TinyParrot: An Integration-Optimized Linux-Capable Host Multicore

Sripathi Muralitharan

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2021

Committee:

Michael Taylor

Scott Hauck

Program Authorized to Offer Degree:

Electrical and Computer Engineering

University of Washington

Abstract

TinyParrot: An Integration-Optimized Linux-Capable Host Multicore

Sripathi Muralitharan

Chair of the Supervisory Committee:

Michael Taylor

Department of Computer Science and Engineering

Recent developments in architecture research warrant the need for efficient host cores to interact and manage multiple accelerators in a system-on-chip design. Existing designs suffer from low configurability, rely on non-standard or proprietary tooling, or require sophisticated mechanisms to interact with accelerators. BlackParrot [1] is a 64-bit Linux-capable, open-source multicore processor that aims to break these barriers to become an accelerator host processor used in state-of-the-art SoCs. One of the critical requirements to achieve this goal is modifying the core quickly to suit the application's needs. The work presented in this thesis discusses ways to optimize BlackParrot for integration with diverse architectures. While BlackParrot offers sufficient configurability to host standalone accelerators in its memory system, it did not have the tools to integrate with larger system designs. The creation of the BlackParrot unicore and the standard cache interface between the L1 cache and its controller, presented in this thesis, allows BlackParrot to integrate with designs with minimal modification. Its parameterizable cache, along with a multi-cycle fill strategy, has enabled the creation of a tiny core that can find uses in systems with physical limitations. The utility of these modifications, among others, was validated by integrating BlackParrot with HammerBlade and OpenPiton.

Contents

Page

List of Tables

List of Figures

1 Introduction

The end of Dennard Scaling [2] and the 'Utilization Wall' have forced computer architects to explore domain-specific architectures [3]. Apple's M1 processor features multiple accelerators for graphics, security and machine learning on the same chip. Google's TPU [4] accelerates machine learning workloads and Microsoft uses FPGAs (Project Catapult) to accelerate cloud-based services and AI. The academic and research communities have many examples showing the growth of hardware specialization [5–11].

The growing popularity of accelerator SoCs requires host processor cores optimized for interacting with and managing multiple accelerators. Current commercial cores provide interfaces that are not configurable for the needs of the user and require elaborate mechanisms to communicate with an accelerator which adds overhead to the system's performance. Additionally, they are expensive and inaccessible for research and to the open-source communities. BlackParrot [1] is a free and open-source, Linux capable RISC-V multicore that aims to break these barriers and become the default host core in state-of-the-art accelerator SoCs.

While [1] shows that BlackParrot provides the best-in-class performance, area and energy efficiency in comparison to popular open-source and commercial RISC-V cores [12–14], its configurability was limited. BlackParrot's multicore system offered the flexibility to add and remove customized tiles for memory, accelerator and I/O to the network as long as they adhered to the network protocol. This flexibility was, however, only at the system level. The aim was to enable the core itself to be flexible. There are multiple academic, industrial and open-source research projects that require a core to connect to a pre-existing network. Other projects have restrictions on on-chip (silicon/FPGA) resources for a host core. In both cases, the lack of core-level configurability resulted in resorting to solutions that motivated the development of an accelerator host core in the first place (i.e. using commercial cores with sophisticated mechanisms to interact with the accelerator).

This thesis summarises my contributions to the research group and addresses BlackParrot's configurability at the core level by motivating the need for an efficient unicore, creating a standard interface to make the system more plug-n-play friendly and adding features to the cache including size parameterization, modifications to the iterative filling strategy, and sub-bank masking to make the memory system more configurable. It provides a detailed overview of three standard controllers that BlackParrot supports of which I personally contributed to the development of the controller for the unicore and the transducer for the OpenPiton integration. I conclude by presenting two real-life examples of using these developments -

- A size-configurable core, codenamed TinyParrot verified to synthesize using an open-source 45nm process node and a low-cost FPGA. BlackParrot's size configurability creates an ecosystem for a broader base of users to include on-chip accelerator host cores in their designs.
- An accelerator host core codenamed HammerParrot for a tiled manycore architecture specialized for machine learning and graph analytics where BlackParrot's tininess and flexibility improves overall system performance.

