An Open Source Non-Blocking Manycore L2 Cache

Kangli Li

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2024

Committee:

Michael Taylor (Chair)
Scott Hauck

Program Authorized to Offer Degree:

Electrical and Computer Engineering

Acknowledgement

First, I would like to thank my parents, Lixin Chen and Tianming Li, and my dog PP for their tremendous support. Without their support and love, I would not have come such a long way and successfully completed my graduate studies.

I would like to thank my thesis advisor, Professor Michael Taylor. As I reflect on my master's thesis journey, I am filled with immense gratitude for his unwavering support and guidance. His mentorship has been a cornerstone of my academic growth, shaping my approach to doing research.

I would also like to extend my gratitude to Professor Scott Hauck for serving on my thesis defense committee. His insightful questions and constructive feedback have been invaluable in refining my research and enhancing its overall quality.

I would also like to thank everyone in the Bespoke Silicon Group, where I truly feel like part of a family. I am especially grateful to Tommy Jung, Scott Davidson, and Huwan Peng for their invaluable help and advice. They were always willing to spend time explaining even the most basic concepts to me and consistently provided valuable insights.

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]), ASIC Clouds ([19], [20], [21], [22], [23], [24], [25], [26]), open source hardware ([27], [28], [29]) RISC-V ([30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]), Network-on-Chips ([42], [43], [44], [45], [46]), security ([47], [48], [49], [50]), benchmark suites ([51], [52], [53]), dark silicon ([54], [55], [54], [56], [57], [58], [15], [18]), multicore ([30], [59], [60], [61], [62], [45], [63], [64], [65], [46], [66], [67], [68], [69]), compiler tools ([70], [71], [72], [73], [74], [75], [76], [77], [78], [79]) and FPGAs ([80], [81], [53]).

University of Washington

Abstract

An Open Source Non-Blocking Manycore L2 Cache

Kangli Li

Chair of the Supervisory Committee:

Michael Taylor

Computer Science and Engineering

This thesis presents the RTL implementation and evaluation of a non-blocking L2 victim cache for the open-source HammerBlade manycore architecture. The primary objective of this research is to address the inefficiencies associated with blocking caches, which often result in network congestion and reduced performance. By transitioning to a non-blocking cache design, we aim to improve memory system efficiency, increase concurrency, and reduce delays. The proposed non-blocking cache incorporates features such as Miss Status Holding Registers (MSHR), Read Miss Queue, and advanced Direct Memory Access (DMA) system. Verification and performance evaluation are conducted using a suite of ten Single Program, Multiple Data (SPMD) benchmark programs. The results demonstrate significant performance improvements, highlighting the effectiveness of the non-blocking cache in enhancing the overall efficiency and throughput of the HammerBlade architecture.

Contents

1 Introduction

Recent advancements in computer architecture have increasingly focused on low-power designs and the development of specialized accelerators to maximize energy efficiency. With growing demands for reduced power consumption, the strategic and efficient use of limited, yet speedy, local SRAM caches has become more crucial than ever.

Another significant trend is the open-source hardware movement. Over the past few decades, open-source software projects like Linux and GCC have revolutionized the software industry, providing free and open tools that are indispensable in today's technology landscape. Despite the lack of a fully open-source pathway for ASIC development due to the reliance on proprietary EDA tools, efforts like BaseJump STL by University of Washington aim to simplify and accelerate the design process by providing a library of reusable SystemVerilog modules.

The research focus has also shifted towards manycore processors recently due to the performance limitations of single-core systems. [82] In the rapidly evolving landscape of computing, manycore architectures have emerged as a cornerstone for high-performance computing. Among these, the HammerBlade architecture from the University of Washington, an open-source platform designed for efficient computation on general-purpose workloads, stands out with its unique composition and capabilities. While the manycore architecture like HammerBlade offers significant performance advantages [83] , it also presents distinct challenges and opportunities, particularly in the realm of cache management and network efficiency.

This thesis builds on previous work of the blocking cache system in HammerBlade by Dai Cheol Jung [37] , focusing on transforming the L2 victim caches from blocking to non-blocking arrays on RTL level. The goal of this transformation is to address the inherent inefficiencies of blocking caches, particularly their tendency to cause network congestion. By introducing a non-blocking cache design, this work aims to improve memory system efficiency, increase concurrency, and reduce delays, ultimately enhancing the overall performance of the HammerBlade architecture.

The non-blocking cache design proposed in this thesis incorporates several features to manage cache misses more efficiently. These include the use of a Miss Status Holding Register (MSHR) queue, Read Miss Queue, multiple Miss Handling Units (MHUs), and an advanced Direct Memory Access (DMA) system. By ensuring that primary and secondary cache misses are handled smoothly, the new design maintains continuous network operations, thereby mitigating congestion issues.

Verification and evaluation of the new non-blocking cache system are critical components of this research. This thesis employs a combination of isolated unit tests, hand-written regression tests, and constrained random testing to ensure the correctness and robustness of the cache design. Performance improvements are demonstrated using ten "Single Program, Multiple Data" (SPMD) benchmark programs, showcasing significant gains in both efficiency and resource utilization.

The transition from blocking to non-blocking cache arrays in HammerBlade marks a significant step towards optimizing manycore architectures for high-performance computing. By addressing the challenges of network congestion and enhancing cache efficiency, this work contributes to the broader goal of advancing manycore processor design and research within the open-source and RISC-V community.

2 Background

2.1 BaseJump STL

BaseJump STL [28] is an open-source standard template library written in SystemVerilog, designed to provide a comprehensive collection of reusable hardware building blocks for ASIC design. The library aims to drastically reduce RTL development and verification time by offering highly composable and parameterizable modules. These modules follow a consistent coding style guideline, ensuring portability and synthesis across various EDA tools. BaseJump STL includes components for memory, dataflow, asynchronous operations, network-on-chip implementations, and more, facilitating rapid design-space exploration and enhancing design efficiency. These modules are designed to be highly composable and parameterizable, significantly reducing the time required for hardware development and verification. By leveraging BaseJump STL, developers can quickly assemble and customize hardware components, ensuring consistency and efficiency across different projects.BaseJump STL modules are used heavily in this work, and many new modules are created as part of this work.

https://github.com/bespoke-silicon-group/basejump_stl

2.2 HammerBlade

The HammerBlade manycore architecture, developed by BSG group at the University of Washington, is an open-source, scalable manycore processor designed for high-performance and energy-efficient computing. HammerBlade is a tiled manycore processor. Tile architecture first originated from MIT Raw project [62] . The architecture features a 2D mesh network that interconnects a large array of tiles, each tile containing a RISC-V core, local memory, and a router. Processors can send packets to remote locations to access other tiles' data memory or shared DRAM. Network Physical Address (NPA), which consists of a (x,y) coordinate and an endpoint physical address (EPA), maps the entire address space that can be reached on the network. Between the network and the shared DRAM, there is a row of the L2 caches. These caches are connected to the router at the bottom of each column. This design supports efficient execution of parallel workloads, with a focus on minimizing memory latency and maximizing throughput.

The HammerBlade manycore utilizes several key components from the BaseJump STL library, such as memory controllers, network interfaces, and dataflow elements, to optimize its performance.

https://github.com/bespoke-silicon-group/bsg_manycore

3 Vanilla Core

HammerBlade is built using interconnected pods. Each pod is like a mini-computer made up of a grid of tiles. There are two kinds of tiles: compute tiles and vcache tiles. Compute tiles are the workhorses, containing processors and network connections. Vcache tiles, on the other hand, are all about data access. They act as a special kind of memory cache, helping the processors quickly retrieve information from the off-chip DRAM.

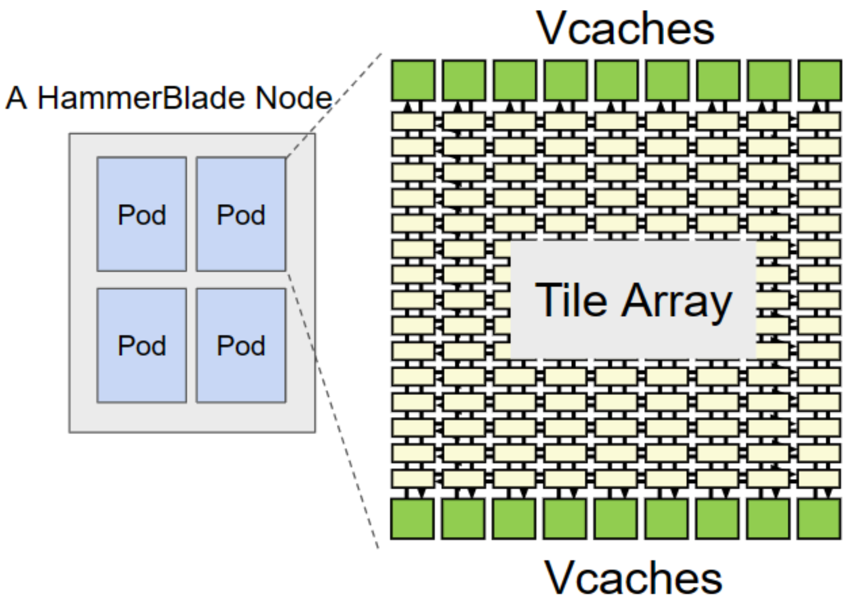


Figure 1: Organization of The HammerBlade Manycore Architecture [84]

The compute tiles each have a V5 core, which is like the brain of the tile. This brain has two important parts: a small 4KB instruction cache (1024 instructions) and a local workspace (4KB data memory). The V5 core is efficient – it can grab instructions and data while it's working on other non-dependent instructions to overlap the memory latency, thanks to non-blocking loads and stores. When the V5 core needs something that is not present in its tile, it sends requests over a special grid-like network (2-D mesh) to find the information it needs, whether it's in another tile's cache, local memory, or the main system memory (DRAM).

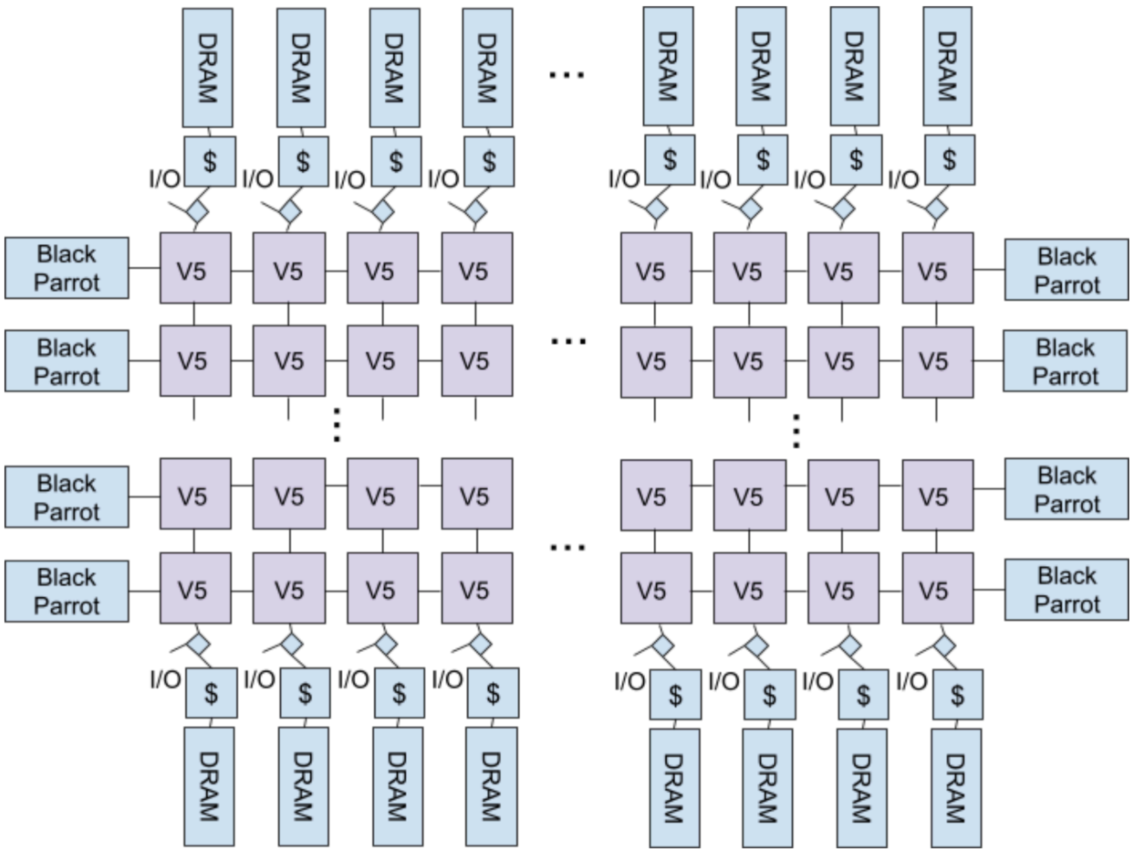


Figure 2: Interconnection of The Components [84]

The Vanilla-5 core, as the workhorse of HammerBlade Manycore, is a light-weight 5-stage pipelined RISC-V processor implementing RV32IMA ISA. Vanilla Core programs execute in SPMD (Single Program Multiple Data) mode in a 32-bit virtual address space, called Endpoint Virtual Address (EVA). Each thread in a core has its own EVA space. EVA provides a mapping to the remote and local data memory and the L2 caches.

