Concurrent Dynamic Memory Coalescing on GoblinCore-64 Architecture

Xi Wang
Department of Computer Science
Whitacre College of Engineering
Texas Tech University
Box 43104
Lubbock, Texas 79409-3104
xi.wang@ttu.edu

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

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

ABSTRACT

The majority of modern microprocessors are architected to utilize multi-level data caches as a primary optimization to reduce the latency and increase the perceived bandwidth from an application. The spatial and temporal locality provided by data caches work well in conjunction with applications that access memory in a linear fashion. However, applications that exhibit random or non-deterministic memory access patterns often induce a significant number of data cache misses, thus reducing the natural performance benefit from the data cache.

In response to the performance penalties inherently present with non-deterministic applications, we have constructed a unique memory hierarchy within the GoblinCore-64 (GC64) architecture explicitly designed to exploit memory performance from irregular memory access patterns. The GC64 architecture combines a RISC-V-based core coupled with latency-hiding architectural features to a memory hierarchy with Hybrid Memory Cube (HMC) devices. In order to cope with the inherent non-determinism of applications and to exploit the packetized interface presented by the HMC device, we develop a methodology and associated implementation of a dynamic memory coalescing unit for the GC64 memory hierarchy that permits us to statistically sample memory requests from non-deterministic applications and coalesce them into the largest possible HMC payload requests.

In this work, we present two parallel methodologies and associated implementations for coalescing non-deterministic memory requests into the largest potential HMC request by constructing a binary tree representation of the live memory requests from disparate cores. We present the coalesced HMC memory request results from applications that exhibit linear and non-linear memory request patterns compiled for a RISC-V core in contrast with a traditional memory hierarchy.

MEMSYS 2016 October 03-06, 2016, Washington, DC, USA

DOI: http://dx.doi.org/10.1145/2989081.2989128

CCS Concepts

•Computer systems organization → Architectures; Parallel architectures; •Computing methodologies → *Parallel computing methodologies;*

Keywords

Memory Coalescing, Microcode, GoblinCore-64, RISC-V, Parallel Computing, Data-Intensive Computing

1. INTRODUCTION

Mainstream modern microprocessor architectures are constructed with the memory systems that consist of multi-level data caches and traditional DDR main memory devices. The native hardware concurrency mechanisms present in the respective micro-architectural implementations only provide a low degree of hardware managed concurrency. Further, these mechanisms are often difficult or entirely not visible from the application layer or instruction set architecture. These mechanisms often promote efficient utilization or near-optimal performance for applications with significant memory reuse or linear memory access patterns.

Conversely, applications generally considered to be *data-intensive* access memory in irregular and non-deterministic patterns or in strides that exceed the size of modern data caches. Executing this class of application on a traditional micro architecture has the inability to make use of the on-chip data caches, resulting in inefficient use of the memory hierarchy. In response to these data-intensive applications, we have developed the GoblinCore-64 (GC64) micro architecture with using a large degree of hardware-managed concurrency coupled to a high bandwidth memory subsystem [1].

We introduce the GC64 machine hierarchy in Figure 1. The core machine model and instruction set are based on the RISC-V [2] instruction set architecture. We utilize the three-dimensional stacked memory devices in the form of Hybrid Memory Cube (HMC) devices as the basis for the GC64 main memory. The HMC devices provide uniquely high bandwidth over traditional DDR-based memory units alongside a packetized memory interface. The GC64 system on chip consists of a series of hierarchical hardware modules. Each socket is constructed with one or more GC64 task groups. These task groups are integrated via a net-

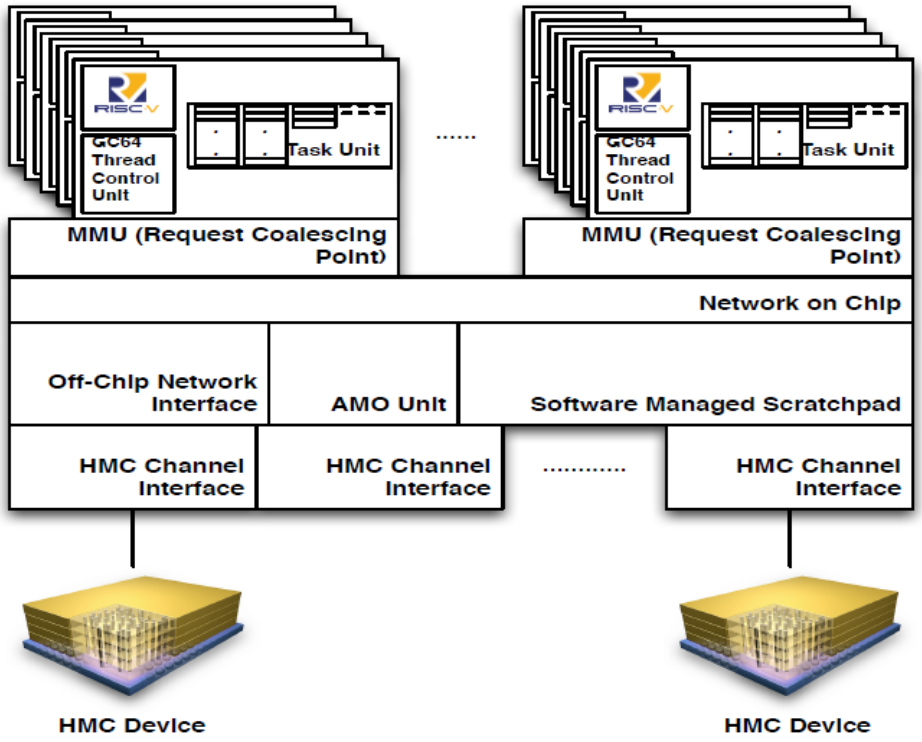

Figure 1: The architecture of GC64.

work on chip interface to four shared on-chip components. The on-chip software-managed scratchpad unit acts as a very high performance, user-mapped storage mechanism for commonly used data. The Atomic Memory Operation (AMO) Unit is responsible for controls queuing, ordering and arbitration of atomic memory operations. The HMC Channel Interface handles the protocol interaction between multiple HMC devices. Finally, the Off Chip Network Interface handles any off chip memory requests that utilize the GC64 memory addressing mechanisms.

