Concurrent Dynamic Memory Coalescing on Goblin-Core 64 Architecture

by

Xi Wang, B.S.E.E

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCES

Approved

Dr. Yong Chen
Chair of Committee

Dr. Noe Lopez-Benitez

Dr. Yu Zhuang

Mark Sheridan
Dean of the Graduate School

May, 2016

ACKNOWLEDGMENTS

I consider myself extremely fortunate to have encountered and worked with many remarkable people during the time I stay at Texas Tech University. While a brief note of thanks is not enough to show my gratitude for their impact on my life, I deeply appreciate their help with my work and life in Tech.

I am starting with my thesis committee members, Dr. Yong Chen, Dr. Yu Zhuang and Dr. Noe Lopez-Benitez by thanking them for all their help, guidance and suggestions during all the courses work and research.

First, I would like to thank my advisor Dr. Yong Chen. It was him who brought me to this supercomputing field and gave me the valuable chance to join this great and competitive project. He is such a kind and helpful advisor who helped me build my career plan and supported me on all the academic works and travels, like SC15 in Austin. I also can't be more thankful to him for appointing me as the research assistant position, especially, I am actually his first RA who is a master student. I cannot forget to mention his trust and appreciation to kindly provide me with the offer to be a doctoral student.

I also need to show my thanks to Dr. Zhuang, who taught me the parallel programming course. He is such a humorous and wise professor, who is also pleased and patient to discuss my ideas with me. I actually learned a lot from his class about the MPI, OpenMP and CUDA, which help me found the idea of my thesis and built this indispensable skills as a computer engineer.

Dr. Benitez is the first professor I met in Texas Tech, and his class on computer architecture is also the first class I registered in computer science department. I cannot thank more his passionate and interesting classes, which helped me build a strong interest in computer architecture design and potentially stimulated my motivation and desire to take the this field as a part of my future career.

Mr. John Leidel is also a very important teammate and also a friend that I have to show my gratitude. John is incredibly helpful and nice, providing the valuable

chance to work as an intern in Mircon. His comprehensive and knowledge in computer architecture and experienced skills in compiler and system level design gave me a very strong impression and he is also my role model who always drives me work hard. I felt so fortunate that I have such an excellent research teammate who is willing to guide me.

Additionally, I would like to devote my thanks for the support by this project of the National Science Foundation under grant CNS-1338078.

At last, I am going to finish this acknowledgment with my family in China. I would like to give all my love and thanks to my parents, my family and my girlfriend. Every time when I am down and feel so weary, they are always there support me, love me. Their love is always my power and motivation to work hard and polish up myself.

TABLE OF CONTENTS

ABSTRACT

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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.

We introduce the GC64 machine hierarchy in Figure 1-1. 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 network 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.

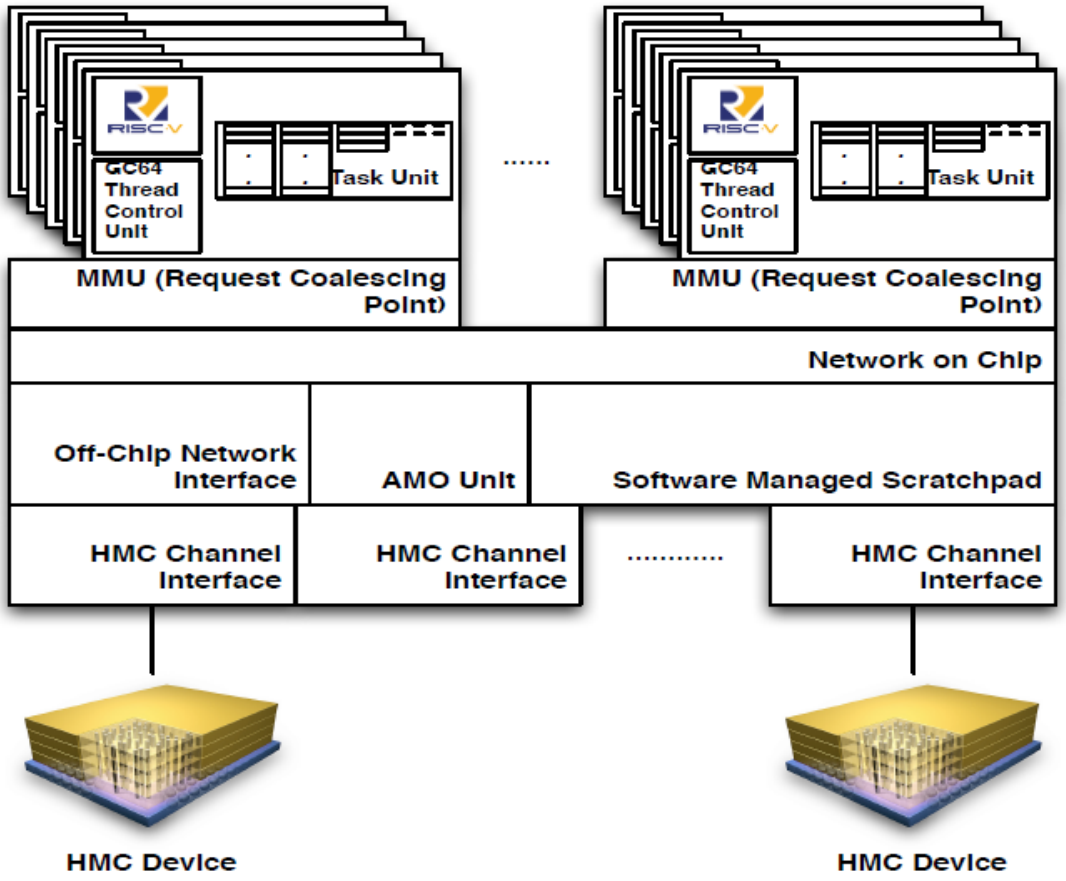

Figure 1-1 Architecture of GC 64

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 [1], 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.

In this research, with a concurrent dynamic memory coalescing unit design, two novel coalescing algorithms are designed and implemented. Also, five benchmarks and applications are utilized to make the comparison between these two algorithms, in order to demonstrate the efficacy of this concurrent model of memory coalescing.

1.1 Contributions

Our work makes the following contributions:

- We build the RISC-V cores on the Linux system, which contains the full tool-chain and compiler and spike simulator.
- We build the Yocto environment, which is a Linux distribution of RISC-V, and test the Symmetric Multi-Processing (SMP) on Yocto.
- We setup the Linux kernel and image on RISC-V, boot the system and have it running the program with OpenMP.
- We design the binary tree based dynamic memory coalescing (DMC) model, make it able to realize the logic of memory tracing and reduce the numbers of requests from RISC-V cores. We make it work as a DMC unit in serial version.
- We design two different parallel algorithms targeting the memory coalescing in Goblin-Core 64, which are implemented in C and OpenMP. Moreover, we evaluate this two algorithms regarding their efficiency compared with the serial DMC unit with all the test cases.

1.2 Organizations

This thesis is divided into six chapters. Following this introductory chapter, Chapter II presents background information on GoblinCore 64. Section 2.1 describes the background of RISC-V, section 2.2 describes the Hybrid Memory Cube, section 2.3 explains about the Architecture of GC64 and section 2.4 describes the previous work of this research.

Chapter III discusses the dynamic memory coalescing including the components of DMC unit, tree structure and the coalescing tree logic.

Chapter IV contains five sections. Section 4.1 introduces the concurrent dynamic memory coalescing. Sections 4.2 demonstrates the mathematical model of the concurrent DMC design. Sections 4.2 to 4.5 briefly introduce the test environment and API used to evaluate the algorithm and design.

Chapter V presents the algorithm design. Section 5.1 describes the Address Partitioned Algorithm (PAA), section 5.2 describes the Work Partitioned Algorithm (PWA).

Chapter VI compares the results of the two algorithms and five test cases. Chapter VII presents the conclusions we have drawn from our research and future work that can be done in this space.

CHAPTER 2

BACKGROUND

2.1 RISC-V

RISC-V (pronounced "risk-five") is an open source instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles.

In contrast to most ISAs, RISC-V is freely available for all types of use, permitting anyone to design, manufacture and sell RISC-V chips and software. While not the first open ISA, it is significant because it is designed to be useful in modern computerized devices such as warehouse-scale cloud computers, high-end mobile phones and the smallest embedded systems. Such uses demand that the designers consider both performance and power efficiency. The instruction set also has a substantial body of supporting software, which fixes the usual weakness of new instruction sets. The project was originated in 2010 by researchers in the Computer Science Division at UC Berkeley, but many contributors are volunteers and industry workers that are unaffiliated with the university [2].

The RISC-V ISA has been designed with small, fast, and low-power real-world implementations in mind [3][4], but without "over-architecting" for a particular microarchitecture style [4][5][6][7]. As of 2014 version 2 of the userspace ISA is fixed [8].

The RISC-V authors aim to provide several freely available CPU designs, under a BSD license. This license allows derivative works such as RISC-V chip designs to be either open and free like RISC-V itself, or closed and proprietary, (unlike the available OpenRISC cores, which under the GPL, requires that all derivative works also be open and free).

By contrast, commercial chip vendors such as ARM Holdings and MIPS Technologies charge substantial license fees for the use of their patents [9]. They also require non-disclosure agreements before releasing documents that describe their designs' advantages and instruction set. Many design advances are completely

proprietary, never described even to customers. The secrecy interferes with legitimate public educational use, security auditing, and the development of public, inexpensive open-source free software compilers and operating systems.