Vanilla-cores are clever when it comes to executing non-blocking remote loads. Even though sending a request and getting a response can take hundreds of cycles, the core can keep working on other things. This is possible because the core maintains a scoreboard of dependencies and continues the execution of the program if the subsequent instructions do not depend on the remote-load (no WAW or WAR hazards).

To communicate with the L2 Cache / DRAM, the address of each core is translated into a network packet sent to the south edge of the network. Here, there's a row of L2 caches, each handling specific memory areas (virtual banks) within the off-chip DRAM. The L2 cache's horizontal position (x-coordinate) combined with the virtual bank address tells us the exact location of the data in the DRAM.

4 On-Chip-Network

The BaseJump network uses a memory model similar to PGAS (Partitioned Global Address Space). In this model, all the processing units (nodes) in the network share one big pool of memory. Each memory location is identified by a unique combination of two parts: the XY coordinates of the node "holding" that memory and a local address within that node: [85]

<X cord, Y cord, local address>

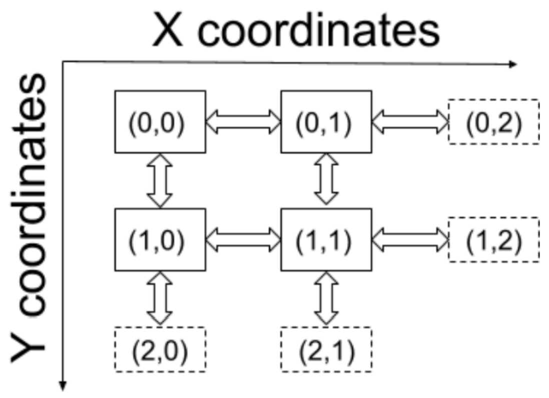


Figure 3: Global Coordinates of Nodes [85]

Each processing unit (node) in this architecture can access memory, registers, or other features based on its own local addressing system. These nodes can be various types, like RISC-V cores, special processors for signals (DSPs), programmable logic blocks (eFPGAs), specialized accelerators, or even memory buffers. Each node also has its own dedicated memory space (memory region) [85] . This allows nodes to perform load, store, and comparison operations on data located in other nodes across the network.

Features like fetching data (remote load) and comparing and swapping data (compare-and-swap) were added to this model, which allows for more complex operations like gather operations and mutual exclusion.

4.1 Mesh Network

Figure 4 shows the mesh architecture. Each tile contains a router and an accelerator, and each accelerator contains an endpoint and a core. Messages on the network are a single wide word that includes header information and payload.

The mesh network uses separate paths for sending requests (source to destination) and receiving replies (destination to source). Requests travel in X then Y directions, while replies follow the reverse path. For smooth operation, accelerators must immediately process incoming messages from the network without waiting to send new ones.

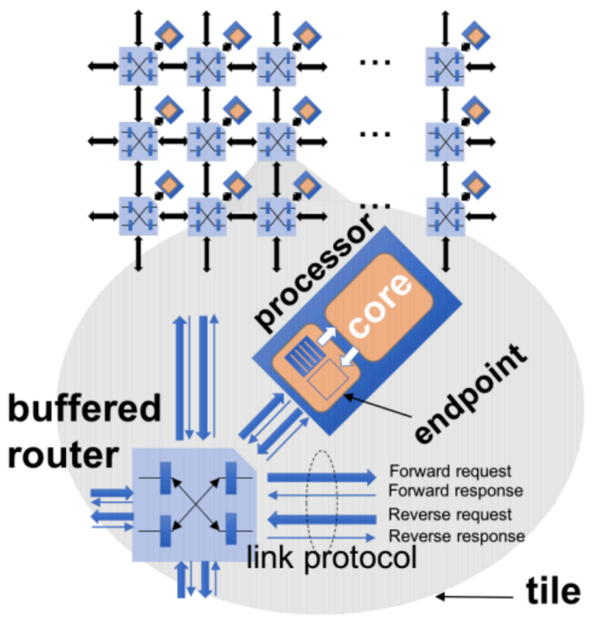


Figure 4: Mesh Network Structure [7]

4.2 Network Congestion

Network congestion is a significant performance bottleneck in the HammerBlade manycore architecture, which arises when the data traffic within the network exceeds its capacity, leading to increased latency and reduced throughput. This congestion impedes the efficiency and performance of parallel computing tasks, as it delays data communication between cores.

In manycore architectures like HammerBlade, network congestion can be caused by several factors as below:

1. ***High Communication Traffic***: Manycore architectures necessitate frequent data exchanges between cores, especially in parallel applications, which can overwhelm the network.
2. ***Inefficient Routing***: Suboptimal routing algorithms can direct excessive traffic through certain network areas, creating bottlenecks.
3. ***Contention for Shared Resources***: When multiple cores attempt to access the same network resources simultaneously, contention occurs, causing delays and congestion.
4. ***Cache Misses***: Frequent cache misses in L1 or L2 caches increase network traffic as cores fetch data from remote memory, further contributing to congestion.

The Flat arrangement of the conventional HammerBlade is largely motivated by the ease of physical design. However, the network capacity ultimately limits the scalability. In N×N mesh, each tile can only inject packets at the average rate of 2/N per cycle before the network channels on the edge become completely saturated [86] . In manycore architectures like HammerBlade, managing the challenges of network congestion is crucial for optimal performance, especially when the number of cores that can fit on a full-reticle chip will keep increasing and reach about 100K+ range in the next few years.

Cellular Manycore architecture, recently developed by BSG group, with its replicated macro units or 'Cells' comprising core arrays and cache bank strips, marks a significant step towards addressing network congestion challenges in manycore systems. This architecture, through its partitioned global address space and flat cache hierarchy, enhances physical locality and facilitates efficient data sharing, which is crucial in manycore environments. [87]

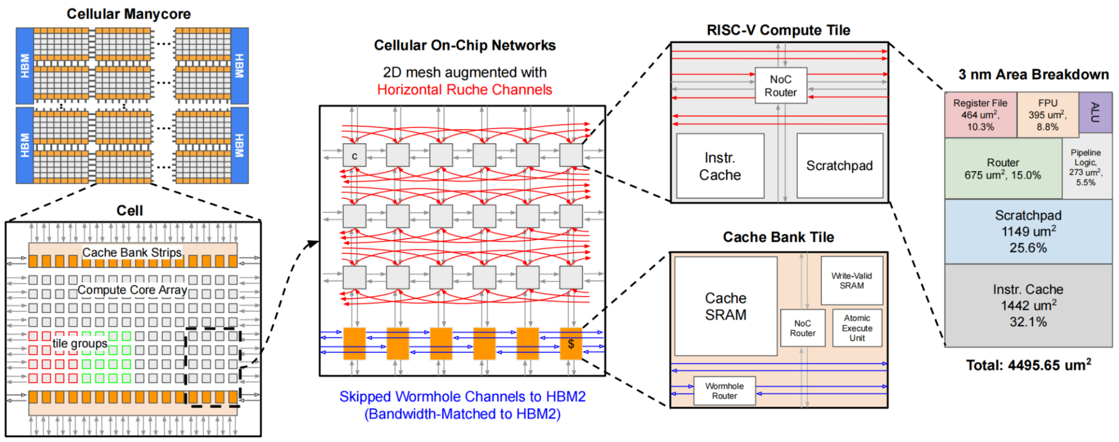


Figure 5: Overview of Cellular Manycore Architecture [87]

However, while Cellular Manycore effectively addresses high-level architectural concerns, there remains ample room for improvement at the memory system level, particularly from the standpoint of vcache arrays.

A significant contributor to network congestion in HammerBlade is the use of blocking L2 caches. Blocking caches stall the entire pipeline during a cache miss, waiting for data retrieval from memory or another cache. This stalling has several adverse effects on network performance:

1. ***Increased Traffic Bursts***: Cache misses in blocking L2 caches generate bursts of network traffic as they fetch the missing data, which can overwhelm the network and cause congestion.
2. ***Pipeline Stalling***: Blocking caches cause processor pipelines to stall while waiting for cache misses to resolve, reducing effective throughput and underutilizing computational resources.
3. ***Reduced Concurrency***: Blocking caches handle only one cache miss at a time, limiting the concurrency of memory operations and leading to inefficient network use.
4. ***Propagation of Delays***: Delays from cache misses and pipeline stalls can propagate through the network, causing additional delays and increasing overall congestion.

To alleviate network congestion, transitioning from blocking L2 caches to non-blocking ones is essential. Non-blocking caches allow multiple outstanding cache misses to be handled concurrently, significantly reducing traffic bursts and improving overall network efficiency. By enabling the processor to continue executing other instructions while waiting for data, non-blocking caches help maintain high throughput and reduce the negative impact of cache misses on network performance.

5 L2 Victim Cache

5.1 Blocking L2 Vcache

The original L2 blocking victim cache used in current HammerBlade was inspired by the data cache in Raw [5] , and implemented by Dai Cheol Jung. It is two-stage pipelined. There are tag-lookup and tag-verify stages. A stage before the tag-lookup stage, we refer to it as an input stage. This pipeline has throughput of one load or store every cycle with an absence of miss [37] .

