Enabling Energy Efficient Hybrid Memory Cube Systems with Erasure Codes

Shibo Wang[1] Yanwei Song[2] Mahdi Nazm Bojnordi[2] Engin Ipek[1,2]
[1]Department of Computer Science
[2]Department of Electrical and Computer Engineering
University of Rochester
Rochester, NY 14627 USA
[1]{swang, ipek}@cs.rochester.edu [2]{yanwei, bojnordi}@ece.rochester.edu

Abstract

The Hybrid Memory Cube (HMC) is a promising alternative to DDRx memory due to its potential to achieve significantly higher bandwidth. However, the high static power of an HMC device compromises power efficiency when the device is lightly utilized. Activating a sleeping HMC takes over 2μs, which makes it challenging to manage HMC power without a substantial degradation in system performance.

We introduce a new technique that alleviates the long wake-up penalty of an HMC by employing erasure codes. Inaccessible data stored in a sleeping HMC module can be reconstructed by decoding related data retrieved from other active HMCs, rather than waiting for the sleeping HMC module to become active. This approach makes it possible to tolerate the latency penalty incurred when switching an HMC between active and sleep modes, thereby enabling a power-capped HMC system. Simulations show that the proposed architecture outperforms a current HMC-based multicore system by 6.2×, and reduces the system energy by 5.3× under the same power budget as the multicore baseline.

Keywords

HMC, erasure codes, energy efficiency

1. Introduction

The Hybrid Memory Cube (HMC) is a high bandwidth, energy-efficient memory interface that has the potential to replace the DDRx based DRAM modules in high performance systems [1]. HMC can provide an order of magnitude higher bandwidth than DDRx by leveraging narrow, high speed SerDes links operating at tens of GHz [2]. HMC modules have started to appear in servers and high performance computing systems, such as the Intel Knights Landing [3] and the FUJITSU next generation HPC systems [4]. HMC modules, however, consume significant static power (25.5W per node in the FUJITSU system[1] [4–6]). For an average processor power dissipation of 80W [4, 6, 7] and a 20% bandwidth utilization [8], the static power of the HMC subsystem accounts for 22% of the total system power.

The maximum allowable power consumption of high-performance systems is often constrained by power delivery, packaging, and cooling requirements. *Power capping* and *power shifting* are two synergistic techniques that aim at maximizing system performance under a power budget [9]. Power capping makes it possible to impose a maximum power budget on different subsystems such as the processor, the main memory, and the I/O devices [10–12]. It allows the power supply and the cooling subsystem to be provisioned for the average rather than the worst case, which significantly reduces cost. Power capping also provides the capability to upgrade a system (*e.g.*, increase the number of nodes, or hardware within a node) without the risk of demanding more power than the supply can provide, and to tolerate the failure of a subset of the power supply units. Many servers and high-performance systems (*e.g.*, IBM Power [13], FUJITSU PRIMERGY [14], and HP ProLiant [15]) adopt power capping and shifting techniques to improve reliability, availability, and cost.

The HMC subsystem is a promising target for runtime power management since (1) HMCs consume significantly lower static power in sleep mode as compared to active mode (*e.g.*, 5.6W vs. 25.5W in an eight-HMC system), and (2) the load on the memory system varies among different workloads and among different phases of a given workload. The current HMC standard provides the capability to transition an HMC into sleep mode, but requires μs latencies to switch between sleep and active modes. This high latency overhead makes it a significant challenge to manage HMC power without incurring a substantial performance degradation.

This paper examines a novel HMC power management solution based on erasure codes. The key idea is to encode multiple data blocks in a single *coding block* that is distributed among all of the HMC modules in the system, and storing the check bits in a dedicated, *always-on* HMC. The data stored in a sleeping HMC module can be reconstructed by decoding a subset of the remaining memory blocks that reside in the active HMCs, thereby avoiding the long power mode switching latency on the critical path of the application. A novel data selection policy is used to decide which data to encode at runtime, significantly increasing the probability of reconstructing otherwise inaccessible data. The coding procedure is optimized by leveraging the near memory computing capability of the HMC logic layer. The result is a power-capped architecture that achieves a 6.2× performance improvement and a 5.3× energy reduction over a current, power-capped HMC system.

Erasure coding requires extra storage space to store the check bits. Alternatively, that extra storage space could be used to cache parts of the working set that reside in the sleeping HMCs. This alternative approach would make it possible to service a memory request quickly by using the cached copies. The evaluation indicates that the proposed mechanism based on erasure codes makes more efficient use of the extra storage space than caching does, providing a higher probability of reconstructing inaccessible data under a fixed storage overhead. Specifically, compared to a power-capped HMC system that caches parts of the working set in an always-on HMC, the proposed architec-

[1]Each node directly connects a processor to eight single-link HMC modules.

ture improves the performance by 2.2× and reduces the energy consumption by 2.0×.