Developing a CPU requires expertise in several specialties: logic design, compiler design and operating system design. It is rare to find this outside of a professional engineering team. The result is that modern, high-quality general-purpose computer instruction sets have not recently been widely available anywhere or even explained except in academic settings. Because of this, many RISC-V contributors see it as a unified community effort. This need for a large base of contributors is part of the reason why RISC-V was engineered to fit so many uses.

The RISC-V authors also have substantial research and user-experience validating their designs in silicon and simulation. The RISC-V ISA is a direct development from a series of academic computer-design projects and was originated in part to aid such projects [4] [10] [11].

2.2 Hybrid Memory Cube

Hybrid Memory Cube (HMC) is a high-performance RAM interface for through-silicon via (TSV)-based stacked DRAM memory competing with the incompatible rival interface High Bandwidth Memory (HBM).

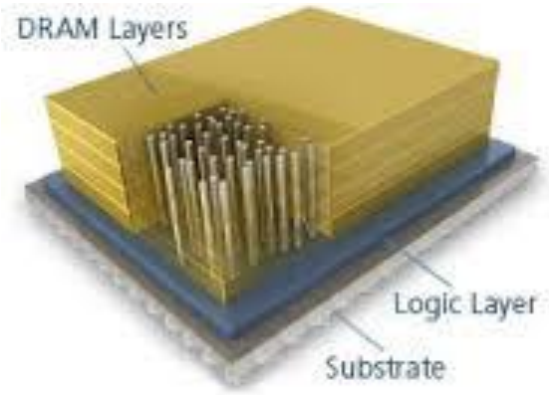

Figure 2-1 HMC architecture [18]

Hybrid Memory Cube was announced by Micron Technology in 2011[12] and promises a 15 times speed improvement over DDR3.[13] The Hybrid Memory Cube

Consortium (HMCC) is backed by several major technology companies including Samsung, Micron Technology, Open-Silicon, ARM, HP, Microsoft, Altera, and Xilinx.[14] HMC combines through-silicon via (TSV) and microbumps to connect multiple (currently 4 to 8) dies of memory cell arrays on top of each other [15]. The memory controller is integrated as a separate die [12]. HMC uses standard DRAM cells but it has more data banks than classic DRAM memory of the same size. The HMC interface is incompatible with current DDRn (DDR2 or DDR3) implementations [16] [17].

Within an HMC, as shown in Figure 2-2, memory is organized into vaults. Each vault is functionally and operationally independent. Each vault has a memory controller (called a vault controller) in the logic base that manages all memory reference operations within that vault. Each vault controller determines its own timing requirements. Refresh operations are controlled by the vault controller, eliminating this function from the host memory controller.

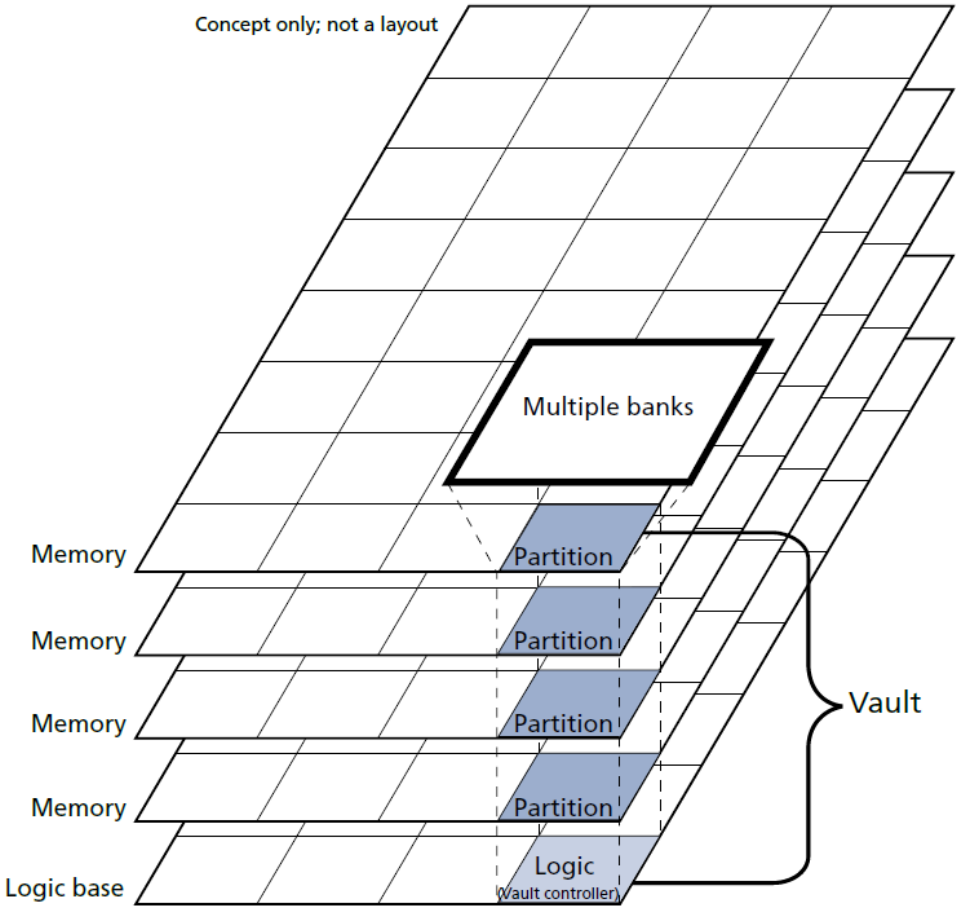

Figure 2-2 Example HMC Organization [18]

Each vault controller may have a queue that is used to buffer references for that vault's memory. The vault controller may execute references within that queue based on need rather than order of arrival. Therefore, responses from vault operations back to the external serial I/O links will be out of order. However, requests from a single external serial link to the same vault/bank address are executed in order. Requests from different external serial links to the same vault/bank address are not guaranteed to be executed in a specific order and must be managed by the host controller. [18]

One big advantage of Hybrid Memory Cube is its high bandwidth, the maximum read and write size per request is 128 bytes in HMC specification 1.0 [18] and 256 bytes in HMC specification 2.0 [19]. In our research, the specification 1.0 is

adopted. With 8 Links, the HMC can reach the bandwidth of 320GB/Cube, which is a fairly impressive performance in comparison with other traditional memory devices.

2.3 GoblinCore 64 Architecture

The GoblinCore64 (herein referred to as *GC64*) is originally designed to facilitate the construction of a high performance core architecture that was well- suited to executing applications traditionally known as “data intensive” These applications generally refer to algorithms that operate on sparse data structures such as graphs, sparse matrices and/or perform nonlinear combinatorial operations. We consider all of the aforementioned target application areas to share the following two general characteristics.

Non-Unit Stride: All of the applications we consider as design targets for GC64 perform a disproportionate number of non-unit stride computations. These computations may simply be non-unit stride, scatters, gathers or completely random. In all cases, the data elements are not generally well-suited to traditional long SIMD or data caching architectures.

Memory Intensive: Given the first characteristic, we also assume a latent characteristic with respect to the memory bandwidth requirements. Given the sparsity or non-linear access requirements, we assume that the design targets operate with a disproportionally high bandwidth to compute ratio. As such, we consider them to be memory intensive rather than computation intensive.

In addition to the core design requirements, we also sought to build a completely open source architecture and tool chain suitable for architectural research in academia and possible commercial implementations. As such, we sought to build BSD-like licensing around the core ISA, simulation infrastructure, tools and toolchain. Given this, we found that our implementation goals aligned well with the RISC-V project. These include, but are not limited to the following:

- A completely open ISA that is freely available to academia and industry.
- An ISA separated into a small base integer ISA.

- Support for the revised 2008 IEEE-754 floating-point standard.
- An ISA with native support for highly-parallel multicore or many core implementations.

In addition to the core RISC-V goals, we also wanted to achieve the following architectural goals (as related to our target design requirements):

- Provide simple architectural structures that are conducive to constructing highly (MIMD) parallel and concurrent applications
- Provide simple ISA extensions conducive to compiler optimization of concurrent applications
- Provide a low-level, mutable parallel construct in hardware that can be easily mapped to higher level parallel programming models (threads, tasks, etc.)
- Provide hardware mechanisms to minimize context switch latency to a very small number of cycles (goal of single cycle context switching events)
- Provide a well-defined mechanism when context switch events occur
- Provide a well-defined mechanism for user applications to explicitly induce context switch events

The GoblinCore64 project is sponsored by the Data Intensive Scalable Computing Laboratory in the Department of Computer Science at Texas Tech University [20].