In order to make best use of the packetized interface presented by the aforementioned main memory HMC devices, we present a concurrent processing methodology and associated implementation in order to coalesce memory accesses from disparate GC64 cores into the largest potential HMC memory requests. We utilize a parallel [3], tree-based methodology in order to optimize the process of coalescing disparate read and write requests prior to dispatch HMC requests in a *dynamic memory coalescing* unit or *DMC*. This concurrent DMC mechanism maintains the same logic in the memory coalescing unit as before, but further decreases the number of the requests flushed into the hybrid memory cubes. Further, the new approach increases the efficiency of the memory coalescing, especially when coupled to multiple HMC devices or when executing applications whose memory request patterns are unusually non-deterministic.

The contribution of this research study is three-fold:

- Build the tree-based memory coalescing model with the tree logic design for correctly coalescing the memory accesses;
- Construct the architecture for concurrent DMC. Two different parallel algorithms are also designed for the concurrent DMC unit;
- Implement the concurrent DMC and prove the superiority of memory coalescing unit, in the perspective of efficacy through the test applications.

The remainder of this paper is organized as follows. Section 2 introduces the previous work. Section 3 introduces the concurrent dynamic memory coalescing concept and design. Evaluations and results are discussed in the Section 4. Section 5 discusses relevant work and compares this study with them. We conclude this research and future work in Section 6.

2. PREVIOUS WORK

Following the release of the open-source, RISC-V instruction set architecture, several research studies have been conducted in order to implement and extend the ISA for application-specific workloads. Full VLSI studies [4] and energy efficiency studies have shown that the RISC-V micro architecture is competitive with other RISC-style system on chip designs [5]. As a general-purpose ISA, RISC-V is also used as the vector processor for the DC-DC converter [6], or extended for threads reallocation in the mixed-criticality systems [7].

However, there has yet been no previous research on attempting to utilize RISC-V as the basis for a scalable high performance or supercomputing system infrastructure. In response to this, we developed the GC64 architecture as a scalable, flexible and open architecture for efficiently executing data-intensive computing applications and algorithms.

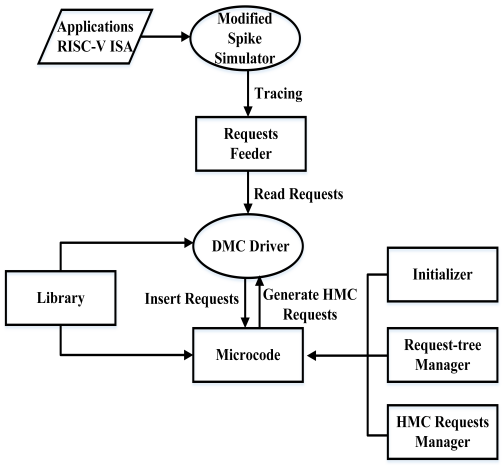

Figure 2: Dynamic memory coalescing components.

As a part of our continued GC64 research efforts, we have extended the core RISC-V simulation infrastructure in order to support our efforts to extend the RISC-V micro architecture with latency-hiding capabilities and a high degree of concurrency. Finally, several previous efforts have focused on understanding the potential performance [8] [9] [10], power [11] and thermal [12] properties of HMC devices as connected to modern core processors or FPGAs. In addition to the core hardware properties of HMC devices, several efforts have focused on understanding the inherent effects of the HMC packetized interface has on certain pathological classes of applications [13]. In addition to examining the existing HMC specification, recent efforts have focused on modeling the potential future HMC logic layer functionality in the form of *Custom Memory Cube* or *CMC* operations that stand to significantly increase performance for typical operations such as mutexes, barrier synchronization and pointer chasing [14] [15] [16].

3. CONCURRENT DYNAMIC MEMORY COALESCING

In this section, we introduce the design and methodology of concurrent dynamic memory coalescing, including two different concurrent coalescing algorithms, in order to coalesce the memory accesses between the RISC-V cores and Hybrid Memory Cube in GC64 highly efficiently.

3.1 Dynamic Memory Coalescing

The dynamic memory coalescing, or *DMC*, unit found in the GC64 micro architecture consists of several components as shown in Figure 2. First, the DMC unit is constructed using a small, embedded RISC-V core that implements the basic RISC-V integer instruction set. As such, the DMC operates using a lightweight, embedded microcode library that implements the core DMC methodology.

Next, in order to provide basic simulation and debugging capabilities, we provide a driver interface to the core DMC microcode library. The DMC driver permits us to instantiate the microcode with synthetic memory traces, gather internal statistics of the microcode runtime and print basic debugging information during simulated microcode execution. The DMC is also responsible for the parallel operation

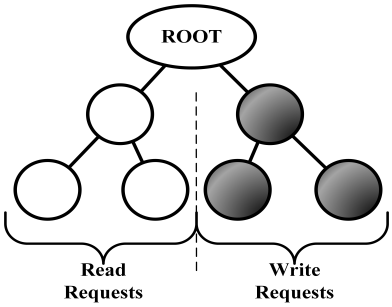

Figure 3: Tree structure of memory requests.

of the microcode. We describe the concurrent coalescing algorithms later.

Finally, the DMC infrastructure consists of the request tree manager and the HMC request manager units. The former ingests the incoming memory requests and manages the internal tree state during live microcode operations. The tree management is performed via the rules and methodology set forth in the next sub-section. The latter utilizes the coalesced memory request structures after being flushed by the tree logic to construct the equivalent HMC memory requests in the correct packetized form.

