttu.edu

ABSTRACT

The recent advances in multi-dimensional or *stacked* memory devices have led to a significant resurgence in research and effort associated with exploring more expressive memory operations in order to improve application throughput. The goal of these efforts is to provide memory operations in the logic layer of a stacked device that provide pseudo processing near memory capabilities to reduce the bandwidth required to perform common operations across concurrent applications.

One such area of concern in applications is the ability to provide high performance, low latency mutexes and associated barrier synchronization techniques. Previous attempts at performing cache-based mutex optimization and tiered barrier synchronization provide some degree of application speedup, but still induce sub-optimal scenarios such as cache line contention and large degrees of message traffic. However, several previous architectures have presented techniques that extend the core physical address storage with additional, more expressive bit storage in order to provide fine-grained concurrency mechanisms in hardware.

This work presents a novel methodology and associated implementation for providing in-situ extended memory operations in an HMC Gen2 device. The methodology provides a single lock, or *tag* bit for every 64-bit word in memory using the in-situ storage. Further, we present an address inversion technique that enables the tag-bit operations to execute their respective read-arbitrate-commit operations concurrently with a statistically low collision between the tag-bit storage and the data storage. We conclude this work with results from utilizing the commands to perform a traditional multi-threaded mutex algorithm as well as a multi-threaded static tree barrier that exhibit sub-linear scaling.

CCS Concepts

•Hardware → Memory and dense storage; •Computing methodologies → *Modeling methodologies;*

MemSys '16 October 3–6, 2016, Alexandria, VA, USA

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

Keywords

Memory architecture; concurrency; barrier synchronization; tag-bits; full-empty bits; 3D memory; Hybrid Memory Cube

1. INTRODUCTION

Recent advances in memory architecture and CMOS manufacturing have yielded a new class of multi-dimensional memory device that stands to change the landscape of bandwidth, latency and functionality with respect to core processing. These new memory architectures, such as the Hybrid Memory Cube (HMC) [6], couple multiple layers of data storage constructed using traditional DRAM layers on a DRAM manufacturing process with a logic or controller layer that is manufactured on a traditional ASIC process.

One of the unique features of the aforementioned HMC device architecture is the non-JEDEC protocol stack utilized to perform I/O between a core processor and the logic layer of the device. This protocol is more analogous to a traditional network protocol as opposed to a traditional DDR-style JEDEC protocol. This new protocol stack has begun to drive a resurgence of research associated with more expressive memory operations behind simple read and write I/O [15] [10] [13] [9] [19] [18] [4]. This processing in or near memory [8] [17], 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.

In this work, we present an extension to the current HMC Gen2 specification that enables additional commands and data storage to perform extended or *tag-bit* operations in-situ within an HMC device. These tag-bits provide an additional tag or lock bit for each 64-bit word in an HMC device. We provide a set of ten additional HMC commands that are encoded using unused opcodes in the current HMC specification that perform common *read-arbitrate-commit* operations that consider the tag bits and the associated data storage within a single operation.

These aforementioned tag-bit operations are implemented using a novel *address inversion* methodology that stores the extended bit payloads separate from the respective data payloads in order to maintain memory concurrency within the target device. This methodology enables the implementation to exist without changes to the core HMC physical addressing structure, packet structure or arithmetic logic functionality. We present an implementation of the ten operations using the HMC-2.0 custom memory cube (CMC)

functionality [13] and demonstrate two common algorithms that represent traditional mutex operations and barrier synchronization.

The remainder of this work is organized as follows. Section 2 presents previous work in system architectures supporting extended-bit memory hierarchies. Section 3 presents an overview of tag-bit memory theory as it relates to promoting fine grained concurrency. Section 4 presents our proposed tag-bit implementation within the HMC specification. Section 5 presents an evaluation of our approach using several common synchronization algorithms. We complete the work with our conclusions and potential future work.

2. PREVIOUS WORK

Several previous architectures have utilized similar approaches to annotating physical memory storage with additional information. The original Tera MTA architecture [2] [1] combined barrel processors with two memory tags and two trap tags in order to optimize the latency hiding capabilities of the system at scale. Rather than focusing on single-bit lock tags, the MTA also included an additional tag to signal whether the associated memory location contains an address (pointer) or a data value. The result was a highly efficient hardware methodology to efficiently execute algorithms such as graph searching and pointer chasing.

The core architecture presented in the MTA was later adapted into the Cray XMT/XMT2 platform [11] [3]. The XMT system architecture includes a similar barrel processor that consists of 128 hardware streams, each of which is permitted to have a single instruction in the pipeline at any given time. The XMT also includes traditional full-empty lock bits associated with each 64-bit word in memory. The end result was a tightly coupled processor and memory subsystem with unique latency hiding capabilities. The XMT series of platforms, however, require a unique programming environment and tool chain in order to optimize machine-resident code.

The J-Machine developed at MIT had a very unique method of extended bit addressing implemented as a part of its on-chip memory [7] [20]. The J-Machine's *Message Driver Processor*, or *MDP*, included a 4K by 36-bit static memory integrated on chip. The processor also included a separate, 256K by 36-bit off-chip memory. The unique nature of the memory subsystem was annotated in the 4 additional tag-bits within each 36-bit word. Rather than representing mutex values, the tag-bits marked whether the respective word included data types such as booleans, integers and user-defined types (objects). Further, two bits were reserved for *future* and *cfuture* values. These bits would cause a trap when accessed. The system was programmed using either Concurrent Smalltalk or Message-Driven C.

The latest system architecture to include extended bits in the physical memory was the Convey MX-100 platform [14] [12]. The MX-100 was a heterogeneous platform that included an FPGA-based coprocessor attached to an Intel x86_64 Xeon host via PCI-Express. The Convey platform implemented full-empty lock bits for each 64-bit word in memory utilizing standard DDR3 DRAMs. Special instructions were utilized to access the contents of the tag-bit and the data payload concurrently with a single request. The special memory instructions were made available from both the coprocessor and the x86_64 host platform. The coprocessor contained native memory instructions via the CHOMP instruction set architecture. The host system utilized special PCI-Express command payloads that were exposed via compiler instructions. The result was a heterogeneous platform that had the ability to perform fine-grained operations concurrently within the coprocessor and the host.