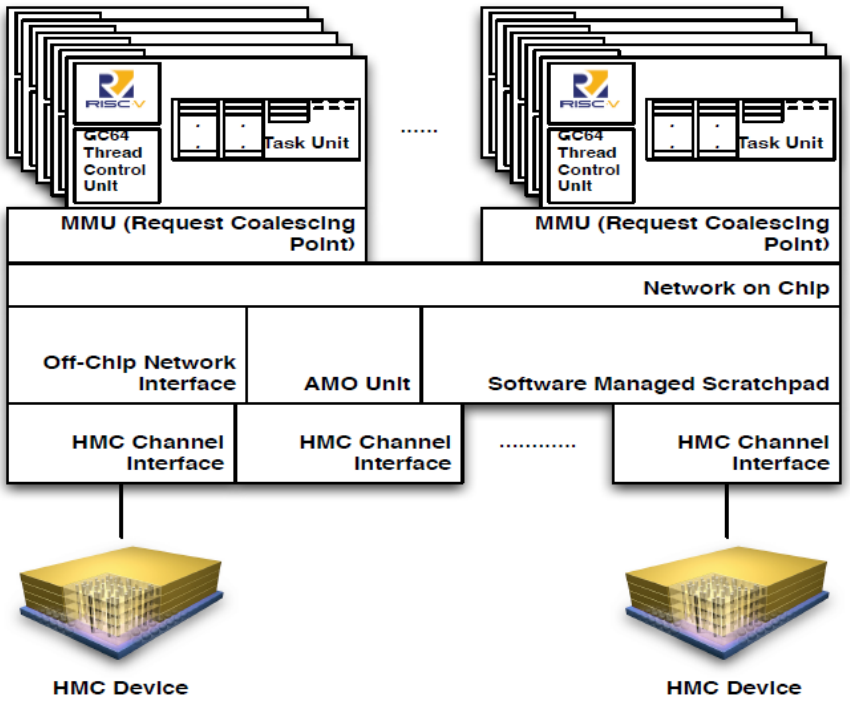

Figure 2-3 Architecture of GC 64

2.4 Previous Work

The following work has been done before this thesis research:

- Design the whole architecture of the GC-64, which is shown in Figure 2-3, to provide a scalable, flexible and open architecture for efficiently executing data intensive computing applications and algorithms.
- One extended simulator of RISC-V has been built to realize the support to scattering and gathering memory requests, task concurrency and task management.
- A serial dynamic memory coalescing unit is built as the memory management unit working in the GC64 architecture with HMC simulator built by our own
- The HMC simulator: HMC-Sim version 2.0 has been released, and it supports HMC specification 2.0 device layout and packet format, 256 byte

Read/Write packets, atomic operation and also adds the support for users to develop their own “Custom Memory Cube” (CMC) operations.

CHAPTER 3

DYNAMIC MEMORY COALESCING

Memory coalescing is defined as the act of merging two consecutive free blocks of memory. When an application frees memory, gaps can fall in the memory segment that the application uses. The purpose of dynamic memory coalescing is to coalesce the requests from processors to decrease the number of memory accesses. So that, the latency between the CPU and memory will be minimized. In this thesis research, the DMC structure consists of the following components as shown in the Figure 3-1.

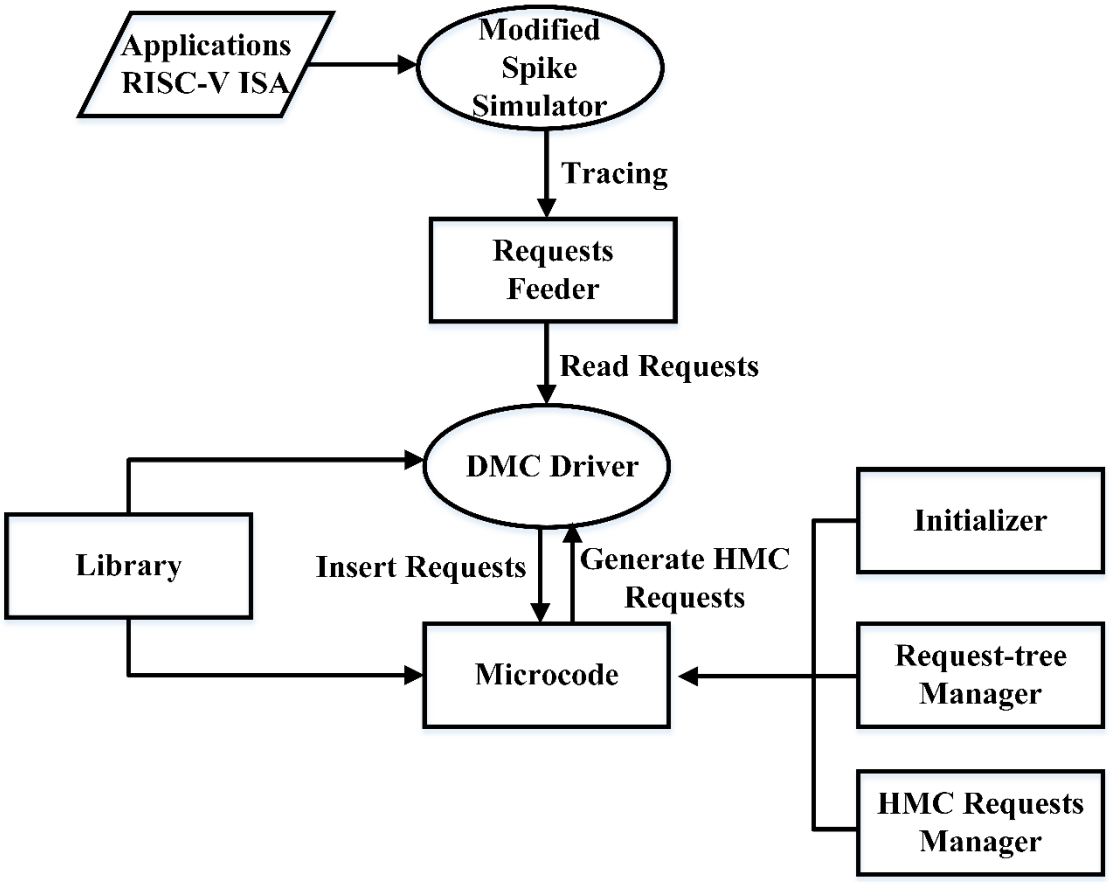

Figure 3-1 The component of DMC

First, the applications will be complied with the RISC-V instruction set architecture and then running on the modified spike simulator, which has been extended to support the memory tracing.

Second, after the generation of the memory requests are generated, requests feeder will feed the requests into the DMC driver with the standard input approach or read as a file according to the user's preference. DMC driver are responsible to operate the DMC logic, parallelized algorithms described in section 6 and insert the input requests by calling the microcode, which contains the initializer, request-tree builder and HMC request builder. Initializer provides the function of memory locality and variables initializations. After initialization, the request-tree manager will build the memory requests tree as required by DMC driver. Additionally, request-tree manager also takes charge of the tree management followed the rule of the tree logic demonstrated in section 4: coalescing tree logic.

Finally, HMC requests manager will build the HMC requests based on the memory requests tree and manages the HMC requests in the HMC list.

The library designed for the memory coalescing in GC-64, which contains the data structures including the tree structure of memory requests from the processor, application requests and HMC requests etc. In the library, the binary tree structures are used to store the requests from the processors. The following three tree structures are defined: Local Operation Tree, Global Operation Tree and Atomic Operation Tree. Each tree is built based on the following rules:

- Each tree contains at minimum of one root node that is null.
- All left nodes of the top-level root are for read operations.
- All right nodes of the top-level root are for write operations
- An exception to the rules is the AMO tree.

The trees are built as the structure shown in the Figure 3-2.

Each node in the binary tree contains the following data:

- Task operation, which could be Read requests or write requests, the minimum and maximum read / write bytes per request are 1 byte and 16 bytes respectively.
- Task ID, which is utilized to identify each specific requests.
- Address, the address of the data which is required by this request.

The requests from the processors were inserted into the tree in a sorting manner to make the binary tree structure to be a sorting binary tree, which means the left most child will store the smallest request address.

When the tree reaches the max read or write bytes limitation, which is 128 bytes per request, according to the HMC specification 1.0 from Micron[18], or the time that the request nodes stay in the tree exceeds the timeout predefined, the sorting binary tree will be expired, and form the HMC requests.

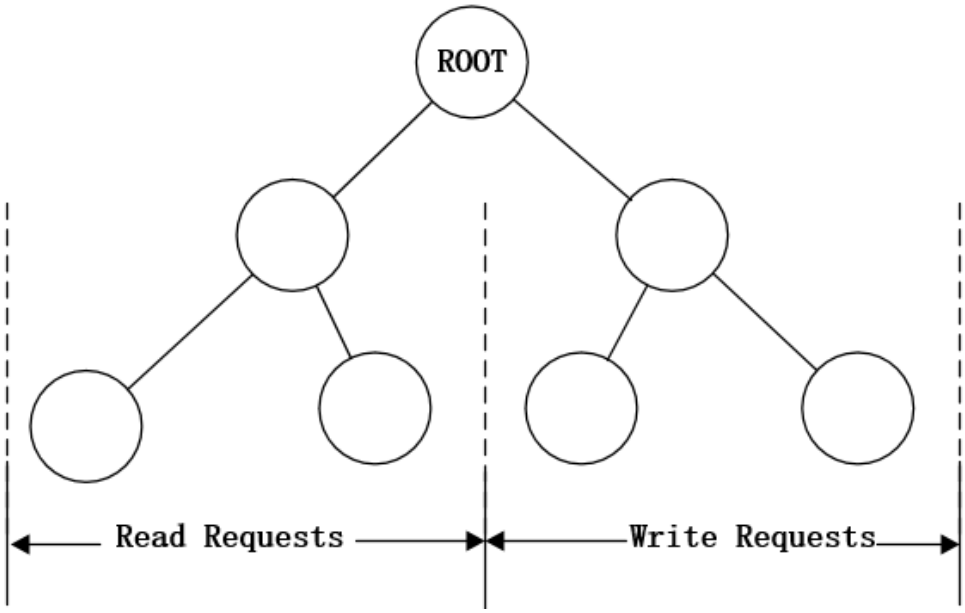

Figure 3-2 The tree structure

An HMC request is defined as a structure which contains:

- Memory operation, which could be a read request or a write request. The minimum and maximum read / write bytes per request are 16 bytes and 128 bytes;
- Address; the address of the data in the memory which is required by this request.