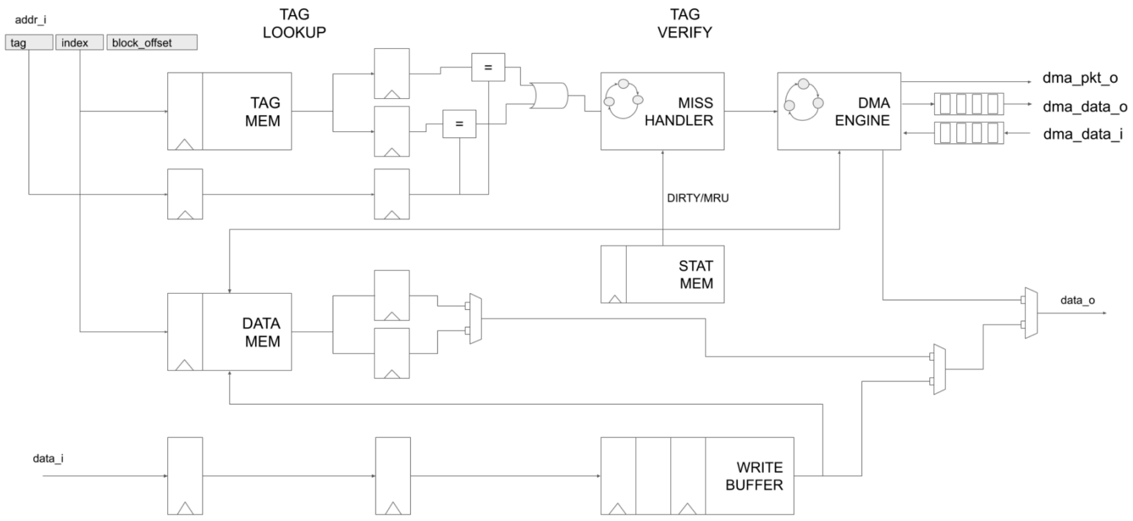


Figure 6: Blocking L2 Cache High Level Schematic [37]

In the tag-lookup stage of the blocking L2 cache, the tag memory, addressed by the index portion of the input address, determines cache hits or misses. Each tag entry includes a valid bit, lock bit, and tag. The data memory is read for load instructions. Cache size is influenced by data width, address width, block size, and number of sets. For instance, a 32 KB cache with 32-bit address/data, 8-word blocks, and 512 sets uses a 9-bit index and an 18-bit tag, requiring a 40 × 512-bit tag memory. The cache also features status memory with dirty and LRU bits.

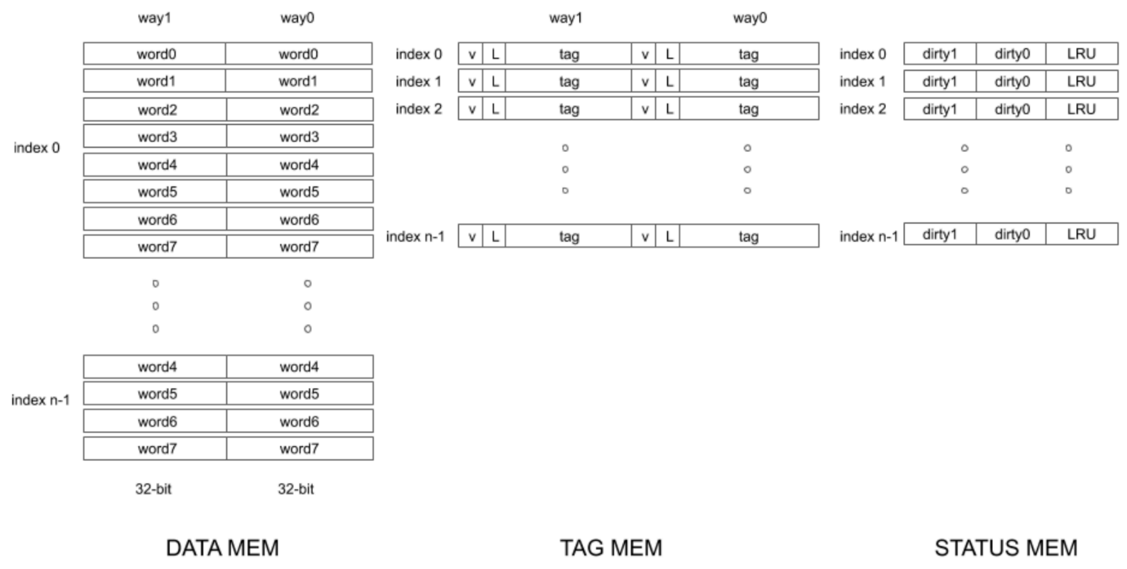


Figure 7: Blocking L2 Cache SRAM Layout [37]

The L2 cache uses a ready-valid handshaking system for both incoming and outgoing data, ensuring smooth communication and handling different data sizes. Software can manage the cache using instructions like TAGST for updating tags, TAGLA/TAGLV for checking tags and validity, and AINV, AFL, and AFLINV for flushing or invalidating specific cache lines.

When the L2 cache misses a requested data piece (store miss), it retrieves the entire block containing that piece. Even if only part of the block is being updated, the whole block is loaded. The cache keeps track of dirty blocks (modified data) and clears the dirty bit when a new block is requested. Once the new block arrives and the update is complete, the dirty bit is set again.

Based on the infrastructure of the original blocking L2 victim cache, I developed the non-blocking one which shares the same basic workflow, but with the capability of holding outstanding misses and going on with the following cache requests, without stalling the pipeline in most cases.

5.2 Non-Blocking Cache Design Overview

In HammerBlade manycore structure, the vcache array plays a key role, and the current HammerBlade uses blocking caches for the vcache arrays on the north and south side. Traditional blocking caches, which stall their pipeline upon encountering a miss, can cause a pause in the functioning of node routers, leading to network congestion as new packets cannot be injected into the network. This situation highlights a critical need for a more efficient cache system. Thus, developing a non-blocking cache becomes imperative. Such a cache design would not only handle misses more efficiently but also maintain continuous network operations, significantly alleviating network congestion issues. The goal of the development of our non-blocking cache design is to improve memory system efficiency, increase concurrency, reduce delay and mitigate network congestion, particularly within the context of the HammerBlade Manycore architecture.

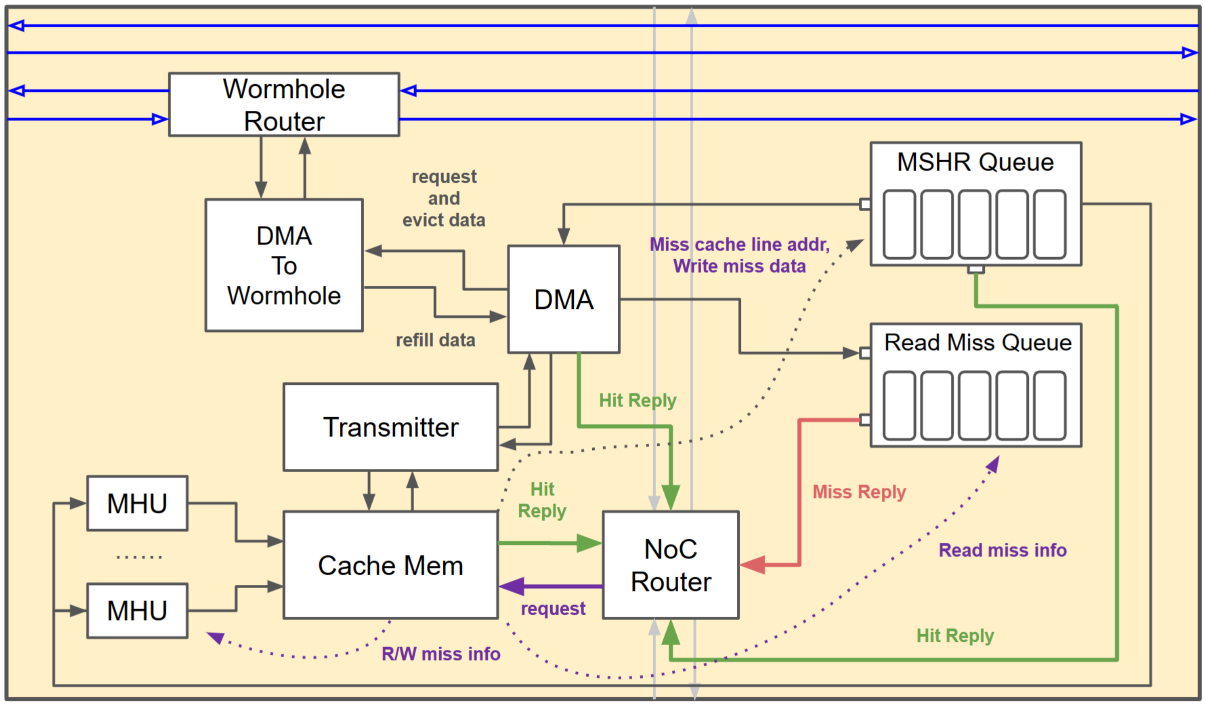


Figure 8: Overview of Non-Blocking Vcache Organization

Figure 8 shows the overall structure of a non-blocking vcache tile in HammerBlade. Central to the cache is an advanced Miss Status Holding Register (MSHR) queue, paired with a read miss queue, multiple Miss Handling Units (MHUs) linked to each corresponding MSHR entry, an efficient Direct Memory Access (DMA) system, and a data transmitter used for eviction/refilling data to go smoothly between DMA and Data Memory. DRAM Requests and data are sent through Mesh / Ruche Network in the form of wormhole packets by wormhole routers. Cache Hit and Miss send the data responses to the right compute tiles by NoC Router. The configuration uniquely addresses the limitations of traditional cache systems by effectively managing both cache hits and misses, and the new design provides enough buffer to ensure that primary and secondary misses are drained from the network so that the hit requests can proceed.

Each vcache tile acts like a device on the network, identified by a special address (NPA). This address has a built-in flag: a 0 in the most significant bit (MSB) means you're talking to the DRAM memory itself. If the MSB is a 1, you're actually interacting with the vcache's tag memory. This allows the main system (host) to clear tags during setup or even examine them for debugging purposes using TAGST and TAGLA instructions.

The non-blocking L2 victim cache retains the two-stage pipeline and most of the basic flow as the old blocking cache. It checkers TAG MEM for the existence of a cache line in the tag look up stage and, in the tag verify stage if the cache line exists, it works in the same way as the original blocking cache; however, if there is a cache miss, it takes 3 cycles to determine which way to evict, update the new tag in advance, record the miss information into Miss Handling Unit(MHU), MSHR and Read Miss Queue(for read misses). The non-blocking cache will then keep on executing the next request, without stalling the pipeline and waiting for the missed cache line to return. The MHU will wait behind the scene till its turn to turn activated, and then it will use the miss information inside to do eviction/refilling properly.

5.3 Non-Blocking Cache Walkthrough

When a compute core sends a remote request through the 2D network and arrives at the destination vcache tile, the NoC router in that vcache tile will unpackage that request into a cache request which the cache can understand.

As shown in Figure 8, the cache will look up the cache line address of this request in the tag look up stage. If it doesn't find any match in the TAG MEM, it means that it is a primary cache miss. In this case, STAT MEM will be read to get the LRU and dirty bits, in order to determine which way to evict. At the same time, new tag data will be directly updated in the TAG MEM in advance, even though this miss hasn't been processed, and the cache line hasn't returned from DRAM yet. This will lead any following cache request to the evicted line to be a cache miss. Also, some information about this cache miss will be stored in an allocated Miss Handling Unit. The cache line address will also be recorded in an allocated MSHR entry, so by checking the MSHR CAM, any future cache request will be taken as a secondary miss, if it finds the same cache line has already been in the MSHR CAM, even if it finds a match in the TAG MEM in the tag look up stage. For a write miss, the data it wants to write will be recorded in the data field of the MSHR entry; for a read miss, the information about this read request will be stored in an allocated Read Miss Queue entry. The details will be discussed later in the following sections.

When a MHU waits until its turn to be activated, it will then start the eviction and refilling process. It will first communicate with the DMA to let it send DRAM read and write requests respectively. The DRAM read request will be packaged as a wormhole packet in the DMA_to_Wormhole module and injected into the wormhole network. The DRAM write request will be kept in the DMA_to_Wormhole module until all the evicted data arrives, and then be packaged as wormhole packets together and sent to the DRAM controller. The MHU will coordinate with the Transmitter to read the evicted data from

DATA MEM. The Transmitter will reorder the words into correct order and transfer them to the DMA. The DMA will wait for all the words in the evicted cache line to arrive and send them to the DMA_to_Wormhole module.