3. TAG-BITS IN APPLICATIONS

As shown in Section 2, several previous architectures have demonstrated implementations of extended bit annotation in physical memory subsystems. However, the aforementioned implementations were utilized in different application or algorithmic constructs. We classify these approaches into one of *memory analysis*, *mutex analysis* and *message analysis*.

The memory analysis classification of algorithms is most commonly associated with the Tera MTA system architecture [21]. The pointer or *forward* bit present in the extended memory storage is used to evaluate whether the data in the associated physical memory storage represents a 47-bit memory address. When the forwarding bit is enabled, the CPU attempts to retry the memory operation using the contents of the data pointer as the next target memory operation. This extended metadata support within the hardware alleviates the requirement for the application to explicitly store separate data to support values *and* pointers, thus saving the physical memory storage required to do so. Moreover, this forward pointer analysis permits efficient execution of algorithms that are required to recursively evaluate pointers. This is a common technique found in graph traversal. In this manner, a single *forward pointer* operation has the ability to derive the final destination of nested sets of connected components.

In addition to the MTA's forward pointer analysis techniques, we also find the tag-bits found in the XMT [11] and MX-100 [12] architectures useful for similar graph analysis techniques. For each of the aforementioned architectures, the physical memory storage includes an additional tag bit used to signal whether the memory location is *full* or *empty*. This extended duality of state can be utilized for graph algorithms that require simple, binary visitation. Consider an algorithm that performs a *breadth-first-search* of a graph. Traversing the graph can be simplified into a set of read operations of subsequent connected nodes and evaluating state of the tag-bits. The XMT and MX-100 provide native instructions where the goal is to read a target memory address and return the data if and only if the tag-bit is empty (*ReadEF*). If the operation is successful, the tag-bit is set of full. In which case, the entire traversal can be structured as a recursive set of these *ReadEF* instructions.

The mutex analysis classification of algorithms is the most common use of the extended tag-bits. In this use case, the extended bit is utilized as a single mutex or *lock* value in order to determine whether the respective data value is locked or unlocked. The XMT architecture implements native instructions that naturally pend in the memory subsystem until the lock value is available. This is analogous to the traditional blocking *pthread_mutex_lock* operation. When the arbitration of the lock fails, the operation pends in the memory subsystem in a queue until it can become a successful event. Conversely, the MX-100 platform implements a methodology analogous to the traditional non-blocking *pthread_mutex_trylock*. When a tag-bit operation is requested and the arbitration of the lock fails, the operation returns

an unsuccessful status value. As such, the MX-100 requires that the application programmatically handles the return value of the tag-bit operation.

Finally, the message analysis classification is likely the least common use of the extended physical address bits. The J-Machine implemented a set of two tag-bits that represent slots for values that have not yet been computed [16]. If a target application thread attempts to read a memory value before it has been supplied, the processor issues a trap in order to force the thread to suspend and wait for the value to arrive. The arrival of the data value in the memory slot clears the pending thread to resume execution. The J-Machine also supported a more expressive type, termed a *future*, that could be annotated programmatically and thus copied between data values and used as return values from function calls without causing a trap. As such, the future tag provided the ability to write algorithms that executed across a global memory namespace where the algorithm could selectively pend on fine-grained, concurrent data values.

4. TAG-BIT ARCHITECTURE

4.1 Requirements

Given that the current HMC 2.0 specification has been made available as a standard, we were very cognizant of the impact of the potential tag-bit CMC operations on the HMC packet structure and the core HMC logic layer. As such, we define a core set of requirements for the implementation in order to avoid perturbing the current HMC device architecture as follows:

- *Utilize existing HMC packet structure*:

  In order to avoid perturbing the current device architecture, we sought to maintain the existing packet infrastructure as is outlined in the HMC-Sim 2.0. This requirement permits us to maintain the efficacy and stability of the simulation of traditional memory commands alongside the annotated CMC commands for tag-bits. It also permits a potential implementor to utilize our methodology with a minimal impact to current production devices.

- *Utilize existing HMC logic structure*:

  In addition to maintaining the existing packet structure, we also sought to maintain the current logic structure. While the logic layer is somewhat abstract with respect to the user and the packet interface, it contains basic ALU functions to operate the existing atomic memory operation commands. As such, we sought to utilize the in-situ logic rather than forcing potential implementors to further expand the functionality present in the HMC logic layer.

- *Utilize existing HMC DRAM structure*:

  Previous attempts to implement extended memory bits have made use of ECC bits or non-standard addressing modes in order to co-locate the extended bits with their respective data payloads. The former method often degrades the potential for a system architecture to detect and/or correct bit errors in the memory system. The latter often results in memory performance degradation. In order to avoid this, we sought to maintain the existing DRAM structure in the HMC specification. This permits the memory architecture to maintain throughput with the existing command structure.

- *Maintain consistency with existing HMC packets*:

  One of the primary concerns with implementing native full-empty memory operations is their existence alongside normal read and write operations. Full-empty operations often require additional logic or register stages in order to process the state of the lock and/or pointer bits. This often results in higher memory latency for normal memory operations. Further, in memory systems that maintain weak ordering, it is imperative that the logic pipeline designed to handle the extended bits adhere to the natural ordering primitives set forth by the architecture. The access to extended bits and their associated data bits must be atomic and operations such as *fences* must not interrupt the processing of the extended bits.

- *Maintain memory concurrency*:

  In addition to maintaining concurrency with existing memory requests, we also sought to maintain as much inherent memory concurrency as possible. Given that we sought to maintain the existing logic and DRAM architecture, with the existing data path, it would be difficult to co-locate extended bits alongside the respective data payload without multiple requests, thus inducing a performance degradation. As such, we sought to define the implementation such that the extended bits and the data payload could be read and written concurrently. In this manner, the memory system would maintain its inherent concurrency with the current physical hardware architecture.