After one HMC request is built, a unique Traction ID (TID) will be attached to the HMC request in order to be used to identify different requests when receive the data back from the memory.

Eventually, the HMC requests will be converted into the HMC request packet [18] which is defined as the structure which consists of request header and request tail, and Cube ID, address, and TID.

3.1 Tree logic

In consideration of the fact that, there is no data cache in GC64, directly accessing the data from memory can potentially cause a high latency. Thus, the sorting binary trees are not just used for storing the request address, but also allows to reduce the number of the requests from the cores and build a small amount of HMC requests. Because HMC allows the user to read/write 128 bytes at a time, which is 8 times of the maximum read/write bytes from RISC-V cores per request. Further, the principle of the tree logic design is concluded as: less requests are more efficient. In this manner, the time used to access the memory will be reduced.

Once the tree reaches the condition that the expiration will be triggered, the left most child will be found as the base address first, and then its parent's request address and corresponding read/write bytes are evaluated, to determine whether these two requests are consecutive. Then the tree will be checked following the same tree logic.

The address of the read requests in the tree may have the following occasions which are shown in Figure 3-3 to 3-6.

In the case a, two requests address are completely consecutive, as shown in the Figure 3-3.

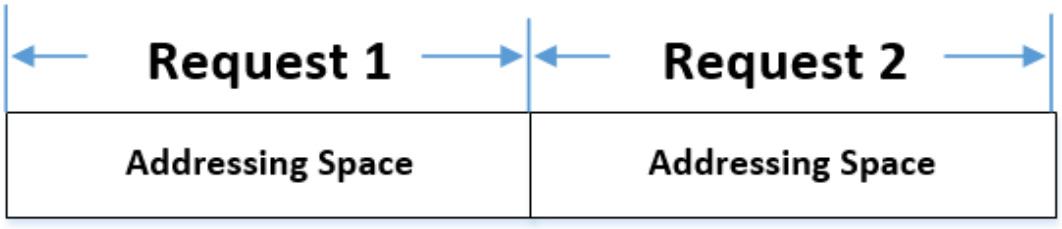

Figure 3-3 Case a

In the case b, addresses of request 1 and request 2 have an overlap, as shown in the Figure 3-4;

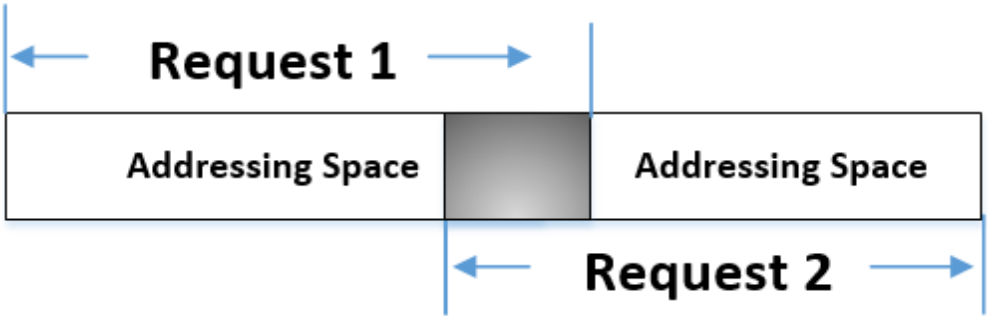

Figure 3-4 Case b

In the case c, as shown in the Figure 3-5, 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.

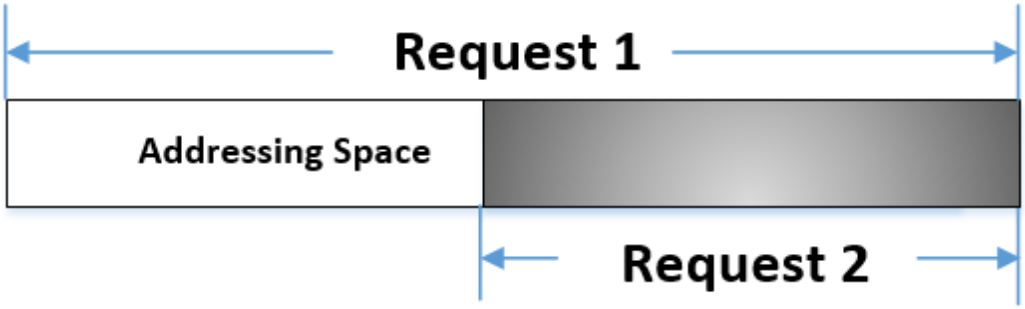

Figure 3-5 Case c

The above case a, b and c are considered as the consecutive occasions, which can be reduced as one HMC request if the total read bytes is not greater than 128 bytes.

As show in the Figure 3-6, 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, then request 1 and request 2 can still be formed in one HMC request. Otherwise, these two requests will be broke into two HMC requests.

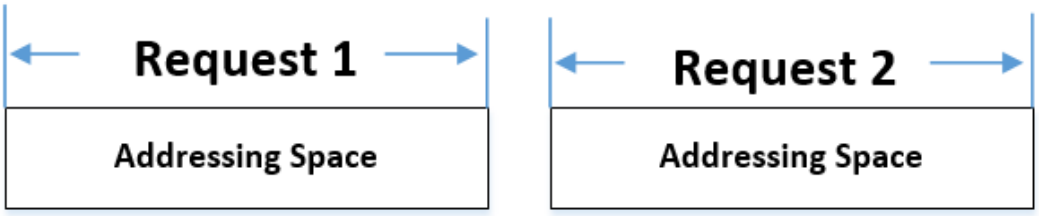

Figure 3-6 Case d

For the case of write requests, in the case a, b and c, these two requests can still be reduced as one HMC request. However, for the case d, the addresses of two requests are not consecutive, if only one HMC request is built, which will change the data in the memory that should not be changed. So, in this case, more than one HMC requests will be built.

This tree logic will run recursively until tracing back to the root node. Eventually, the tree will be expired, except the root node, which does not contain any data.

CHAPTER 4

CONCURRENT DESIGN

In this chapter, we introduce the design and methodology of concurrent dynamic memory coalescing, including the architecture of concurrent DMC, a mathematical modeling and two concurrent approaches to coalesce the memory accesses between the RISC-V cores and Hybrid Memory Cube in GC64 highly efficiently. It is originated from the current dynamic memory coalescing unit, which performs the coalescing logic and converts the requests from processors to the HMC requests serially. Additionally, the API and the environment for evaluation are also introduced in this chapter.

4.1 Concurrent tree-based memory tracing

The previous sections describe the dynamic memory coalescing and the logic of the tree. One limitation of this tree based memory coalescing is that when the application accesses the data totally randomly, even different hybrid memory cubes, the request from the processors could be totally separated, which will lead to the consequence that even after the memory coalescing, the number of requests will not be effectively reduced as expected.

For instance, let us suppose the following equation is the instruction in one application:

$$a[i] = b[i] + c[d[i]]$$

let n equal to the value of d[i], then c[d[i]] is equal to c[n], n will be a totally random numbers that are not consecutive. In this way, the addresses between different requests could be far away from each other, which may not be reduced into one HMC request, which means the total number of the HMC requests will increase, and the total latency will expand too. In order to solve this problem, tree based algorithm is optimized by making the memory coalescing unit work in parallel.

As widely acknowledged that the task parallelism is closely related to a divide-and-conquer approach, where a big problem is chopped into many sub-problems. The

sub-problems are often independent and parallelizable [21]. Following this manner, the tasks of dynamic memory coalescing are assigned to different threads based on range of the address of requests, which are independent. Each thread will build its local tree, which is shown in Figure 4-1.

