Caches for Complex Open Source System-on-Chip Designs

Dai Cheol Jung

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

Master of Science in Electrical Engineering

University of Washington
2019

Committee:

Michael Taylor (Chair)
Scott Hauck

Program Authorized to Offer Degree:

Electrical and Computer Engineering

University of Washington

Abstract

Caches for Complex Open Source System-on-Chip Designs

Dai Cheol Jung

Chair of the Supervisory Committee:

Michael Taylor

Computer Science and Engineering

This thesis describes the RTL implementation of the cache system in a system-on-chip design that consists of a shared memory multi-core processor and a tiled manycore processor. It explores the physical design space and the microarchitectural features used in implementing such system. This work contains the details of the L2 victim cache between the manycore array and the DRAM controller, and the 8-way set-associative L1 data cache in the RISC-V multi-core processor. It also explains the implementation detail of the classic 5-stage pipelined processor in each manycore tile, related to memory access and floating-point unit extension. Verifying the correctness of the cache system can be a challenging task on its own. This work documents various testing strategies used to uncover and track down the bugs found in the memory subsystem. Such method involves using an open source memory consistency checking software and devising constrained randomized tests. This project is an extension of the open source hardware projects, BaseJump STL, which aims to reduce the amount of time it takes to create a custom chip by creating a library of reusable SystemVerilog modules commonly used in ASIC designs, and BaseJump Manycore, an open source tiled architecture designed for computing efficiency, scalability and generality.

Contents

1 Introduction

Recent trends in computer architecture have focused on low-power designs and new custom accelerators for extreme energy efficiency. Reading from the off-chip DRAM could incur hundreds of clock cycles, and it could dissipate energy that is orders of magnitude higher than reading from the SRAM. Many convolutional neural network accelerators have large banks of SRAM to store the weights of the synapses. As the requirements on power consumption have become stricter, the importance of efficiently and intelligently caching data in the small and fast local SRAM blocks has become more relevant than ever.

Another important trend is the advent of open source hardware movement. In the past few decades, the open source software movement, such as Linux and gcc, has completely transformed the landscape of software industry. It's hard to think about building a new application without any of these free and open software tools. Imagine everyone having to write their own database engines and web servers to build a simple website.

There is a famous lore, surrounding the development of Google's TPU [1]. People at Google forecasted that if everyone with a smartphone used voice search for 3 minutes a day, the current datacenter capacity will need to double. General-purpose CPUs were simply not powerful enough to handle the deep neural network inference task used for speech recognition. Datacenters are densely packed with servers, networking equipments, power distribution network along with cooling systems with multiple redundancy to guarantee high availablity. It is often easy to forget how much of an investment is required to benefit from the economies of scale. It became apparent that a new ASIC had to be built. The catch was that not only it had to be superior in performance, but also it had to be built in an insanely short timeframe.

Currently, there is no fully open source way to build a new ASIC. Virtually all processes in the ASIC design flow heaviliy rely on the proprietary EDA tools, which are complicated and expensive. The barriers to entry are very high in developing these tools from the scratch. Despite this setback, we focused on building a library of reusuable basic building blocks in SystemVerilog to reduce the amount of time spent on RTL development and verification.

My work in this thesis includes the RTL implementation and verification of the L2 victim cache for the manycore array and the L1 data cache in BlackParrot RISC-V processor. I have also extended the Vanilla Core pipeline with the IEEE-754 floating-point unit that I have ported and verified.

Shaolin Xie was leading the manycore development before I stepped in. Mark Wyse came up with the whole cache coherence protocol and the microcoded cache coherence engine. Dan Petrisko designed the BlackParrot backend. Farzam Gilani worked on building the TLB in BlackParrot. Scott Davidson and Chun Zhao are the main physical design people.

Paul Gao designed the wormhole router, and he mainly works on the designing off-chip IO system and building high-speed PCBs for our chips. Dustin Richmond and Leonard Xiang worked on the FPGA side of things.

2 Background

2.1 BaseJump STL

BaseJump STL [2] is an open source standard template library written in SystemVerilog. This library consists of highly composable building blocks that are used ubiquitously in ASIC design. By reusing highly optimized and verified hardware blocks, it aims to drastically reduce RTL development and verification time. The modules have a simple handshaking interface so that users can piece them together without having to worry about the timing inside the modules. Extensive parametrization (e.g. bit-width, number of inputs, size of memory) allows rapid design-space exploration. The source code in this library follows a consistent coding style guideline for improving readability and maintainability of the code base, and for making sure that the design is portable and synthesizable across numerous EDA tools (e.g. DC compiler, Vivado). BaseJump STL modules are used extensively in the work of this thesis, and many new modules are created as part of this work.

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

2.2 BaseJump Manycore

BaseJump Manycore is an open source RISC-V tiled manycore processor. Tile architecture first originated from MIT Raw project [3]. Each tile contains a 32-bit classic 5-stage pipelined processor, called Vanilla Core. It has been augmented with a single-precision IEEE-754 floating-point unit. These cores are interconnected by the 2D mesh network. 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. These processors have lightweight synchronization mechanisms (e.g. LR-LBR) to synchronize with other cores.

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

2.3 Celerity

Celerity [4] is an open source SoC design, composed of three tiers of computational fabric: general-purpose, massively parallel, and specialization tier. The general-purpose tier contains 5 Berkeley Rocket cores [5]. The massively parallel tier contains 496 manycore processors. The massively parallel tier and the general-purpose tier together provide a generic platform where a custom accelerator can be added for a special flavor. Lastly, Celerity has a binarized neural network (BNN) accelerator as a specialization tier to target realtime computer vision application.

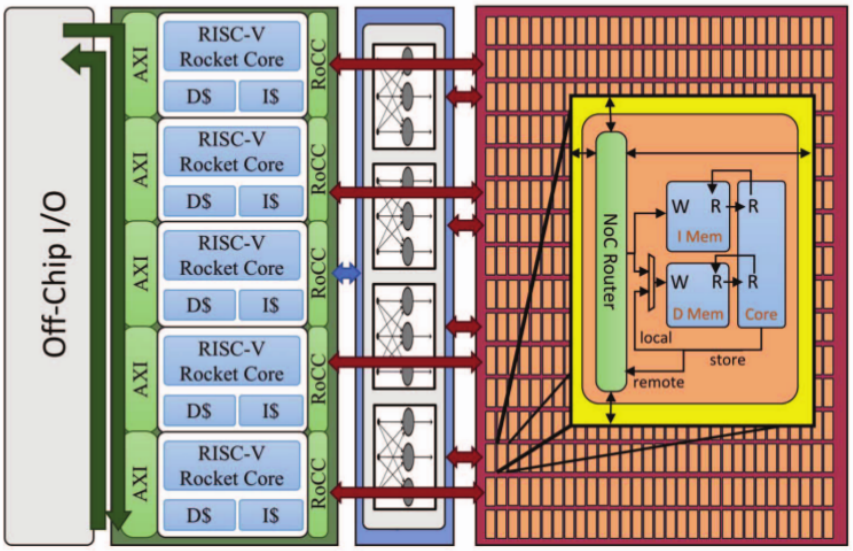

Figure 1: Celerity block diagram

3 Vanilla Core

3.1 Programming Model

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. There are four major subspaces in EVA mapping.

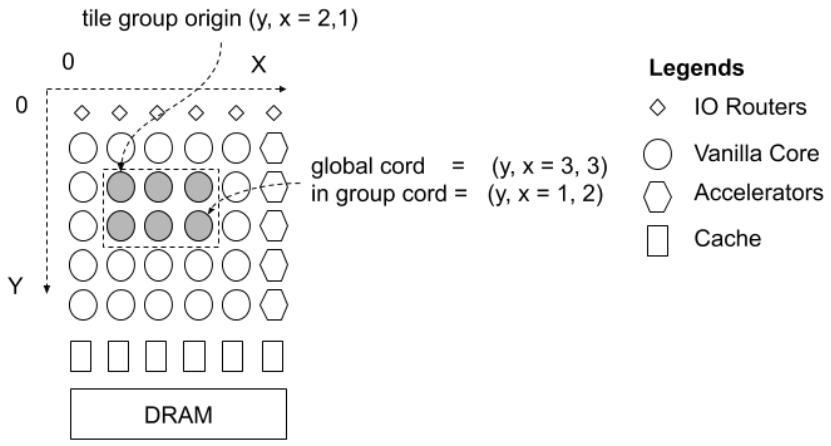

Figure 2: Top-level Manycore Diagram

- Local Data Memory: read and write from a local 4 KB single-ported synchronous read SRAM (bsg_mem_1rw_sync).
- Global: This accesses remote endpoints using the x,y-coordinate and an Endpoint Physical Address (EPA)
- In-Group: Each Vanilla Core has configurable registers to set the tile-group origin coordinate. A tile group is a rectangular subset of an manycore array, which shares the same in-group address space. The x,y-coordinate encoded in the address is added to the tile-group origin, before sent out to the network. This allows placing a tile-group anywhere in the manycore array, and the program can be written in such way that the origin of the tile-group is at (0,0).
- L2 Cache / DRAM: This address is translated into a network packet sent to the south edge of the network where there is a row of L2 caches. Each L2 cache covers mutually exclusive physical address spaces in an off-chip DRAM. These spaces are called virtual banks. The x-coordinate of the L2 cache and the virtual bank address combined provides a physical address in a DRAM.