4.2 Architecture

We utilize the HMC-Sim Hybrid Memory Cube simulation infrastructure as the basis for our tag-bit design and implementation. The 2.0 version of the HMC-Sim infrastructure [13] includes the ability to define custom, arbitrary operations beyond the base set of Gen2 HMC packet commands [6]. This functionality, termed *custom memory cube* or *CMC* functions, is exploited to implement our tag-bit operations. Each of the individual tag-bit operations is implemented as a separate CMC commands as per the HMC-Sim 2.0 CMC specification.

Each of the respective tag-bit operations is implemented in a similar manner per the flow diagram in Figure 1. Each of the commands, with the exception of the *ClrXX* command, is implemented as a normal memory operation attached to CMC opcodes. In this manner, each command has a request packet and a subsequent response packet. The command flow is also designed to support our concurrency requirements as outlined in Section 4.1.

Once the determination has been made that the command is, in fact, a tag-bit CMC command, we begin by decoding the target physical address. The target address is saved for the normal memory request and a second physical address, herein referred to as the *bit address*, is derived from the original physical address per the methodology outlined in Section 4.3. At this stage in the tag-bit operation, we split the single incoming request into two requests to represent the *bit address* and *data address*. Bit payloads are always

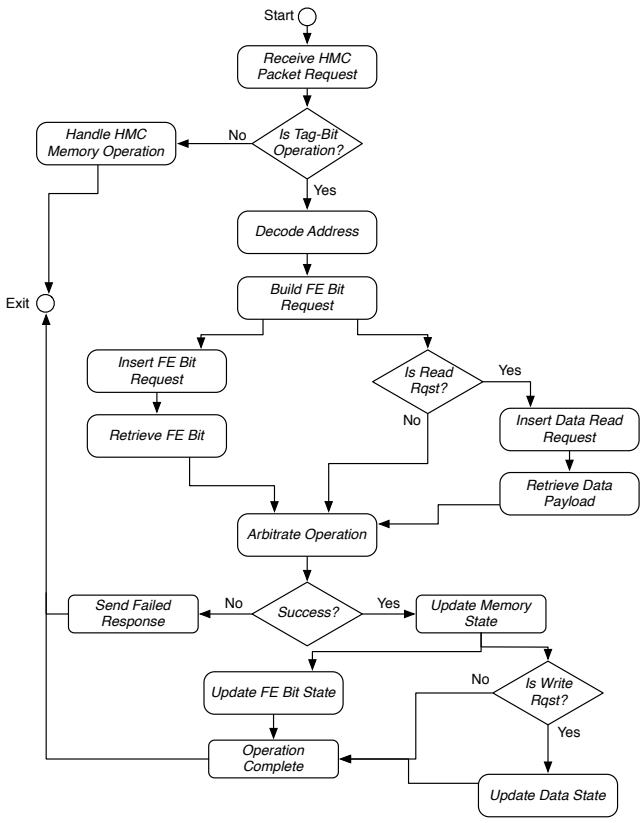

Figure 1: Full-Empty Request Processing

fetched regardless of the operation. At this stage, only tag-bit requests that desire to read the memory location execute a parallel memory read.

Once all the data from both potential memory fetch operations are complete, we arbitrate the state of the full-empty bit in order to determine whether the operation will be a success or failure. If the operation is targeted as a successful request, we update the value of the full-empty bit and the target data. Otherwise, we return an error state. The arbitration is a combination of the request type and the state of the bit. For example, if the operation requires that the bit be set (*full*), then an operation will become successful if the bit address contains a value of '1'; otherwise, the target physical memory will not be modified. Further, full-empty *exclusive* operations (designated with an X in the operation table) ignore the initial value of the tag-bit.

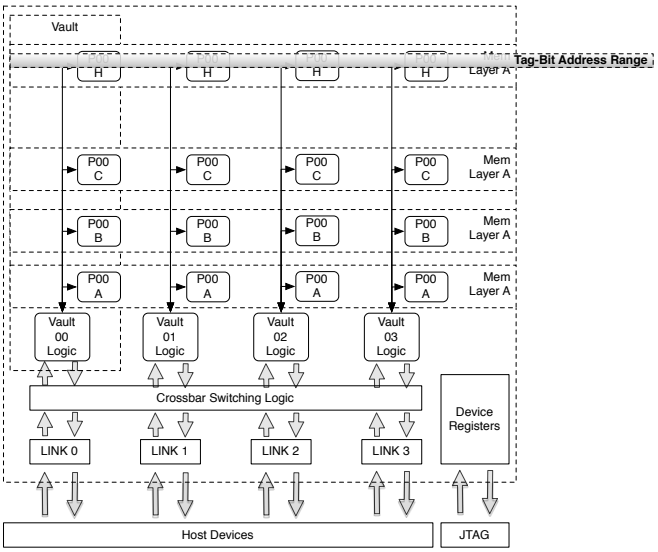

Figure 2: Tag-Bit Physical Address Range

4.3 Addressing

The key to maintaining concurrency within our tag-bit implementation is the ability to derive the location of a physical address's corresponding tag-bit. A single tag-bit CMC request will generate two subsequent memory operations corresponding to the tag-bit and the target data. Fetching both payloads from the same physical bank within the same physical vault would cause an inherent serialization of the requests. This serialization would result in significant performance degradation of the tag-bit request.

In order to avoid the aforementioned request conflict, we define a two-stage methodology that provides natural data and tag-bit addressing capabilities with a low statistical probability of address collisions between the target data payload and its respective tag-bit. The first stage of our methodology requires us to define and reserve the appropriate address range to store the tag-bits. As previously mentioned, we split the storage of the tag-bits into physical storage that is separate from the target data physical storage. This permits us to maintain the current HMC array design and provide the ability to issue concurrent memory requests to the various DRAM layers. We also restrict the tag-bit addressing to the maximum block sized addressing mode, or 256 byte blocks.