In addition to the core functionality provided by the DMC microcode implementation, the associated library infrastructure and the DMC driver, we also provide basic extensions to the RISC-V functional simulation infrastructure, *Spike*. The extensions enable us to output native memory tracing from actual application execution and import the traces directly into the DMC driver interface. The DMC driver ingests the request via standard input or static file input.

Once initialized, the DMC driver will ingest the requests into the request-tree manager. The request-tree manager will build and organize the memory requests as a binary tree structure. The request-tree manager maintains three internal binary trees, representing local memory requests, global or *off-chip* memory requests and atomic memory operation requests. Each of the tree structures maintains a single null node that represents the root of the tree. For each new memory request inserted into the tree, read operations are inserted to the left of the root node and all write operations are inserted to the right of the root node.

The trees are constructed using the form shown in the Figure 3. Each node in the binary tree contains the following data:

1. *Task Operation*: Represents the type of requests (read, write AMO) and its associated memory operation size (1,2,4,8 bytes)
2. *Task ID*: Identifies the source task ID for each specific request
3. *Address*: The source address of the respective memory request

As requests are individually inserted into the respective request tree, they are sorted according to their source address. As such, the left most leaf node will contain the lowest request address.

As memory requests are inserted, the tree will eventually reach a point of saturation. This occurs when 128 bytes

of memory requests are resident within the read or write portion of the tree. The 128 byte limit corresponds to the maximum theoretical memory request size in the HMC 1.0 specification [17]. Additionally, we track the number of insertion cycles that a tree has been active. If we reach a tree saturation or exceed a maximum timeout value, the microcode forces a *tree expiration*, or the point at which HMC requests are flushed from the coalesced memory requests in the tree.

The HMC requests are defined in a structure which contains:

1. *Memory oepration*: read or write requests corresponding to HMC request packets
2. *Address*: the address corresponding to the respective HMC request.

Eventually, the HMC requests will be converted into the HMC request packet [17] which is defined as the structure that consists of request header, request tail, Cube ID, address, and TID.

3.2 Coalescing Tree Logic

Given that the GC64 micro architecture lacks inherent data caches as associated with the core memory hierarchy, the memory coalescing provided by the DMC unit is vital to optimizing the access to main memory from the concurrent task units. The HMC packetized request format provides a unique capability in that posting larger memory requests, up to 128 bytes each, significantly reduces the control overhead required to access main memory. As such, coalescing the memory requests from multiple, concurrent task units increases the efficiency of accessing main memory.

Once the tree reaches the condition that triggers the expiration, the most left child will be found as the base address. We traverse the tree in order, checking each subsequent node in order to determine its spatial distance from the previous address. We do so recursively until we find the maximum possible request to inject into one or more HMC devices.

The address of the read requests in the tree may occur in one of the following forms as shown in Figure 4. In the case a, two requests address are completely consecutive. In the case b, the address of request 2 totally overlaps with request 1, which means the ending address of request 2 is not greater than that of request 1. As shown in the case c, addresses of request 1 and request 2 have an overlap.

The above case a, b and c are considered consecutive occurrences, which can be reduced to a single HMC request if the total read bytes is not greater than 128 bytes. As shown in the case d, the address of two requests are not consecutive, if the distance between the starting address of the request 1 and the ending address of request 2 is not greater than 128 bytes. In this case, request 1 and request 2 can still be formed in one HMC request. Otherwise, these two requests will be separated into two HMC requests. In the event of write requests, cases a, b and c can still be reduced to one HMC request. However, for the case d, if only one HMC request is built, then the intermediate addresses would be modified or *poisoned*. In order to avoid corrupting the state of memory, we force the algorithm to generate multiple HMC requests. This tree logic will run recursively until it reaches the root or null node. Eventually, the entire tree will be marked as expired, leaving only the root or null node remaining.

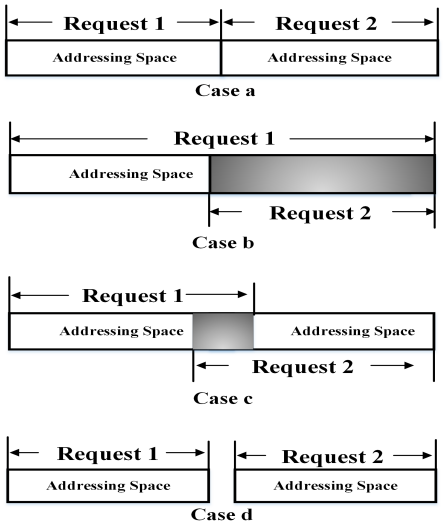

Figure 4: Cases of memory accesses.

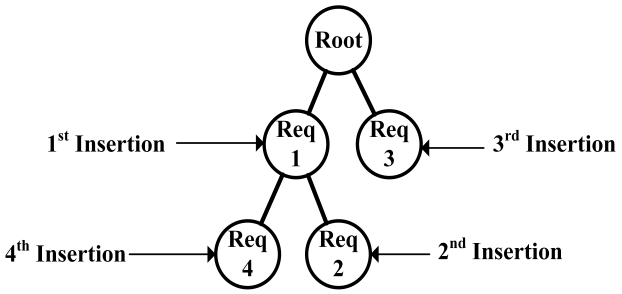

Figure 5: Example of the tree logic execution.

For example, assume four requests are being handled by the memory coalescing unit:

- Request 1: read 8 bytes at address 0x0000100F
- Request 2: read 16 bytes at address 0x00001018
- Request 3: write 32 bytes at address 0x000010FF
- Request 4: read 16 bytes at address 0x00001008

As shown in the Figure 5, the binary tree will first insert request 1 at the left side of the root node because request 1 is a read operation. Then insert request 2 as the right children of request 1 following the formation of sorted binary tree. At the third step, request 3 is inserted as the right child of the root as it is a write memory request. Finally, due to the address of request 4 being smaller than request 1, it will be inserted as the left child of request 1. After expiring the binary tree, one HMC read request will be generated, which will read 32 bytes starting at the address 0x00001008, and one HMC write request will be generated, which will write 32 bytes at the base address 0x000010FF.