2 Background

2.1 BlackParrot

BlackParrot [1, 15] is a 64-bit RISC-V multicore processor designed to function as an accelerator host in state-of-the-art SoCs. It is an 8-stage pipelined, in-order, single-issue processor that implements the I (Integer), M (Multiply), A (Atomic), F (Single-precision Floating Point), and D (Double-precision Floating Point) extensions of the RISC-V ISA [16]. BlackParrot is a Linux-capable processor supporting three privilege modes (Machine, Supervisor, User) and RISC-V's SV39 virtual memory system [16]. BlackParrot consists of 3 major components - the Front End, the Back End, and the Memory End to enforce modularity in the design. The Front End consists of the PC generation unit, instruction fetch, and branch prediction logic. The Back End has multiple execution lanes, each handling instructions belonging to a different class like integer, floating-point, branching, load/store, Control Status Register (CSR) access, and exception handling. BlackParrot's Front End injects instructions into the Back End through the Issue Queue, and the Back End communicates information such as branch decisions and PC redirections back to the Front End via the Command Queue. On events like a cache miss or an exception, the issue queue rolls back to the last committed instruction and replays the instructions once the miss/exception has been resolved. Figure 1 illustrates the core microarchitecture. The Memory End consists of the directory-based, race-free, programmable cache coherence engine or a lightweight FSM-based cache controller and an L2 slice. The BlackParrot multicore consists of multiple core tiles, each consisting of the core, L1 caches, and controllers, and a cache coherence engine backed by an L2 slice, all arranged in a scalable 2-D mesh network topology, codenamed Bedrock, that supports diverse tile types and interfaces to aid in easy integration with accelerators. BlackParrot offers the best-in-class performance, area, and energy efficiency [1] on silicon and extensively uses BaseJumpSTL and modern software engineering practices to ease adoption and achieve its goal of becoming the default host in accelerator SoCs.

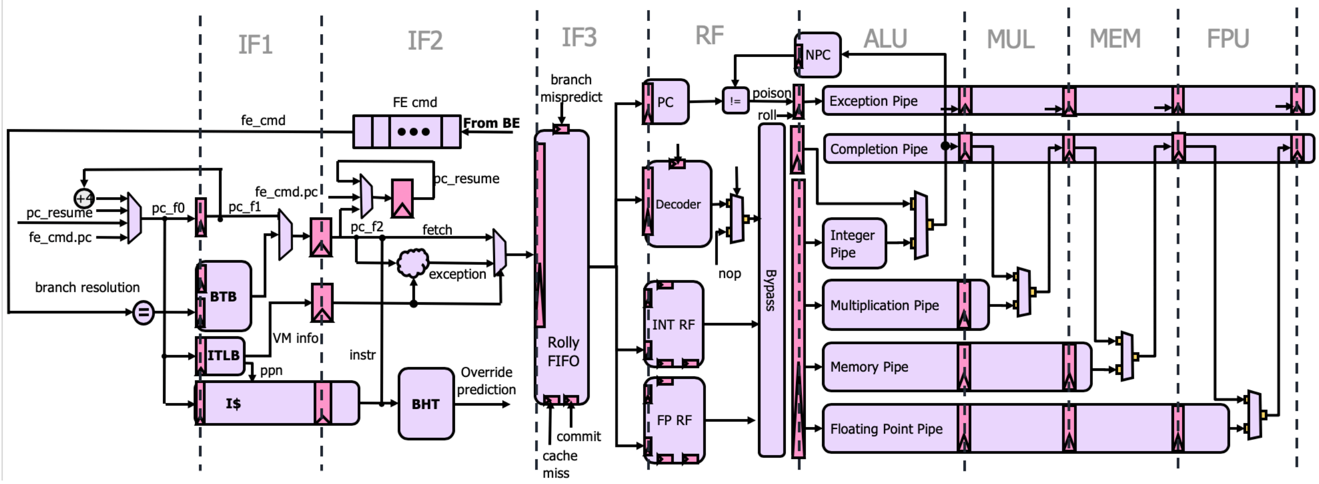


Figure 1: BlackParrot core microarchitecture [1]

2.2 BaseJump STL

Modern software programming languages like C/C++ provide repositories of commonly used, thoroughly validated, and highly optimized data structures and algorithms as a Standard Template Library (STL) to aid in the agile development of software. BaseJumpSTL [17, 18] aims to achieve the same for hardware development by providing highly composable building blocks that allow the designer to devote more time to the development of their microarchitecture. The library provides many synthesizable modules ranging from simple logic gates, flip flops, and counters to more complicated structures like caches and memory controllers while also hosting many non-synthesizable building blocks to aid in design verification. All building blocks are written in SystemVerilog following a consistent set of coding guidelines that enhances readability while still employing best practices to achieve Pareto optimal power, performance, and area. BaseJumpSTL ensures the user does not have to worry about timing diagrams by providing a standardized, latency-insensitive handshaking interface and allows for quick design space exploration through extensive parameterization. The hardware modules are process node agnostic, and the library provides a specialization layer to fine-tune implementations to a specific technology. These features make the library amenable for quick adoption, as evidenced by the multiple research projects that have taped out chips using this library as the foundation.

3 Motivation for the BlackParrot Unicore

As stated previously, BlackParrot is an efficient multicore with system-level configurability that supports the integration of coherent and non-coherent, standalone accelerators. Figure 2 shows a single-core tile in the BlackParrot multicore system. Bedrock, a standard interface for the network and memory system, supports this integration.