Also note that our tag-bit implementation requires a single mutex bit for every 64-bit word. As such, for an 8GB HMC device, the maximum number of mutex bits required is $8,589,934,592/8$ or 1,073,741,824 lock bits. This is a non-trivial amount of storage required to store all the necessary mutex bits. Given this and our requirement to maintain consistent concurrency within an individual tag-bit request, we reserve sufficient bits in the upper most physical addressing range to store all the mutex values. For 8GB devices, we reserve the upper most 134,217,728 bytes of the address range. For 4GB devices, we reserve the upper most 268,435,456 bytes of address range. This generally resides in the upper most DRAM layer of the respective device configuration as depicted in Figure 2.

The second stage of the methodology is the core of the

addressing logic. The goal of this stage of the tag-bit processing is to ingest a physical address request and generate a secondary physical address and associated bit offset for the corresponding tag-bit in a deterministic manner. This address *mangling* is explicitly designed to generate tag-bit addresses where the probability of collision with the corresponding target data address is low. We utilize a method, herein referred to as *address inversion*, to correlate specific byte addresses and bits within the respective bytes to target data addresses within the array. Inverting a target physical address in order to derive its corresponding bit and byte address can be described as follows:

1. *Normalize the Physical Address*: We shift the base physical address right by decimal 4 in order to normalize the address to 31 or 32 bits for 4GB and 8GB devices, respectively.
2. *Obtain Byte Offset*: We divide the normalized base address by 8 in order to obtain the necessary byte address from the end of the array.
3. *Derive the Target Tag-Bit Byte Address*: We utilize the offset from Step 2 and subtract it from the largest potential normalized address (*0xFFFFFFFF* for 4GB and *0x1FFFFFFFF* for 8GB devices, respectively). This becomes the tag-bit byte address.
4. *Derive the Target Tag-Bit Bit Address*: We again utilize the offset from Step 2 and perform a modulo by decimal 8 in order to determine the correct bit within the tag-bit byte address corresponds to our target address.
5. *Correct the Tag-Bit Address*: We have to correct the new tag-bit address by shifting it left 4 in order to conform with the HMC Gen2 physical addressing format.

An example of performing the aforementioned method on a candidate address (4GB device), *0x000000000005b5b0* is as follows:

- *Step 1*: 0x000000000005b5b0 >> 4 = 0x5b5b
- *Step 2*: Byte Offset = (0x5b5b / 8) = 0xb6b
- *Step 3*: Tag-Bit Address = (0xffffffff - 0xb6b) = 0xfffff494
- *Step 4*: Bit Address = (0xb6b % 8) = 3
- *Step 5*: Final Tag-Bit Address = Bit 3 at 0xfffff4940

Given the aforementioned addressing structure, we can determine where addressing conflicts occur. Addressing conflicts result in a target physical address sharing a vault and bank address with its respective tag-bit address, thus forcing a serialization of the subsequent requests. For 4GB devices, our methodology has a 0.392% probability that a requested physical address will induce a conflict with its respective tag-bit address. Similarly, for 8GB devices, our methodology has a 0.195% probability that a requested physical address will induce an addressing conflict. We summarize the probability for addressing conflicts in Table 1. We obtain this data by analyzing every potential target memory address, generating the complementary tag-bit address and determining whether the two addresses reside on the same vault and bank pair, thus resulting in an address conflict.

Table 1: Tag-Bit Addressing Conflict Probability

Device	*8 Byte Words*	*Address Conflicts*	*Probability of Conflict*
4GB	4026531832	1966080	1 in 255 or 0.392%
8GB	7516192760	1835008	1 in 511 or 0.195%

4.4 CMC Command Structure

Following a survey of previous attempts to implement full-empty style memory annotation, we chose the implementation found in the Convey MX-100 [12] as the basis for our HMC implementation. We design and implement a series of ten individual CMC commands that read and/or write memory and the associated tag-bit concurrently. Commands are given names based upon their core function as well as the current and future state of the tag-bit. We utilize a naming convention whereby we choose a base command name (*Command*) followed by two characters to designate the current (C) and successful (S) state of the tag-bit. As such, commands are designated in the form: *CommandCS*.

- E: Designates that the bit shall be empty
- F: Designates that the bit shall be full
- X: Designates that the bit is read, but unused. This state maintains *exclusivity* of the command with respect to other commands in flight.

In addition to defining the basic tag-bit functionality, we group our commands into four respective groups: commands that read memory, commands that write memory, commands that perform read-modify-write operations and posted commands. Our memory read commands include *ReadEF*, *ReadFE*, *ReadFF* and *ReadXX*. ReadXX is a special case that permits host processors to read memory values regardless of their respective mutex state. Our memory write commands include *WriteEF*, *WriteFF*, *WriteXE*, and *WriteXF*. Note that we do have a corresponding write command for ReadXX. It would be inherently dangerous to permit an application to preemptively write memory using a tag-bit operation without updating the tag-bit state. We include a single read-modify-write command, *IncFF*, that becomes very useful for atomic counters. Finally, we add a single posted command, *ClrXX* that permits an application to zero a memory location and the corresponding lock bit.

All non-posted commands include 16-byte response payload (*Response Length* in Table 2). As shown in Figure 3, the least significant 8-bytes includes the data payload from a read or read-modify-write operation. The data portion of the response is all zero for write operations. The most significant 8-bytes includes a value to signal the success (1) or failure (0) of the corresponding operation. The only command that does not return a response packet is the *ClrXX* command. This is a posted command that sets the target memory location to zero and sets the corresponding tag-bit to empty (0). This is equivalent in function to a *WriteXE* operation with no response.