3.3 Concurrent Coalescing Methodology

The previous sub-sections describe the dynamic memory coalescing and the logic of the tree coalescing algorithm. However, applications may attempt to access memory resident in different areas of a single HMC device or across multiple HMC devices, leading to inefficient coalescing as these

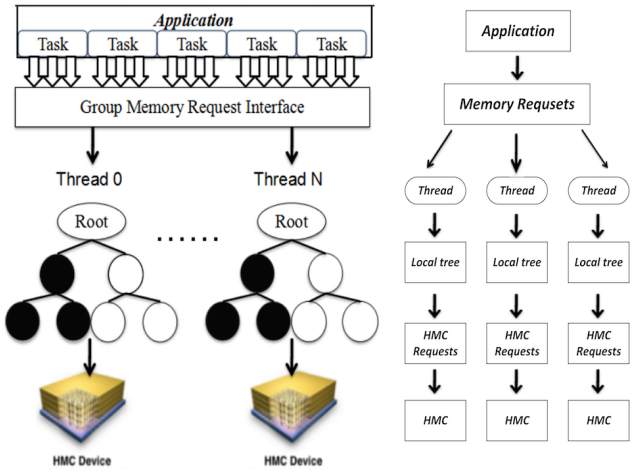

Figure 6: The architecture of concurrent DMC.

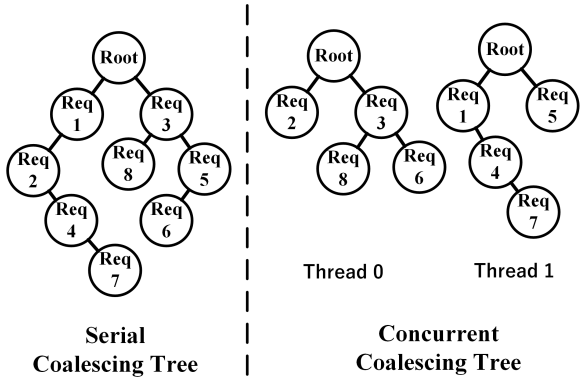

Figure 7: Example of concurrent DMC

addresses can be far away from each other and decrease the chance of being coalesced. The fundamental idea of concurrent coalescing is to introduce multiple coalescing units and logic, with each coalescing unit dedicated to handle memory coalescing for a smaller address space (e.g. for one partition in the case of a single HMC device, or for one HMC device in the case of multiple HMC devices), and thus increase the chance and efficiency of memory coalescing. For instance, let us assume an instruction as below:

$$a[i] = b[i] + c[d[i]] \quad (1)$$

Let n equal to the value of *d[i]*, then *c[d[i]]* is equal to *c[n]*. Since n can be a totally random number, the memory accesses can be in different HMC devices or different partitions of one HMC. The addresses between different data items accessed can spatially distant from each other. The end result can be low efficiency or inability in reducing the tree into a single request. Therefore, in order to coalesce the memory accesses highly efficiently, we partition the address space and further optimize the memory accesses using concurrent coalescing logic to increase its overall efficacy.

Following this methodology, the tasks of dynamic memory coalescing are assigned to different and independent threads based on the address of memory requests. As shown in Figure 6, each thread will construct its own local tree and perform the necessary coalescing per the aforementioned logic. For thread 0 to thread N, each thread will only handle these addresses that fall within their assigned HMC address space partition and insert these addresses into their local trees, increasing the chance and efficiency of coalescing memory requests. We use an example to illustrate this methodology.

For instance, assume 8 requests need to be coalesced by the DMC unit, as shown in Table 1. For a single DMC unit, it will build a serial coalescing tree that contains all requests, as shown in the left part of Figure 7. In this way, when generating the HMC requests following the aforementioned tree logic, request 4 and 7 will be coalesced into one HMC read request, and request 3 and 8 will be reduced into one HMC write request. However, all the other requests cannot be coalesced, which will generate 6 HMC requests in total. With a concurrent DMC methodology, suppose thread 0 is responsible for coalescing the requests with the base address between 0x00000000 and 0x0FFFFFFF. Similarly thread 1 will coalesce the requests with the base address between 0x10000000 and 0x1FFFFFFF. Then, the concurrent coalescing tree will be built as shown in the right side of Figure 7. Afterwards, according to the tree logic, thread 0 will coalesce requests 3, 6, 8 into one HMC write request and one HMC read request. Similarly, the thread 1 will build one HMC read request from requests 1, 4, 7 and one HMC write request. Therefore, the concurrent DMC methodology will generate only 4 HMC requests in total, with a reduction of two requests compared against the case of a single DMC unit with generating 6 HMC requests.

We design two different concurrent coalescing algorithms for the concurrent DMC methodology, a *address partitioned algorithm (APA)* and a *work partitioned algorithm (WPA)*. Each algorithm maintains the aforementioned DMC components, memory coalescing tree model and coalescing tree logic. In order to ensure correct behavior in the concurrent coalescing algorithms of the DMC microcode, we restrict each of the global variables utilized by the coalescing functions to remain thread private prior to entering the parallel code region. Each of the parallel threads is also forced to maintain its own file pointer within the driver infrastructure such that multiple threads may make independent and asynchronous forward progress. In addition to the variables required for the microcode core, we also create copies of the timing and statistics variables for each individual thread. This enables us to record and tune the efficacy of individual microcode threads as they coalesce memory requests from different application workloads. These two concurrent coalescing algorithms are discussed in detail below.