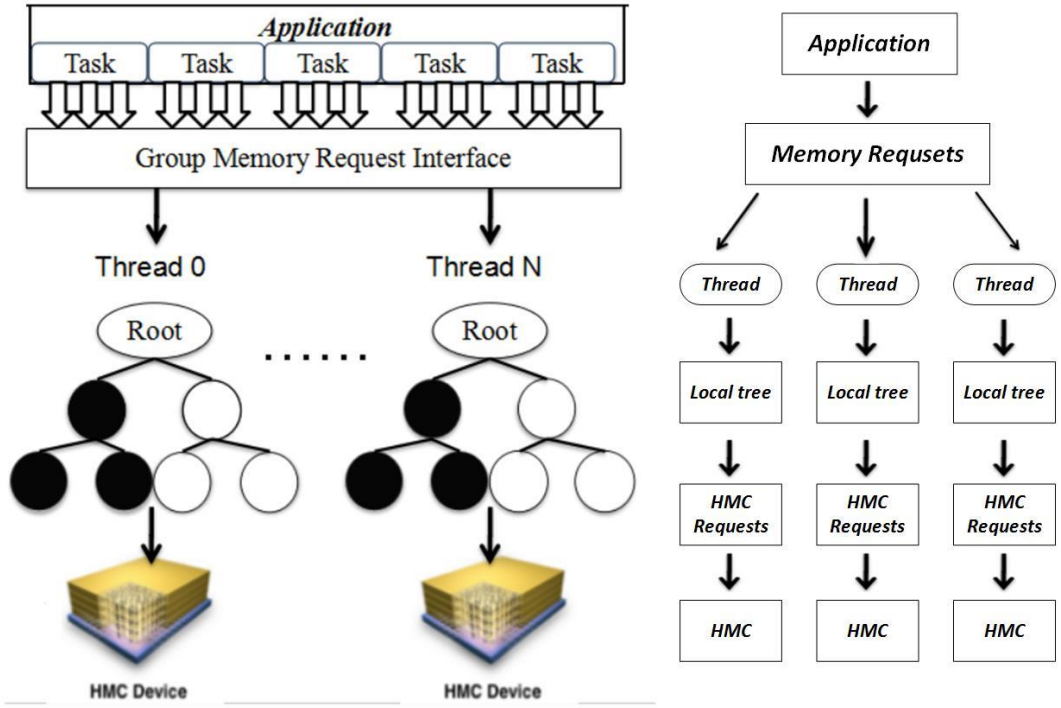

Figure 4-1 The architecture of concurrent DMC

4.2 Mathematical Modeling

Define r is the total number of requests produced by the processer, S_{hr} and C_{hr} are the number of HMC requests generated by the serial DMC and concurrent DMC, respectively. N is defined as the total number of different HMC devices, which is also the number of threads. P_k stands for the proportion of the requests in the k_{th} HMC device among all the requests read into the DMC unit, and follows the equation below:

$$\sum_{k=1}^{N} P_k = 1, k \in [1, N]$$

Suppose r is much larger than 128 bytes and all the requests accessing one HMC devices are consecutive. In this way, two cases are defined for modeling, which

are the best case and worst case. The serial DMC and concurrent DMC can be demonstrated by the equations in both cases.

- Best case: for each 128 bytes fed into the DMC unit, there exist both read and write requests accessing each HMC devices. In other words, for every coalescing tree, and $k \subseteq [1, N]$, $P_k > 0$. Due to the fact that the max read/write bytes in the memory coalescing tree is 128, the number of HMC requests targeting the address in each HMC will be two(One HMC read request and one HMC write request). Thus, we can form the following equations:

$$S_{hr} = \frac{r}{128} \times \sum_{k=1}^{N} P_k \times N \times 2 = \frac{r \times N}{64}$$

$$C_{hr} = 1 + \frac{r}{128} \times 2 = \frac{r}{64}$$

  As shown in the equation, $S_{hr} = C_{hr} \times N$. Therefore, in the best case the concurrent DMC is N times as efficient as serial DMC.

- Worst case: for each 128 bytes fed into the DMC unit, there exist both read and write requests targeting the same HMC device. In this way, in every coalescing tree, there will be 128 bytes requests targeting the same HMC device. Since the max read/write bytes in the memory coalescing tree is 128, the number of HMC requests generated by the each coalescing tree will be two (One HMC read request and one HMC write request). Thus we can form the following equations:

$$S_{hr} = 1 + \frac{r}{128} \times 2 = \frac{r}{64}$$

$$C_{hr} = 1 + \frac{r}{128} \times 2 = \frac{r}{64}$$

As shown in the above equation, $S_{hr} = C_{hr}$. So, in the worst case the concurrent DMC and serial DMC have the same performance in the perspective of the efficacy.

There are two concurrent approaches designed for the memory coalescing in this research.

- The first approach is to partition the addressing space of the memory and force each thread to take care of a corresponding partition. In this way, each thread will only build a local tree which contains the requests with the base address fall into its own partition. Further, this approach may also be extended for multiple hybrid memory cubes. For example, thread 1 will take care of the requests towards the address in the first HMC device and thread 2 will take care of the requests targeting the address in the second HMC device etc. In this manner, each process will only insert the request that requires the data in its own HMC device and the difference between different address of requests will be smaller. In this way, less HMC requests will be built based on the tree logic mentioned in the previous section. In other words, the efficiency are supposed to be increased with this improvement.
- The second approach is to insert the read requests and write requests separately. Some threads are used to insert the read requests and the rest threads are responsible for coalescing the write requests. This approach will increase the possibility of building larger HMC requests by partitioning the read and write requests. Additionally, this concurrent methodology may also take advantage of the PAA approach by assigning the specific addressing space to each thread.

4.3 Environment

An OpenEmbedded RISC-V port: Yocto is first considered as the platform to implement the parallelization, which is a Linux distribution generator. The Yocto

Project is a Linux Foundation workgroup whose goal is to produce tools and processes that will enable the creation of Linux distributions for embedded software that are independent of the underlying architecture of the embedded software itself. The project was announced by the Linux Foundation in 2010 [22]. In March 2011, the project aligned itself with OpenEmbedded, an existing framework with similar goals, with the result being The OpenEmbedded-Core Project.

The Yocto Project is an open source project whose focus is on improving the software development process for embedded Linux distributions. The Yocto Project provides interoperable tools, metadata, and processes that enable the rapid, repeatable development of Linux-based embedded systems.

The Yocto Project has the aim and objective of attempting to improve the lives of developers of customised Linux systems supporting the ARM, MIPS, PowerPC and x86/x86 64 architectures. A key part of this is an open source build system, based around the OpenEmbedded architecture which enables developers to create their own Linux distribution specific to their environment. This reference implementation of OpenEmbedded is called Poky.

There are several other sub-projects under the project umbrella which include EGLIBC, pseudo, cross-prelink, Eclipse integration, ADT/SDK, the matchbox suite of applications, and many others. One of the central goals of the project is interoperability among these tools.

The project offers different sized targets from "tiny" to fully featured images which are configurable and customizable by the end user. The project encourages interaction with upstream projects and has contributed heavily to OpenEmbedded-Core and BitBake as well as to numerous upstream projects, including the Linux kernel. The resulting images are typically useful in systems where embedded Linux would be used, these being single-use focused systems or systems without the usual screens/input devices associated with desktop Linux systems.

As well as building Linux systems, there is also an ability to generate a toolchain for cross compilation and a Software Development Kit (SDK) tailored to their own distribution, also referred to as the Application Developer Toolkit (ADT). The project tries to be software and vendor agnostic. Thus, for example, you can choose which package manager format you intend to use (deb, rpm, or ipk).

Within builds, there are options for various build-time sanity/regression tests, and also the option to boot and test certain images under QEMU to validate the build [23].

The Yocto for RISC-V developed by RISC-V team from UC Berkeley is fully installed and tested [24]. However, this environment was found that it did not support running the program with SMP, which is not as expected, even though it is able to compile the multithreading program.

4.4 Linux/RISC-V

This is a port of Linux kernel for the RISC-V instruction set architecture. Development is currently based on the 4.1 long term branch [25]. This Linux system built on RISC-V is able to run the program with installed API, like OpenMP, MPI etc. Thus, this platform is chosen to run the test cases for the results.

4.5 API: OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran [27], on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [26][28][29]. The structure of multithreading with OpenMP is shown in Figure 4-2.

OpenMP as an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The

threads then run concurrently, with the runtime environment allocating threads to different processors.

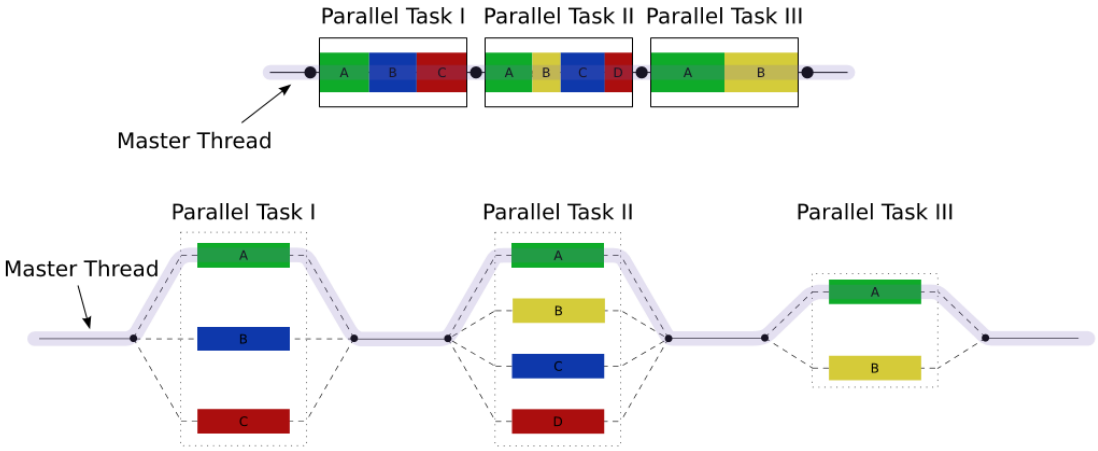

Figure 4-2 An illustration of multithreading with OpenMP [31]

The section of code running in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed [27]. Each thread has an id attached to it which can be obtained using a function (called omp_get_thread_num()), as shown in the Figure 4-3. The thread id is an integer, and the master thread has an id of 0. After the execution of the concurrent code, the threads join back into the master thread, which continues onward to the end of the program. OpenMP provides several ways to synchronize the threads like: critical, barrier, atomic. Specifically, the critical synchronize the threads by forcing only one thread is able to run the specific code segment; after that, another thread will be allowed to run this code segment. In this way, threads take turns to run this critical code segment. Moreover, OpenMP provides the keywords “private, thread private, first private, last private, shared, etc.” to distinguish the local variables and global variables. Private variables will be created inside of the parallel code segment for each threads, other threads cannot access these private variables of other threads. Shared variables will be shared for each threads and the variables in the serial codes are accessible by each thread. First private and last private variables will be initialized by the original assignment and written into the original variables, respectively. Thread

private variables will be private to the same thread through all the parallel code segments. It should be noticed that, the master thread's thread local variables will be the variables in the serial code.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way [31].