When the data is read from DRAM and arrives at the DMA_to_Wormhole module in the form of wormhole packets, it will parse the wormhole packets into cache line words and send them to the DMA. The MSHR ID is sent along with DRAM read packet and returns with the cache line data, so the cache will know which MSHR the data arrives at the DMA belongs to. The DMA will first combine the returned data with the most up-to-date data in the MSHR CAM, reorder the combined data, and transfer them to the Transmitter. The Transmitter will write these words into DATA MEM and the MHU will be notified everything is done. At the same time the DMA will use the combined data to serve all the read misses under this cache line in the Read Miss Queue. When the cache finishes both writing the data into DATA MEM, and serving all the read misses, a cache miss handling process is finished. All these steps except serving the Read Miss Queue are being executed in parallel behind the scene, without stalling the cache pipeline to wait for this miss to be done.

For a cache hit, there could be multiple possible cases. The simplest situation is that the cache finds a match in the tag look up stage and doesn’t find a match in MSHR CAM, which means this is a legit cache hit. So it can directly output the corresponding data for a read hit, or load the write data and mask into the store buffer for a write hit.

The second case is that it finds a match in the TAG MEM but also finds a match in the MSHR. This means this is a secondary miss. For a write request, since we can simply write the data into the data field in the proper MSHR entry, it can just be taken as a normal cache hit. For a read request, if all the bytes it wants to read are written in the MSHR entry by previous write misses, it can directly output the data in the MSHR, which can also be taken as a cache hit.

Another case is that when a read request comes, it finds that the cache line is in the DMA right now, which means this is a secondary miss and the cache line has already returned from DRAM and arrived at the DMA. In this case, it can take advantage of the combined data in the DMA to grab the data it wants. This can also be taken as a hit case.

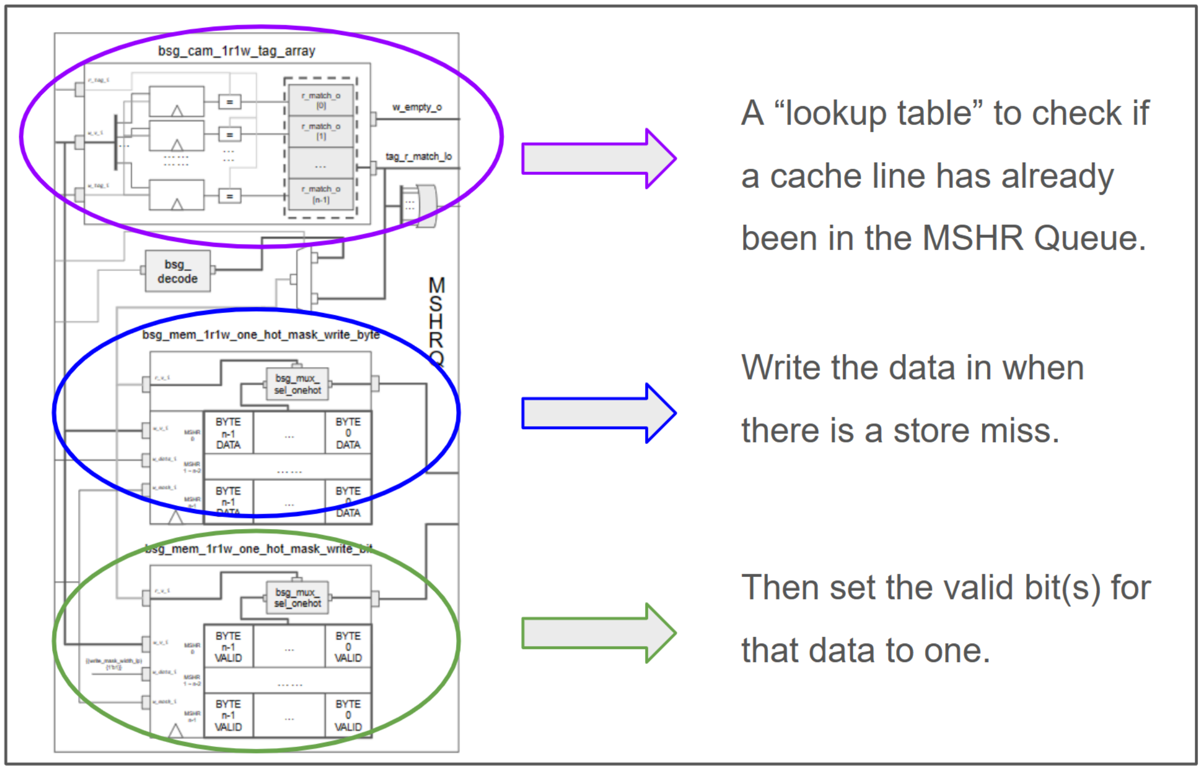
5.4 MSHR

In non-blocking caches, Miss Status Holding Registers (MSHRs) play a crucial role in managing cache misses efficiently. When a cache miss occurs, an MSHR is allocated to track the status of the outstanding memory request. Each MSHR holds information about the pending miss, including the address, the type of request (read or write), and any additional data necessary for handling the request once the data is fetched from the lower memory hierarchy.

MSHRs enable the cache to continue servicing other requests while waiting for the data associated with the miss to arrive. By allowing multiple misses to be tracked simultaneously, MSHRs significantly enhance the throughput and performance of the cache. This non-blocking behavior reduces the stall time for the processor, as it does not need to wait idly for a single miss to be resolved before processing subsequent memory requests.

The MSHR Queue in the non-blocking cache is essentially implemented as a Content Addressable Memory (CAM). It uses the cache line address offset as the tag to look up matched entries, and has two separate SRAMs inside to store the data byte(s) from primary or secondary store misses, and update the corresponding valid bits for each byte respectively.

Traditionally, MSHRs have limitations in handling multiple cache misses efficiently. They allocate a new entry for each cache miss, with certain bits indicating whether the miss is a secondary miss related to a primary one. This approach, however, leads to a storage issue: a large number of secondary misses consume many MSHR entries. With limited MSHR resources, the system quickly becomes saturated, stalling the pipeline and resulting in network congestion. To overcome these challenges, by dividing MSHRs into MSHR Queue and Read Miss Queue, and implementing a data field in MSHR entries, ensured that primary or secondary store misses could directly store their data [88] , reducing the need for allocating new MSHR entries. Meanwhile, this approach can firstly transform some secondary load misses into hits and for others, it allows more space for each MSHR entry to allocate the load miss, since the Read Miss Queue is essentially an 1RW SRAM and will have much less area overhead when the number of entries increases.



A “lookup table” to check if a cache line has already been in the MSHR Queue.

Write the data in when there is a store miss.

Then set the valid bit(s) for that data to one.

Figure 9: MSHR Queue Schematic

5.5 Read Miss Queue

The Read Miss Queue is a separate SRAM aside from the MSHR Queue. Everytime there is a primary or secondary read miss, a new entry is allocated in the Read Miss Queue. The space of Read Miss Queue is evenly distributed to each MSHR entry so each MSHR entry will have the same count of Read Miss Queue entries for allocation. When a read miss occurs but the corresponding space in the Read Miss Queue is full, the pipeline has to be stalled, and wait until the refill data comes back to DMA.

When the data of a cache line returns from DRAM, the Read Miss Queue will immediately be served until all the read misses linked to that cache line are issued, during which period if there’s a read going in the tag verify stage, it has to be stalled.

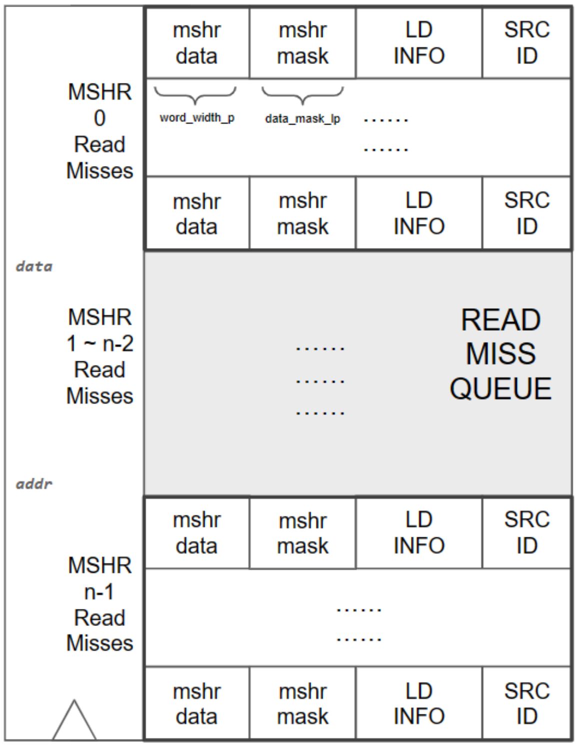


Figure 10: Read Miss Queue Schematic

Each Read Miss Queue entry is made up of four parts: MSHR data, MSHR mask, LD information, and SRC ID.

- MSHR data and mask - When a secondary read miss occurs, it is possible that a previous write primary/secondary miss has written some bytes of the data that this secondary read miss wants to access, to the MSHR data field, so these bytes are the most up-to-date data to be output, even when the cache line returns. This means we have to store these data as part of the information in the Read Miss

Queue entry, since any following secondary write miss could potentially write newer data to some of these bytes again.

- LD information - This could consist of information such as, whether this read request is a *read byte*, *half* or *double*, whether it has sign extension or not, whether it is a *read mask* and if so, what the mask bits are…
- SRC ID - This is the destination coordinates that comes along with each cache request. With this ID, the cache can know where the response data should route to, when the cache line returns and the read misses are served.

5.6 Evict/Refill Transmitter

The Evict / Refill Transmitter connects the DMA and data memory in the cache, and works for delivering the evict / refill data between data memory and DMA. This process can be done behind the scenes so the pipeline doesn't necessarily need to be stalled anymore.

We added this transmitter because of an interesting modification made to the data memory, compared with the original blocking cache. We partitioned the data memory (DMEM) into two banks, which store even and odd words in cache lines respectively. Due to SRAM's area limitations, each cache line is split into bursts to store in the DMEM, thus resulting in a long stall for the incoming requests when the cache conducts eviction/refilling. By partitioning the DMEM into even and odd banks, it allows eviction/refilling and incoming requests to access different banks concurrently without stalling, by letting the data transmitter read or write data from the other bank after knowing the bank that the incoming request is going to access, thus significantly mitigating network jams in HammerBlade. The challenge of potential resource contention and data re-order due to partitioning is addressed by implementing a two-tier pipeline with one FIFO tied to each bank in the transmitter, and arbitrarily setting priorities for different operations, ensuring both efficient data flow and minimal area overhead.

5.7 Word Tracking Strategy

The word tracking strategy utilizes a separate SRAM, known as track memory(TRACK MEM), to keep track of the validity of each word within a cache line. This approach allows store misses, where a whole word is stored, to be treated as hits. The information about the cache line to be evicted is recorded in the miss handling unit. The store data is recorded in the MSHR (Miss Status Holding Register), and the track memory is updated by setting all bits in the corresponding way to zero except for the bit corresponding to the stored word, which is set to one.

When the data for the evicted cache line is read from the data memory, the data stored in the MSHR will then be written into the data memory. If there is a subsequent read request for the data in this word, it is treated as a read hit. If there is a subsequent store request for this word, or similarly a store word request for other words, it is treated as a store hit and the corresponding bit in the track memory is updated in the same manner.

This strategy continues until a read request is made for a word whose corresponding bit in the track memory is not set to one, or a store miss occurs where less than a whole word is stored. By leveraging this word tracking strategy, a significant amount of unnecessary memory traffic can be avoided, enhancing the performance of write-intensive programs like *memcpy*. This optimization can lead to substantial time savings, particularly for operations that frequently update specific words within cache lines.

The word tracking strategy also introduces some new situations and corresponding solutions, among which one case that we will most frequently encounter is called track miss. Track miss means in the tag look-up stage, the cache finds the cache line has already been in the cache, but after checking the valid bits in TRACK MEM, it finds that not all the words that this request wants to access are valid. In this case, some words in this cache line were already written by previous *store word* requests, and these words are more up-to-date, compared to those in the off-chip DRAM. So after we fetch the cache line from DRAM, we should keep these words in the DATA MEM, instead of updating them. Same, if the way we are going to evict only has some parts of its words valid, we should only update these words in the DRAM, instead of the whole cache line. To do so, we will have to store the valid bits somewhere for later use. In this design, we load them into the Miss Handling Unit when a cache miss is detected and allocated in the tag verify stage.