3.4 Address Partitioned Algorithm

As shown in the algorithm 1, in the partitioned address algorithm (APA), the base address of each request is utilized to determine whether the respective thread will insert the request into its tree. The logic utilizes the respective thread ID (TID) combined with the following condition to make the determination:

$$TID \times \alpha \leq addr < (TID + 1) \times \alpha \quad (2)$$

As shown above, the *addr* represents the target address of the respective request and α represents the total addressing space divided by the N, the number of threads in the execu-

Table 1: Example of requests

Request	OP	Address
1	RD16	0X10009FFF
2	RD8	0X000F1000
3	WR16	0X00001008
4	RD8	0X1000A008
5	WR8	0X100F0008
6	WR16	0X0000101F
7	RD16	0X1000A010
8	WR8	0X00001000

tion context. This implies that the physical address space is separated into N contiguous blocks of size α and assigned to an individual thread.

```
Input: Requests from processors
Parallel code segment starts;
Initialization;
while there are pending requests do
    rqst = read_requests();
    if TID × α ≤ addr ≤ (TID + 1) × α then
        insert_request();
        if The tree meets the condition of expiration
        then
            build_hmc_requests();
            expire_tree();
        end
    else
        Continue;
    end
end
Free memory;
Parallel code segment ends;
Output: HMC requests
```

Algorithm 1: Address Partitioned Algorithm

For example, suppose we utilize N=16 threads in the execution context, α is 0x01000000 and there are eight requests to be inserted as shown in the Table 1. The *OP* column represents the respective operation of the requests, where *RD* is a read request and *WR* is a write request. In this manner, *RD8* would represent an 8-byte read request and *WR16* would represent a 16-byte write request.

In this example, thread 0 would be responsible for the address space falling between 0x00000000 and 0x0FFFFFFF. Similarly, thread 1 would be responsible for the address space falling between 0x10000000 and 0x1FFFFFFF. An example of inserting these aforementioned requests as listed in Table 1 is shown in Figure 8.

During the concurrent coalescing, threads do not block or wait for each other. Rather, they continue processing the incoming requests delivered by the driver. This process will continue iterating until all requests have been received by the DMC unit, which implies that each thread in the execution context will read and process each incoming memory request.

The final stage of the algorithm requires an explicit barrier in order to ensure that all threads have completed their respective processing and all the potential memory requests have been flushed to the equivalent HMC requests. We also

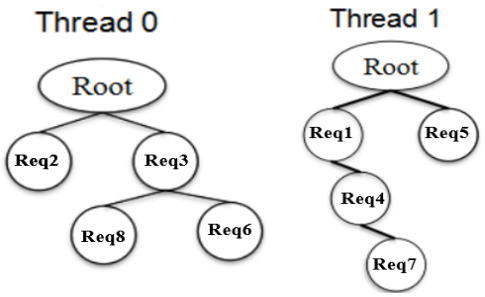

Figure 8: Example of partitioned address algorithm.

implement a single data reduction for the purpose of collecting the statistical timing and tuning values from each of the parallel threads after processing has been completed.

3.5 Work Partitioned Algorithm

The work partitioned algorithm (WPA) assigns the address spaces to the individual thread ranks in a similar manner to the APA approach. However, it also separates the read and write requests. In this algorithm, the thread ranks are also split into two halves. The lower half of the thread ranks handle the read requests and the upper half of the thread ranks handle the write requests. In the following equation, γ represents N divided by 2 and α represents the the size of each partition, which is the quotient of the total addressing space and γ.

```
Input: Requests from processors
Parallel code segment starts;
Initialization;
while there are pending requests do
    rqst = read_requests();
    if TID ≤ γ then
        if (rqst.op = read)and(TID × α ≤ addr ≤
        (TID + 1) × α) then
            insert_request();
            if Thetreemeetstheconditionofexpiration
            then
                build_hmc_requests();
                expire_tree();
            end
        end
    else
        If (rqst.op = write)and((TID − γ) × α ≤
        addr ≤ (TID − γ + 1) × α) insert_request();
        if Thetreemeetstheconditionofexpiration then
            build_hmc_requests();
            expire_tree();
        end
    end
end
Free memory;
Parallel code segment ends;
Output: HMC requests
```

Algorithm 2: Work Partitioned Algorithm

As shown in Algorithm 2, these threads whose TIDs are greater than or equal to γ only consider the incoming write requests. These threads with TIDs less than γ only consider

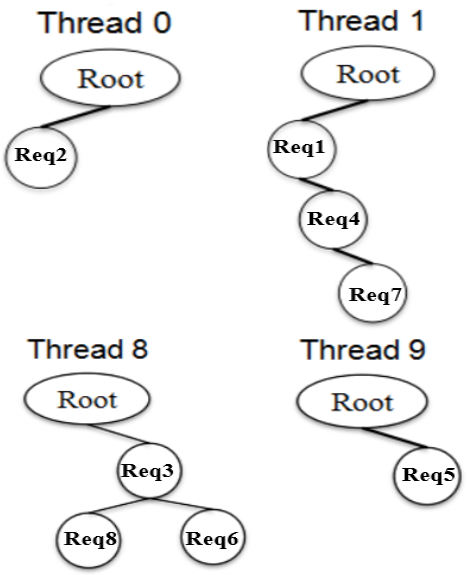

Figure 9: Example of partitioned work algorithm.

the read requests. We show this relation in Equations (3) and (4), respectively.

$$(TID - \gamma) \times \alpha \leq addr < (TID - \gamma + 1) \times \alpha \quad (3)$$

$$TID \times \alpha \leq addr < (TID + 1) \times \alpha \quad (4)$$

As an example, assume there are sixteen threads utilized in an execution context and α is 0x01000000. Algorithm 2 is utilized to insert these eight requests found in Table 1. In this manner, thread 0 and thread 1 will only insert the read requests within the address space spanning addresses 0x00000000 to 0x0FFFFFFF and 0x10000000 to 0x1FFFFFFF, respectively. Conversely, threads 8 and 9 will only consider write requests from the address space spanning 0x00000000 to 0x0FFFFFFF and 0x10000000 to 0x1FFFFFFF, respectively. All of these threads will follow the same approach until all requests are inserted, coalesced and subsequently expired to be dispatched to one or more HMC devices. Figure 9 provides an illustration of Algorithm 2.