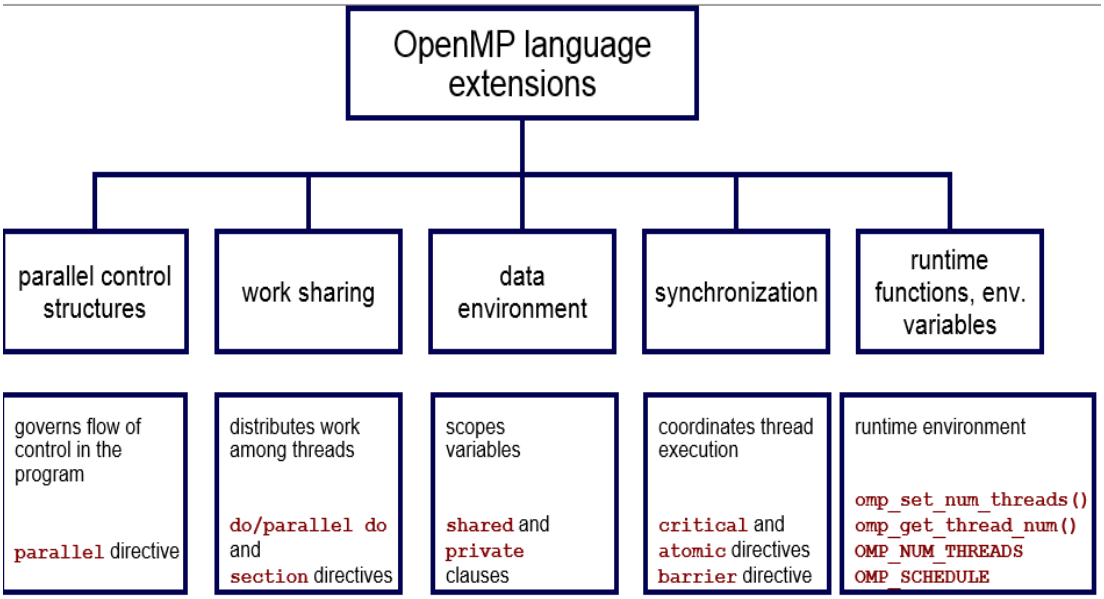

Figure 4-3 OpenMP constructs [31]

4.6 Implementation Plan

The OpenEmbedded RISC-V port Yocto is first considered as the platform to implement the concurrent DMC. However, this environment was found that it did not support running the program with SMP, even though it is able to compile the multithreading program.

Thus, instead of the Yocto, another port of Linux kernel for the RISC-V instruction set architecture which supports the SMP, is used as the environment to evaluate the concurrent DMC with multiple threads. After building the kernel and

image, the OpenMP programming model and library [30] are adopted in this research to parallelize the memory coalescing.

CHAPTER 5

ALGORITHM DESIGN

We design two different concurrent coalescing algorithms for the concurrent DMC methodology, a partitioned address algorithm and a partitioned work algorithm. 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.

5.1 Partitioned Address Algorithm

In the partitioned address algorithm (PAA), as shown in the Figure 5-1, each thread begin running in parallel in order to read the input memory trace after all the private copies are created. 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:

$$TID \times \alpha \leq address \leq (TID+1) \times \alpha$$

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

execution 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
```

Figure 5-1 Partitioned Address Algorithm

Table 6-1 Example of input requests

Request	Operation	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

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 5-2.

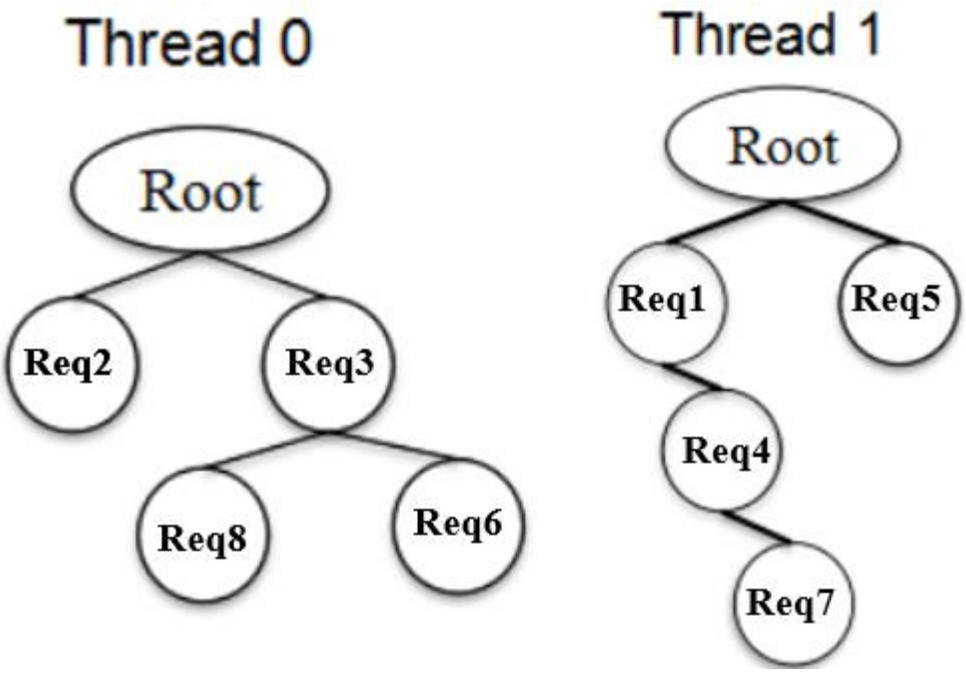

Figure 5-2 The example of PAA

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 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.

5.2 Partitioned Work Algorithm

As shown in the Figure 5-3, the partitioned work algorithm (PWA) assigns the address spaces to the individual thread ranks in a similar manner to the PAA 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 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
```

Figure 5-3 Partitioned Work Algorithm

For the threads with the TID greater than or equal to γ, the threads will only insert the write requests which meets the following condition, as shown in the relation:

$$(TID - \gamma) \times \beta \leq addr \leq (TID - \gamma + 1) \times \beta$$

For the threads with TID smaller than γ, the threads will only insert the read requests which meets the following condition, as shown in the following relation:

$$TID \times \beta \leq addr \leq (TID + 1) \times \beta$$

For instance, assume there are sixteen threads are used and β is 0x01000000. There are also eight requests which are read and write, as listed 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. And an example are shown in the Figure 5-4 to demonstrate all the requests listed in the table 1 inserted by algorithm PWA.

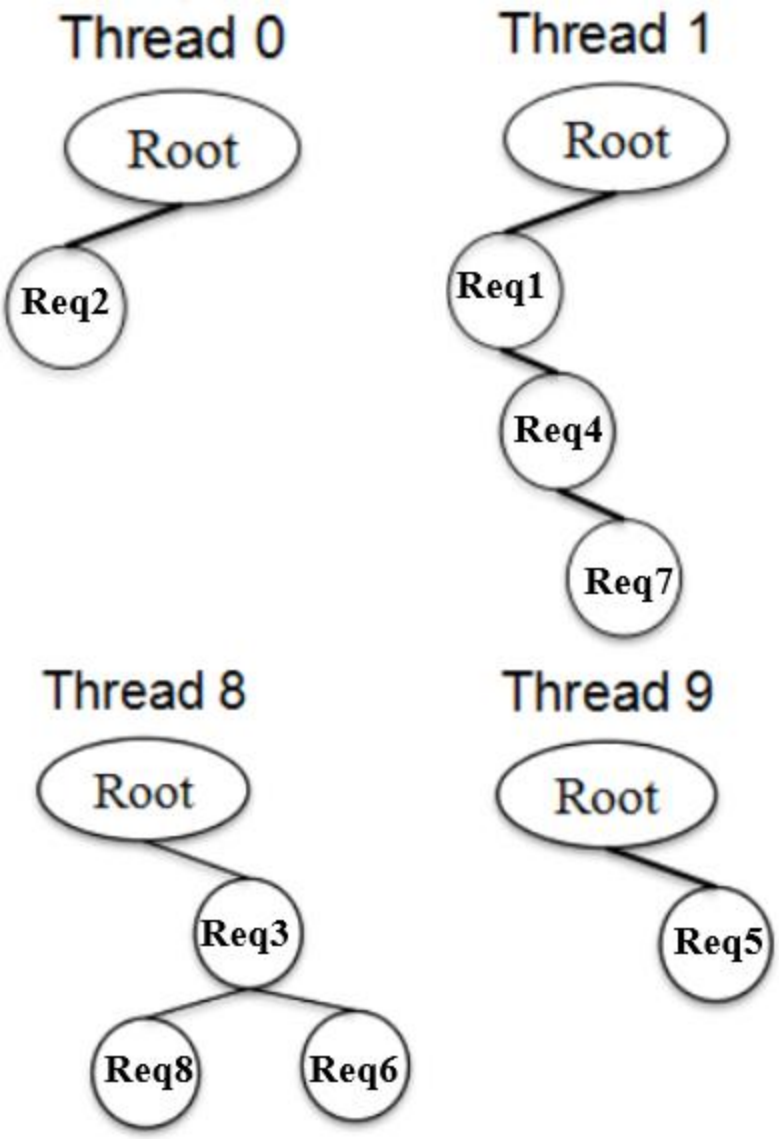

Figure 5-4 Example of PWA

CHAPTER 6

EVALUATION

There are 5 test cases are used to make comparison between the two concurrent DMC algorithms and the serial DMC on the perspective of efficiency.