5.8 Miss Handling FSM Logic

1. `MHU_START` : Each free miss handling unit waits in MHU_START state, until there is a cache miss detected in the tag-verify stage, and allocated to that miss handling unit. Some information about this miss, such as the cache line address, whether it

is a track miss or not, and the track bits of the way which is chosen to be evicted, will be loaded into this MHU. The MHU will then keep waiting in this state till its turn to be activated. There's a FIFO keeping record of the order of each cache miss, and the earlier cache miss will be processed first. When it gets to its turn, the MHU will be activated. It will jump onto sending refill address(MHU SEND REFILL ADDR) if it is a normal miss, or jump over store tag miss branch if it is a primary full word store miss(MHU STORE TAG MISS).

2. `MHU_SEND_REFILL_ADDR` : When the cache needs a block of data that's not available (DMA read request), it picks an unused slot (invalid way) or the least recently used block (LRU) to replace. If the LRU block has dirty data (modified), the cache sends an eviction message (MHU SEND EVICT ADDR) to prepare for writing it back to memory. Otherwise, the cache moves on to fetching the new data (MHU WRITE FILL DATA). This way, the cache can kick off evicting the old block while the new data is on its way from DRAM, hiding some of the delay. Finally, the cache updates the tags and dirty bits to reflect the new data.
3. `MHU_WAIT_SNOOP_DONE` : In some cases, there are still some words waiting to write into the chosen evicting cache line in the store buffer, when everything is all set for the eviction process in the MHU. This state ensures that all these kinds of words are written into that cache line before we evict it, so the data written into DRAM would be correct.
4. `MHU_SEND_EVICT_ADDR` : As soon as the DMA write request is sent out, it shifts to MHU SEND EVICT DATA. In some rare cases, when the miss in a MHU is a store tag miss, and before it starts the eviction process, another secondary partial

store / load to one of the other words happens, we should directly transition it to a normal miss and send DMA read request to get the whole cache line(MHU SEND REFILL ADDRESS).

5. MHU_SEND_EVICT_DATA : During this state, the transmitter will read from one data bank at a time until all the odd and even words in a cache line are gotten, and at the same time it will reorder these words in the correct order and transfer them to DMA. The DMA will wait and accumulate all the words in a cache line before it sends them out.
6. MHU_WRITE_FILL_DATA : Similarly, in this state, the DMA reorders the words in the returned cache line in correct order and transfers them to the transmitter. The transmitter will also access one data bank at a time in order to avoid congestion. While the cache line is being written to the data memory, it saves the entire cache line and combines it with the most up-to-date data in the MSHR CAM, and then uses the combined cache line to serve the read misses in the Read Miss Queue. For a store tag miss, there is no need to fetch the cache line from DRAM. In that case, it will directly transfer the word(s) and mask(s) into the transmitter, and the transmitter can write the word(s) into data memory. When all the data is written into data memory, it jumps back to the initial state and waits for the next cache miss(MHU START).
7. MHU_STORE_TAG_MISS : Typically for a primary store full word miss, it will jump onto sending evict address if there is a cache line needed to be evicted. In the same rare cases as mentioned in MHU_SEND_EVICT_ADDR, it will directly transition to a normal miss and send DMA read request(MHU SEND REFILL ADDR).

8. MHU_STORE_TAG_MISS_FILL_HOLD : This is used for some corner cases where the MHU should wait for some time due to special needs, before it starts evicting data, for a primary store full word case.

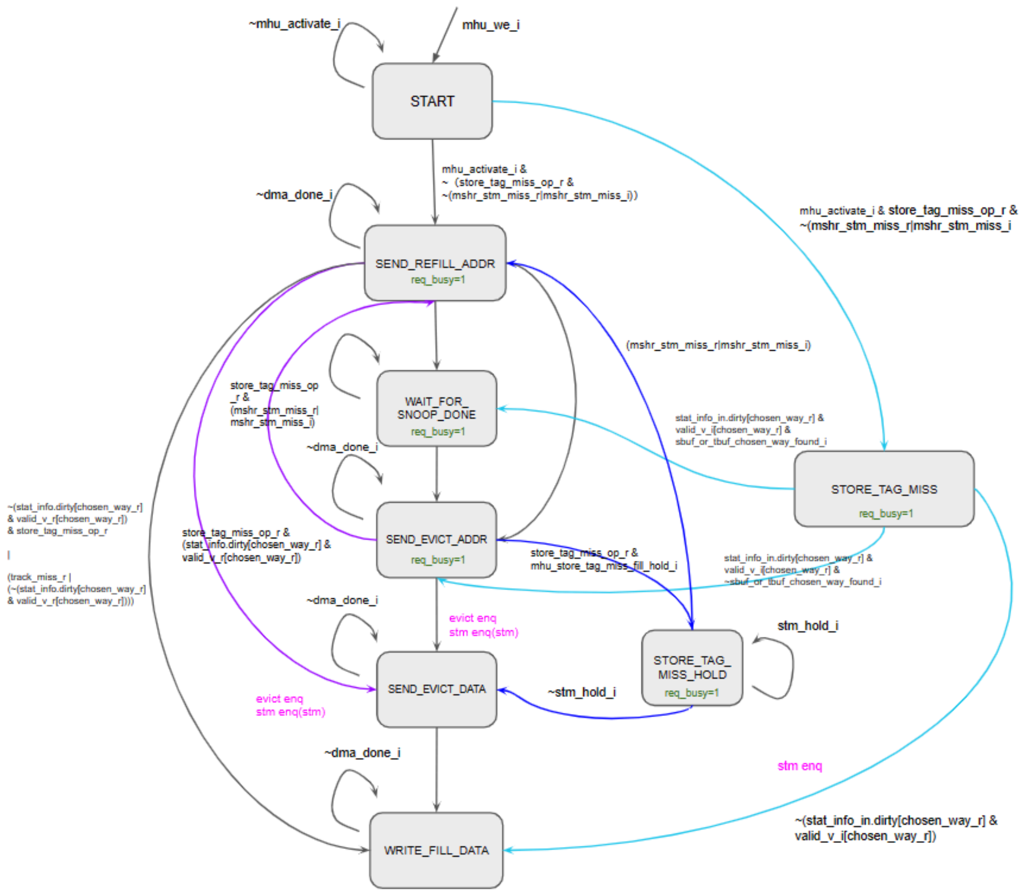

~mhu_activate_i
mhu_we_i
START
mhu_activate_i &
~ (store_tag_miss_op_r &
~(mshr_stm_miss_r|mshr_stm_miss_i))
~dma_done_i
SEND_REFILL_ADDR
req_busy=1
mhu_activate_i & store_tag_miss_op_r &
~(mshr_stm_miss_r|mshr_stm_miss_i
(mshr_stm_miss_r|mshr_stm_miss_i)
store_tag_miss_op
_r &
(mshr_stm_miss_r|
mshr_stm_miss_i)
WAIT_FOR_
SNOOP_DONE
req_busy=1
~dma_done_i
stat_info_in.dirty[chosen_way_r] &
valid_v_i[chosen_way_r] &
sbuf_or_tbuf_chosen_way_found_i
STORE_TAG_MISS
req_busy=1
~(stat_info.dirty[chosen_way_r]
& valid_v_r[chosen_way_r])
& store_tag_miss_op_r
|
(track_miss_r |
(~(stat_info.dirty[chosen_way_r]
& valid_v_r[chosen_way_r])))
store_tag_miss_op_r &
(stat_info.dirty[chosen_way_r] &
valid_v_r[chosen_way_r])
SEND_EVICT_ADDR
req_busy=1
store_tag_miss_op_r &
mhu_store_tag_miss_fill_hold_i
stat_info_in.dirty[chosen_way_r] &
valid_v_i[chosen_way_r] &
~sbuf_or_tbuf_chosen_way_found_i
~dma_done_i
evict enq
stm enq(stm)
stm_hold_i
STORE_TAG_
MISS_HOLD
req_busy=1
SEND_EVICT_DATA
~stm_hold_i
evict enq
stm enq(stm)
~dma_done_i
stm enq
~(stat_info_in.dirty[chosen_way_r] &
valid_v_i[chosen_way_r])
WRITE_FILL_DATA

Figure 11: MHU FSM Diagram

5.9 Cache Line Locking

The cache line locking mechanism is implemented in the similar way as the original blocking cache. The locking mechanism is designed for two main purposes. First, if a cache line is invalid and locked, it prevents new data from being placed in that line. This is particularly useful in the rare case of a defective SRAM cell, rendering a specific cache line unusable. Second, if a cache line is valid and locked, it cannot be selected for eviction. This can be advantageous when we know in advance that a specific address will be accessed frequently, allowing us to lock the cache line and prevent it from being evicted. It is assumed that the user cannot lock both ways within a set [37] .

5.10 Replacement Policy

The non-blocking cache succeeds the Least-Recently-Used(LRU) replacement policy to determine which way to evict, when a new cache miss occurs and needs to be allocated. This cache replacement trick uses a tree structure with n-1 bits (for an n-way cache) to identify the least recently used (LRU) block. Each bit acts like a direction sign: 0 for left, 1 for right. By following this path through the tree, you can find the LRU block with simple logic (see the table for details). When you access a specific block, the tree is "flipped" to point away from that block, updating the LRU information. [89] This update only requires modifying a few bits, three for an 8-way cache. The beauty of this approach is that you can update the LRU information without even needing to read the existing bits first.

LRU encoding		Next LRU state	
curr_state [6:0]	LRU way	Referred way	next_state [6:0]
zzz 0z00	0	0	--- 1-11
zzz 1z00	1	1	--- 0-11
zz0 zz10	2	2	--1 --01
zz1 zz10	3	3	--0 --01
z0z z0z1	4	4	-1- -1-0
z1z z0z1	5	5	-0- -1-0
0zz z1z1	6	6	1-- -0-0
1zz z1z1	7	7	0-- -0-0

Table 1: Tree pseudo-LRU encoding. ’z’ means ”don’t care”. ’-’ means unmodified. [84]

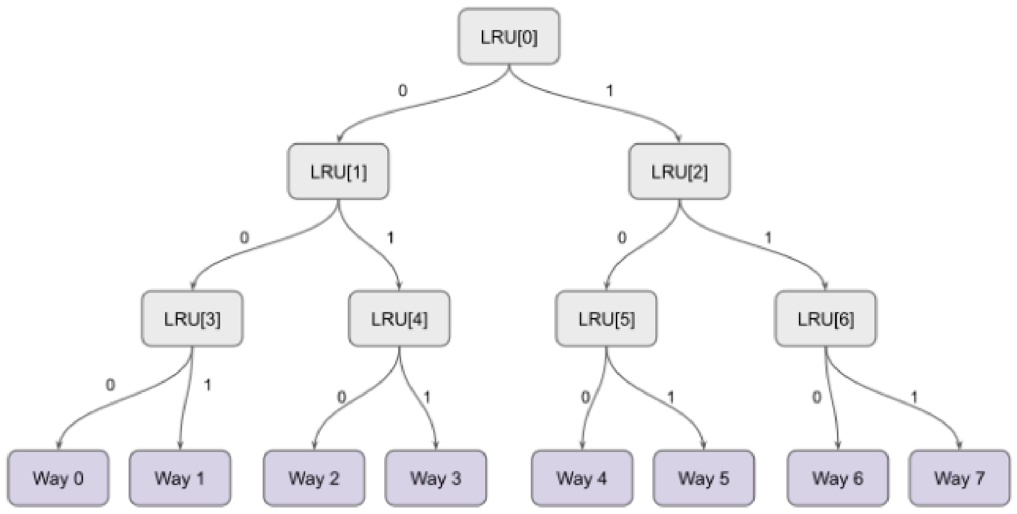


Figure 12: Tree Pseudo-LRU Algorithm [84]

5.11 Interface Converters

5.11.1 Attaching to Manycore Link

Every vcache tile can be reached on the network using a special address (NPA). This address has a built-in code: a 0 in the most important bit (MSB) tells the system you're talking to the DRAM memory itself. But if the MSB is a 1, you're actually interacting with the vcache's special memory for tags. This lets the main system (host) wipe clean all the tags during startup or even examine them for troubleshooting purposes using TAGST and TAGLA instructions [37].