2. Background and Related Work

HMC takes advantage of three dimensional (3D) integration and SerDes links to achieve high bandwidth at the expense of high static power dissipation (Section **2.1.**). Erasure codes, discussed in Section **2.2.**, are leveraged to manage HMC power with a minimal impact on performance.

2.1. Hybrid Memory Cube

HMC relies on high-speed SerDes links to increase the bandwidth between the memory controller and the processor. However, related work [1,5] shows that a large fraction of the HMC power is due to the SerDes links. Since NULL FLITs have to be generated, scrambled, and transmitted when no other packets are pending [2], the SerDes link power in the active mode is constant at all times [16]. This static link power constitutes a large percentage of the total power when the memory is underutilized. The current HMC standard provides two low-power modes, *sleep* and *power down*. Both of them suffer from long wake-up latencies (*e.g.*, 2μs from the sleep mode, and hundreds of μs from the power down mode [2]). Due to the significant latency to exit the power down mode, only the sleep mode is considered in this paper.

Ahn *et al*. [17] design a mechanism to disable the HMC links according to the queuing delay observed at runtime. However, this approach requires maintaining connectivity to each HMC at all times. Under a power budget in which full connectivity cannot be maintained, or in a power-capped system in which each HMC module has a single link connected to the processor (such as the FUJITSU [4] system), the long power mode switching latency becomes difficult to hide.

Malladi *et al*. [18] introduce a DRAM interface modification to support low power modes with short wake-up latencies. This is a solution at the circuit level; however, it requires a non-trivial hardware change to the interface circuitry, and requires support from both the memory and the processor vendors. The proposed approach, which solves the power problem at the architecture level, requires no changes to the underlying link circuitry.

2.2. Erasure Codes

Erasure codes are a type of error correcting code (ECC) that are broadly applicable to large, distributed storage systems [19]. In contrast to SECDED codes, an erasure code is based on an *erasure channel* model in which data at a specific, known location is lost. Therefore, an erasure code has a higher (2×) correction capability as compared to an ECC based on a *binary symmetric channel* with the same amount of redundant data. Among all of the erasure codes, the Reed-Solomon (RS) code [20] is chosen to reduce the capacity overhead of the proposed HMC system. Figure 1 shows an example of the encoding and decoding procedures for a systematic (n, k) RS code [20]. A *distribution matrix* contains an identity matrix in the first k rows, and the rest of the rows are chosen such that any $k \times k$ sub-matrix is invertible. Encoding is accomplished by multiplying the distribution matrix by a source data vector. Since the codeword is 2^w-bits long (2^3-bits in the example), the data is partitioned into 2^w-bit codewords, and the matrix-vector multiplication is performed separately for each codeword. To recover the lost data, the corresponding rows are deleted from the distribution matrix, and the remaining rows are selected to form a $k \times k$ matrix. The generated matrix is inverted and multiplied by the surviving data vector that corresponds to the selected rows. Let $a_{i,j}$ ($i \in [0, n-k-1]$, $j \in [0, k-1]$) represent an element in the lower (non-identity) part of the distribution matrix. On a write, the new value of C_i is computed as follows [19]:

$$C_i' = C_i + a_{i,j}(B_j' - B_j), \tag{1}$$

where B_j' and C_j' respectively are the new values that replace B_j and C_j. The RS code requires Galois Field arithmetic, which makes a hardware encoder/decoder expensive to implement. The Cauchy Reed-Solomon (CRS) code, a variant of the RS code that converts all of the Galois Field operations to XORs, is employed to reduce the coding overhead [20]. Figure 2 shows the CRS version of the encoding example in Figure 1 (a).

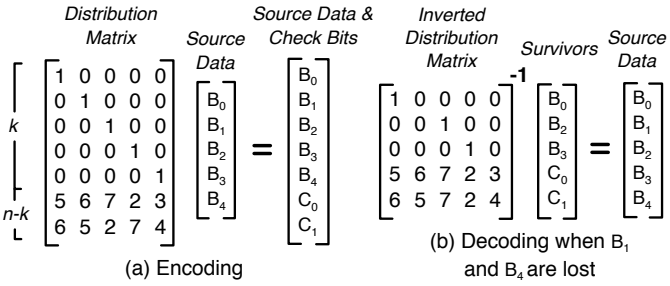

Figure 1: Illustrative example of RS encoding and decoding (k = 5 and n = 7).

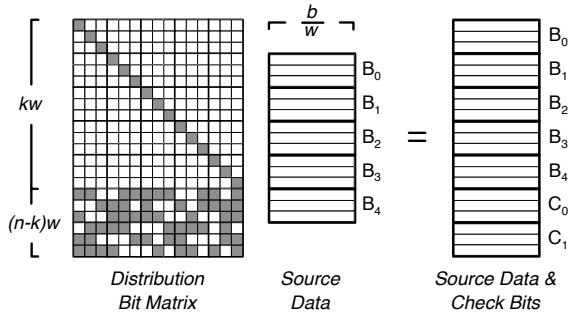

Figure 2: CRS version encoding of the same coding scenario in Figure 1 (assuming the size of data block is b bits). The dark unit in the bit matrix represents 1; the light one represents 0.