4. EVALUATION

In order to evaluate the efficacy of our concurrent DMC design and aforementioned parallel algorithms, we construct two simulated environments that exchange data. The first environment is hosted by the RISC-V *Spike* functional simulator and is initialized to host a running application. We compile a set of known test cases using the RISC-V GCC compiler port and execute them on the Spike simulator. The application-level simulator generates memory traces which is directed into the second simulator instance via a named pipe. The second simulator instance, also using Spike, executes the microcode and the microcode driver. The DMC microcode driver receives the incoming memory requests, dynamically coalesces them via our methodology described above and outputs a set of statistics that reports the efficacy of the coalescing for the target application.

We utilize a set of four applications that represent dense (linear) memory requests, random memory requests and real application workloads to test the efficacy of our approach. These applications are described as follows:

- *High Performance Conjugate Gradient*: The HPCG benchmark [18] performs a fixed number of Gauss-Seidel preconditioned conjugate gradient iterations using double precision (64-bit) floating point arithmetic. This benchmark is designed to represent more traditional high performance computing workloads typically found in large-scale solvers.
- *SSCA2*: The SSCA2 benchmark [19] was developed in conjunction with the DARPA High Productivity Computer Systems (HPCS) program as the basis for a non-deterministic graph benchmark. The benchmark is composed of four kernels that operate on a large-scale, directed multi-graph.
- *STREAM*: The STREAM benchmark [20] is designed to measure the sustainable memory bandwidth for contiguous, long-vector memory accesses. It performs a series of four internal benchmarks that represent copying two vectors, vector-scalar multiplication, vector-vector addition and triadic operations with vector-scalar multiplication and vector-vector addition.
- *ScatterGather*: The final two test cases represent a pathological *scatter* and *gather* operation, respectively. Each benchmark initializes two long vectors that represent an index vector and a storage vector. The indices are initialized using random numbers generated from a known polynomial. The benchmark executes 1024 iterations of full scans across the entire storage vector where each lookup performs A[i] = B[Idx[i]] for the scatter and A[Idx[i]] = B[i] for the gather, respectively.

We execute the concurrent DMC microcode using 2, 4 and 8 threads. Given that the maximum HMC configuration attached to a single socket is currently 8, there is no apparent motivation to drive microcode concurrency beyond 8 parallel units. Further, the chip area required to implement such a parallel execution unit directly in the memory pipeline would outweigh the marginal benefit. For each of the address partitioned algorithm (*APA*) and the work partitioned algorithm (*WPA*), we record the number and distribution of incoming memory requests as well as the number and distribution of outgoing HMC requests. We calculate the relative efficiency of the given simulation by dividing the number of coalesced requests by the total number of input requests. The requests of the APA and the WPA approach are presented in Figure 10 and Figure 11, respectively.

As shown in Figure 10, the APA approach performs relatively well on the STREAM, scatter and gather tests. The performance and scalability are stable and outperform the lack of coalescing as the thread concurrency scales from 2 to 8. The scatter and gather test case provides particularly good scalability at 8 threads as it coalesces 72.48% of the incoming memory accesses. However, the test cases demonstrating the HPCG and the SSCA2 benchmarks only exhibit slight increases in overall efficiency.

Upon further analysis, we find that the HPCG and SSCA2 tests demonstrate much larger input sets for the aforementioned simulation results. Analyzing the raw request data

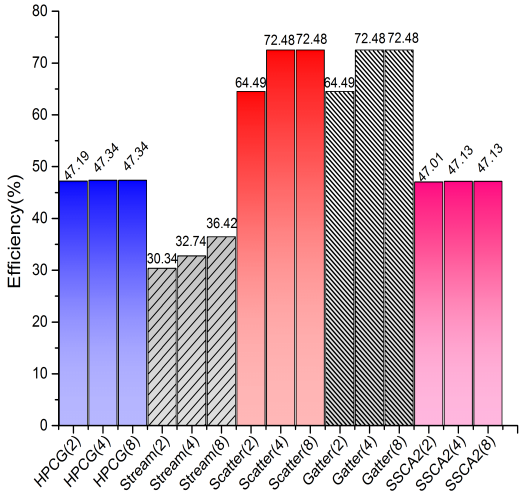

Figure 10: Scaled APA results.

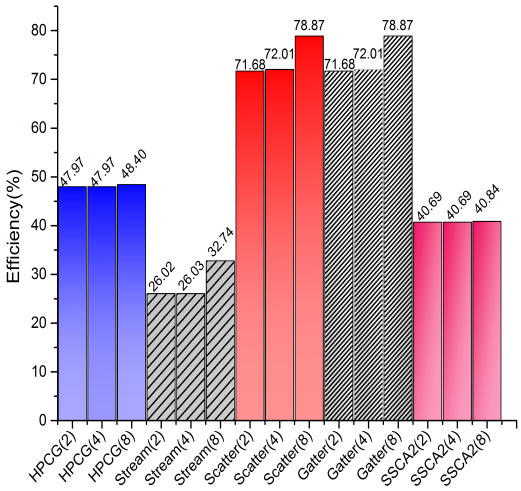

Figure 11: Scaled WPA results.

shows us that individual memory requests generated by these two tests tend to be unique and discrete in spatial address space. As such, they represent a much more difficult memory request pattern than that of the STREAM, scatter and gather test cases.