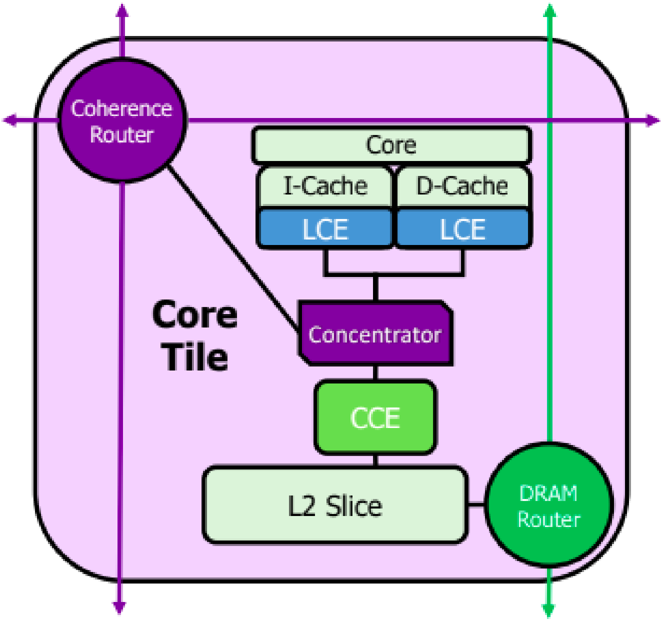


Figure 2: A core tile in the BlackParrot multicore system. LCEs are the Local Cache Engines and a CCE is the directory-based Cache Coherence Engine [1].

However, not only system-level configurability is enough, but also core-level configurability is necessary to expand the scope of BlackParrot as an accelerator host and cater to more full-featured systems that require host core(s) to connect to a network. The first roadblock in this path was the lack of an efficient single-core implementation of BlackParrot.

A study that measured the area of BlackParrot's core tile (The snapshot of BlackParrot used for this experiment is commit: c8c67bedb5) exposed the area inefficiency of the core tile for a single core BlackParrot. Figure 3 shows the results of this experiment. The core occupies only 64% of the total area, while the directory controller and the network components consume the rest. These are components that are not required in the single-core implementation. They add to the overall power consumed by the core tile, the cycles incurred during memory operations, and make the system more complex than necessary.

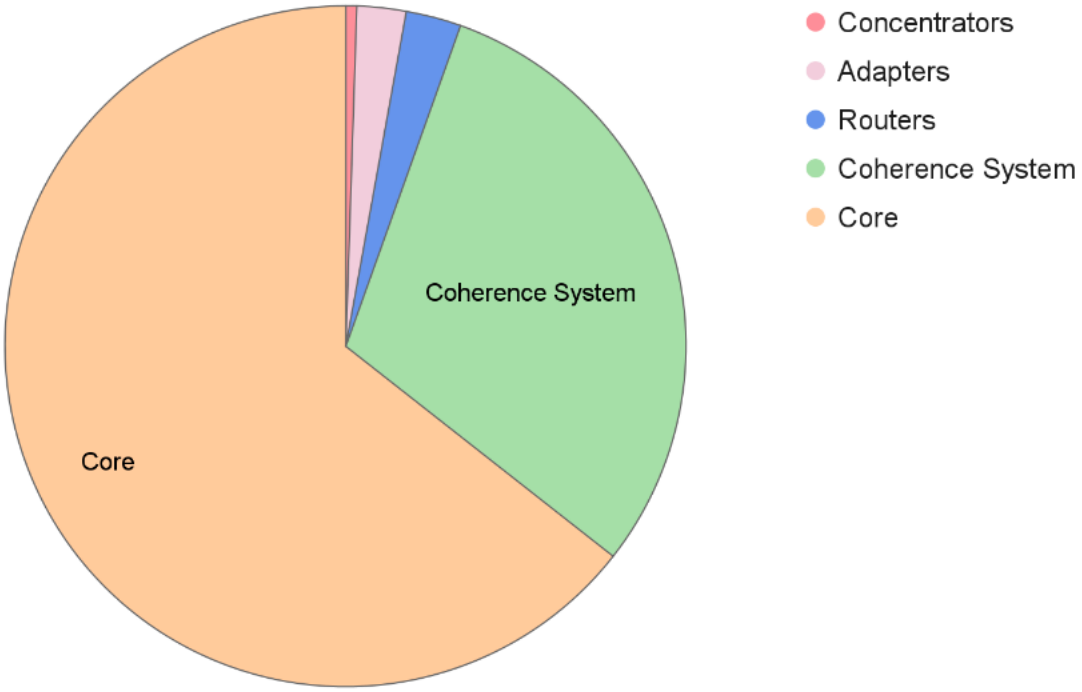


Figure 3: Area breakdown of a single BlackParrot core tile (no L2 slice). The area numbers were generated using the Global Foundaries 14nm process node. Note: BlackParrot is a **continuously evolving** processor core and the version used here was accessed on Jan 31, 2020.