Erasure codes are widely used in RAID systems to provide reliability and serviceability [19, 21], but these systems do not rely on erasure codes to manage power. In these systems, the number of active disks is always greater than or equal to the number of disks that hold the source data; thus, these methods cannot be applied to a power-capped HMC system whose power budget does not permit activating all of the HMCs simultaneously.

The RAIM system from IBM [22] protects a server from single-channel DRAM errors by using erasure codes. Different

from prior work, the proposed encoded HMC system employs CRS codes to enable efficient power capping.

3. Overview

The proposed approach hides the long latency to activate a sleeping HMC by retrieving and decoding a subset of the data within the same coding block from active HMCs. Figure 3 shows an example HMC system with and without the proposed encoding mechanism, in which the memory power budget allows only two of the HMCs to be active at any time. Two memory requests, one to HMC 0 and the other to HMC 1, are shown in the example. Initially, HMC 1 is in sleep mode, and HMC 0 is active. In a standard HMC system, after the completion of the first request, the second request must wait until the power manager turns off HMC 0 and subsequently turns on HMC 1, which incurs a 2.6μs latency. However, in the proposed encoded HMC system, the second memory request does not have to wait: the system retrieves and decodes a part of the coding block from HMC 0 and HMC 3, and incurs a latency much shorter than the standard HMC system does.

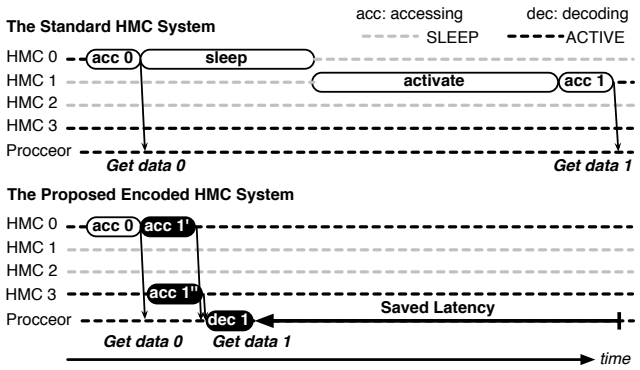

Figure 3: Illustrative example of the key idea.

Figure 4 illustrates an example multicore system connected to four single-link HMCs using the proposed architecture. In a system with $k+1$ HMCs, a coding block is formed by encoding k data chunks (cache blocks B_0, B_1, B_2), each of which comes from a different HMC. The cache blocks storing the check bits (C_0, C_1) are stored in the $(k+1)^{\text{st}}$ HMC, called the *always-on HMC*. This always-on HMC is always active to support the coding capability. A centralized control unit, comprising an *encoding/decoding manager* and a *power manager*, controls the power mode of the individual HMCs, and services memory requests based on the current power mode of each HMC.

When a memory request is received, the HMC manager first checks whether the HMC that stores the data is active. If so, the memory request is sent to the destination HMC controller. If the destination HMC is sleeping and the memory request is a read operation, the encoding/decoding manager checks whether the data is decodable. If so, the memory request is serviced immediately by decoding. Otherwise, the encoding/decoding manager sends an HMC wake-up request to the HMC power manager, and waits until the HMC is active.

4. Implementation

An HMC power manager that transitions the HMCs between active and sleep modes (Section **4.1.**) is employed by all of the evaluated systems. An encoding/decoding manager (Section **4.2.**) supports the coding related operations with the optimizations described in Section **4.3.**.

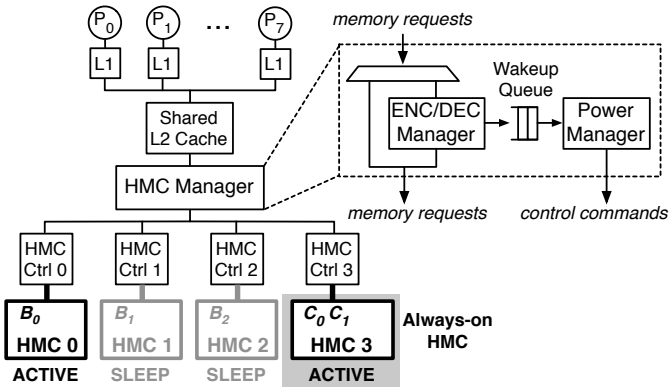

Figure 4: An example multicore system comprising a shared L2 cache and four single-link HMCs using the proposed block encoding mechanism.