Table 1: EVA mapping (x = x-coord, y = y-coord, ? = 1 or 0)

Resource	Address Format
Local DMEM	0000_0000_0000_0000_0001_????_????_????
Global	01yy_yyyy_xxxx_xx??_????_????_????_????
In-Group	001y_yyyy_xxxx_xx??_????_????_????_????
L2/DRAM	1xx?_????_????_????_????_????_????_????

When the chip is powered on, every tile waits in a frozen state. During this state, the PC (program counter) and the pipeline registers reset to a default value. The PC cannot increment until the tile receives an unfreeze command from the host. During the initialization step, the host loads instruction and program data into the tiles and the DRAM. This instruction cache (icache) is a 1024-entry, direct-mapped cache, and has 12-bit tag width. It has a block size of one word. Given these parameters, the PC width is 24-bit wide (byte address), and the maximum size of the kernel program that can run without address aliasing is 16 MB. The following list describes the procedure to load and run the program on a manycore array.

1. Every tile waits in a frozen state.
2. The host loads the text section of the program in DRAM.
3. The host loads the first 4KB of the program in an icache of each tile in a tile group.
4. The host loads the private data section of the program in a local scratchpad data memory.
5. The host loads the shared data section of the program in DRAM.
6. The host configures the tile-group origin CSRs in each tile.
7. The host unfreezes the tiles and the program starts running.
8. When the program finishes running, it sends a 'finish' packet to the host.

3.2 Floating-Point Unit

Most machine learning algorithms we are targeting heavily rely on floating-point operations. It is hard to imagine getting any kind of speedup using the software implementation of floating-point operations, which takes orders of magnitude more number of cycles and bloats the program size. I would argue that the FPU is the "first class accelerator" that benefits nearly all signal processing and machine learning applications. I think this FPU is a vital enhancement to make our system more enticing to many research groups.

Before we added the FPU, we made some careful estimation on how much this will increase the tile area, because this will determine how many tiles we can fit on a single die. We have designed the FPU to be parametrizable by exponent and mantissa width. We also considered adding the FP register file. Table 2 shows the area estimation for the FPU in various precisions. We estimated the area of FP register file to be 6782 umˆ2. Table

3 shows the area breakdown for a single manycore tile without the FPU. Based on these numbers, we estimated that the FPU will increase the tile area by 19.4%.

Table 2: FPU Components Cell Area (TSMC40)

	Cell Area (um^2)		
Component	**bfloat16**	**bin32**	**bin64**
add_sub	2009	4206	7000
mul	1313	5126	17726
cmp	658	1047	1614
i2f	1134	2405	4536
f2i	807	1623	3413
TOTAL	**5921**	**14407**	**34289**

Table 3: Manycore Tile without FPU Area Breakdown (TSMC40)

Type	Area (um^2)
Combinational	16124
Noncombinational	3829
Macro/Black Box	78908
TOTAL	**109011**

3.3 Processor Pipeline

Vanilla Core is optimized for general computation and inter-core interaction. It is compatible with RV32IMF ISA [6] with some exceptions:

1. Fully support RV32I except FENCE.I
   (a) Loads: `LB, LH, LW, LBU, LHU`
   (b) Stores: `SB, SH, SW`
   (c) Shifts: `SLL, SLLI, SRL, SRLI, SRA, SRAI`
   (d) Arithmetic: `ADD, ADDI, SUB, LUI, AUIPC`
   (e) Logical: `XOR, XORI, OR, ORI, AND, ANDI`
   (f) Compare: `SLT, SLTI, SLTU, SLTIU`
   (g) Branches: `BEQ, BNE, BLT, BGE, BLTU, BGEU`
   (h) Jump & Link: `JAL, JALR`
   (i) Synch: `FENCE`
2. Fully support RV32M extension
   (a) Loads: `MUL, MULH, MULHSU, MULHU`
   (b) Stores: `DIV, DIVU`
   (c) Shifts: `REM, REMU`
3. Subset of RV32A with modification

(a) Load Reserve: `LR, LBR`

4. Partially support RV32F extension

   (a) Move: `FMV.S.X, FMV.X.S`
   (b) Convert: `FCVT.S.W, FCVT.S.WU, FCVT.W.S, FCVT.WU.S`
   (c) Load: `FLW`
   (d) Store: `FSW`
   (e) Arithmetic: `FADD.S, FSUB.S, FMUL.S`
   (f) Sign Inject: `FSGNJ.S, FSGNJN.S, FSGNJX.S`
   (g) Min/Max: `FMIN.S, FMAX.S`
   (h) Compare: `FEQ.S, FLT.S, FLE.S`
   (i) Classify: `FCLASS.S`

This section describes the implementation detail of the 5-stage pipelined processor that is related to memory access. The pipeline splits into two independent paths after ID stage. The usual integer and some float instructions that eventually write back to the integer register file go to EXE stage (with the exception of `FLW`). All float instructions that write back to the FP register file go to FP-EXE stage.

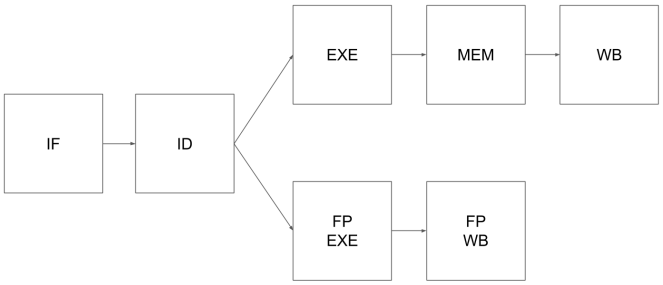

Figure 3: Vanilla Core Pipeline

- IF (Instruction Fetch)

  The instruction cache (icache) is accessed in IF stage. If the tag does not match the upper portion of PC, an icache miss signal is raised, and an "icache bubble" is inserted in the pipeline. In EXE stage, this bubble sends a remote packet to the DRAM to fetch the missing instruction. It waits for the response in MEM stage. The fetched instruction is written into the icache, and the program execution resumes.

- `ID` (Instruction Decode)

  There are register files for integer and FP operation. Each register file is accompanied by a scoreboard. The integer scoreboard is marked when a load instruction is issued

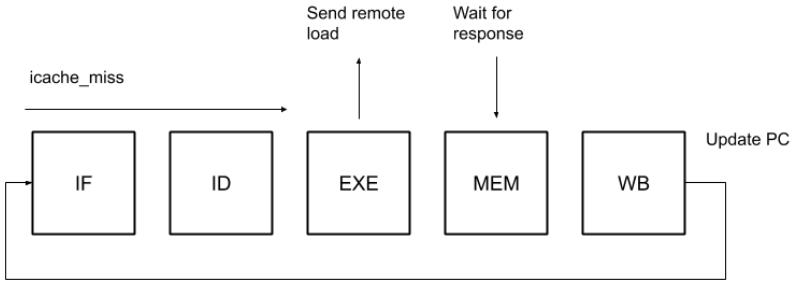

Figure 4: Instruction Cache Miss

to EXE stage. The scoreboard is cleared when a load result is either written back to the register file or has become available for forwarding. Any subsequent instruction that has data dependency on a marked register in the scoreboard stalls in ID stage until the scoreboard is cleared.

Likewise, the FP scoreboard is marked when the FP instruction issued. All instructions through FPU take three fixed cycles to compute. The FP scoreboard is cleared when the result from FPU or the load result (`FLW`) is written in FP-WB register.

For some instructions, checking the scoreboard is not sufficient to determine whether these instructions can be issued or not. For example, the float-to-int convert instruction reads a floating-point value from the FP register file, does the conversion, and writes back to the integer register file. Since there is no forwarding path from FP-WB to EXE stage, it needs to check FP-WB does not have any data with dependency. Similarly, for a move instruction from the integer to FP register file, we need to check that there is no pending writeback data in EXE, MEM, and WB, since there is no forwarding path from EXE, MEM, and WB to FP-EXE stage.

- `EXE` (Execute)

  EXE stage has four functional units: ALU, LSU (load-store unit), an iterative multiplier and divider, and FPU-int.

  LSU inspects the load/store address, and decides whether to access local data memory or to send out a remote packet. It also checks if the current instruction is an icache bubble that originated from IF stage. If the remote packet cannot be sent, because the network is busy, the pipeline stalls until the network is ready.