5.11.2 Attaching to DRAM Controller

The non-blocking cache interacts with the external world through its DMA. The DMA will send the read/write requests and evicted data to a module called DMA_to_Wormhole, which connects the DMA with the wormhole network. The requests and data from DMA will then be packed into the form of wormhole packets to go through the network, arriving at the DRAM controller.

The DRAM controller will schedule and dispatch all the DRAM requests and at the same time, buffer and reorder the returned data from DRAM, and pack the data into wormhole packets to send back to the corresponding vcache's DMA.

A round-robin module takes vcaches' DMA requests, and sends commands to the DRAM controller. There are two submodules called RX and TX. RX routes data received from the DRAM controller to the correct cache. The read data is expected to return out of order so the RX will have to buffer the data and reorder them to be in the correct order before sending them back to the corresponding vcache. TX routes the evicted data from L2 vcache to the DRAM controller. This separation of RX and TX logic is a very convenient abstraction, because usually read and write channel operations are independent from each other.

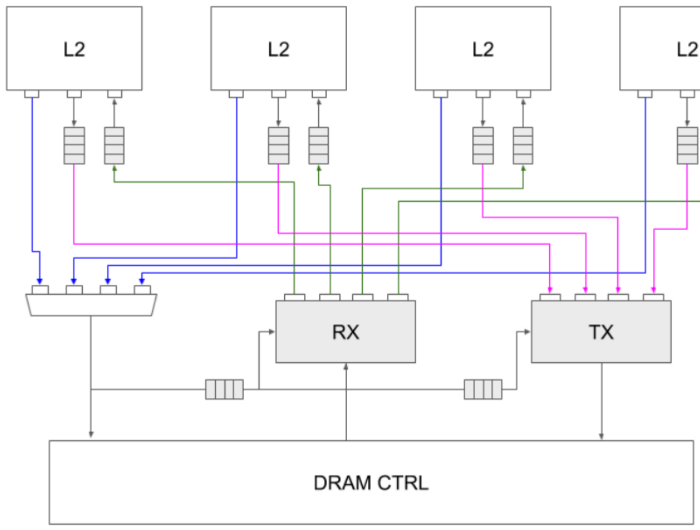


Figure 13: L2 DMA Interface Converter [37]

6 Verification Strategy

6.1 Isolated Unit Test

To verify the functionality of the cache itself, before integrating the non-blocking cache into the current HammerBlade, I first conducted isolated unit testing on it. By using a non-synthesizable fake DMA model to simulate the function of a real DRAM, we generated test traces for different cases, and compared the theoretical results and the actual outputs to verify the correctness.

6.2 Hand-written Regression Test

In order to test the new non-blocking cache design, we created a testbench designed to dispatch requests and monitor the corresponding responses. This setup facilitates various checks including tag clearing, writing data of different sizes (such as bytes, halves, and words), and executing flush or invalidate commands. It assesses if the write buffer properly bypasses or if the miss handler accurately replaces the LRU cache line. This verification process involves examining the tags through TAGLA and TAGLV following a cache line replacement. Additionally, when new bugs were identified during the integration process with the manycore array, specific tests replicating them were developed and added to the regression testing suite to ensure comprehensive coverage and validation.

6.3 Constrained Random Test

To make the verification process easier and comprehensive enough(i.e. cover all the possible cases), we utilized a coverage-based verification method called constrained random testing.

The idea is, instead of thinking about how to set up a test to get the design into a bunch of weird corner cases to get rid of bugs, we can make use of randomized input generators; and then use coverage metrics to measure whether those randomized generators have sufficiently covered the hardware. We can tweak the tests until we get where we need to be. So, assuming that writing coverage scripts is easier than trying to write weird corner case tests, using a coverage methodology actually allows us to converge faster on fully coverage hardware.

After setting up a bunch of coverpoints, covergroups and illegal/ignore bins, we ran the simulation with VCS. By using DVE to view the functional coverage results, we found out that it achieved 100% coverage and at the same time all test cases passed, so we can say the non-blocking cache design is tested sufficiently and proven to be functionally correct.

7 Benchmarking

7.1 Non-Blocking Cache Profiler

Based on the original blocking cache profile, a non-blocking one is developed to accommodate different cache utilization situations. There are mainly five different cases for the non-blocking cache:

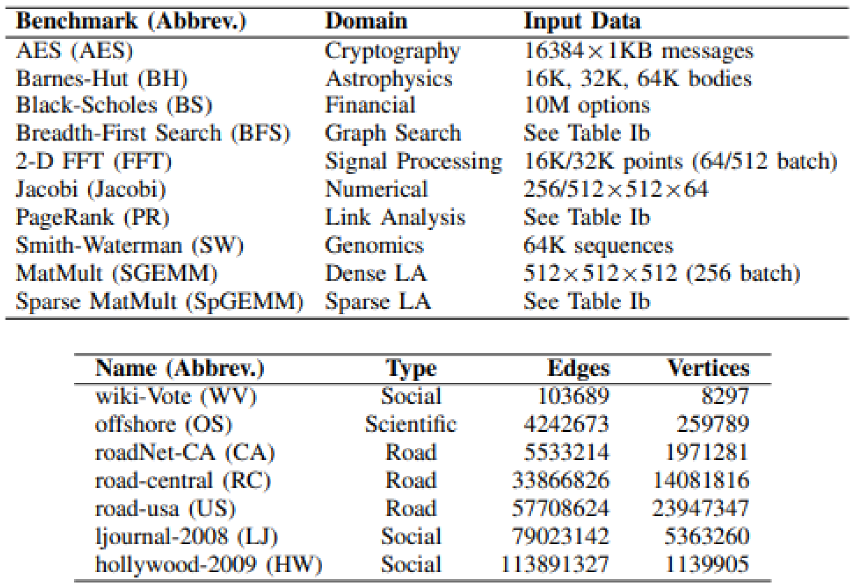
1. *Utilized* - read,write,atomic
2. *Idle* - no cache hit, and no cache miss in progress
3. *Miss* - no cache hit, and there is at least one cache miss in progress, but no request is stalled in TV, because there is space in MSHR.
4. *MSHR full* - no cache hit, and there is at least one cache miss in progress, but there is a request is stalled in TV, because not enough space in MSHR
5. *Stall rsp* - cache hit, but output FIFO is full.

The cache profiler tracks and stores all the related periodic data and this data will later be thrown into profiling scripts and tools for a comprehensively statistical and visualized analysis.

7.2 Benchmark Programs

The hypothesis was that, with the transition from blocking vcache arrays to non-blocking ones, HammerBlade will gain great benefits on most of the SPMD benchmark programs, in respect of total runtime speedup, DRAM utilization increase and network stall reduction. It will benefit most from those programs which are memory-intensive, and least from those which are compute-intensive with very limited memory requests. Moreover, the benefit should increase almost linearly as the number of MSHRs increases, because when the number doubles, the cache's ability to handle concurrent memory data requests also approximately doubles. Table I summarizes the ten parallel benchmarks used to evaluate HammerBlade with optimizations. These benchmarks generally fall into one of three groups:

1. *Compute-intensive, Low-communication*: AES, BS, and SW have high operational intensity and require very little memory access. Utilizing the local scratchpad is crucial for frequently accessed data. AES tiles store S-boxes locally, BS heavily utilizes the FP divider and square-root unit, and SW's high branch-miss rate necessitates scratchpad usage.
2. *Compute-intensive, Sequential-access*: SGEMM, FFT, and Jacobi are characterized by different phases of the program, where all tiles initially load large, sequential blocks of data, compute for a long time, and then dump out the results.
3. *Memory-intensive, Irregular-access*: SpGEMM, PR, BFS, and BH operate on sparse and irregular data structures that are difficult to partition. Cells copy data structures (full or partial) to local DRAM for faster access. The multi-iteration algorithm requires synchronization between Cells at the end of each round to exchange results for the next iteration. [87]



Benchmark (Abbrev.)	Domain	Input Data
AES (AES)	Cryptography	16384×1KB messages
Barnes-Hut (BH)	Astrophysics	16K, 32K, 64K bodies
Black-Scholes (BS)	Financial	10M options
Breadth-First Search (BFS)	Graph Search	See Table Ib
2-D FFT (FFT)	Signal Processing	16K/32K points (64/512 batch)
Jacobi (Jacobi)	Numerical	256/512×512×64
PageRank (PR)	Link Analysis	See Table Ib
Smith-Waterman (SW)	Genomics	64K sequences
MatMult (SGEMM)	Dense LA	512×512×512 (256 batch)
Sparse MatMult (SpGEMM)	Sparse LA	See Table Ib

Name (Abbrev.)	Type	Edges	Vertices
wiki-Vote (WV)	Social	103689	8297
offshore (OS)	Scientific	4242673	259789
roadNet-CA (CA)	Road	5533214	1971281
road-central (RC)	Road	33866826	14081816
road-usa (US)	Road	57708624	23947347
ljournal-2008 (LJ)	Social	79023142	5363260
hollywood-2009 (HW)	Social	113891327	1139905

Table 2: Ten parallel benchmarks used to demonstrate the parallel programmability [87]

In the evaluation, We incrementally improve each of these parameters (router, cache, density opt), then add the architectural features in the following order: Non-blocking loads, Ruche Network, Write-validate policy, Load Packet Compression, Regional IPOLY, and finally Non-blocking cache. Here we focus on the Non-blocking Cache feature. We can evaluate the benefit from Non-blocking cache by comparing the performance of the design of Non-blocking cache, with the one previously with Regional IPOLY only.

7.3 Results and Evaluation

The evaluation of the new design showcased notable improvements in the performance. The 10 parallel benchmarks run on the HammerBlade system revealed that even with just 2 MSHR entries, the non-blocking cache achieved approximately a 1.5X speedup in single-cell performance compared to the highest-performing baseline, for programs with moderate memory requests. Importantly, it shows that about 70% of the benchmarks showed almost no network stall contributions to the core utilization, with the highest being under 20% in the remaining benchmarks. This performance boost is a direct result of the optimized memory access and reduced network congestion enabled by our cache design.