In order to construct our full-empty command packets within the existing HMC Gen2 packet structure, we utilize a set of opcodes that are not currently assigned to standard

Table 2: CMC Full-Empty Operations

Operation	*Pseudocode*	*Command Enum*	*Request Command*	*Request Length*	*Response Command*	*Response Length*
IncFF	if (full) mem=old+data; full=1; ret(1,old) else ret(0,0x00ll)	CMC70	70	2 FLITS	WR_RS	2 FLITS
ReadEF	if (!full) full=1; ret(1,mem) else ret(0,0x00ll)	CMC71	71	1 FLIT	RD_RS	2 FLITS
ReadFE	if (full) full=0; ret(1,mem) else ret(0, 0x00ll)	CMC72	72	1 FLIT	RD_RS	2 FLITS
ReadFF	if (full) full=1; ret(1,mem) else ret(0, 0x00ll)	CMC73	73	1 FLIT	RD_RS	2 FLITS
ReadXX	ret(1, mem)	CMC74	74	1 FLIT	RD_RS	2 FLITS
WriteEF	if (!full) mem=data; full=1; ret(1,0x00ll) else ret(0, 0x00ll)	CMC75	75	2 FLITS	WR_RS	2 FLITS
WriteFF	if(full) mem=data; full=1; ret(1,0x00ll) else ret(0,0x00ll)	CMC76	76	2 FLITS	WR_RS	2 FLITS
WriteXE	mem = data; full = 0	CMC77	77	2 FLITS	WR_RS	2 FLITS
WriteXF	mem = data; full = 1	CMC78	78	2 FLITS	WR_RS	2 FLITS
ClrXX	mem=0; full=0	CMC85	85	1 FLIT	NONE	0 FLITS (POSTED)

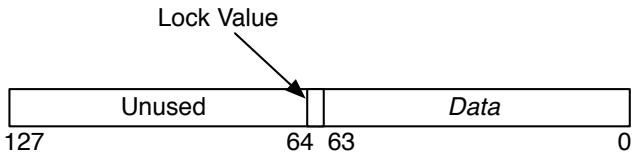

Figure 3: Full-Empty Response Payload

packet types. These opcodes are mapped using the CMC functionality present in the HCM-Sim 2.0 simulation infrastructure [13]. We summarize our full list of the commands and their respective CMC packet structure in Table 2.

5. TAG-BIT CMC EVALUATION

5.1 Synchronization Methods

We construct a series of parallel synchronization simulations to elicit the efficacy and utility of our tag-bit design and implementation. The first algorithm is designed to represent a mutex operation used to protect a critical section of code.

The algorithm utilizes two tag-bit CMC operations to protect the critical code section from interference. A central lock value is used as the target for the mutex. While this will undoubtedly induce a memory hot spot once the degree of parallelism reaches a sufficient level, this test serves to elicit the efficacy and simplicity of utilizing tag-bit operations within an HMC device. First, we initialize the mutex variable to a negative value. We then utilize the *ReadEF* command to obtain the lock from any given thread. A successful lock will return a successful return value as well as the current value in the mutex memory location (*ADDR*). Once a successful lock has been obtained, the respective thread will utilize a *WriteXE* and store its thread id (*TID*) to the mutex location and clear the lock bit. We present this approach in Algorithm 1.

Algorithm 1 Central Lock Algorithm

```
Init ADDR = -1
for Nthreads do
  ReadEF(ADDR)
  if LOCK_SUCCESS then
    WriteXE(ADDR,TID)
  else
    ReadEF(ADDR)
    while LOCK_FAILED do
      ReadEF(ADDR)
    end while
    WriteXE(ADDR,TID)
  end if
end for
```

The second algorithm is designed to mimc a traditional static tree barrier synchronization commonly found in shared and distributed memory programming models [5]. The threads are organized into a binary tree structure in order to minimize the lock contention on individual mutex variables. The goal of the hierarchical structure is to alleviate hot spots on hardware components such as memory buses and caching hierarchies, thus improving the overall scalability of the approach.

The algorithm is constructed using the standard *entry* and *exit* phases. In the entry phase, each of the leaf nodes signal their respective parents that they have entered the barrier and subsequently execute a spin-wait on the sense value. Once an individual parent acknowledges that all its respective leaves have arrived, it signals its parent. If a node is designated as a root node, it waits for all the children to arrive. Once all the child nodes arrive at the barrier, the root node begins the exit phase of the algorithm. The root node signals all the children to exit the barrier by changing to sense variable, thus completing the barrier. We provide a sample diagram using six candidate threads with the respective lock command states in Figure 4.

We utilize *ReadEF*, *ReadFF* and *IncFF* commands to

Algorithm 2 Static Tree Barrier Algorithm

```
MySense = ReadXX(Sense)
for Nthreads do
  if PARENT_NODE then
    Arrived = ReadEF(LocalLock)
    while Arrived != NUM_LEAVES do
      Arrived = ReadFF(LocalLock)
    end while
    if ROOT_NODE then
      if MySense==1 then
        WriteXE(Sense,0)
      else
        WriteXF(Sense,1)
      end if
    else
      Arrived = IncFF(ParentLock,1)
      while Arrived != Success do
        Arrived = IncFF(ParentLock,1)
      end while
      CheckSense = ReadXX(Sense)
      while CheckSense == MySense do
        CheckSense = ReadXX(Sense)
      end while
    end if
  else
    Arrived = IncFF(ParentLock,1)
    while Arrived != Success do
      Arrived = IncFF(ParentLock,1)
    end while
    CheckSense = ReadXX(Sense)
    while CheckSense == MySense do
      CheckSense = ReadXX(Sense)
    end while
  end if
end for
```

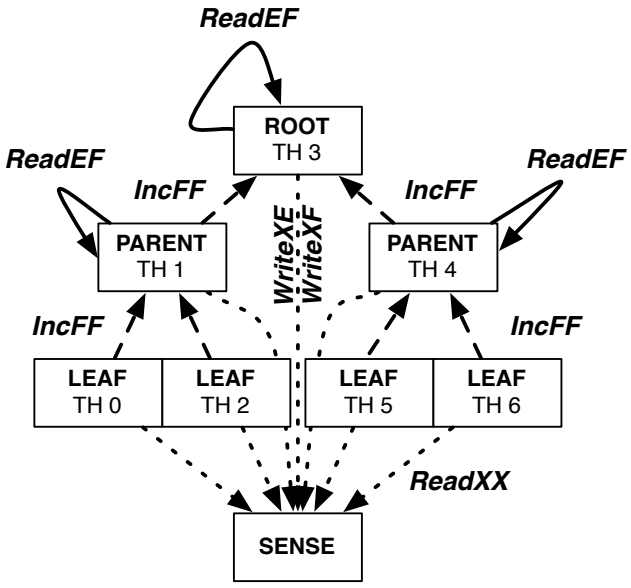

Figure 4: Six Thread Static Tree Barrier

manage the relationship between a child and its parent lock. In this manner, a child task cannot update the lock value of its parent until the parent arrives at the barrier. Similarly, we utilize *ReadXX* commands from each of the children to read the value of the sense variable. The root node utilizes a combination of *WriteXE* and *WriteXF* commands to switch the sense of the barrier during the exit phase. We describe the full algorithm in Algorithm 2.