  A remote packet has a flag to indicate whether this is for a remote store/load or an instruction fetch. There is also a flag to distinguish between a load to integer or FP register file. These flags are returned with the responses, so the pipeline knows how to handle the responses that arrive asynchronously.

  Once the remote load packet is sent out, it could take a long time to get the word back. Events, such as cache miss in L2, writing back the dirty block for eviction,

processing earlier requests from other L2 caches, and closing and opening a new row in the DRAM all accrue to this cost. Instead of blocking the pipeline, the remote load exits the pipeline without waiting for the response. The scoreboard remains set until the response comes back, stalling any subsequent data-dependent instructions in ID stage.

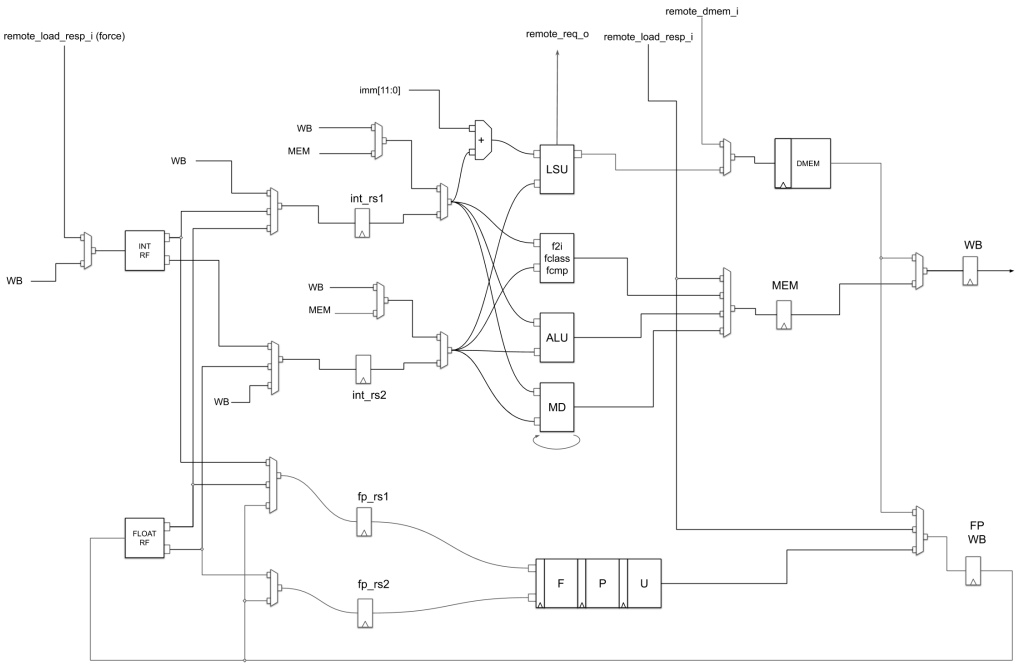

Figure 5: Vanilla Core Forwarding Path

- MEM (Memory)

  MEM stage pipeline could get data from three different sources. One of these are forwarded to WB stage. First, it could be an ALU or a multiplier result. Second, it could be the read output from local data memory. Third, it could be a remote load response. The remote load response can also be inserted in FP-WB stage, in case of `FLW`.

- WB (Writeback)

  WB stage only stalls when the remote load response stalls the pipeline to write back to the integer register file.

- FP-EXE (FP-Execute)

  FPU has a single-precision floating-point adder-subtractor, multiplier, and int-to-float converter. FPU has three pipeline stages inside.

- FP-WB (FP-Writeback)

  FP-WB never stalls. It can always write back to the FP register file. Writeback data could be originating from remote and local `FLW` or FPU. FPU stalls if it has

valid output, but there is also a load response, which has higher priority.

4 L2 Victim Cache

4.1 Design Overview

We designed this L2 cache to be two-stage pipelined. The architecture of this cache was inspired by the data cache in Raw [3]. 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. Here are some high-level descriptions of this cache:

- Two-way set associative
- Write-back, Write-allocate, Fetch-on-write
- LRU replacement policy
- Write buffer
- Flush/Invalidate instruction
- Cache line locking

In the tag-lookup stage, the tag memory is read in order to determine cache hit or miss. The tag memory is addressed by the index portion of the input address. Each entry in the tag memory contains a valid bit, lock bit, and tag for every cache block. A data memory is also read for load instructions.

There are four parameters that can vary the size of SRAM blocks in the cache: data width, address width, block size, and a number of sets. For example, if the address and data are 32-bits wide, each cache block has 8 words, and there are 512 sets in total, the capacity of this cache is 32 KB. In this case, the index is 9-bit wide, and the tag is 18-bit, so the tag memory should be 40×512 bits large (including valid and lock bits). There is also a status memory, which holds a dirty bit for every block and a LRU bit for every set. Since this is two-way associative, the LRU way can be encoded by using only one bit.

The L2 cache has a ready-valid handshaking interface at its input and output. Requests arrives in a form of a packet that contains address, data, opcode, mask, and sign-extend fields. The L2 cache can handle load and store of various data sizes: word, half, and byte. For each request accepted, the L2 cache produces one output. For instructions like store or invalidate, which has no return value, the output is simply zero. There are also instructions for software cache management. `TAGST` instruction can directly modify an entry in tag memory. This is necessary for manually invalidating every cache block during the boot-up process. `TAGLA` and `TAGLV` can directly read the tags and valid bits, so that the programmer can inspect the tag memory content for a debugging purpose. There are

Figure 6: L2 Cache SRAM layout

also instructions to manually flush or invalidate a cache line. `AINV`, `AFL`, and `AFLINV` check if the input address is cached or not, and invalidate and/or flush the block that contains the input address.

When there is a store miss, the entire missing block is brought into the cache, including the part of the block that is not being written. The dirty bit is cleared when the new block is requested. After the new block arrives and the new data is written, the dirty bit is set to one.

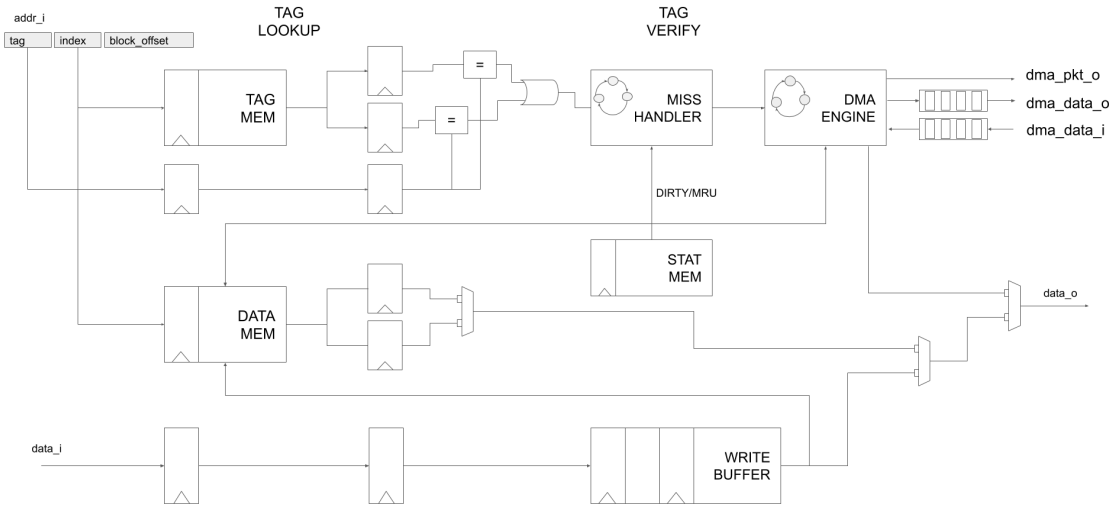

Figure 7: Victim cache high-level schematic

4.2 Miss Handling FSM Logic

1. `START` : The miss handler waits in `START` state, until there is a cache miss detected in the tag-verify stage. Once it transits to the next state, the pipeline stalls, and the cache no longer accepts any new requests. If load or store arrives with a cache miss, it shifts to `SEND_FILL_ADDR` state. If invalidate or flush instruction arrives, which does not require refilling a cache block, it shifts to `FLUSH_INSTR`. Dirty bits and LRU bit are read in this state to make a decision on eviction and replacement later.

2. `FLUSH_INSTR` : Valid and dirty bits are set to zero, if it's an invalidate operation. If it's a flush operation (`TAGFL`, `AFL`, `AFLINV`), it shifts to `SEND_EVICT_ADDR` state. For `AINV`, it shifts directly to `RECOVER` state.

3. `SEND_FILL_ADDR` : A DMA read request is sent out. The invalid or LRU way is chosen to replace the block if needed. If the LRU way is dirty, it shifts to `SEND_EVICT_ADDR` state. Otherwise, the miss handler shifts to `GET_FILL_DATA`. To mitigate a long read latency from DRAM, the DMA read request is sent out first, so that a block can be evicted, while the read data is in-flight. The tags and dirty bits are also updated in this state.