A unicore system minimally needs the core pipeline, the L1 caches, and their controllers and expose a suitable interface to the rest of the memory system. Additionally, an integration-optimized unicore needs to provide the necessary configurability to communicate with its connected system efficiently. A standard interface between the L1 cache and its controller can achieve the required configurability while satisfying the minimum requirements of the unicore. The following sections describe how to do this in a maximally efficient and modular way.

4 BlackParrot Cache Optimizations for Integration

4.1 Design Overview

BlackParrot contains an Instruction Cache (I$) and a Data Cache (D$). The rest of this section gives an overview of the current version of the D$ (commit: 414747b058) since the I$ inherits the base design from the D$.

BlackParrot's D$ is a Virtually Indexed, Physically Tagged (VIPT), write-back (default) and write-allocate cache that can be configured in different ways as shown in Table 1[1][2]

Cacheline Size (bytes)	Associativity	Sets	Cache Size (kB)
64	8-way	64	32
	4-way	64	16
	2-way	64	8
	Direct Mapped	64	4
32	4-way	128	16
	2-way	128	8
	Direct Mapped[1][2]	128	4
16	2-way	256	8
	Direct Mapped[1][2]	256	4
8	Direct Mapped[1][2]	512	4

Table 1: Different cache configurations

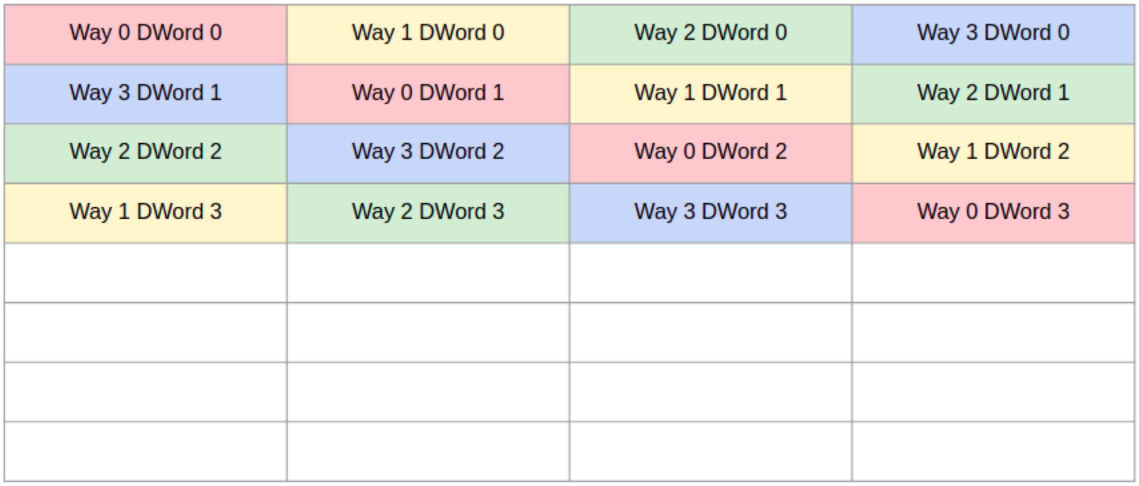
The cache consists of three primary memories - data, tag, and stat and each memory has one read/write port. The data memory holds the cached data. It has one bank per way in the cache, and the cache lines are interleaved among the banks to speed up cache line access. Each doubleword (8 bytes) from a cache line is written to a separate bank in the data memory. The bank ID of this write is governed by Equation 1. The interleaving of a single cache line across multiple banks helps to read a single index in the data memory, retrieving the same 64-bit doubleword of every cache line across all the ways while still giving the benefit of writing an entire cache line at once. Figure 4 illustrates this interleaving for a 4-way cache. Before evicting a cache line, it must be

[1] These configurations are valid but untested

[2] Support for more configurable Direct-Mapped caches is a work in progress (link)

de-interleaved, and this is achieved by rotating the entire cache line to the right by the appropriate number of double words.

$$\text{Bank ID} = \text{Way ID} + \text{DWord Offset} \tag{1}$$



Way 0 DWord 0	Way 1 DWord 0	Way 2 DWord 0	Way 3 DWord 0
Way 3 DWord 1	Way 0 DWord 1	Way 1 DWord 1	Way 2 DWord 1
Way 2 DWord 2	Way 3 DWord 2	Way 0 DWord 2	Way 1 DWord 2
Way 1 DWord 3	Way 2 DWord 3	Way 3 DWord 3	Way 0 DWord 3

Figure 4: Data Memory Organization

The tag memory holds the tag and the coherence state for a cache line. The D$ uses a practical implementation of the Least Recently Used (LRU) replacement policy called tree pseudo-LRU [19] that encodes the LRU way of the cache. The pseudo-LRU tree is traversed in a breadth-first manner, and each bit in this format encodes the direction of traversal (to compute the LRU way ID) for that node. A 0 indicates to go left in the tree, and a 1 indicates to go right in the tree. Figure 5 and Table 2 illustrates the LRU format and how it maps to a way ID. When a way is accessed, the LRU information is updated by traversing the tree for the referred way and creating a mask to update the current LRU way. Effectively this process flips the bits while traversing the tree and guarantees that the referred way is not the LRU way. Figure 6 shows how the LRU way is updated given the referred way ID. The stat memory holds this LRU tree state and the dirtiness of the cache line. Any read to the stat memory passes through the LRU tree decoder to obtain the way ID, and any read/write to a given way encodes the referred way ID and writes it to the stat memory. Therefore, encoding and decoding the LRU way prevents the need for a read-modify-write operation to the stat memory.