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 in order to observe the scalability and the potential bottlenecks presented by different hardware configurations. For 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.

5.2 Synchronization Results

For each of the aforementioned algorithms, we measured the min, max and average latency to perform the respective operation. The results were scaled using thread counts from two to one hundred threads.

After executing the central lock algorithm on both 4GB and 8GB device configurations, we find that the resultant

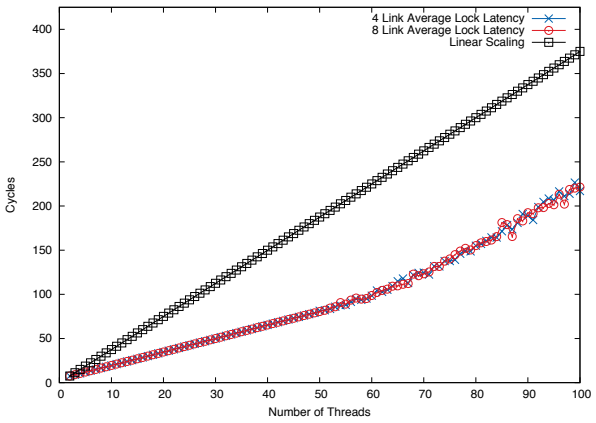

Figure 5: Mutex Average Lock Cycles

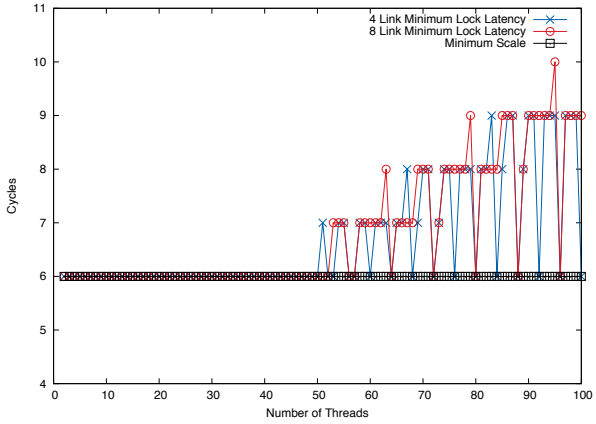

Figure 6: Mutex Minimum Lock Cycles

scaling curves are very similar. We find in both cases that the aggregate scaling of the approach performs better than the equivalent linear timing. The average lock cycle timing presented in Figure 5 indicates that the average timing at the largest thread count (100) is 42.4% better than the linear scale for the 4GB device and 41.06% better than the linear scale for the 8GB device. As such, we can determine that the weak ordering of the HMC device coupled with the tag-bit operations result in an efficient implementation for naive central locking mechanisms despite the inherent hot spotting that occurs with this approach.

Further analysis of the scaled results of the central locking algorithm reveals interesting details regarding the difference between the 4GB and 8GB device performance. As mentioned above, the 4GB device exhibits slightly better average performance at the largest thread counts. This is likely due to the larger number of queueing slots present in the 8GB device configuration, thus promoting more concurrency and more contention on the central lock variable. The minimum lock latency presented in Figure 6 reveals that the minimum latency to perform the lock is essentially static from 2 to 50 threads. The 4GB device begins to increase its minimum latency at 51 threads and the 8GB device at 53 threads, respectively. The most interesting value in this statistic is the largest minimum value. The 8GB device yields its worst minimum latency at 10 cycles and the 4GB device at 9 cycles. Architectures that are particularly sensitive to memory latency with a smaller degree of concurrency may elect to configure two 4GB devices rather than a single 8GB device in order to minimize latency.

Finally, we analyze the maximum scaling results from the central lock tests. Figure 7 presents the scaled results that exhibit the maximum number of cycles, or worst thread performance, required to complete the central lock. The 4GB device peaks at 99 threads with 392 cycles. However, the 8GB device peaks at 100 threads with 387 cycles. As such, we can determine that the 8GB device exhibits slightly lower maximum latency at scale. This implies that architectures with a high degree of concurrency may exhibit better performance with 8GB devices.

In addition to the analysis performed on the central locking algorithm, we also present results for the static tree barrier algorithm. The similarity between the scaling curves of the 4GB and 8GB results is also evident with the barrier results. We find that both device configurations perform significantly better than theoretical linear performance at scale. The average barrier timing presented in Figure 8 shows us that distributing the memory requests amongst across multiple address destinations serves to alleviate the hot spot contention we observed in the simple mutex algorithm above.