4. `SEND_EVICT_ADDR` : As soon as the DMA write request is sent out, it shifts to `SEND_EVICT_DATA`.

5. `SEND_EVICT_DATA` : The miss handler triggers the DMA engine to start reading the evicted block from the data memory and send out one word every cycle. It must wait for the write buffer to be empty, before triggering the DMA engine.

6. `GET_FILL_DATA` : Also in here, the write buffer must be empty, before initiating refilling the block. For a load miss, while the block is being written to the data memory, it saves the word that needs to be presented at the output in a snoop register, so that the data memory does not have to be read again.

7. `RECOVER` : This is an intermediate state before `DONE` state. During this cycle, the instruction in the tag-lookup stage reads tag and data again in case it was modified by the miss handler.

8. `DONE` : Finally, the return data is presented at the output, and the pipeline continues when the output is acknowledged.

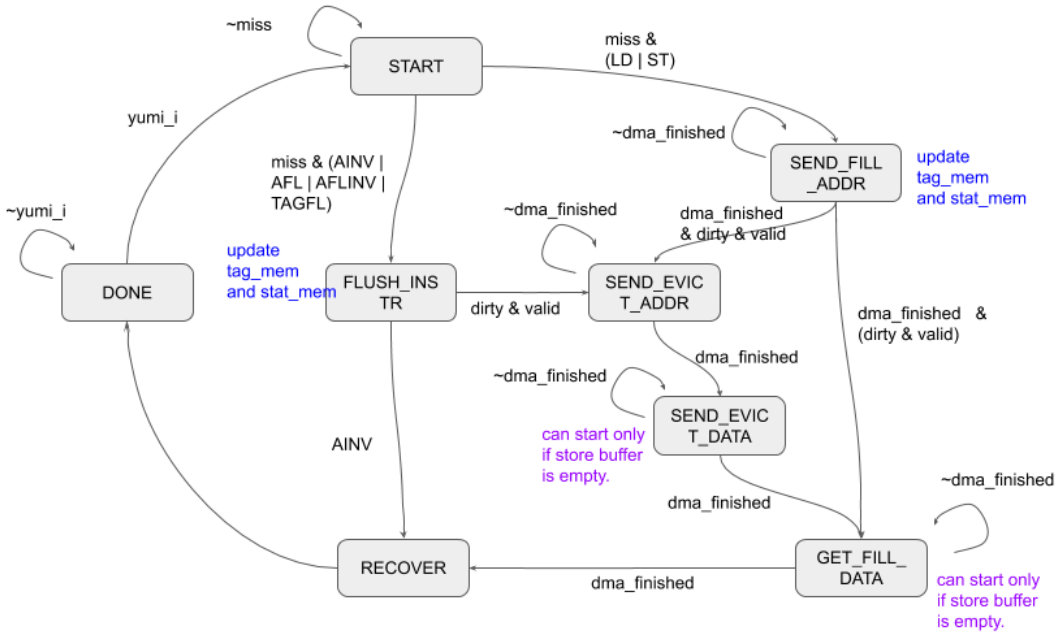

Figure 8: Miss handler Finite state machine

4.3 Write Buffer Operation

Load instructions read from the data memory in the input stage, and store instructions write to the data memory in the verify stage. Since the data memory is implemented using a single-port SRAM, there is a structural hazard between the load and store. This problem could be solved by stalling the incoming load instruction, when the store instruction is writing to the data memory in the verify stage, but this approach will create an extra latency for the load instruction. Instead of stalling, the store instruction in the verify stage queues its data, byte mask, way, and address in the write buffer, so that the incoming load instruction can read from data memory without causing structural hazard. The data read by the incoming load instruction may not be up-to-date, because the data queued in the write buffer has not been written to the data memory. The load instruction also snoops the entries in the write buffer to see if there is any entry with a matching address, and bypasses the latest matching data. The DMA operation by the miss handler also reads and writes from the data memory. If the DMA operation evicts a block while there is a valid entry in the write buffer, the evicted data might still be stale. Therefore, the DMA engine must wait until the write buffer is completely empty.

The write buffer needs to hold up to two store entries. At first, it may not be obvious why only two entries are required. Figure 9 illustrates the write buffer operation in detail. Figure 9 shows the pipeline stage of the cache not the write buffer. Conceptually, the write buffer is just a FIFO on the side, where the store data is queued if the data memory is not available to be written. The store data can directly bypass the write buffer, if there is no other entry in the write buffer, and the data memory is available. Assume there is

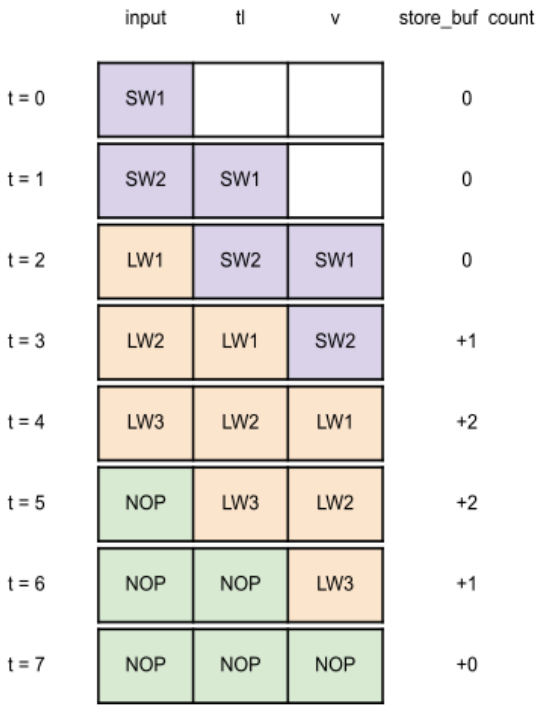

Figure 9: Write buffer operation

no miss, and all loads and stores are to the same address. At $t = 2$, SW1 wants to write to the data memory, but LW1 is in the input stage, so SW1 is queued in the write buffer. As a result, at $t = 3$, the write buffer counter is incremented to +1. At $t = 3$, since there is another load (LW2) in the input stage, SW2 is queued to the write buffer as well, and the counter incremented to +2 at $t = 4$. At $t = 3$, LW1 in the tag-lookup stage needs to check if SW2 and the write buffer has any matching address. At this time, the write buffer has only one valid entry, which is SW1. At $t = 4$, yet another load (LW3) waits at the input. The write buffer still cannot write to the data memory at this time, and the write buffer counter remains at +2. At $t = 5$, since there is no incoming instruction, the entry in the write buffer can be written to the data memory, and the counter can decrement. If there had been another load at $t = 5$, then the counter would remain at +2. If there had been a store at input, then the write buffer would be able to write to the data memory.

4.4 Cache Line Locking

We added cache line locking mechanism for two main purposes. If a block is invalid and locked, a new block cannot be placed in this block. This is useful for a rare occasion when a SRAM cell is defective, and a particular cache line cannot be used. If the block is valid and locked, this block cannot be chosen for eviction. If we know some address is accessed very frequently a priori, we could take advantage by locking the block to prevent the block getting evicted. We assume that the user cannot lock both ways in a set.

4.5 Interface Converters

4.5.1 Attaching to Manycore Link

Each L2 cache is an endpoint on the network, which can be addressed by the NPA. If the MSB of the endpoint address is `1'b0`, then this accesses the DRAM. Otherwise, it accesses the tag memory space by using `TAGST` and `TAGLA`. This allows the host to clear the tags during initialization or to inspect the tags for debugging.

4.5.2 Attaching to DRAM Controller

A round-robin module takes L2 DMA requests, and sends commands to the DRAM controller. There are two submodules called RX and TX. RX routes data received from DRAM controller to the correct cache. It is expected that the read data will return in the order the requests are sent. TX routes the eviction data from L2 to the DRAM controller. When the round-robin module sends the read and write request, it queues the id of the cache to RX and TX, respectively. Reading from this queue, RX and TX modules know the order to route the data. This separation of RX and TX logic is a very convenient abstraction, because usally read and write channel operations are independent from each other. Data bus width on L2 DMA interface and DRAM controller is 32-bit wide. The data transfer is burst-based, and each request starts one data transfer of a size equal to one cache block.

4.5.3 Attaching to AXI-4 Interface

Converting the L2 DMA interface to AXI-4 [7] uses the RX/TX abstraction in a very similar way, except that it requires an extra logic to convert different data width. The L2 DMA interface has data width of 32-bit. AXI-4 data bus is usually wider (256, 512 bits). The RX module serializes wider data into smaller words, and the TX module does the opposite by deserializing.