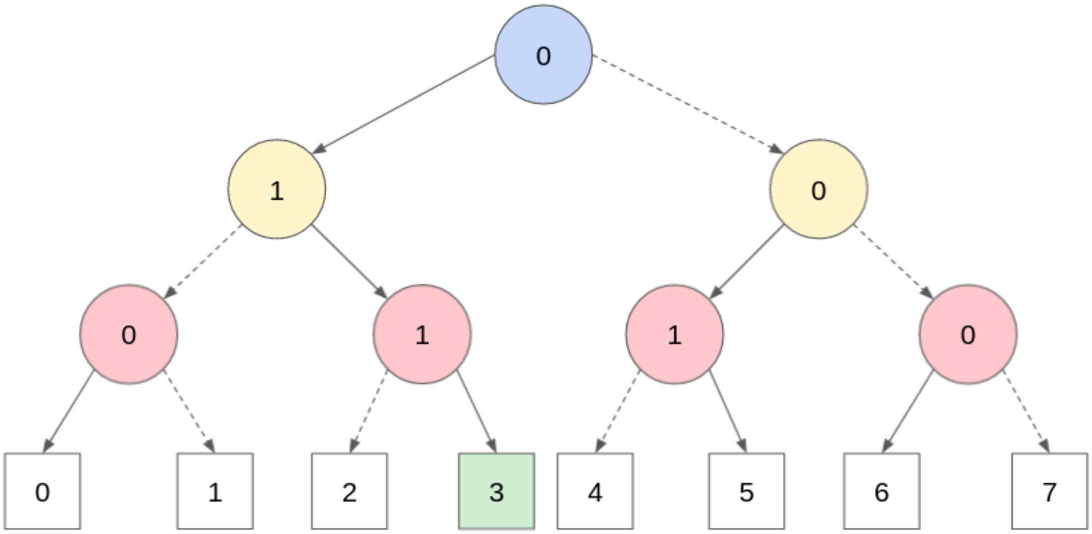


Figure 5: The LRU encoding format for an 8-way set-associative cache. The figure shows that the current LRU way is way 3 (marked in green)

LRU Encoded Format	Way ID
xxx_0x00	Way 0
xxx_1x00	Way 1
xx0_xx10	Way 2
xx1_xx10	Way 3
x0x_x0x1	Way 4
x1x_x0x1	Way 5
0xx_x1x1	Way 6
1xx_x1x1	Way 7

Table 2: LRU encoded format (MSB-first) to Way ID conversion for an 8-way associative cache. ‘x’ denotes a don’t care.

The D$ has a pipelined datapath consisting of 3 pipeline stages - Tag Lookup (TL), Tag Verify (TV), and Data Mux (DM).

The cache decodes the incoming cache packet, and the data and decoded packet are registered at the negative edge of the clock giving only half a cycle for address calculation since the packet arrives on a positive edge of the clock. Figure 7 shows the partitioning of a cache address.