Similarly, the results of the WPA approach provide nearly identical results. Figure 11 demonstrates that there is little difference between the second and fourth threads and the overall efficiency peaks at roughly 6% for the STREAM, scatter and gather test cases. However, the overall efficiency for the HPCG and the SSCA2 tests demonstrate a slight increase of 0.15% and 0.43%, respectively.

The HMC request distribution of the APA algorithm executing the HPCG benchmark using 2 and 8 threads is illustrated in Figure 12 and Figure 13, respectively. As the concurrency scales from 2 to 8 threads, we observed that more HMC requests were generated for the maximum request size of 128-bytes. However, it is also apparent that the majority of the requests were coalesced into smaller 16-byte write requests (WR16). We attribute this behavior to application divergence to perform operations such as constructors, destructors, and system calls.

In addition to the aforementioned results, we present the raw data from the 8 thread simulations of the APA and WPA algorithms in Tables 2 and 3, respectively. In the APA approach, we achieved a maximum efficiency of 72.48% with the scatter and gather tests and the lowest efficiency in the STREAM test of 36.42%. Across all the test cases, we find an average increase in efficiency of 55.17%. Similarly, with the WPA approach, we maximize our efficiency with the scatter and gather tests at 78.86% and our lowest recorded efficiency with the STREAM test at 32.73%. Our average efficiency in the WPA approach is found to be 55.94%. As such, the WPA approach is found to be slightly more efficient across our diverse set of application test cases.

Given that the cost of memory requests are a first order performance bottleneck when accessing main memory, we also take into account the total costs of the HMC requests generated by the concurrent DMC unit. The HMC device architecture has a unique packet request structure for read and write requests that require an additional 32-bytes of control overhead for each memory request, regardless of the memory payload size. As such, the total cost of the HMC requests are calculated by adding the total control overhead and the total data overhead.

We measure the relative cost decrease and present data that represents the proportional decrease in memory request cost scaled from two to eight threads. Figure 14 elicits these results. In the APA approach, the largest overall cost decrease is 17.42% with the scatter and gather test results. Additionally, the minimum cost decrease is associated with the HPCG test with a cost decrease of 0.11%. The average cost decrease across all APA tests is 8.83%.

We also present the cost decrease results with the WPA approach. The total request costs of the scatter and gather tests decreased by 20.21%. However, the SSCA2 test only achieved a cost decrease of 0.25%. The average cost decrease across all WPA tests is 10.04% and, as such, the WPA approach is considered to be more efficient with respect to the overall cost decrease.

5. RELATED WORK

Given the inherent efficacy of modern data caches, memory coalescing is rarely utilized for core SoC designs. However, the topic about memory coalescing is extremely popular for improving the GPU memory performance. In order to achieve a faster parallel programming in GPU, reads and writes within each warp should be arranged sequentially [21], and the data is reordered in global memory in such a way that the words read by consecutive threads fall into consecutive address locations [22]. When consecutive threads access consecutive global memory region, then a single transaction may be implemented, and accesses are coalesced. This concept is fairly similar to our approach as implemented with GC64 architecture. Similarly, targeting at the providing a better performance for non-contiguous access pattern of

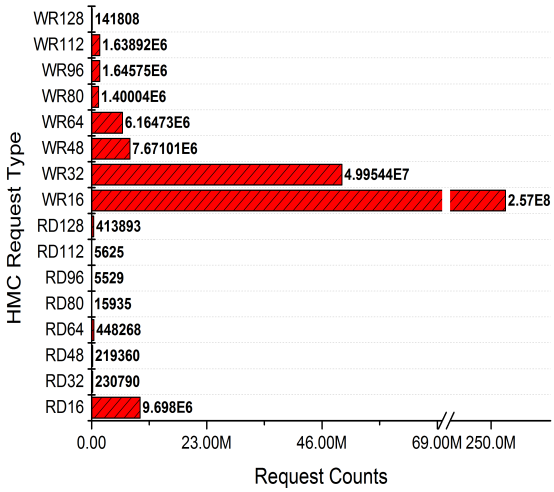

Figure 12: HMC request distribution with the APA algorithm and two threads.

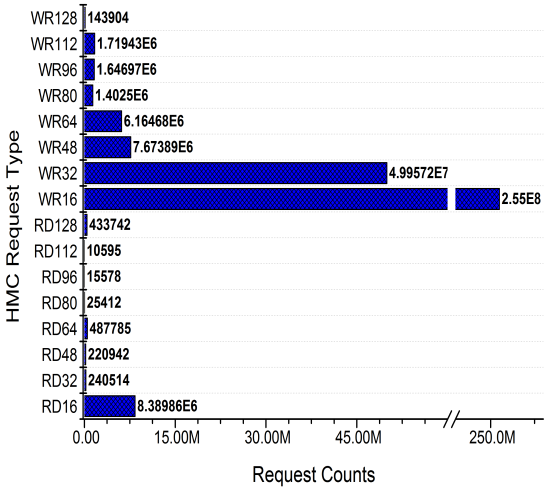

Figure 13: HMC request distribution with the APA algorithm and eight threads.

Table 2: Evaluation of APA with 8 threads

Algorithm	APA		
Test Case	Total input	Total output	Efficiency(%)
gather	608280	167388	72.481752
scatter	608280	167388	72.481752
hpcg	646840285	340632587	47.338996
ssca2	1936197717	1023581527	47.134452
stream	3292791	2093492	36.421959

Table 3: Evaluation of WPA with 8 threads

Algorithm	WPA		
Test Case	Total input	Total output	Efficiency(%)
gather	608280	128538	78.868613
scatter	608280	128538	78.868613
hpcg	646840285	333785917	48.397475
ssca2	1936197717	1145435847	40.840967
stream	3292791	2214737	32.739825

many scientific applications, there are also many researches are conducted based on hierarchical collective I/O scheduling to reduce the large number of I/O requests [23] [24].