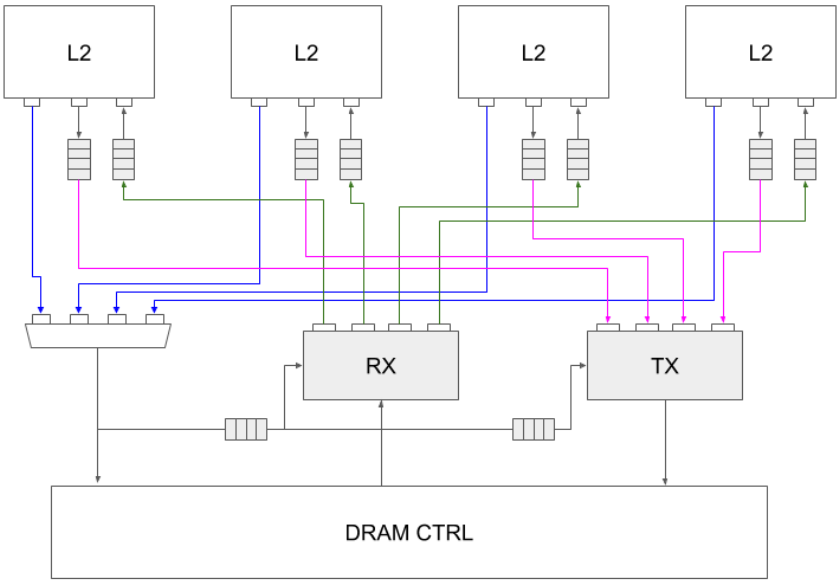

Figure 10: L2 DMA Interface Converter. RX/TX Abstraction. Blue line is for request. Pink line is for evict data. Green is for refill data.

5 L1 Data Cache

5.1 Design Overview

BlackParrot is a 64-bit RISC-V multi-core processor in a nascent stage of development. It is meant to be a Linux-capable open source core, that is modular and area-efficient. BlackParrot is divided into three major components. The Frontend is responsible of PC generation, instruction fetch, and branch prediction. The Backend implements the execution pipeline. The Memory-end contains the directory and the cache coherence engine (CCE). The CCE makes all the decisions in the coherence system, and connects to the next-level memory. The L1 data cache in BlackParrot borrows many microarchitectural ideas from the L2 cache. It has two pipeline stages (tag-lookup and tag-verify). It also has the same write buffer in L2. It has a ready-valid handshaking interface at input, and a valid-only interface at output.

It is 8-way set-associtive with pseudo-LRU replacement policy. It has 64 sets and 8-word block size. The 32 KB data memory is divided into 8 SRAM banks for block interleaving to speed up block access. L1 instruction cache is a simpler version of L1 data cache without the logic related to store capability.

The L1 cache is virtually-indexed and physically-tagged. In a scheme for a page-based virtual memory system, called "Sv39" [8], the 39-bit virtual address space is divided into 4 KB pages. With the block size of 8 64-bit words, the maximum width of an index that can be used without causing virtual address aliasing is 6-bit. For a direct-mapped cache, this limits the capacity of the cache at 4 KB. By increasing the associativity, we

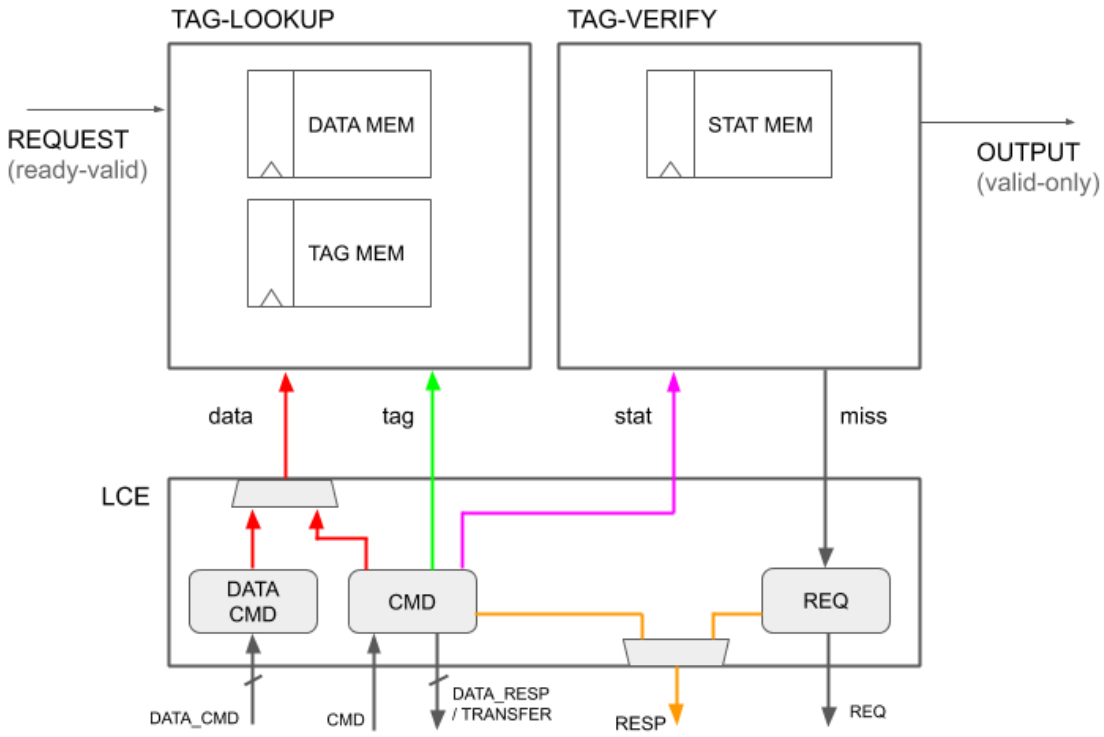

Figure 11: BlackParrot Data Cache Block Diagram highlighting Resource Contention

can increase the cache capacity. The bottom 12-bits are the untranslated portion of the virtual address. TLB (Translation Lookaside Buffer) is also accessed outside the cache to translate the 27-bit virtual tag into a physical tag, as an instruction enters the tag-lookup stage. Either TLB miss or translated tag arrives at the end of the tag-lookup stage.

The main difference from the L2 cache pipeline is that the L1 pipeline does not stall in an exceptional events like cache miss or TLB miss. When those events occur, the pipeline is flushed, and the processor waits until the miss is resolved before it starts reissuing the instructions that has caused the miss.

The L1 cache has two parts: the cache pipeline where load and store requests go through. Local cache engine (LCE) that handles MESI cache coherence protocol. Cache coherence is explained extensively in this book [9]. LCEs and CCEs are connected on the coherence network, where messages are passed back and forth. There are five channels on this network.

1. Outgoing

    - Request: When there is a cache miss, the request is sent to the CCE using this channel. The request contains the block address, LRU way id, a dirty bit of the LRU way, and whether it's load or store miss.
    - Response: the LCE sends an acknowledgement to the CCE through this channel.

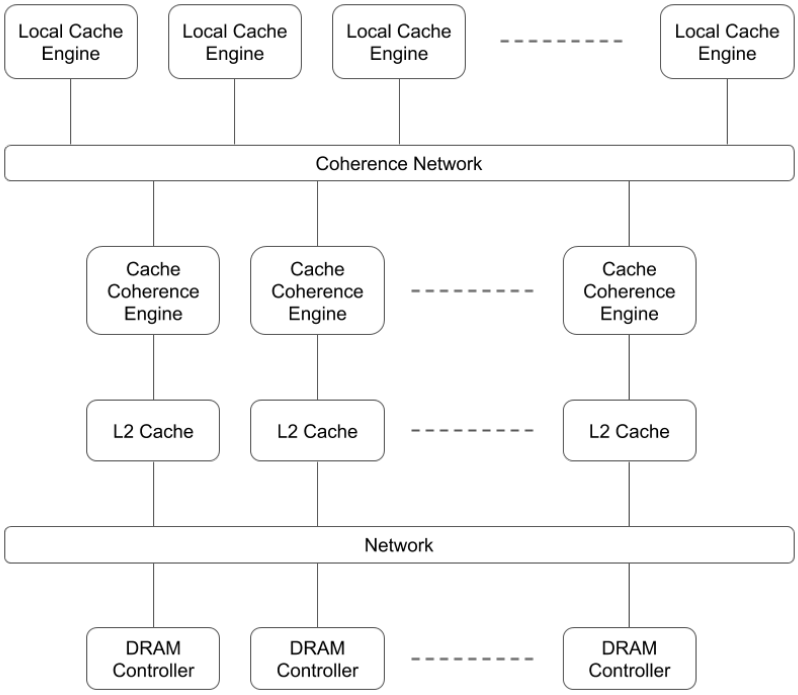

Figure 12: BlackParrot Memory End

- Data Response: When the LCE is asked to write back a cache block, the block data is sent to the CCE via this channel. The routers in this channel is wormhole flow-controlled. The large cache block (512-bit) is broken down into small word-sized flits.