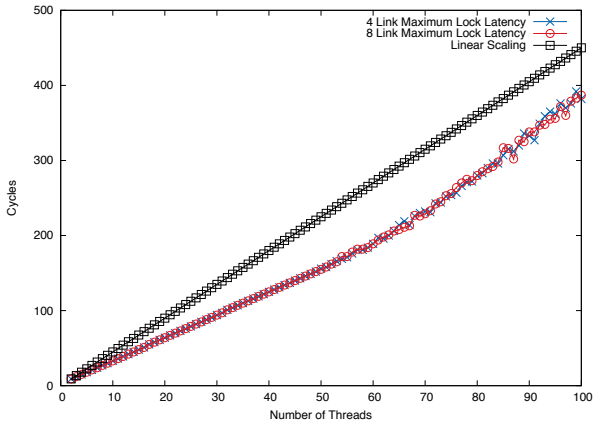

Figure 7: Mutex Maximum Lock Cycles

Further, the relative performance difference with respect to linear timing increased dramatically over the observed results with the mutex algorithm. The average cycle timing presented in Figure 8 and the maximum cycle timing presented in Figure 9 show the dramatic gap between the linear scale and the observed timing. Upon further analysis, we see that the 4GB device average timing is 93.2% faster than the linear timing at 100 threads. Similarly, the 8GB device average timing is 94.4% faster than the linear timing at the

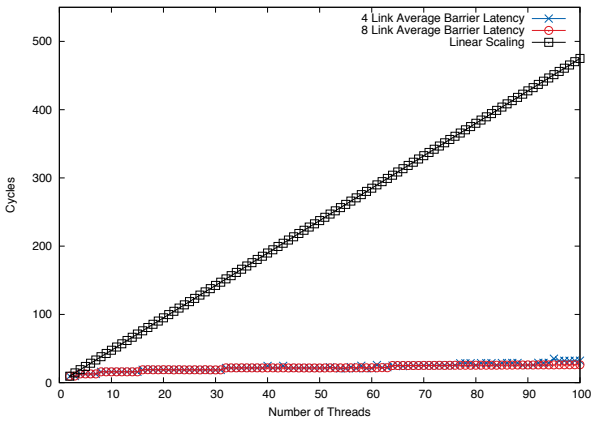

Figure 8: Barrier Average Lock Cycles

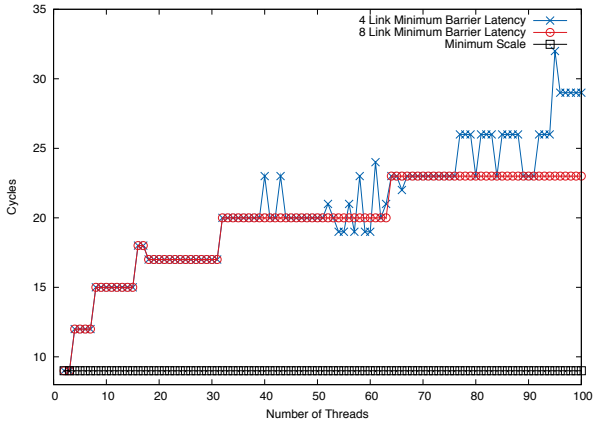

Figure 10: Barrier Minimum Lock Cycles

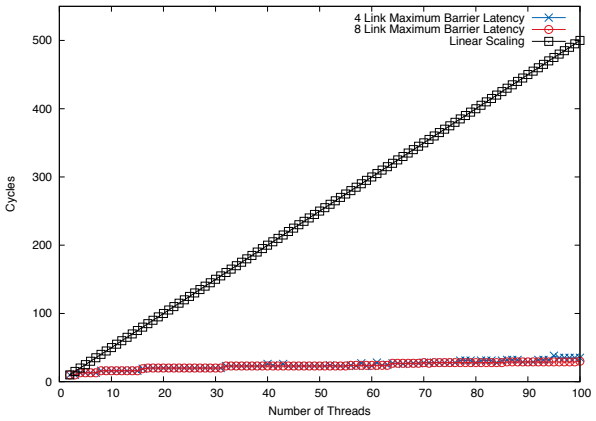

Figure 9: Barrier Maximum Lock Cycles

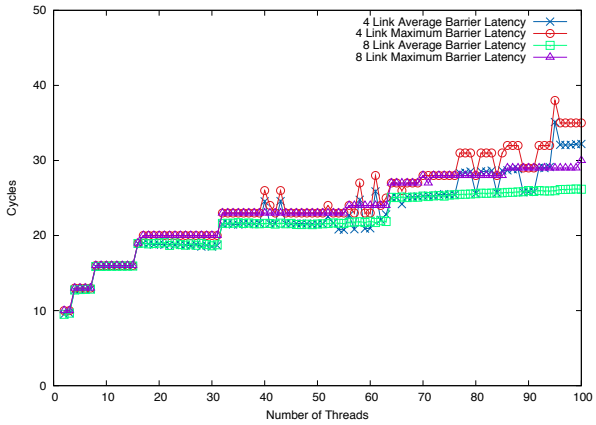

Figure 11: Barrier Scaled Cycle Comparison

same 100 thread scale, or 1.2% faster than the 4GB device. We also find that the maximum thread timing for each of the 4GB and 8GB results follows the same curve. The 4GB and 8GB devices are 93% and 94% faster than the linear timing results. Given this, we can conclude that the additional queuing capacity and link connectivity provided by the 8GB device results in roughly a 1% performance speedup over the smaller 4GB device configuration.

Despite their inherent similarity in scaled performance, the results from characterizing the minimum barrier cycle timings between the 4GB and 8GB devices showcase interesting behavior. First, we observe in Figure 10 that the minimum cycle performance for each device configuration very quickly deviates from the minimum cycle timing of four cycles. This is clearly due to the phased approach utilized in the static tree algorithm that requires multiple full-empty operations, even at small thread counts. The scaling curves between the two device configurations remains identical up to 40 concurrent threads. This is where the 4GB device results begin to deviate from the smooth step function presented in the 8GB results. The 4GB device exhibits a worst case minimum latency of 32 cycles at 95 threads, or 28.1% higher than the 23 maximum cycles recorded with the 8GB device.

Upon further analysis, we find that this is caused by a combination of link saturation and our link choice algorithm. The simulation utilizes a naive algorithm that chooses links via a round-robin method. Subsequent memory requests from threads may experience *HMC_STALL* signals given queue saturation on the target link. However, adjacent links may have available queue slots for requests. A more intelligent memory controller unit may provide a more appropriate approach that trades off request injection bandwidth for latency to route requests across the HMC device's crossbar interconnect.

Finally, we see in Figure 11 that the 8GB device clearly outperforms the 4GB device configuration. For those seeking to maximize concurrent performance of a potential sys-

tem architecture, the 8GB-8Link device is the clear choice. Further, we see that more care needs to be given with device configurations that utilize a small number of links in order to optimize the placement of requests such that the request queuing infrastructure present in the device is optimally utilized.