In addition to analysis of the static transformation approach, additional research has been performed to enhance the coalesced access at runtime. Given that the static approach has been observed to be somewhat restricted to certain classes of applications, additional analysis has been done to optimize the memory accesses that fail to coalesce at runtime [25]. This trend of dynamic memory coalescing helps increase the efficacy of highly concurrent processing systems and will likely increase in popularity as more applications are ported and optimized for heterogeneous architectures.

6. CONCLUSION AND FUTURE WORK

In this work, we have presented a new methodology to the core dynamic memory coalescing approach using parallelized microcode that increases the overall efficacy and scalability of coalescing memory requests designed for one or more hybrid memory cube devices. The two approaches presented form a methodology that provides highly concurrent architectures, such as the GoblinCore-64, the ability to dynamically optimize incoming memory request traffic and make best use of available memory link bandwidth. We demonstrate the approach using several pathological kernels such as vector-scalar multiplication (in STREAM benchmark) and scatter/gather memory requests. We also demonstrate the approach using common application benchmarks in the form of STREAM, HPCG and SSCAv2.

The future direction of the research will focus on the extension of the research based on the architectural direction of the GoblinCore-64 project. The GC64 architecture will be expanded and improved as an open architecture for advanced data analytics. We will continue to utilize and expand our support for the RISC-V ISA and our use of hybrid memory cube devices as main memory. We will also continue to improve the efficacy of the DMC methodology by optimizing the microcode implementing the microcode directly in RISC-V assembly in order to remove as much of the microcode latency as possible.

7. ACKNOWLEDGMENT

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

8. REFERENCES

[1] John D. Leidel, Xi Wang, and Yong Chen. GoblinCore64: Architectural Specification. Technical report, Texas Tech University, September 2015.

[2] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0. Technical report, 2014.

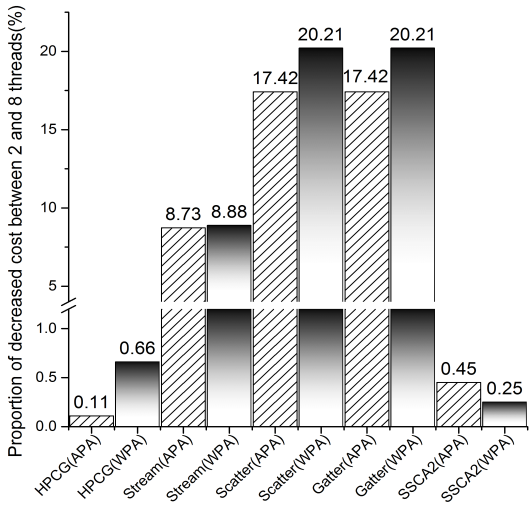

Figure 14: Proportion of the overall cost decrease of WPA and APA approaches.

[3] Rohit Chandra. *Parallel Programming in OpenMP*. Morgan kaufmann, 2001.

[4] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanovic. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pages 199–202. IEEE, 2014.

[5] Andrew Waterman. *Improving energy efficiency and reducing code size with RISC-V compressed*. PhD thesis, Master's thesis, University of California, Berkeley, 2011.

[6] Brian Zimmer, Yunsup Lee, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Ben Keller, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Hanh-Phuc Le, et al. A RISC-V vector processor with tightly-integrated switched-capacitor DC-DC converters in 28nm FDSOI. In *VLSI Circuits (VLSI Circuits), 2015 Symposium on*, pages C316–C317. IEEE, 2015.

[7] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. FlexPRET: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 101–110. IEEE, 2014.

[8] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, 2012.

[9] Maya Gokhale, Scott Lloyd, and Chris 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] Paul Rosenfeld, Elliott Cooper-Balis, Todd Farrell, Dave Resnick, and Bruce 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.

[11] Yinhe Han, Ying Wang, Huawei Li, and Xiaowei Li. Data-aware DRAM refresh to squeeze the margin of retention time in hybrid memory cube. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 295–300. IEEE Press, 2014.

[12] Mushfique Junayed Khurshid and Mikko Lipasti. Data compression for thermal mitigation in the Hybrid Memory Cube. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 185–192. IEEE, 2013.

[13] John D Leidel and Yong Chen. HMC-Sim: A Simulation Framework for Hybrid Memory Cube Devices. *Parallel Processing Letters*, 24(04):1442002, 2014.

[14] John Leidel. Hybrid Memory Cube Simulator 2.0. http://gc64.org/?p=137, December 2015.

[15] Maya Gokhale, Scott Lloyd, and Chris 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.

[16] Lifeng Nai and Hyesoon 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.

[17] Hybrid Memory Cube Specification 2.0. Technical report, July 2015.

[18] Toward a New Metric for Ranking High Performance Computing Systems. Technical report, Sandia National Laboratories, 2013.

[19] David Bader and Kamesh Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. *High Performance Computing : HiPC 2005*, 3769:465–476, 2005.

[20] John D. McCalpin. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers, 1995.

[21] Victor Podlozhnyuk. Black-Scholes option pricing, 2007.

[22] NSLP Kumar, Sanjiv Satoor, and Ian Buck. Fast parallel expectation maximization for Gaussian mixture models on GPUs using CUDA. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 103–109. IEEE, 2009.

[23] Jialin Liu, Yu Zhuang, and Yong Chen. Hierarchical collective i/o scheduling for high-performance computing. *Big Data Research*, 2(3):117 – 126, 2015. Big Data, Analytics, and High-Performance Computing.

[24] Yin Lu, Yong Chen, Yu Zhuang, Jialin Liu, and Rajeev Thakur. Collective input/output under memory constraints. 2014.

[25] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. Characterizing and enhancing global

memory data coalescing on gpus. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 12–22. IEEE Computer Society, 2015.