2. Incoming

   - Command: Commands from the CCEs to invalidate, write back, transfer, set tag arrive through this channel. Only the CCE commands can modify the coherence states and tags.

3. Bi-directional

   - Data Command: Only the LCEs are the destination in this channel. Two sources are the CCEs and other LCEs. The cache block for refill goes through here. The routers in this channel are also wormhole flow-controlled.

5.2 Preventing LCE Starvation

There is a contention between the LCE and the cache pipeline trying to access the memory resources (tag, data, status). In this situation, the pipeline always gets the priority. The LCE waits for a bubble in the pipeline to access the resource. To prevent the LCE from waiting for too long, we added a counter that increments when the LCE needs to access something but did not get any chance. Otherwise, the counter resets to zero. When this counter hits the max value, the L1 cache deasserts the ready signal at the input for one cycle to artificially create a bubble down the pipeline so that the LCE can make a progress.

5.3 Arbitration inside LCE

Two coherence requests could contend for the same resource (e.g. network channel), and correct arbitrations may be required. Figure 13 illustrates a case, where LCE1 sends a load miss for a block that has already been cached in LCE2 in an Exclusive state. The CCE first invalidates the block in LCE2, and then sends a transfer command to LCE2. LCE2 responses with an invalidate ack. After LCE1 receives both data transfer and set_tag command from the CCE, it responds with a transfer ack. We could imagine a situation where LCE has to send out both transfer ack and invalidate ack. In this case, which ack should be sent out first? We prefer to choose an event that does not trigger more events, or an event that happens later in the sequence of events. Otherwise, spawning more messages could lead to the congestion in the network. In this example, the coherence ack gets higher priority, as it happens later in the sequence.

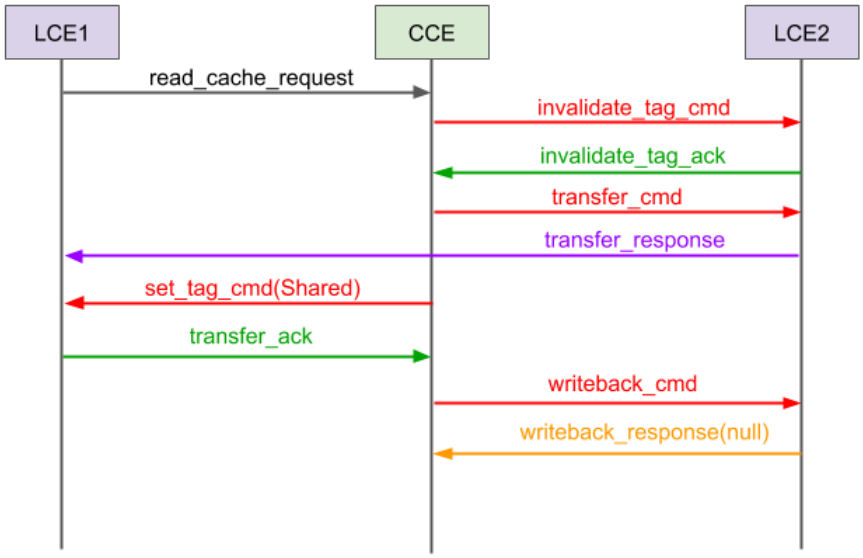

Figure 13: LCE-CCE Sequence Diagram. Load-miss. Block cached in another LCE in Exclusive state.

5.4 Data Memory Organization

There are two major data memory access patterns. First, load and store access the nth word from each block in a set. Second, eviction and refill access an entire block. If all the words in a block were in the same data memory bank, then it would take multiple cycles to read out the block. In order to speed up the block-sized accesses, the cache blocks are interleaved across the banks.

Figure 14 shows the address interleaving scheme for the 4-way set-associative cache, but the same idea can be easily extended to the 8-way case. In this organization, each bank is one word wide (64-bit). The number of words in a block and the number of banks are set equal to the associativity. The key idea is to use the bank id as a hash key. For load and store, we XOR the block offset with the bank id to obtain the way id. For eviction and refill, we XOR the way id with the bank id to obtain the block offset. This relationship can be expressed as (`way_id ^ block_offset = bank_id`).

The cache blocks need to be de-interleaved before they go out to the network. This de-interleaving logic can be implemented by n of n-input 64-bit multiplexor for n-way associative cache, but this could end up taking lots of combinational logic area. Instead, we take the advantage of the special property in this interleaving scheme. We created a butterfly multiplexor (Figure 15), which has the stages of muxes which interleave the input data, based on the select input. The output data width is equal to the input data width. The unit of swapping increases in the later stages. The lowest-order select bit swaps odd and even words. The highest order select bit swaps the upper half and lower half of the input array. This pattern mirrors that of the FFT butterfly network. This approach saves the hardware resources in the order of $O(\log_2 n)$.

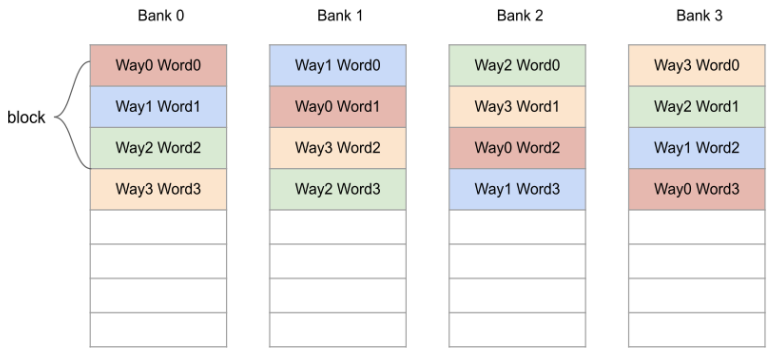

Figure 14: Block Interleaving Scheme

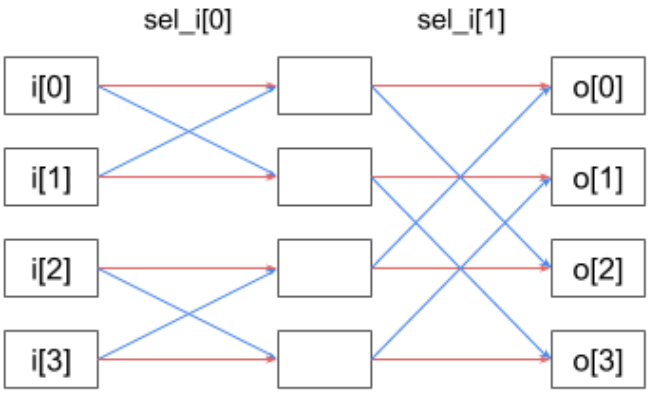

Figure 15: Butterfly Multiplexer

5.5 Tree Pseudo-LRU Algorithm

Tree pseudo-LRU [10] algorithm uses 7-bits to keep track of the least recently used (LRU) block in the 8-way set-associative cache. These bits are organized in a tree-like structure (Figure 16), so that '0' means 'take left', and '1' means 'take right' to find the LRU way. Using these bits, we can calculate the LRU way using a simple combinational logic represented in Table 4. When a way is accessed, LRU bits are updated in such way that the traversal of the tree is directed away from the referred way. This requires updating only three bits. The important advantage of this algorithm is the fact that we can update the LRU bits without having to read them first.

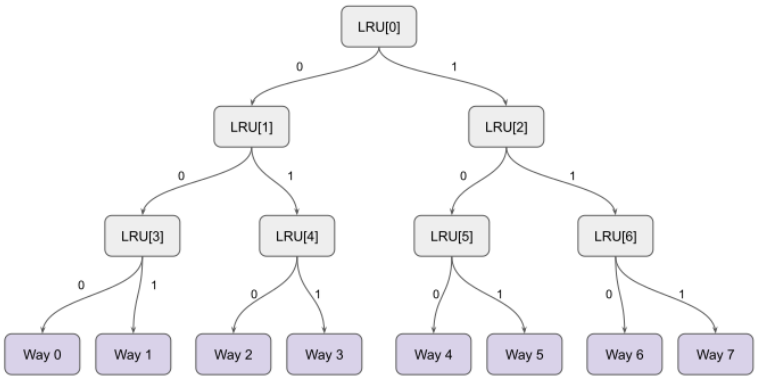

Figure 16: Tree Pseudo-LRU Algorithm

Table 4: Tree pseudo-LRU encoding. Next LRU state. 'z' means "don't care". '-' means unmodified.

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

5.6 Uncached Access

The L1 cache can make an uncached access to the physical address space without bringing in the cache block. This is useful for accessing the memory-mapped IO devices, or storing data in the manycore tile scratchpad memory. The uncached load and store cause cache misses, and flush the pipeline until the response comes back from the CCE.