4.1. Power Manager

The power manager receives wake-up requests. It decides which HMC to activate or put to sleep based on the current state of the HMCs, and the HMC power allocated by an upper-level power manager (*e.g.*, an OS power manager). In addition to the links, the DRAM layers on a sleeping HMC are also put into the low power, self-refresh mode to reduce the HMC static power consumption. The operation of the HMC power manager is based on related work [11, 12], and is decoupled from the memory controller by a simple interface. The power manager implements wake-up, sleep, and time-out logic. A status block records the number of active HMCs, and the number of cycles for which the oldest pending request to each HMC has been waiting. Algorithm 1 gives an overview of the power management procedure. [2] The wake-up logic picks the head entry in the wake-up queue, and sends an activate request to the corresponding HMC controller when the number of active HMCs is less than the power budget permits. When the number of active HMCs equals the maximum number allowed under the HMC power budget, and there are HMCs with no pending memory requests (*free HMCs*), the sleep logic puts the least-recently-used (LRU) HMC to sleep before activating a sleeping HMC. The sleep logic blocks new requests to the LRU (active) HMC to prevent starvation when the head entry of the wake-up queue has been waiting for a period longer than a configurable time-out threshold, and all of the active HMCs are busy.

4.2. Encoding/Decoding Manager

Directly applying the encoding mechanism to an HMC subsystem cannot save any power, since the number of always-on HMCs that store the check bits would equal the number of HMCs that are allowed to sleep. Therefore, the proposed encoded HMC system must maintain a small *encoded region* to which the erasure code is applied. An encoding/decoding manager is implemented as shown in Figure 5. The decoding cache

[2] We made the conservative assumption that the HMC static power is the same in stable and transition states.

Data: $activeHMCs$, $sleepingHMCs$
Result: power management commands to the HMC controllers
`// Wake-up logic: FIFO selection policy`
while $wakeupQ \neq \varnothing$ *and* $|activeHMCs| < numHMCCap$ **do**
  $HMC \leftarrow$ dequeue($wakeupQ$);
  wakeup(HMC);
  $sleepingHMCs \leftarrow sleepingHMCs - \{HMC\}$;
  $activeHMCs \leftarrow activeHMCs \cup \{HMC\}$;
end
`// Sleep logic: LRU selection policy`
if $wakeupQ \neq \varnothing$ *and* $\exists\ freeHMC \in activeHMCs$ **then**
  $victimHMC \leftarrow$ LRU($freeHMCs$);
  sleep($victimHMC$);
  $activeHMCs \leftarrow activeHMCs - \{victimHMC\}$;
  $sleepingHMCs \leftarrow sleepingHMCs \cup \{victimHMC\}$;
end
`// Time-out logic: LRU selection policy`
if *waitTime* $>$ *timeoutThreshold* *and* $\nexists\ freeHMC \in activeHMCs$ **then**
  block(LRU($activeHMCs$));
end

Algorithm 1: HMC power management.

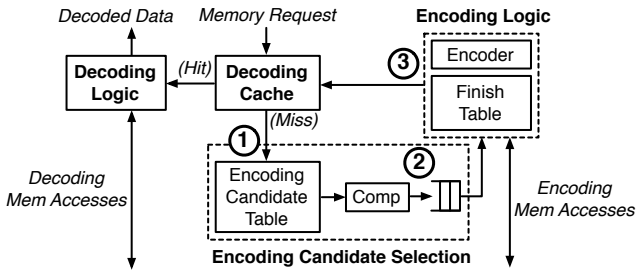

Figure 5: Encoding/Decoding Manager. The numbered labels are used for the encoding procedure.

is a small, set associative cache that is used to check whether a memory request is accessing the encoded region, and if so, to determine the address of the corresponding check bits. In order to reduce the decoding cache overhead, encoding is done at the page granularity. A group of encoded pages (including the generated check bits) is called a *coding group*. Decoding is performed at the individual cache line granularity.

4.2.1. Encoding

When a memory request can neither access a sleeping HMC nor be decoded, the corresponding coding group ID is sent to the *Encoding Candidate Table* (step ① of Figure 5). Each entry of the table has a counter to keep track of the number of accesses to the corresponding coding group. Every few cycles, the controller identifies the coding group with the highest counter value as a candidate, and empties the table. This simple—yet effective—scheme exploits data locality.

Encoding data too frequently not only increases the memory traffic, but also pollutes the decoding cache. Therefore, the counter value associated with the candidate is compared to a threshold to filter out data that are unlikely to be accessed in the near future. If the counter value exceeds the threshold, the encoding process begins (step ② in Figure 5). A *Finish Table* is used to keep track of the encoding status. When all of the relevant data have been received and the check bits have been generated, the coding group ID is inserted into the decoding cache along with the address of the check bits (step ③ in Figure 5). The encoder consists of a set of AND and XOR gates. The AND gates operate on the stored distribution matrix and the source data, and feed the results into an XOR tree, which produces the check bits. This simple encoding procedure is performed in the background, and is not critical to performance.