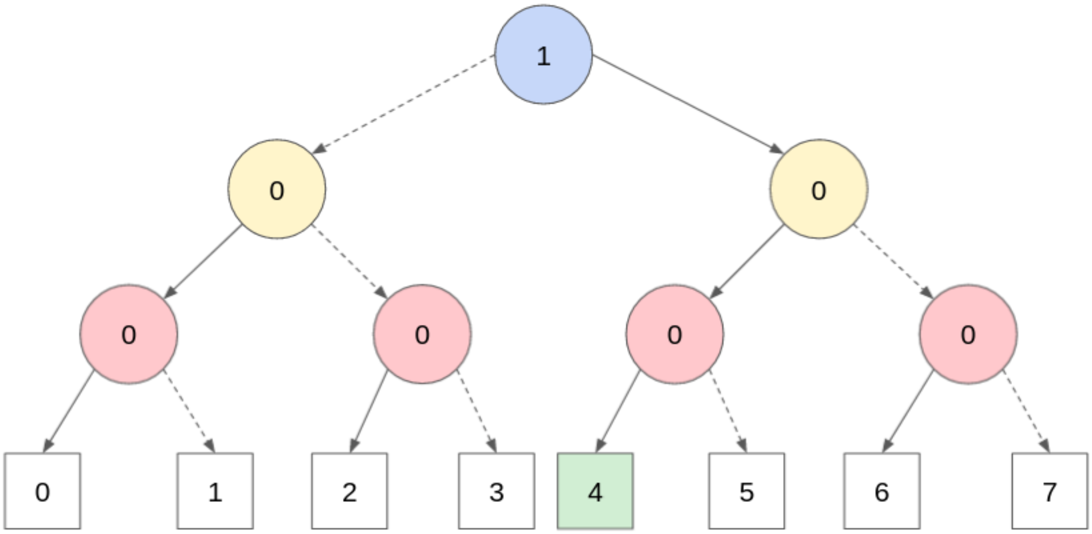


Figure 6: Updating the LRU tree from Figure 5 for an 8-way set-associative cache when the referred way has ID 3. The figure shows that the new LRU way is way 4 (marked in green)

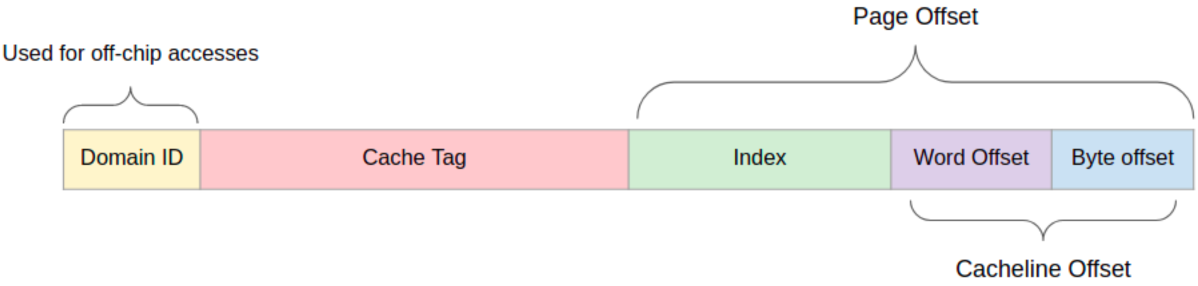


Figure 7: Address breakdown for the cache

In the TL stage, the cache reads the tag and data memories simultaneously and detects a hit or a miss. This information, along with the data read from the data memory, is stored to the pipeline register on the negative clock edge, giving an entire cycle for this stage. The physical tag for the cache translated by the Translation Lookaside Buffer (TLB) and the Physical Memory Attributes (PMA) also arrives in this stage and is latched. The data is muxed down to the correct double word or word in the TV stage and injected into the core pipeline on a positive clock edge giving the cache half a cycle to mux the data making its hit time two cycles for double word and word ops. Half cycles for address calculation and data multiplexing are sufficient since they involve either bit slicing or a single mux operation, while an entire cycle is required for memory reads. The LRU information in the stat memory is updated during the TV stage. If there was no hit in the cache, then the miss information is sent to the controller in this stage. The double word selected in the TV stage is muxed down for sub-byte operations or recoded for floating-point operations in the DM stage to reduce the critical path.

The cache also contains an in-built ALU used to perform atomic operations at the L1 level. The supported operations include AND, OR, XOR, ADD, MIN, and MAX. However, the cache can offload these functions to the next level of the memory system through uncached operations.

The cache also has a write buffer that stores the incoming write data until the data memory becomes free from the incoming loads. The incoming loads also snoop the write buffer for valid data while determining a hit, preventing data hazard stalls.

BlackParrot's caches are currently blocking and can handle only one miss at a time. The cache sends a miss request to the controller and waits until a response is received.

The I$ design is similar to the D$ design. It only supports load operations and does not use a combination of positive and negative clock edges, which makes its hit time three cycles. The positive and negative clock edge trick does not benefit the I$ because of a feedback path from the output of the I$ to its input due to the branch prediction logic.

The cache datapath is partitioned into a fast path and a slow path. The fast path describes the cache datapath during a hit as described above. The slow path describes the logic required to handle a miss and fill the cache with data from the next memory level. The L1 cache-controller interface forms the backbone to create an area-efficient single-core and improve the core's overall integration potential. The following section gives a detailed overview of the capabilities of this interface.

4.2 Cache Interface

The following requirements are imposed to define an interface that caters to the current and future goals of the processor,

1. It must service all combinations of caches and controllers. requiring that the cache/controller only adhere to the interface.
2. The interface must support an exhaustive set of operations to support various use cases.
3. On the cache side, it must be agnostic to the type of cache (e.g., blocking/non-blocking, coherent/non-coherent) and must also be easily configurable to support different cache configurations and organizations.
4. On the controller side, the interface should offer a low overhead so that transducing does not become a bottleneck in the system.
5. The interface must implement latency insensitive handshaking to hide cache and controller timing information.

4.2.1 Cache to Controller Interface

The interface from the cache to the controller handles the requests from the cache. It should contain all information required by the controller to handle the request. The request interface currently (commit: 414747b058) has the fields given in Table 3. Requests can be of different types, and Table 4 highlights the different types supported by the current version of the interface.