6 Verification Strategy

6.1 Hand-written Regression Test

In order to test the L2 cache, we developed a testbench to send requests and check the responses at the output. It tests clearing the tags, writing different data granularity (e.g. byte, half, word), and using flush/invalidate instructions. It tests whether the write buffer bypasses correctly, or whether the miss handler correctly evicts the LRU way. This can be verified by inspecting the tags using `TAGLA, TAGLV` after the replacement. When new bugs were found during the integration with the manycore array, new tests that replicate the bugs were created, and appended to this regression test suite.

6.2 Constrained Random Testing

In order to test coherence protocol, we set up a testbench, where multiple L1 data caches are connected to multiple CCEs backed by block SRAMs. These L1 caches are fed with random load and store requests to a random set of addresses. Each store value is unique, and modding the value by the number of cache gives the id of the requestor. When the load and store are committed, the simulation prints a trace in a format that the AXE consistency checker [11] can understand. Each line of the trace contains a thread id, a memory address, and a load or store value. AXE reads the trace file, and checks if the memory consistency has been violated. We expect that the output traces are sequentially consistent for this setup. The topic of memory consistency model is discussed at length in this book [9]. Since every store value is unique, we can easily track down where the request originated from, and narrow down the root cause of the bug. This is a very powerful method, because it tests the memory subsystem in isolation, rather than trying to debug the full system for the bug found in millions of cycles into the program execution.

```
0: M[0]  :=  4       // {threadId}: M[{addr}]
1: M[64] :=  1       // ':=' means store.
1: M[3]  ==  0       // '==' means load.
2: M[0]  ==  4
0: M[0]  :=  8
3: M[7]  :=  3
2: M[64] ==  1
```

6.3 Not Random Enough

After running millions of randomized traces, we were almost convinced that the cache was bug-free. However, we quickly realized that the randomized testing by itself is not a panacea for verifying the memory system. Initially, there was no interval between two cache requests, so we added some intervals of random length, which uncovered some more bugs in the cache. Also, we were using an ideal synthesized block memory with no timing characteristics. We replaced the memory end with a Verilog DRAM simulation model provided by Micron [12] that is representative of more realistic timing behavior of the

DRAM devices. This effort not only helped find more timing related bugs in the L2 cache, but also found several bugs in the open source DRAM controller we were developing.

7 Synthesis

In this section, I take a step back and explore the common themes between the L2 victim cache and the L1 data cache – the challenges encountered and the solutions to overcome them, some interesting questions, and take-home lessons.

Caches are very complex and large in terms of code size. The L2 victim cache was 1,799 lines of code, and the L1 data cache was 2,736 lines of code. This is comparable with the lines of code for the core logic of Vanilla Core, which is 1,942 lines of code. It depends on what is considered to be the "core" logic.

Both L1 and L2 caches have a two-stage pipeline and a write buffer to allow load and store every cycle without stall. What could be an alternative pipeline structure? One downside of this pipeline design is that for load instructions, all the data memory banks need to be accessed in the tag-lookup stage, even for a cache miss. Charging up all the large SRAM banks ($>$ 4 KB each) may not be ideal for the dynamic power consumption. We could think of a pipeline design that first reads the tags, and then decides whether to read only one bank with a cache hit, or to raise a cache miss signal. This pipeline could possibly be implemented with one extra stage with tag-verify and data-read in the same cycle. Whether this extra pipeline stage is worth the improvement of the overall power consumption is an open question.

In hardware design, it is natural to design things in a power of two (e.g. data bus width, address space size), because it is the most efficient use of number space given by the binary numbers. However, there are situations where we can only fit non-power of two physical structures on a chip. For example, it may only fit 6×6 manycore array on a given die area. One of the main features of BaseJump STL is extensive parameterization. If we were to place the L2 victim cache at the bottom of each column, what would be the best way to divide up the DRAM physical space among the six L2 caches?

Going back to the two-stage pipeline implementation, the pipeline structure boils down to the different memory blocks (tag, data, status), a cache miss handler that accesses those memories, the pipeline logic, and the write buffer. The L1 data cache has a virtual tag translation interface that arrives in the tag-lookup stage. The L2 cache is just a special case where the translated tag is same as the untranslated tag. The next-level memory interface is different for those two caches. The L2 cache has very minimal request, read data, write data interface, and it does not receive any command from the DRAM controller. The L1 cache has more number of network channels, and it receives various commands from CCEs for the coherence protocol. If we were to build another cache with more sophisticated replacement policy or a non-blocking pipeline, how would the pipeline structure will need

to adjust? Is there a possible hybrid design where the two designs can converge?

8 Conclusion

BaseJump Manycore with the L2 caches has been successfully implemented on AWS F1 Instance [13]. A row of the L2 caches connects to the AXI-4 DRAM interface in the F1 shell interface. The same architecture has also been mapped to the Xilinx VCU128 Evaluation kit, which has the similar UltraScale+ FPGA with the high-bandwidth-memory interface. Because of the 32-bit endpoint virtual address space, the maximum DRAM space we can utilize is severely limited. We will add support for the 64-bit virtual pointer to fully utilize all the DRAM space available. There is an on-going effort in hacking compilers and programming languages to make the manycore processors more programmable and accessible. We chose to create an FPGA image so that people can validate if the current specification of the manycore architecture is sufficient for the programming capabilities we desire. By quickly programming on the FPGA, we can get feedbacks from the programmers and iterate, before we send out the design to the foundries.

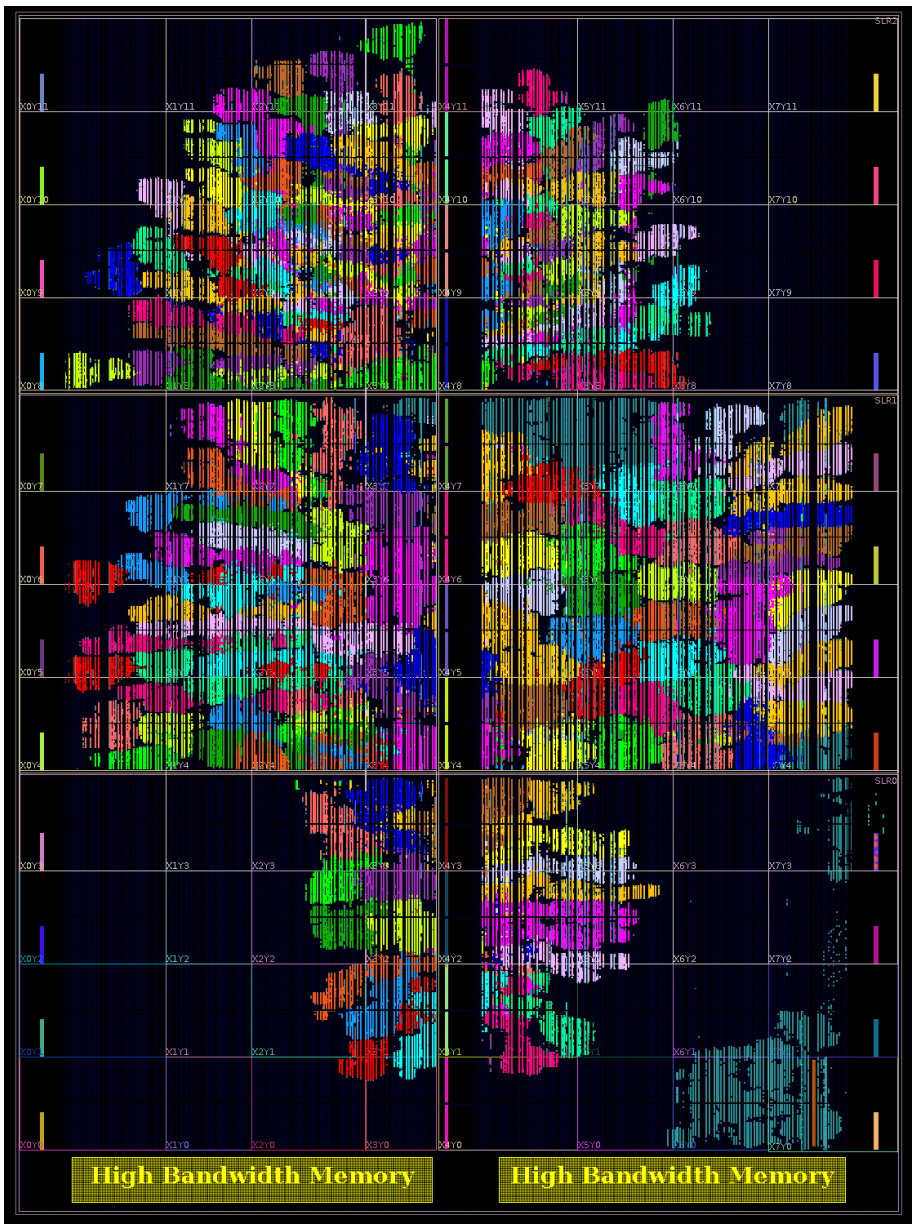

Figure 17: 12x12 Manycore on Xilinx VCU128 board (XCVU37P) (By courtesy of Leonard Xiang)

This is a floorplan of the 16 KB L2 victim cache. The aspect ratio of the layout has been deliberately chosen to match the dimension of one manycore tile. The synthesized logic is almost as large as the status and the tag memory together. There is a lot of unutilized space, where more logic could fit. We will probably add a parameter to vary the associativity, implement more sophisticated replacement policy, and make the pipeline non-blocking to handle multiple misses at a time.