4.2.2. Decoding

When a memory request cannot access a sleeping HMC, it accesses the decoding cache to determine whether the required data is decodable. On a hit, the decoding logic sends the memory request to the always-on HMC (which contains the check bits), and to some of the other active HMCs. After all of the related data are retrieved, the source data block is decoded, and the memory request is serviced.

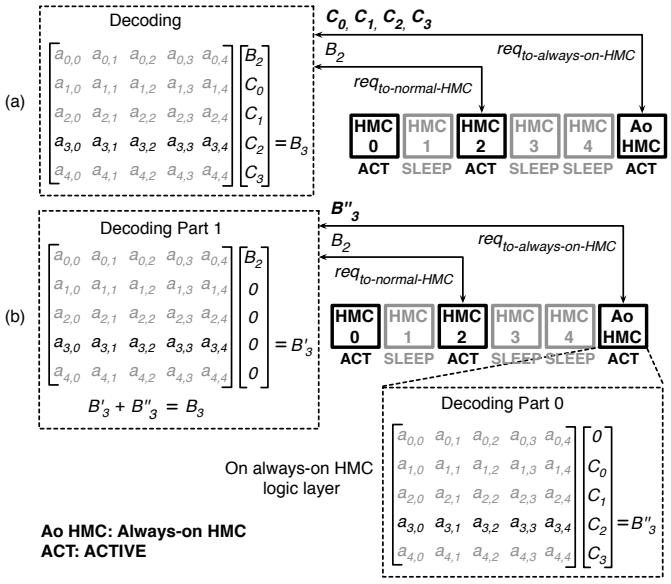

Figure 6: Illustrative example of the decoding procedure: (a) without optimization; (b) with optimization.

Similar to the encoder, the decoder comprises a combination of AND and XOR gates. The precomputed inverse matrices are kept in memory. Because the size of an inverse matrix is small and only a few such matrices are needed, the storage overhead is insignificant. Because the address of the inaccessible cache line is known, instead of reading the entire matrix and computing all of the source data in the coding block, we only need to access the row corresponding to the required cache line to perform the decoding. This method efficiently reduces the decoding latency and the associated energy overhead.

4.2.3. Updating

Since the check bits must be kept up-to-date, each write access needs to check whether it is going to access the encoded region. If so, the old and the new values are XORed, and the result is returned to the always-on HMC, which updates the check bits. The update logic is neither complex nor on the critical path. The evaluation indicates that its latency and power consumption overheads are not critical to the viability of the proposed system.

4.3. Memory Traffic Optimizations

Several optimizations are performed to reduce the extra coding memory traffic. These optimizations significantly reduce the extra bandwidth requirement and energy consumption of the coding process, thereby improving the performance and the energy efficiency of the proposed system.

4.3.1. Encoding

Since a large amount of data must be read and written during encoding, 128B reads and writes are employed to amortize the packet header cost.

4.3.2. Decoding

When more HMCs are allowed to sleep at the same time (Sections **2.** and **4.2.2.**), decoding an inaccessible cache line requires reading more check bits from the always-on HMC, which significantly increases the bandwidth pressure (Figure 6 (a)). To reduce the memory traffic, the decoding procedure is split between the logic layer of the always-on HMC and the processor, as shown in Figure 6 (b). The memory request sent to the always-on HMC requires reading the check bits, producing a partial result B_3'' (whose size is the same as the inaccessible cache line), and sending it to the processor. At the same time, the processor generates another partial result B_3' based on the parts of the coding block obtained from the other HMCs (B_2). The source data (B_3) is recovered by XORing the two partial results.

4.3.3. Updating

Two special memory requests are used to reduce the memory traffic generated by the updates; both requests leverage the logic layer to provide a simple computational capability. The first one is a *read-XOR-update* request, which reads the old data, returns the XOR between the old and the new data, and writes the new data into the memory array. The second special request is an *update-check-bits* request, which transfers the result obtained from the read-XOR-update request to the always-on HMC. When the update-check-bits request is received, the check bits are read out, XORed with the updated value, and written back to the memory array.

5. Experimental Setup

Assessing the performance, energy, and hardware overheads of the proposed power management approach requires both circuit- and architecture-level design and evaluation. The encoder and decoder logic is designed and verified in Verilog RTL using Cadence NCSim, and synthesized using the Synopsys Design Compiler. We use a heavily-modified version of the SESC simulator [23] to model the proposed system. (The simulation parameters are shown in Table 1.) We use McPAT to estimate the processor energy at the 22nm technology node [24].