6. CONCLUSIONS

In this work, we present a novel methodology with an associated implementation for new tag-bit addressing and operations that reside entirely within the existing HMC Gen2 device specification. The methodology includes tag-bit addressing design and a novel addressing algorithm, termed *address inversion*, that enables the implementation to maintain memory concurrency within a single tag-bit request while fetching both tag-bit and its respective 64-bit data word. Using this methodology, we design and implement a series of ten tag-bit operations that provide read-arbitrate-commit functionality for reading, writing, querying and incrementing (read-modify-write) data locations while preserving consistent tag-bit memory state.

The implementation of the aforementioned functionality was performed using the HMC-Sim 2.0 infrastructure and its associated custom memory cube or *CMC* library functionality. Each of the ten individual tag-bit commands are implemented as a unique, CMC operation that resides within the existing HMC Gen2 packet specification. All of the CMC operations are defined using existing, unused opcodes and existing request and return types. Further, the implementation utilizes operations that are presumed to be resident in the ALU utilized for Gen2 atomic memory operations. In which case, the implementation minimizes the degree of disruption to the core HMC Gen2 device specification.

Finally, we present results associated with two common algorithms implemented using our CMC tag-bit commands. The first algorithm mimics a traditional central lock approach using a single mutex variable. We exhibit results from 4GB and 8GB devices that scale 40% better than the equivalent linear scale out to 100 concurrent threads.

The second algorithm for which we present results is designed to showcase a traditional barrier synchronization algorithm in the form of a static tree barrier. The barrier hierarchy is constructed using a binary tree structure in order to eliminate the hot spotting observed in the aforementioned mutex algorithm. The resulting data exhibits scaling for 4GB and 8GB device configurations that perform 90% better than the equivalent linear scale out to the same 100 concurrent threads.

In addition to the quantitative results presented in this work, we can also determine that additional care must be taken to construct memory pipelines and memory controllers that deterministically balance the request traffic from a core system on chip to one or more HMC devices in order to maximize the available queue slots on the device or devices. Despite this, the methodologies presented in this work clearly showcase the potential performance benefits for concurrent algorithms by adding extended or tag-bit memory operations to the HMC device specification.

7. FUTURE WORK

There has been significant interest from the current HMC-Sim user community to include relative power simulation capabilities in the various internal stages of the infrastructure. Future development of the internal HMC-Sim infrastructure will include the ability to derive relative peak power information from a given command request and response. The difficulty in providing such an interface is abstracting the definition of individual power values away from a specific HMC device implementation or SKU. As such, we want to provide the user the ability to adapt the infrastructure to simulate future, theoretical devices without any inherent constraints.

In addition to adding basic power metrics to the simulation infrastructure, we would like to perform additional algorithmic simulations using the tag-bit CMC operations included in this work. The algorithms demonstrated in this work include mutex operations and barrier synchronization methods. We would also like to perform simulations based upon common graph traversal and manipulation algorithms such as *breadth-first-search* and *coloring* to demonstrate fine-grained concurrency using techniques outside traditional course-grained synchronization.

8. ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation under grant CNS-1338078.

9. REFERENCES

[1] G. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry. Tera hardware-software cooperation. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–16, New York, NY, USA, 1997. ACM.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 122–127, New York, NY, USA, 2014. ACM.

[3] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experience with scientific programs on the cray mta-2. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 46–, New York, NY, USA, 2003. ACM.

[4] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. High performance axi-4.0 based interconnect for extensible smart memory cubes. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1317–1322, San Jose, CA, USA, 2015. EDA Consortium.

[5] J. Bradford and S. Abraham. Efficient synchronization for multithreaded processors, 1998.

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

[7] W. J. Dally, A. Chien, S. Fiske, W. Horwat, R. Lethin, M. Noakes, P. Nuth, E. Spertus, D. Wallach, D. S. Wills, A. Chang, and J. Keen. Retrospective: The j-machine. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 54–58, New York, NY, USA, 1998. ACM.

[8] M. Gokhale, S. Lloyd, and C. Hajas. Near memory data structure rearrangement. In *Proceedings of the 2015 International Symposium on Memory Systems*,

MEMSYS '15, pages 283–290, New York, NY, USA, 2015. ACM.

[9] M. Gokhale, S. Lloyd, and C. Macaraeg. Hybrid memory cube performance characterization on data-centric workloads. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, pages 7:1–7:8, New York, NY, USA, 2015. ACM.

[10] 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*, pages 145–155, Sept 2013.

[11] P. Konecny. Introducing the cray xmt, 2007.

[12] J. D. Leidel, J. Bolding, and G. Rogers. Toward a scalable heterogeneous runtime system for the convey mx architecture. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1597–1606, Washington, DC, USA, 2013. IEEE Computer Society.

[13] J. D. Leidel and Y. Chen. Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations. In *Proceedings of the 2016 IEEE 30th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '16, Washington, DC, USA, 2016. IEEE Computer Society.

[14] J. D. Leidel, K. Wadleigh, J. Bolding, T. Brewer, and D. Walker. Chomp: A framework and instruction set for latency tolerant, massively multithreaded processors. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 232–239, Washington, DC, USA, 2012. IEEE Computer Society.

[15] L. Nai and 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*, MEMSYS '15, pages 258–261, New York, NY, USA, 2015. ACM.

[16] M. D. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 224–235, New York, NY, USA, 1993. ACM.

[17] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[18] D. Resnick. Opportunities to upgrade main memory. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 55–59, New York, NY, USA, 2015. ACM.

[19] 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 UMD-SCA-2012-10-01, University of Maryland, 2012.

[20] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the j-machine and the cm-5. In L. N. Bhuyan and X. Zhang, editors, *Multiprocessor Performance Measurement and Evaluation*, pages 464–475. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[21] R. Vuduc and E. J. Riedy. Microbenchmarking the tera mta. Technical Report cs2 58-s99, University of California Berkeley, 1999.