- The first one is the High Performance Conjugate Gradient Benchmark, also known as HPCG, which is a software package that performs a fixed number of symmetric Gauss-Seidel preconditioned conjugate gradient iterations using double precision (64 bit) floating point values [32].
- The 2rd test case is a synthetic graph theory benchmark called SSCA#2 which is developed by the DARPA High Productivity Computer Systems (HPCS). This benchmark is composed of four kernels operating on a large-scale, directed multi-graph [34].
- The 3rd test case is the stream benchmark, which is designed to measure sustainable memory bandwidth for contiguous, long-vector memory accesses [35].
- 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 PAA and the work partitioned algorithm PWA, 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 PAA and the PWA approach are presented in Figure 6-1 and Figure 6-2, respectively.

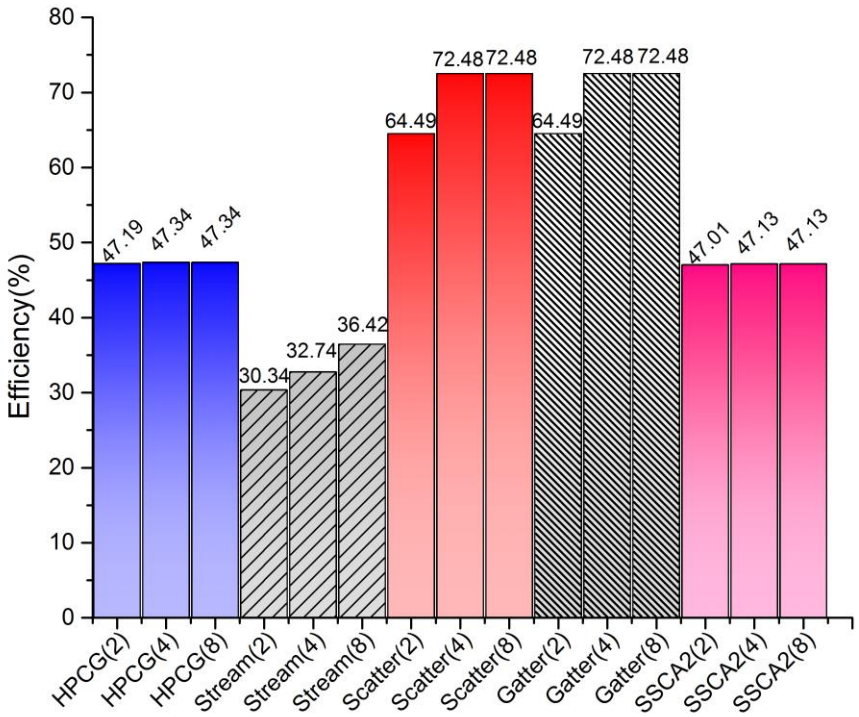

Figure 6-1 PAA Results

As shown in Figure 6-1 the PAA 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 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 PWA approach provide nearly identical results. Figure 6-2 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.

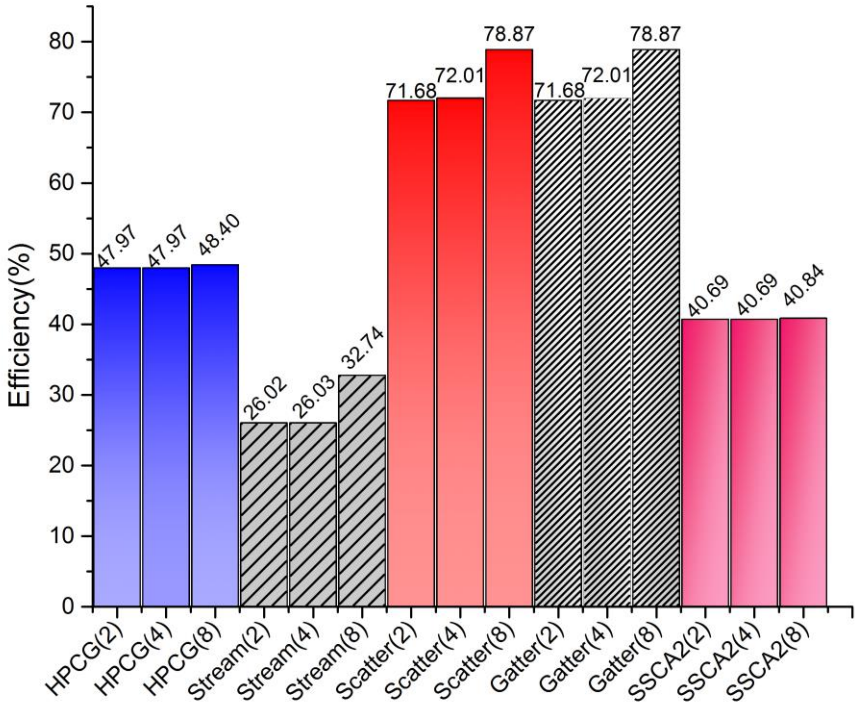

Figure 6-2 PWA Results

The HMC request distribution of the PAA algorithm executing the HPCG

benchmark using 2 and 8 threads is illustrated in Figure 6-3 and Figure 6-4, 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 PAA and PWA algorithms in Tables 2 and 3, respectively. In the PAA 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 PWA 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 PWA approach is found to be 55.94%. As such, the PWA approach is found to be slightly more efficient across our diverse set of application test cases.

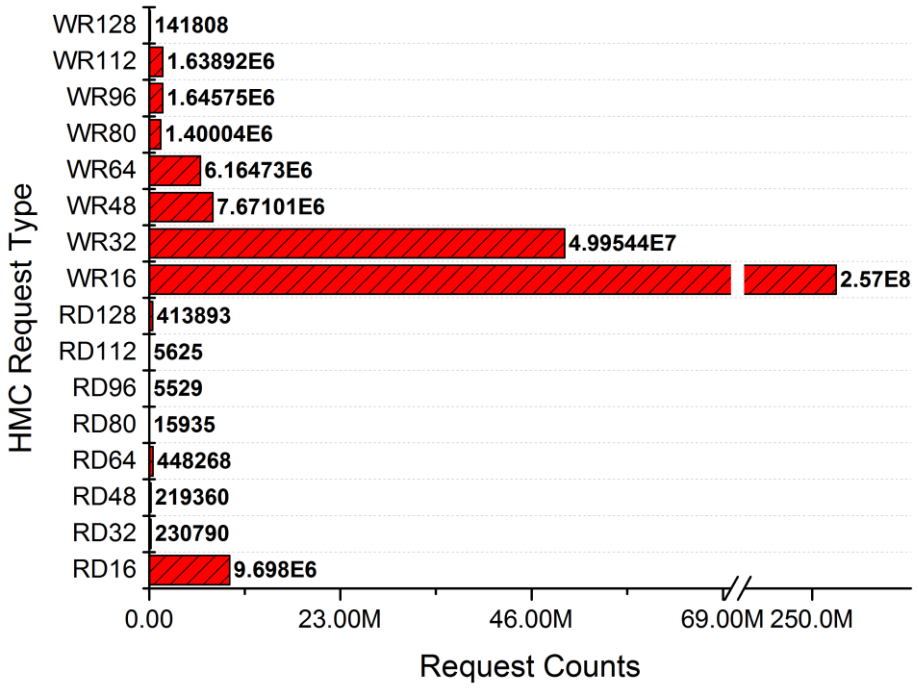

Figure 6-3 HMC request distribution with the PAA algorithm and two threads

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.

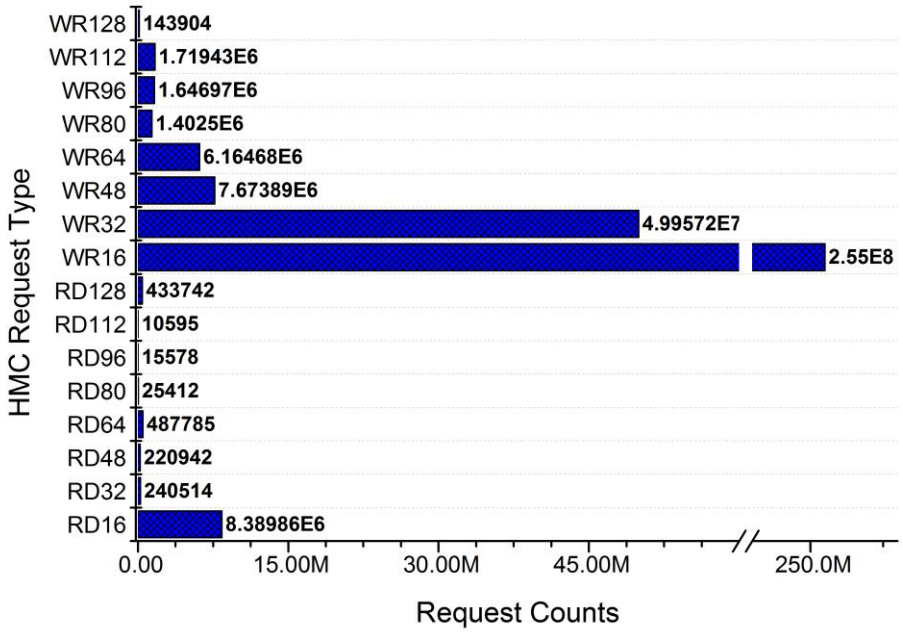

Figure 6-4 HMC request distribution with the PWA algorithm and two threads

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 6-5 elicits these results. In the PAA 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 PAA tests is 8.83%.