CACTI-3DD [25] is used to model the timing of the 3D DRAM die stack within the HMC. A custom address mapping, shown in Figure 7, is used to take advantage of spatial locality, thereby increasing the idleness of the HMC links[3]. The proposed system has eight data HMCs, and one dedicated, always-on HMC that stores the check bits. This system is compared against two baselines. The first baseline employs a standard, eight-HMC memory system with the same power management mechanism as the proposed technique but no special provisions to avoid the long power mode switching latency. The second baseline architecture adds caching on top of the first baseline to cache parts of the working set that reside in the inaccessible HMCs. The memory capacity overhead of the second baseline is the same as that of the proposed architecture. Two caching policies are implemented and evaluated. The first, conventional caching policy, treats the extra HMC as a page cache. The second, enhanced caching policy, uses the data selection and power mode switching mechanisms of the proposed approach, with parameters and threshold values tuned to achieve high performance. The maximum number of active HMCs (including the always-on HMC in the proposed system) at any moment—and thus the HMC power budget—is the same for all of the evaluated systems. As the evaluated benchmarks are not large enough to use the entire physical address space, the size of the extra storage space is restricted to less than 1/9 of the actual memory footprint of each application. [4]

Core	32 out-of-order cores, 2.2GHz, issue width 4, commit width 4
IL1 cache	64KB, 4-way, 64B line, hit/miss delay 2/2
DL1 cache	64KB, 4-way, 64B line, WB, hit/miss delay 2/2
Coherence	Snoopy bus with MESI protocol
L2 cache	16MB, 16-way, 8 banks, 64B line, hit/miss delay 18/4
HMC Interface	HMC manager including link power manager, link controllers
HMC	4GB/HMC, 256 banks/HMC
SerDes Link	15GHz, 2.1W/link (active) [5], 0.1W/link (sleep) [17]
Switching latency	Power mode: active→sleep = 600ns, sleep→active = 2μs [2]
Vault controller	FCFS, close-page policy
DRAM timing	tCK=1ns, tRCD=4, tRAS=11, tRC = 21, tCAS=7, BL=8
DRAM power	0.47W background power [5], 3.7pJ/bit access energy [1]

Table 1: Simulated system parameters.

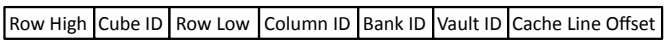

Figure 7: Memory address mapping.

We evaluate a mix of twelve parallel applications that are readily portable to our simulator from the NuMineBench [26], Phoenix [27], SPLASH-2 [28], SPEC OpenMP [29], and NAS [30] suites.

6. Evaluation

This section evaluates the performance, power, energy, and area characteristics of the proposed architecture.

6.1. Hardware Overhead

Table 2 lists the power, area, and latency overheads of the additional hardware units introduced by the proposed architecture. The power overhead of the additional hardware represents less than 0.03% of the entire power-capped HMC system. The latencies are correctly modeled as processor cycles for performance analysis.

[3]We evaluated different address mappings in which the cubeID was assigned different bit positions, including page/block interleaving, and those in [2], and chose the best performing one for the baseline.

[4]One always-on HMC out of nine total HMCs is used to store the check bits or the cached data.

	Power (mW)	Area (mm^2)	Latency (ns)
Encoding Logic	4.9	0.004	3
Encoding Candidate Selection Logic	0.4	0.0001	0.75
Decoding Logic	2.8	0.006	0.5
Decoding Cache	4.0	0.013	0.25
Decoding Logic (on Always-on HMC)	8.2	0.017	0.5
Updating Logic (on Normal HMC)	0.5	0.001	0.25
Updating Logic (on Always-on HMC)	1.2	0.009	0.75

Table 2: Power, area and latency overheads of the additional hardware on the processor and the HMC sides at 22nm.

6.2. Performance

Figure 8 shows the performance of the proposed architecture and the baseline systems that use caching, normalized to the standard HMC system. All of the results in this section and Section **6.3.** are based on the same power budget, which allows four HMCs to be active simultaneously.

On average, the encoded HMC system achieves a 6.2× speedup over the standard HMC system, a 2.9× speedup over the baseline system with the conventional caching policy, and a 2.2× speedup over the baseline system with the enhanced caching policy. 99% of the inaccessible data can be decoded in the proposed system. Among all of the benchmarks, `raytrace` exhibits the least performance improvement (less than 10%). The reason is that its memory access patterns allow most of the HMCs to have long idle times, and the baseline system undergoes only a small performance degradation under power capping, which leaves little room for improvement.

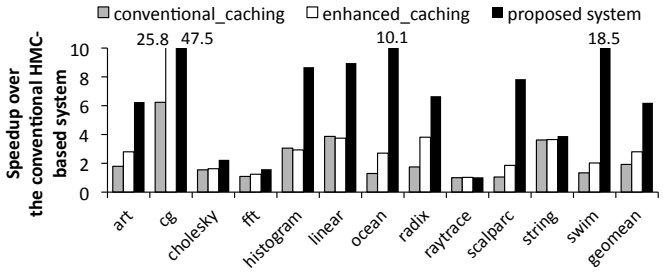

Figure 8: Performance of the proposed architecture.