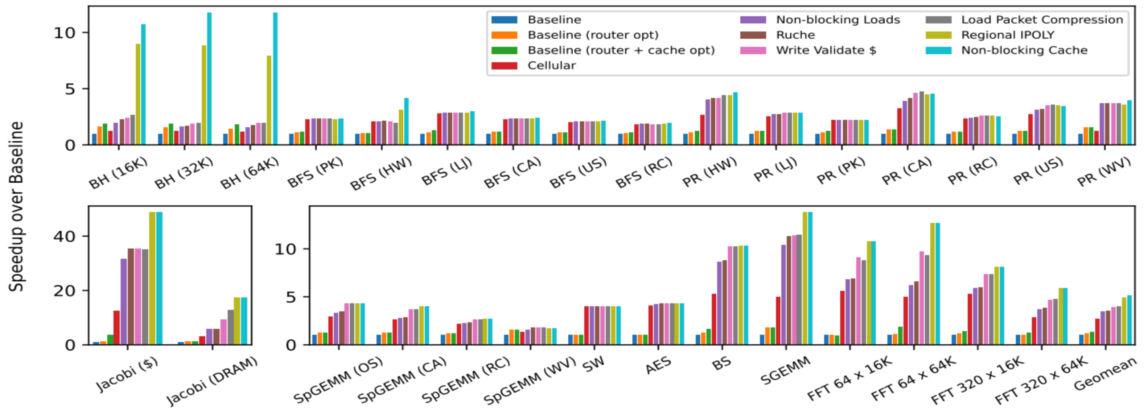


Figure 14: Speedup Over Baseline [87]

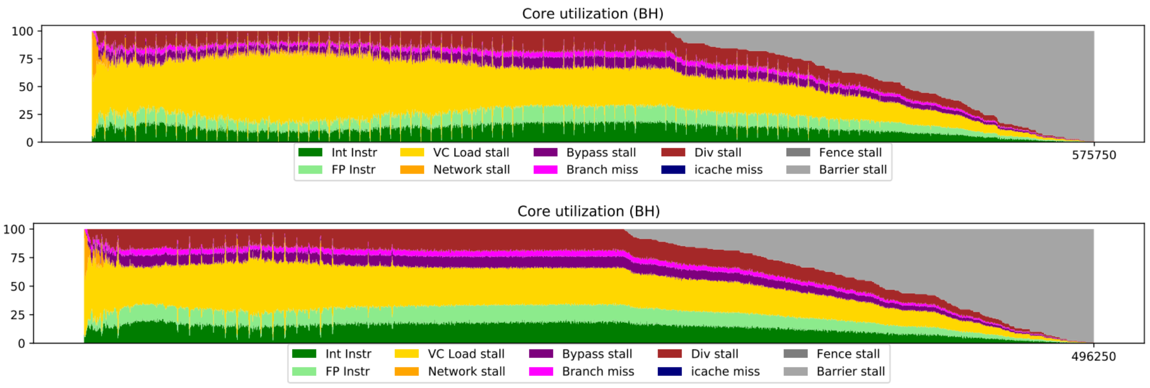


Figure 15: BH Core Utilization

The performance impact of switching to a non-blocking L2 cache varies across different benchmark categories due to a combination of factors, including data access patterns, synchronization requirements, and algorithm sensitivity.

Memory-intensive, irregular-access benchmarks often deal with sparse and irregular data structures, making it challenging to predict data access patterns and efficiently utilize the cache. The non-blocking L2 cache's ability to overlap memory accesses and reduce synchronization overhead can be particularly beneficial for these benchmarks, as it can mitigate the impact of unpredictable data access patterns and improve overall performance.

BH, in particular, exhibits a high degree of synchronization between cores, as it frequently requires data exchange for its computations. The non-blocking L2 cache's ability to hide synchronization latency can significantly improve performance for BH by reducing the time cores spend waiting for data or synchronization operations.

The algorithms used in other benchmarks within the memory-intensive, irregular-access category might not be as sensitive to the specific characteristics of the non-blocking L2 cache. Their data access patterns might not align as well with the cache's structure, or their synchronization requirements might not be as demanding. As a result, the performance gains from switching to a non-blocking L2 cache might be less pronounced for these benchmarks compared to BH.

Factors Contributing to BH's Superior Performance Improvement could be:

- *Intensive Synchronization*: BH's algorithm relies heavily on synchronization between cores, making it more susceptible to the performance benefits of the non-blocking L2 cache's reduced synchronization overhead.
- *Data Access Locality*: BH's data access patterns exhibit better locality compared to other benchmarks in this category, meaning the data it needs to access is more likely to be located within the cache or within a short distance, making the non-blocking L2 cache's ability to overlap memory accesses more beneficial.
- *Algorithm-Cache Fit*: BH's algorithm characteristics align well with the strengths of the non-blocking L2 cache, such as its ability to handle irregular data access patterns and reduce synchronization latency.

Focusing on the BH program which benefited most from the integration of non-blocking vcache arrays, switching from blocking to non-blocking vcache shows a clear trend: DRAM utilization increases significantly. This rise becomes even more apparent with more MSHR entries, indicating the system's improved ability to handle concurrent memory operations. Additionally, the BH program's runtime steadily decreases with both the switch to non-blocking vcache and the increase in MSHR entries. This suggests a more streamlined operation with less wait time for memory accesses and reduced network congestion. Furthermore, larger datasets naturally lead to more significant performance gains in both DRAM utilization and runtime reduction.

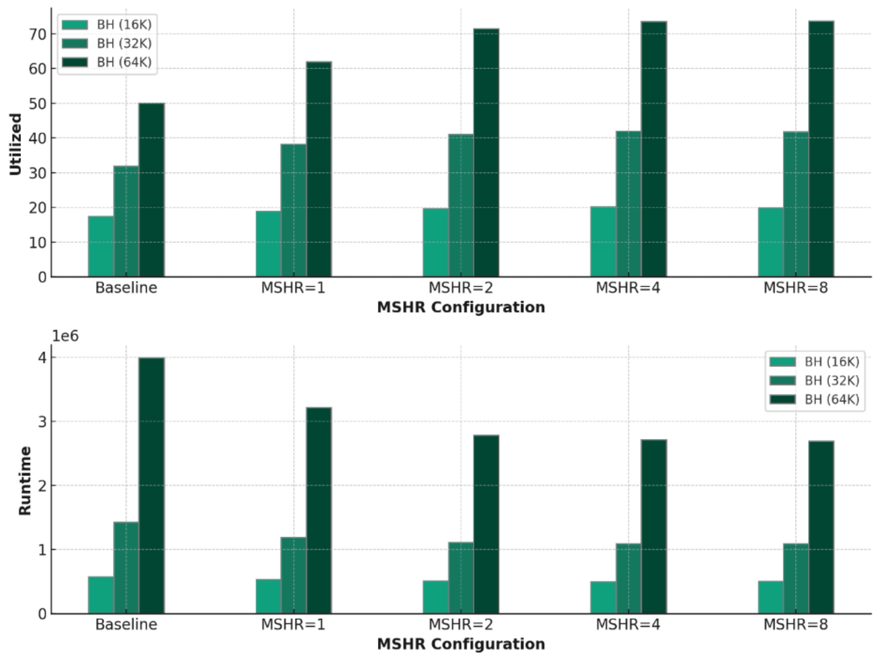
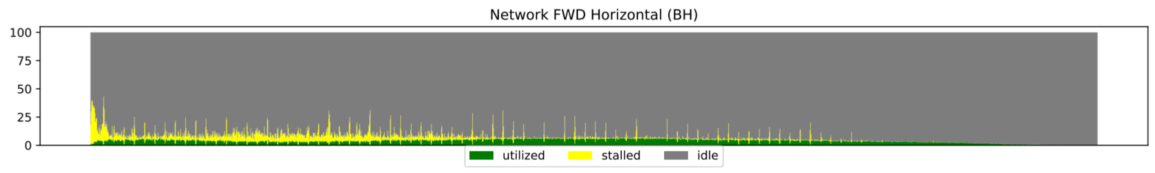


Figure 16: BH Runtime and DRAM Utilization

To take a closer look at the BH benchmark programs with body size of 16KB, we see a significant reduction in vcache load stalls, indicating the non-blocking vcache's efficiency in handling memory requests. The network stall reduction was less dramatic, likely because the 16KB data size still fits comfortably within the blocking cache. However, Figure 16 clearly illustrates the substantial benefits for the FWD network with non-blocking vcache. The figure shows a significant decrease in FWD network stalls, suggesting that non-blocking vcache effectively avoids many of these delays.



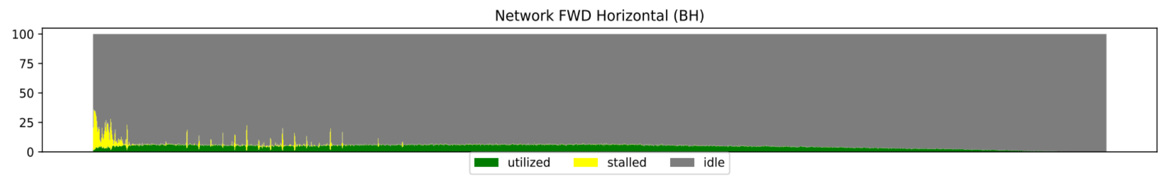


Figure 16: BH Network FWD Over Blocking Cache

8 Conclusion

The transition from blocking to non-blocking cache arrays in the HammerBlade manycore marks a significant step towards optimizing manycore architectures for high-performance computing. By addressing the challenges of network congestion and enhancing cache efficiency, this work contributes to the broader goal of advancing manycore processor design and research within the open-source and RISC-V community.

The non-blocking cache design developed and evaluated in this thesis demonstrates great improvements in memory system efficiency and overall performance, particularly in memory-intensive applications. The introduction of features such as MSHR, Read Miss Queue, and an advanced DMA system allows the HammerBlade architecture to handle multiple cache misses concurrently, reducing delays and improving throughput.

Verification and benchmarking results confirm the effectiveness of the non-blocking cache in mitigating network congestion and enhancing the overall efficiency of the manycore system. The insights gained from this research can inform future developments in manycore architectures, contributing to the ongoing evolution of high-performance computing systems.

References

[1] A. Brahmakshatriya, E. Furst, V. Ying, C. Hsu, M. Ruttenberg, Y. Zhang, T. Jung, D. Richmond, M. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. Amarasinghe, “Taming the zoo: A unified graph compiler framework for novel architectures,” in ISCA, 2021.

[2] E. Furst, “Code Generation and Optimization of Graph Programs on a Manycore Architecture,” Ph.D. dissertation, University of Washington, 2021.

[3] X. Zhang, H. Xia, D. Zhuang, H. Sun, X. Fu, M. Taylor, and S. L. Song, “η-LSTM: Co-designing highly-efficient large lstm training via exploiting memory-saving and architectural design opportunities,” in ISCA, 2021.

[4] C. Xie, X. Li, Y. Hu, H. Peng, M. Taylor, and S. L. Song, “Q-VR: System-level design for future mobile collaborative virtual reality,” in ASPLOS, 2021.

[5] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski, “A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator,” IEEE Journal of Solid-State Circuits, pp. 933–944, April 2020.

[6] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. Dreslinski, “A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm,” in Symposium on VLSI Circuits, 2019, pp. C150–C151.

[7] Q. Zheng, N. Goulding-Hotta, S. Ricketts, S. Swanson, M. B. Taylor, and J. Sampson, "Exploring energy scalability in coprocessor-dominated architectures for dark silicon," Transactions on Embedded Computing Systems (TECS), Mar 2014.

[8] B. Beresini, S. Ricketts, and M. Taylor, "Unifying manycore and fpga processing with the RUSH architecture," in Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on, 2011, pp. 22 –28.

[9] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner," in International Symposium on Microarchitecture (MICRO), 2011.

[10] J. Sampson, M. Arora, N. Goulding-Hotta, G. Venkatesh, J. Babb, V. Bhatt, M. B. Taylor, and S. Swanson, "An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors," in Conference on Field Programmable Logic and Applications (FPL), 2011.

[11] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," Micro, IEEE, pp. 86–95, March 2011.

[12] S. Swanson and M. Taylor, "GreenDroid: Exploring the next evolution for smartphone application processors," in IEEE Communications Magazine, March 2011.

[13] M. Arora, J. Sampson, N. Goulding-Hotta, J. Babb, G. Venkatesh, M. B. Taylor, and S. Swanson, "Reducing the Energy Cost of Irregular Code Bases in Soft Processor

Systems," in IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011.

[14] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor, "Efficient Complex Operators for Irregular Codes," in High Performance Computing Architecture (HPCA), 2011.

[15] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. Taylor, and S. Swanson, "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," in HOTCHIPS, 2010.

[16] N. Goulding-Hotta, "Specialization as a Candle in the Dark Silicon Regime," Ph.D. dissertation, University of California, San Diego, 2020.