SystemVerilog structs and enums help package the request into a packet(s) that the cache can transmit. The interface uses a valid-then-ready (a.k.a valid-yumi) [17] handshake to transmit packets to the controller. The cache asserts the valid signal when it wants to send a request and keeps the valid signal high until the controller sends back an acknowledgment by asserting the yumi signal.

Field	Description
msg_type	Type of Request
addr	Physical address
size	Request size (1 byte - 64 bytes)
data	Data (for uncached, write-through and atomic ops)
subop	Sub-opcode
hit	Hit information

Table 3: Cache request packet

Operation	Sub-op	Description
miss_load	-	Load miss
miss_store	-	Store miss
wt_store	-	Writethrough store
uc_load	-	Uncached load
uc_store	-	Uncached store
uc_amo	-	Atomic operation
	amolr	Load reserved
	amosc	Store conditional
	amoswap	Atomic swap
	amoadd	Atomic add
	amoxor	Atomic XOR
	amoand	Atomic AND
	amoor	Atomic OR
	amomin	Atomic minimum
	amomax	Atomic maximum
	amominu	Atomic unsigned minimum
	amomaxu	Atomic unsigned maximum
cache_flush	-	Cache flush (for fencing operations)
cache_clear	-	Clears the cache tag and stat memories

Table 4: Operations supported by the interface

This handshake is preferred over a ready-then-valid handshake [17] because the controller implementation can choose to accept specific packets at different points in its operation. For example, the controller can handle uncached and write-through stores while waiting to respond to a previous independent load.

In addition to transmitting a request packet, the controller might also require metadata to service the request. This metadata may not be available in the same cycle for a high-performance cache design. If the controller requires metadata to service the miss, the metadata can arrive at any cycle later than or equal to the original cache request. The metadata packet has a valid-only handshake which means that the controller must register the metadata on the cycle that it is sent. The cache must eventually provide the metadata required for the controller to handle the request, and the latency between the original request and the metadata packet must be a fixed, known constant for all request types and under all backpressure conditions. Currently, the metadata packet contains the fields shown in Table 5.

Field	Description
hit_or_repl_way	Hit (for invalidations) or replacement way (for evictions)
dirty	Dirty bit for the corresponding way

Table 5: Cache request metadata packet

4.2.2 Controller to Cache Interface

The interface from the controller to the cache handles the responses to the cache's requests and coherence operations and invalidations to the cache. It acknowledges receiving a cache request and sends back any data to service the requested operation (e.g., a load miss request is serviced by sending back a cache line) that read/write from the cache memories. Additionally, commands from the memory system to update the coherence state or invalidate a cache line can also be sent via this interface. SystemVerilog structs and enums are exploited to provide a user-friendly way to transmit this information from the controller to the cache. The interface requires a valid-then-ready (a.k.a valid-yumi) handshake to send packets to the cache. Cache designs would prioritize servicing the incoming requests from the core over the response packets, both of which access the cache memory. Therefore, to address any structural hazards, the cache interface should handle this backpressure, and a valid-yumi handshake is best suited for this job. A cache can send three response packets, one for each kind of memory in the cache. These ports are independent to allow for flexibility in the cache fill strategy and the controller implementation. A description of the fields of the three interfaces are given in Tables 6, 7 and 8.

Field	Description
opcode	Read/Write cached/uncached data memory
index	Cache index
way_id	Cache way
data	Data

Table 6: Data memory packet

Field	Description
opcode	Read/Write/Clear tags and/or coherence state
index	Cache index
way_id	Cache way
tag	Physical tag
state	Coherence state

Table 7: Tag memory packet

Field	Description
opcode	Read/Clear the LRU and/or dirty bits
index	Cache index
way_id	Cache way

Table 8: Stat memory packet

Apart from responses to requests, the controller also sends back signals to indicate the controller's status. Since a single cache request could trigger multiple fills based on the cache fill strategy, the controller is required to assert a *cache_req_complete* signal for one cycle when the cache request is complete. The interface also supports credit-based flow control by communicating the amount of credits left in the controller for requests via *cache_req_credits_full* and *cache_req_credits_empty*. Empty signifies that all downstream operations are complete and the controller is ready to receive more requests, while full indicates that the controller cannot accept any more requests.

BlackParrot also implements a rudimentary version of the critical word first technique in its cache, filling the request address first. To indicate that the critical words and tag are being sent, the *cache_req_critical_data* and *cache_req_critical_tag* signals are asserted.

4.2.3 Miss Tracking and Cache Locking

The interface does not support passing miss tracking information between the cache and its controller, and therefore the cache is responsible for tracking the status of all outstanding misses. The cache should rely on the *cache_req_complete* signal and other metadata (such as miss ID) to resolve the corresponding miss. The controller can also perform other functions while handling a miss, such as service other requests. The interface assumes that all the intelligence required to perform such optimizations lies within the controller.