The baseline systems with caching exhibit better performance compared to the naïve baseline without caching. However, their performance is still worse than the proposed system. For the system with the conventional caching policy, when an uncacheable HMC is temporarily transitioned into sleep mode to satisfy the wakeup requests from the other HMCs, all of the memory requests to the sleeping HMC are blocked. Because of the long power mode switching latency, the system blocks a significant number of memory requests that cannot be serviced until the destination HMC is activated again. In order to reduce the number of blocked memory requests, the enhanced caching policy avoids having a fixed designation of cacheable and uncacheable HMCs. Therefore, regardless of which HMC is in sleep mode, a memory request can always be serviced if the requested data resides in the always-on HMC. However, the proposed approach can make more efficient use of the always-on HMC than the enhanced caching since it stores the check bits instead of replicating the source data. With the same capacity overhead, the proposed system achieves the highest hit rate in the always-on HMC when the memory requests cannot access the sleeping destination HMC. Therefore, the proposed approach can deliver higher system performance compared to caching under an iso-capacity requirement.

For the benchmark `string`, the hit rates observed under both caching policies are close to that of the proposed approach. This observation explains why the performance gap between caching and the proposed approach is relatively narrow for `string`. Since the enhanced caching policy achieves a better performance than the conventional caching policy, we use the system with the enhanced caching policy as the caching baseline for the remainder of the evaluation.

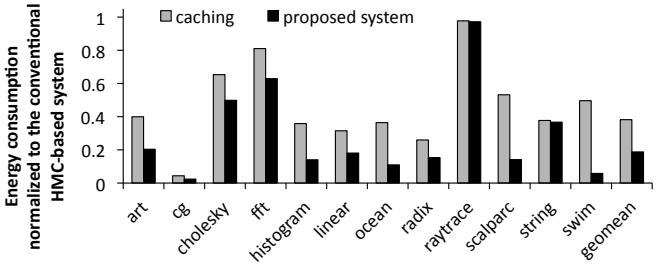

Figure 9: Energy consumption of the proposed architecture.

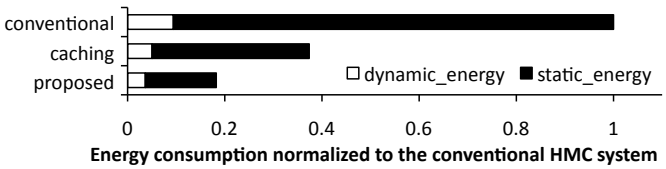

Figure 10: Energy breakdown of the evaluated systems.

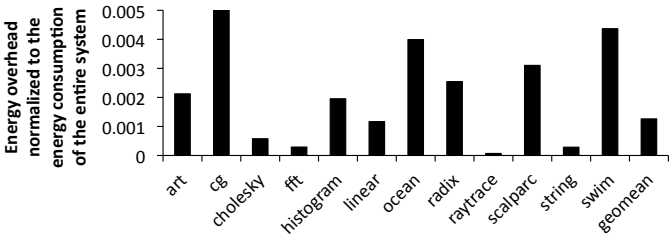

Figure 11: Energy overhead to support the coding capability.

6.3. Energy

The energy consumption of the entire system (including the cores, the shared L2 cache, the memory controller, and the HMC modules) is shown in Figure 9. On average, the proposed system respectively expends 19% and 49% of the energy consumed by the naïve and the caching baseline systems. The baseline systems consume more energy due to the longer execution time, which is shown in Figure 10. The proposed architecture does not save significant energy on `raytrace`, because the naïve baseline system does not degrade the performance of this benchmark significantly. Figure 11 depicts the coding related energy overhead, which is less than 0.15% of the total system energy.

6.4. Sensitivity Studies

Sensitivity to the size of the memory region allocated to the check bits, and to the HMC power budget, are studied in this section.

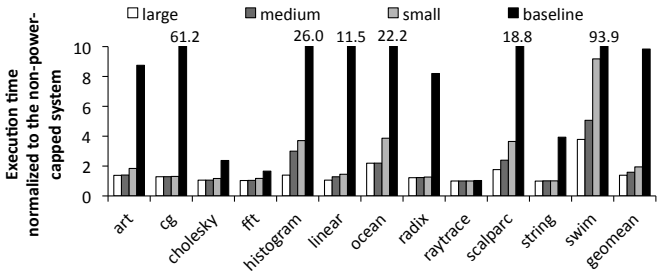

Figure 12: Performance of the proposed architecture with different sizes of the check bits.