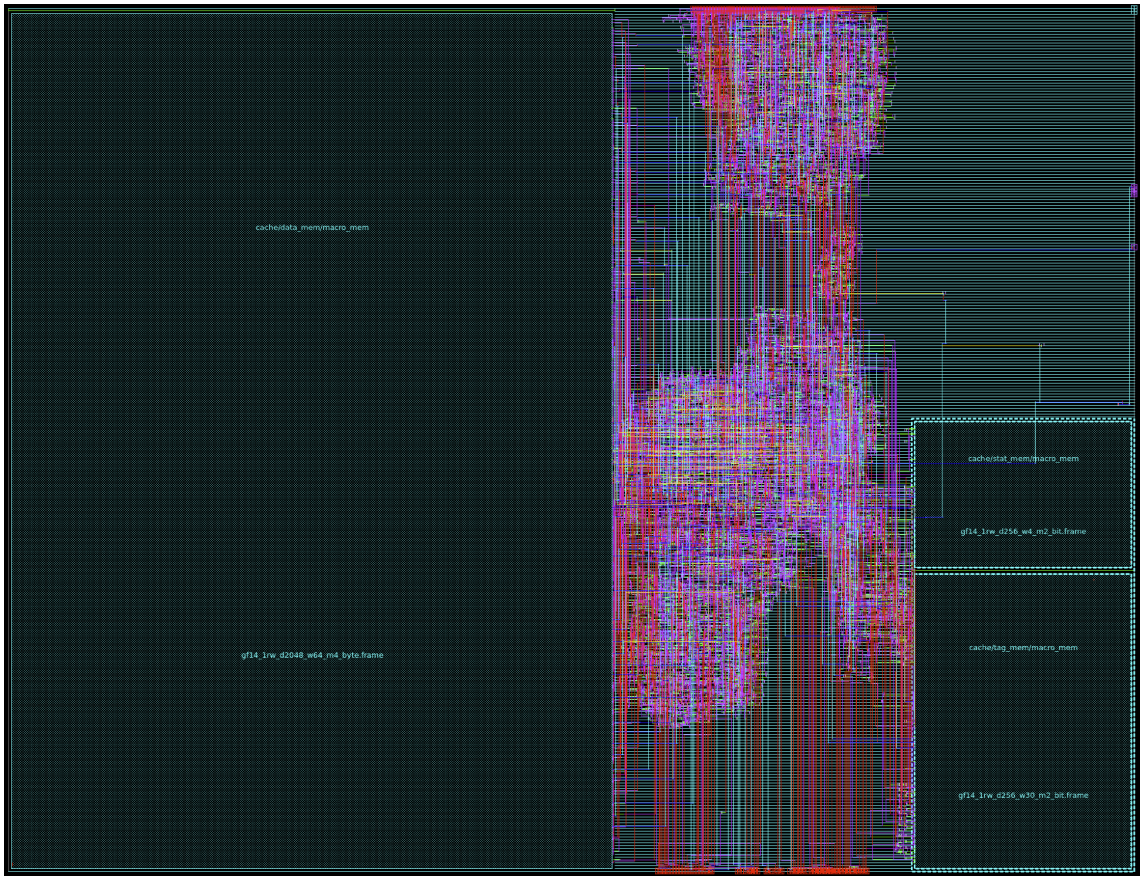

Figure 18: 16 KB L2 Victim Cache Floorplan (185.38 um x 144.96 um) (GF 14 nm) (By courtesy of Scott Davidson)

This is a floorplan of the single-core BlackParrot RISC-V processor. Most of the area is dominated by the SRAM blocks of the L1 instruction and data cache. The pink area on the left is the frontend of the processor, which is responsible for PC generation and instruction fetch. The green area on the right is the backend, which implements the execution pipeline. The microcoded cache coherence engine is highlighted in the middle.

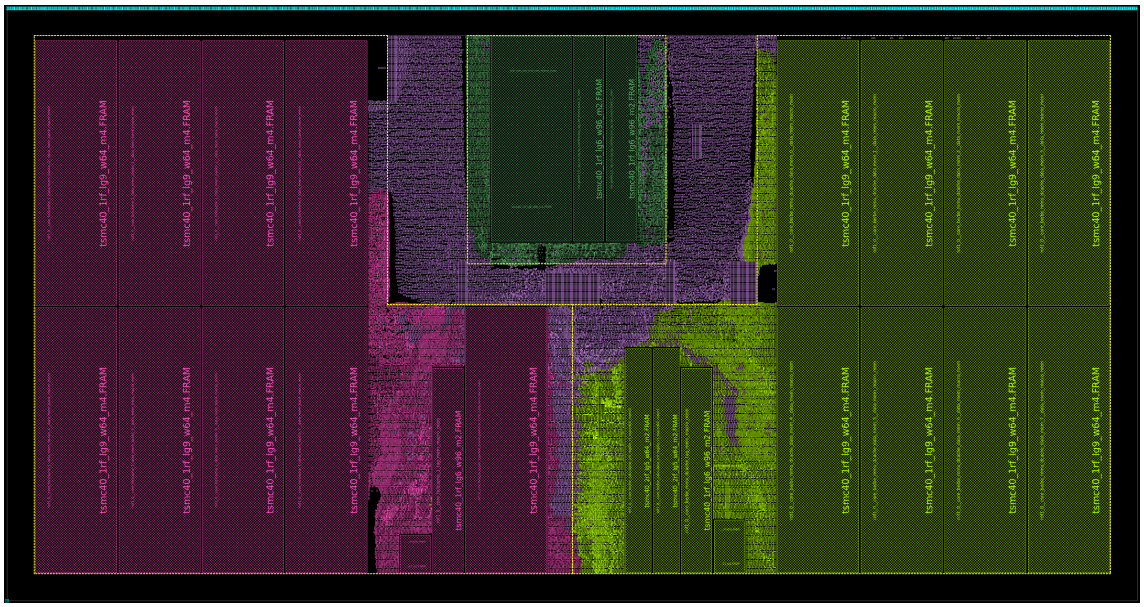

Figure 19: Single-core BlackParrot Floorplan (32 KB I-Cache, 32 KB D-Cache) (1250 um x 655 um) (TSMC 40 nm) (By courtesy of Chun Zhao)

Acknowledgement

First, I would like to thank my parents, Kiro Jung and Sohui Mo, my sister, Jihae Jung, and my dog Haru for their support. Without their support, I would not have this opportunity to study in grad school.

I would like to thank my thesis advisor, Professor Michael Taylor, for truly inspiring me to do great work. He has always challenged me with fascinating and increasingly more difficult problems to solve. He graciously shared his expertise and years of experience in designing elegant system and building computer chips.

I would also like to thank Professor Scott Hauck, who was my advisor during my first year of master's program. He provided me with guidance when I first came to UW. He has been very patient with me, and allowed me to make silly mistakes and gave me chance to figure things out.

I would also like to thank everyone in Bespoke Silicon Group and everyone who helped me with research for this thesis: Shaolin Xie, Chun Zhao, Dustin Richmond, Scott Davidson, Paul Gao, Dan Petrisko, Mark Wyse, Farzam Gilani, Leonard Xiang, Bandhav Veluri, Borna Ehsani, and many others I forgot to include.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.

[2] M. B. Taylor, "BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design," *Design Automation Conference*, June 2018.

[3] M. B. Taylor, *Tiled microprocessors*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2007.

[4] 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.

[5] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[6] `https://riscv.org/specifications/`. Accessed: 2019-06-10.

[7] ARM, *AMBA AXI and AXE Protocol Specification*, 2013. IHI 0022E.

[8] `https://riscv.org/specifications/privileged-isa/`. Accessed: 2019-06-10.

[9] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool, 2011.

[10] "pseudo-LRU." `https://people.cs.clemson.edu/~mark/464/p_lru.txt`. Accessed: 2019-06-05.

[11] M. Naylor, S. W. Moore, and A. Mujumdar, "A Consistency Checker for Memory Subsystem Traces," *FMCAD*, Oct 2016.

[12] Micron Technology, *Automotive LPDDR SDRAM - MT46H128M16LF*, 2013. Rev. H 01/17 EN.

[13] "AWS EC2 FPGA Hardware and Software Development Kit." `https://github.com/aws/aws-fpga`. Accessed: 2019-06-05.