[17] M. Khazraee, "Reducing the development cost of customized cloud infrastructure," Ph.D. dissertation, University of California, San Diego, 2020.

[18] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2010.

[19] M. B. Taylor, L. Vega, M. Khazraee, I. Magaki, S. Davidson, and D. Richmond, "ASIC clouds: Specializing the datacenter for planet-scale applications," CACM, pp. 103–109, 2020.

[20] S. Xie, S. Davidson, I. Magaki, M. Khazraee, L. Vega, L. Zhang, and M. B. Taylor, "Extreme datacenter specialization for planet-scale computing: Asic clouds," in ACM Sigops Operating System Review, 2018.

[21] M. B. Taylor, “Geocomputers and the Commercial Borg,” in SIGARCH Computer Architecture Today, Dec 2017.

[22] M. Taylor, “The Evolution of Bitcoin Hardware,” Computer, IEEE, Sept-Oct. 2017.

[23] M. Khazraee, L. Vega, I. Magaki, and M. Taylor, “Specializing a Planet’s Computation: ASIC Clouds,” IEEE Micro, May 2017.

[24] M. Khazraee, L. Zhang, L. Vega, and M. Taylor, “Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon,” in Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[25] I. Magaki, M. Khazraee, L. Vega, and M. Taylor, “ASIC Clouds: Specializing the Datacenter,” in International Symposium on Computer Architecture (ISCA), 2016.

[26] M. B. Taylor, “Bitcoin and the Age of Bespoke Silicon,” in International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2013.

[27] ——, “Your agile open source HW stinks (because it is not a system),” in ICCAD, 2020.978-1-4673-9030-9/20/$31.00 ©2020 IEEE

[28] ——, “BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design,” in Design Automation Conference, June 2018.

[29] H. Esmaeilzadeh and M. B. Taylor, “Open Source Hardware: Stone Soups and Not Stone Satues, Please,” in SIGARCH Computer Architecture Today, Dec 2017.

[30] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs,” IEEE Micro, pp. 93–102, Jul/Aug. 2020.

[31] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski, "Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL," IEEE Solid-State Circuits Letters, vol. 2, no. 12, pp. 289–292, 2019.

[32] ——, "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS," in 2019 Symposium on VLSI Circuits, 2019, pp. C30–C31.

[33] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric," Micro, IEEE, Mar/Apr. 2018.

[34] R. Zhao, C. Zhao, S. Xie, B. Veluri, L. Vega, C. Torng, N. Sun, A. Rovinski, A. Rao, G. Liu, P. Gao, S. Davidson, S. Dai, A. Amarnath, KhalidAl-Hawaj, T. A. C. Batten, R. G. Dreslinski, R. K.Gupta, M. B.Taylor, and Z. Zhang, "Celerity: An Open Source RISC-V Tiered Accelerator Fabric," in 7th RISC-V Workshop, 2017.

[35] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Lotfi, J. Puscar, A. Rao, A. Rovinski, L. Salem, N. Sun, C. Torng, L. Vega, B. Veluri, X. Wang, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, I. Galton, R. K. Gupta, P. P. Mercier, M. Srivastava, M. B. Taylor, and Z. Zhang, "Celerity: An Open Source RISC-V Tiered Accelerator Fabric," in HOTCHIPS, Aug 2017.

[36] L. Vega and M. B. Taylor, " RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization ," in CARRV, 2017.

[37] D. C. Jung, “Caches for Complex Open Source System-on-Chip Designs,” Master’s thesis, University of Washington, 2019.

[38] S. V. Ranga, “ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor,” Master’s thesis, University of Washington, 2021.

[39] S. Muralitharan, “TinyParrot: An Integration-Optimized Linux-Capable Host Multicore,” Master’s thesis, University of Washington, 2021.

[40] Y.-M. Chueh, “A Complete Open Source Network Stack For BlackParrot,” Master’s thesis, University of Washington, 2022.

[41] R. M. Ramstad, “Enabling Vector Load and Store instructions on HammerBlade Architecture,” Master’s thesis, University of Washington, 2024.

[42] D. C. Jung, S. Davidson, C. Zhao, D. Richmond, and M. B. Taylor, “Ruche Networks: Wire-Maximal, No-Fuss NoCs,” in NOCS, 2020.

[43] D. Petrisko, C. Zhao, S. Davidson, P. Gao, D. Richmond, and M. B. Taylor, “NoC Symbiosis,” in NOCS, 2020.

[44] Y. Zhu, M. Taylor, S. B. Baden, and C.-K. Cheng, “Advancing super-computer performance through interconnection topology synthesis,” in International Conference on Computer-Aided Design (ICCAD), 2008, pp. 555–558.

[45] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, “Scalar Operand Networks,” in IEEE Transactions on Parallel and Distributed Systems, February 2005.

[46] J. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, “Energy Characterization of a Tiled Architecture Processor with On-Chip Networks,” in International Symposium on Low Power Electronics and Design (ISLPED), August 2003.

[47] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, “Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing,” in DAC, 2021.

[48] B. Hawkins, B. Demsky, and M. B. Taylor, “ BlackBox: Lightweight Security Monitoring for COTS Binaries,” in Code Generation and Optimization, 2016.

[49] ——, “A Runtime Approach to Security and Privacy,” in European Security and Privacy, 2016.

[50] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. Taylor, and R. Kastner, “Hiding Intermittant Information Leakage with Architectural Support for Blinking,” in International Symposium on Computer Architecture (ISCA), 2018.

[51] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, “CortexSuite: A Synthetic Brain Benchmark Suite,” in International Symposium on Workload Characterization (IISWC), Oct. 2014.

[52] S. Kota Venkata, I. Ahn, D. Jeon, A. Gupta, and M. Taylor, “SD-VBS: The San Diego Vision Benchmark Suite,” in IEEE International Symposium on Workload Characterization (IISWC), 2009.

[53] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, “The Raw Benchmark Suite: Computation Structures for General Purpose Computing,” in IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 1997.

[54] M. Taylor, “A Landscape of the New Dark Silicon Design Regime,” in Design Automation and Test in Europe, April 2014.

[55] ——, “A Landscape of the New Dark Silicon Design Regime,” Micro, IEEE, Sept-Oct. 2013.

[56] M. B. Taylor, “Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse,” in Design Automation Conference (DAC), 2012.

[57] V. Bhatt, N. Goulding-Hotta, Q. Zheng, J. Sampson, S. Swanson, and M. B. Taylor, “ Sichrome: Mobile web browsing in Hardware to save Energy ,” in Dark Silicon Workshop, ISCA, 2012.

[58] N. Goulding-Hotta, J. Sampson, Q. Zheng, V. Bhatt, S. Swanson, and M. Taylor, “GreenDroid: An Architecture for the Dark Silicon Age,” in Asia and South Pacific Design Automation Conference (ASPDAC), 2012.

[59] A. Gupta, J. Sampson, and M. B. Taylor, “Qualitytime: A simple online technique for quantifying multicore execution efficiency,” in International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.

[60] ——, “DR-SNUCA: An energy-scalable dynamically partitioned cache,” in International Conference on Computer Design (ICCD), 2013.

[61] ——, “Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking,” in International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS), 2013.

[62] M. Taylor, “Tiled Microprocessors,” Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[63] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, “Evaluation of the Raw Microprocessor: An

Exposed-Wire-Delay Architecture for ILP and Streams," in International Symposium on Computer Architecture (ISCA), June 2004.

[64] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs," in IEEE Micro, March 2002.

[65] M. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, V. Strumpen et al., "The raw processor-a scalable 32-bit fabric for embedded and general purpose computing," in Proceedings of Hot Chips XIII, 2001.

[66] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal, "A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network," in IEEE International Solid-State Circuits Conference (ISSCC), February 2003.

[67] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks," in IEEE Transactions on Parallel and Distributed Systems (TPDS), February 2005.

[68] ——, "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures," in International Symposium on High Performance Computer Architecture (HPCA), February 2003.

[69] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, “Baring it all to Software: Raw Machines,” in IEEE Computer, September 1997.

[70] D. Jeon, S. Garcia, and M. B. Taylor, “Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis,” in Conference on Code Generation and Optimization (CGO), 2013.

[71] S. Garcia, D. Jeon, C. Louie, and M. Taylor, “The Kremlin Oracle for Sequential Code Parallelization,” Micro, IEEE, vol. 32, no. 4, pp. 42–53, July-Aug. 2012.

[72] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, “Kismet: Parallel Speedup Estimates for Serial Programs,” in Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA), 2011.

[73] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor, “Kremlin: Rethinking and Rebooting gprof for the Multicore Age,” in Proceedings of the Conference on Programming Language Design and Implementation (PLDI), 2011.

[74] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, “Parkour: Parallel Speedup Estimates from Serial Code,” in USENIX Workshop on Hot Topics in Parallelism (HOTPAR), 2011.

[75] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor, “Kremlin: Like Gprof, but for Parallelization,” in Principles and Practice of Parallel Programming (PPoPP), 2011.

[76] S. Garcia, D. Jeon, C. Louie, S. Kota Venkata, and M. B. Taylor, “Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning,” in USENIX Workshop on Hot Topics in Parallelism (HOTPAR), 2010.

[77] K. Zee, V. Kuncak, M. Taylor, and M. C. Rinard, “Runtime checking for program verification,” in RV, 2007.

[78] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor, “The RAW compiler project,” in Proceedings of the Second SUIF Compiler Workshop, 1997, pp. 21–23.

[79] S. Athrij, “Vectorizing the Hamerblade Compiler,” Master’s thesis, University of Washington, 2024.

[80] Y. Zhu, Y. Hu, M. Taylor, and C.-K. Cheng, “Energy and switch area optimizations for FPGA global routing architectures,” in ACM Transactions on Design Automation of Electronic Systems (TODAES), January 2009.

[81] Hu, Zhu, Taylor, and Cheng, “FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach ,” in ICCD, 2007.

[82] Zexin Fu, Mingzi Wang, Yihai Zhang and Zhangxi Tan, “Cache Coherent Framework for RISC-V Many-core Systems,” Seventh Workshop on Computer Architecture Research with RISC-V (CARRV 2023), Orlando, Florida, USA, 2023

[83] John L Hennessy and David A Patterson. 2011. Computer architecture: a quantitative approach. Elsevier

[84] HammerBlade Manycore Technical Reference Manual, https://docs.google.com/document/d/1b2g2nnMYidMkcn6iHJ9NGjpQYfZeWEmMdLeO_3nLtgo/edit#heading=h.60uv5hnu5sg6

[85] BaseJump Manycore Accelerator Network, https://docs.google.com/document/d/1-i62N72pfx2Cd_xKT3hiTuSilQnuC0ZOaSQMG8UPkto/edit#heading=h.y130vyk7ihvb

[86] D. Seo, A. Ali, W.-T. Lim, and N. Rafique, “Near-optimal worstcase throughput routing for two-dimensional mesh networks,” in 32nd International Symposium on Computer Architecture (ISCA’05), 2005, pp. 432–443.

[87] SPD: Open-Source RISC-V Manycore with Scalable Resource Organization, D. Jung, M. Ruttenberg, P. Gao, S. Davidson, D. Petrisko, K. Li, A. Kamath, L. Cheng, S. Xie, P. Pan, Z. Zhao, Z. Yue, B. Veluri, S. Muralitharan, A. Sampson, A. Lumsdaine, Z. Zhang, C. Batten, M. Oskin, D. Richmond, M. Taylor, ISCA 24, Buenos Aires, Argentina

[88] Sicolo, J. E. (1992). A Multiported Nonblocking Cache for a Superscalar Uniprocessor (Master's thesis). University of Illinois at Urbana-Champaign, Graduate College

[89] “pseudo-LRU.” https://people.cs.clemson.edu/~mark/464/p_lru.txt. Accessed: 2019-06-05.