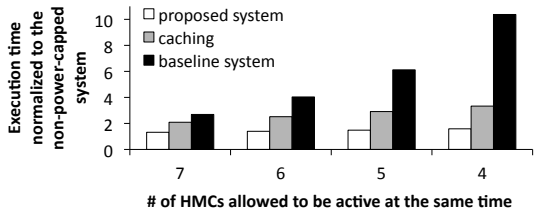

Figure 13: Performance of the proposed architecture under different HMC system power budgets.

6.4.1. Sensitivity to the Size of the Check Bits

Figure 12 shows how the performance of the proposed HMC system changes with the size of the memory region allocated to the check bits. *Medium* represents a system that restricts the capacity of the space allocated to the check bits to less than 1/9 of the memory footprint of each application; *large* increases the size of the allocated space by 25%; and *small* reduces it by 25%. For most of the benchmarks, the performance degradation increases as the capacity of the space allocated to the check bits is decreased. `Histogram`, `ocean`, `scalparc`, and `swim` are more sensitive to the space allocated to the check bits in the system. These applications exhibit poor data locality, which renders the encoding candidate selection logic (Section **4.2.**) less effective. As one would expect, when the size of the encoded region decreases, the likelihood of retrieving inaccessible data by decoding also decreases.

6.4.2. Sensitivity to the Power Budget

Figure 13 shows the effect of the HMC power budget on both the proposed and the baseline systems. On average, all of the three systems perform worse when the HMC power budget is reduced. However, the performance degradation is less severe for the proposed HMC system. The benefits of the proposed system increase as the power budget is lowered.

7. Conclusions

We introduce a power-capped HMC system that employs erasure codes to tolerate the long HMC power mode switching latency. Rather than waiting for an HMC to become active, a memory request to the unavailable data can be satisfied by decoding other data within the same coding block, which can be retrieved from the active HMCs. Overall, the proposed architecture achieves a 6.2× performance improvement and a 5.3× energy reduction over a standard HMC system operating under the same power budget. We conclude that erasure coding holds significant potential to improve the performance and energy efficiency of future high-performance memory systems.

References

[1] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube new DRAM architecture increases density and performance," in *VLSIT*, 2012.

[2] Hybrid Memory Cube Consortium, "Hybrid Memory Cube specification 1.0," 2013.

[3] AnandTech, "Intel's "Knights Landing" Xeon Phi coprocessor detailed," 2014.

[4] Fujitsu, "SPARC64 XIfx: Fujitsu's next generation processor for HPC," 2014.

[5] S. H. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory + logic devices on MapReduce workloads," in *ISPASS*, 2014.

[6] Fujitsu, "A new generation 16-core processor for supercomputing SPARC64 IXfx," 2011.

[7] Advanced Micro Devices, "ACP - the truth about power consumption starts here," 2009.

[8] K. T. Malladi *et al.*, "Towards energy-proportional datacenter memory with mobile DRAM," in *ISCA*, 2012.

[9] C. Lefurgy *et al.*, "Power capping: A prelude to power shifting," *Cluster Computing*, vol. 11, no. 2, pp. 183–195, 2008.

[10] T. Patki *et al.*, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *ICS*, 2013.

[11] B. Diniz *et al.*, "Limiting the power consumption of main memory," in *ISCA*, 2007.

[12] I. Hur and C. Lin, "A comprehensive approach to DRAM power management," in *HPCA*, 2008.

[13] M. Floyd *et al.*, "Adaptive energy-management features of the ibm power7 chip," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 8:1–8:18, 2011.

[14] Fujitsu, "White paper: Reduce environmental impacts through Fujitsu's platforms," 2013.

[15] Hewlett-Packard, "Technologies in HP ProLiant Gen8 c-Class server blades," 2013.

[16] V. Soteriou and L.-S. Peh, "Dynamic power management for power optimization of interconnection networks using on/off links," in *HOTI*, 2003.

[17] J. Ahn *et al.*, "Dynamic power management of off-chip links for Hybrid Memory Cubes," in *DAC*, 2014.

[18] K. T. Malladi *et al.*, "Rethinking DRAM power modes for energy proportionality," in *MICRO*, 2012.

[19] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Softw. Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[20] J. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *NCA*, 2006.

[21] C. Weddle *et al.*, "PARAID: A gear-shifting power-aware RAID," *Trans. Storage*, vol. 3, no. 3, 2007.

[22] P. Meaney *et al.*, "IBM zEnterprise redundant array of independent memory subsystem," *IBM Journal of Research and Development*, vol. 56, no. 1.2, pp. 4:1–4:11, 2012.

[23] J. Renau *et al.*, "SESC simulator," 2005, http://sesc.sourceforge.net.

[24] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[25] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *DATE*, 2012.

[26] J. Pisharath *et al.*, "NU-MineBench 2.0," Northwestern University, Tech. Rep., 2005.

[27] R. M. Yoo *et al.*, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *IISWC*, 2009.

[28] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[29] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[30] D. H. Bailey *et al.*, "NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep., 1994.