We also present the cost decrease results with the PWA 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 PWA tests is 10.04% and, as such, the PWA approach is considered to be more efficient with respect to the overall cost decrease.

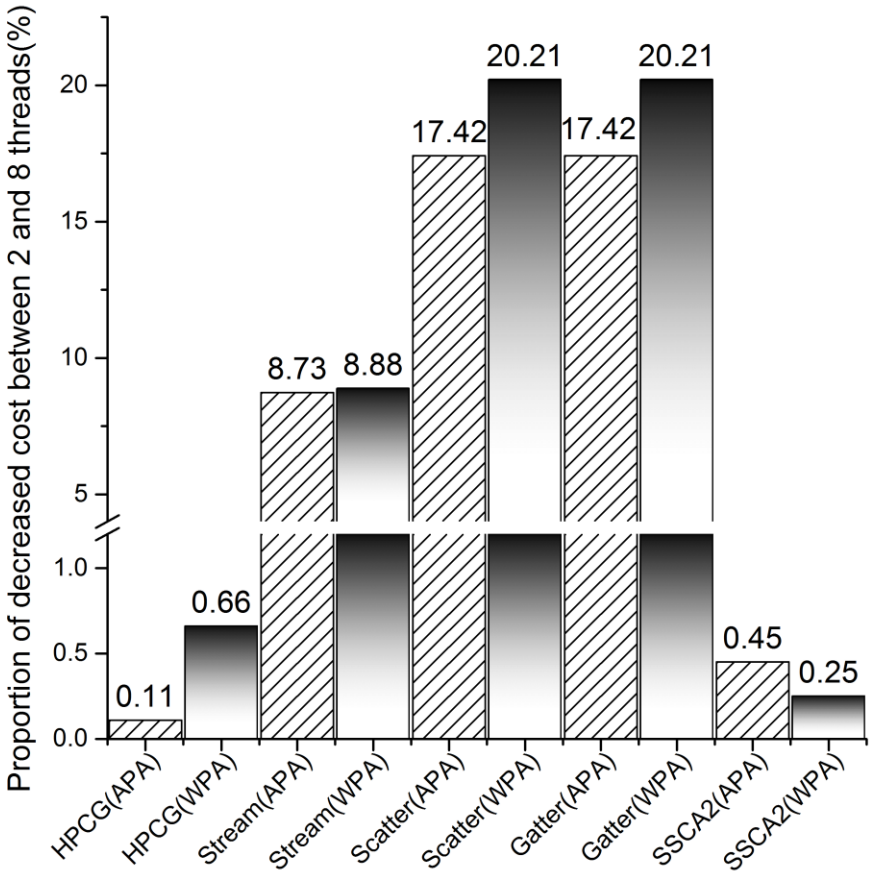

Figure 6-5 Proportion of the overall cost decrease of PWA and PAA approaches

Table 6-1 Evaluation of PAA with 8 threads

Algorithm	PAA		
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 6-2 Evaluation of PWA with 8 threads

Algorithm	PWA		
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

CHAPTER 7

CONCLUSION

In this work, we have presented a new methodology to the core dynamic memory coalescing approach using concurrent 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 [36].

BIBLIOGRAPHY

[1] Chandra R. Parallel programming in OpenMP[M]. Morgan kaufmann, 2001.

[2] riscv.org. Regents of the University of California. Retrieved August 25, 2014.

[3] "Rocket Core Generator". RISC-V. Regents of the University of California. Retrieved 1 October 2014.

[4] Waterman, Andrew, et. al. "The RISC-V Instruction Set Manual Vol. I, User-Level ISA, version 2.0". RISC-V Downloads. Regents of the University of California. Retrieved 2014-08-25.

[5] Celio, Christopher; Love, Eric. "ucb-bar/riscv-sodor". GitHub Inc. Regents of the University of California. Retrieved 12 February 2015.

[6] "SHAKTI Processor Project". Indian Institute of Technology Madras. Retrieved September 15, 2014.

[7] Celio, Christopher. "CS 152 Laboratory Exercise 3" (PDF). UC Berkeley. Regents of the University of California. Retrieved 12 February 2015.

[8] Waterman, Andrew; Lee, Yunsup; Patterson, David A.; Asanovi, Krste. "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2 (Technical Report EECS-2014-54)". University of California, Berkeley. Retrieved December 26, 2014.

[9] Demerjian, C. (2013). “A long look at how ARM licenses chips: Part 1” semiaccurate.com/2013/08/07/a-long-look-at-how-arm-licenses-chips, “How ARM licenses it’s IP for production: Part 2” semiaccurate.com/2013/08/08/how-arm-licenses-its-ip-for-production.

[10] Asanovic, Krste. "Instruction Sets Should be Free: The Case for RISC-V". RISC-V. Regents of the University of California. Retrieved 2014-08-25.

[11] RISC-V on Wikipedia, https://en.wikipedia.org/wiki/RISC-V#cite_ref-sodor_4-0

[12] Micron Reinvents DRAM Memory // Linley group, Jag Bolaria, 12 September 2011

[13] Mearian, Lucas (2013-09-25). "Micron ships Hybrid Memory Cube that boosts DRAM 15X". computerworld.com. Computerworld. Retrieved 2014-11-04.

[14] Microsoft backs Hybrid Memory Cube tech // by Gareth Halfacree, bit-tech, 9 May 2012

[15] Hybrid Memory Cube (HMC), J. Thomas Pawlowski (Micron) // HotChips 23

[16] Memory for Exascale and ... Micron's new memory component is called HMC: Hybrid Memory Cube by Dave Resnick (Sandia National Laboratories) // 2011 Workshop on Architectures I: Exascale and Beyond, 8 July 2011

[17] Hybrid Memory Cube (HMC) in Wikipedia: https://en.wikipedia.org/wiki/Hybrid_Memory_Cube [Accessed 3 February 2016].

[18] HMC Specification 1.0 from Micron, http://www.hybridmemorycube.org/ [Accessed 10 October 2015].

[19] HMC Specification 2.0 from Micron, http://www.hybridmemorycube.org/ [Accessed 15 February 2016].

[20] The overview of GoblinCore-64, http://gc64.org/?page_id=21' [Accessed 15 February 2016].

[21] Phan, Anh-Dung, and Michael R. Hansen. "An approach to multicore parallelism using functional programming: A case study based on Presburger Arithmetic." Journal of Logical and Algebraic Methods in Programming 84.1 (2015): 2-18.

[22] Yocto Project Aligns Technology with OpenEmbedded and Gains Corporate Collaborators. http://www.linuxfoundation.org/news-media/announcements/2011/03/yocto-project-aligns-technology-openembedded-and-gains-corporate-co

[23] Yocto on Wikipedia, https://en.wikipedia.org/wiki/Yocto_Project [Accessed 1 February 2015].

[24] Yocto/OpenEmbedded RISC-V Port. http://riscv.org/software-tools/yocto/ [Accessed 15 December 2015].

[25] Linux/RISC-V, Linux kernel port of RISC-V. http://riscv.org/software-tools/linux/ [Accessed 20 December 2015].

[26] OpenMP Compilers". *OpenMP.org. 2013-04-10*. Retrieved 2013-08-14.

[27] *Gagne, Abraham Silberschatz, Peter Baer Galvin, Greg. Operating system concepts (9th ed.). Hoboken, N.J.: Wiley. pp. 181–182.* ISBN 978-1-118-06333-0.

[28] OpenMP Tutorial at Supercomputing 2008 http://openmp.org/wp/2008/10/openmp-tutorial-at-supercomputing-2008/

[29] Using OpenMP – Portable Shared Memory Parallel Programming – Download Book Examples and Discuss. http://openmp.org/wp/2009/04/download-book-examples-and-discuss/

[30] Muddukrishna, Ananya, et al. "Grain Graphs: OpenMP Performance Analysis Made Easy." 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'16). Association for Computing Machinery (ACM), 2016.

[31] OpenMP on Wikipedia, https://en.wikipedia.org/wiki/OpenMP [Accessed 15 February 2016].

[32] Dongarra J, Heroux M A, Luszczek P. A new metric for ranking high performance computing systems[J]. National Science Review, 2016: nwv084.

[33] Graph 500 Benchmark http://www.graph500.org/specifications [Accessed 9 March 2016].

[34] J. Kepner, D. P. Koester, and et al. HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.0, April 2005.

[35] McCalpin J D. A survey of memory bandwidth and machine balance in current high performance computers[J]. IEEE TCCA Newsletter, 1995: 19-25.

[36] Xi Wang, John D. Leidel, Yong Chen, Concurrent Dynamic Memory Coalescing on GoblinCore-64 Architecture, under review of MEMSYS 2016.