BlackParrot additionally implements a cache locking mechanism to allow forward progress on LR/SC operations. Load Reserved (LR) and Store Conditional (SC) operations are primitives used to implement synchronization structures like locks. When a core reserves a cache line, the core should be given some time to progress in the critical section of the program and prevent incoming invalidations from breaking the core's reservation and creating a situation where the lines keep jumping between cores' caches. The cache is locked on a successful load reservation operation to prevent cache line ping-ponging. A counter starts counting up to some pre-determined maximum value, and the cache is deemed 'locked' and does not process any incoming response packets from the controller during counting. If the corresponding store conditional is successful, or the counter reaches its maximum value, then the cache is 'unlocked'. The maximum value of this counter can be configured to prefer local or remote operations over the other.

4.3 Iterative Filling/Eviction

BlackParrot's default configuration uses an 8-way, 32KB cache with a 64B cache line. Cache line filling and eviction are done at the cache line granularity, meaning that 64B chunks of data are transferred at any point in time. Filling/evicting lines this way was sub-optimal because -

1. It is not practical to expect full cache lines to be available from the lower levels of the memory system because of the network's flit size and wormhole streaming[3].

2. It occupies silicon die area and requires more energy. The downstream logic in the memory system uses multiple buffers and FIFO queues to handle different data processing rates between modules. Moving 64B chunks of data would therefore require large buffers and FIFO memories.

One of the immediate solutions would be to pick another cache configuration that BlackParrot supports. For example, using a 2-way, 8KB cache can handle data chunks required by systems such as OpenPiton (Sec. 5.3) while still moving data around in cache line width. However, this still does not solve the issue with the physical network limitation and decreases the cache's hit rate since it is less associative and smaller in size.

The solution was to implement an iterative filling strategy. By supporting partial fills in the L1 caches, power consumption and area utilization can be minimized. However, miss latency increases since each fill/eviction incur more cycles. The increase in miss latency can be amortized by implementing critical word first or early restart mechanisms in the cache. BlackParrot supports a rudimentary version of the critical word first mechanism where the controller fills the cache with the requested address first, but the core remains stalled until the miss request is complete.

An additional field, *fill_index*, was introduced into the data memory response packet in the cache interface to implement iterative line filling. This index tells which position in the cache line the fill data is supposed to replace. The fill widths can be a minimum of 8 bytes and a maximum of 64 bytes and can be no smaller than the bank width. The controller can even fill multiple data memory banks at the same time. When a cache request packet is received, the controller first fetches the data for the address requested (i.e., the critical word) and then proceeds to fill all the other sub-blocks within the cache line iteratively. Figure 8 shows the sequence of operations required to fill a single cache line.

[3] NoC symbiosis breaks this barrier with some modifications to the system configuration [20]

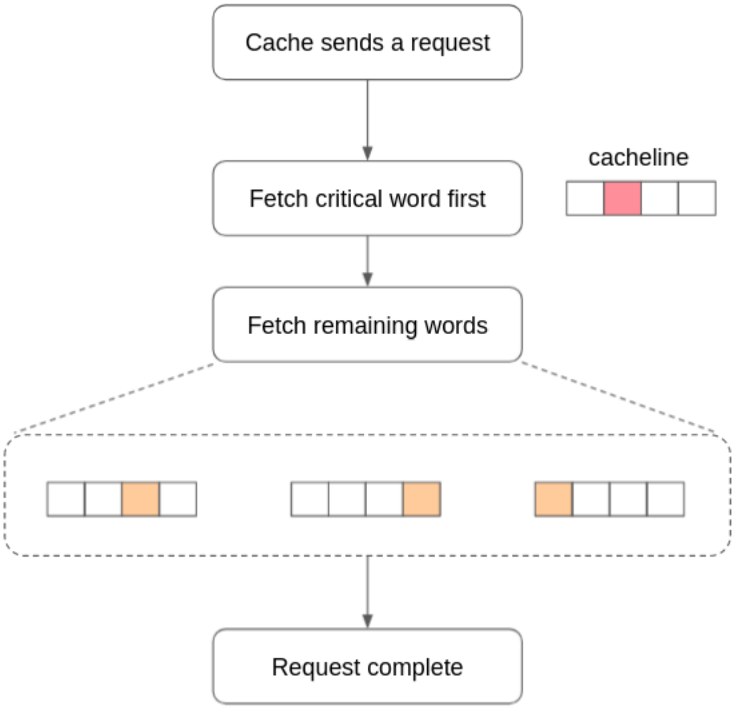


Figure 8: Iterative filling of a cacheline. Each colored box in the cacheline indicates the portion of the line that is being filled. The red box indicates the critical word.

4.4 Sub-bank Filling/Eviction

The iterative filling method described above enforced that the fill width can be a minimum of the bank width. BlackParrot's minimum bank width is 8 bytes since the maximum data width that the core can request is 8 bytes. However, most systems expose interfaces that follow standard protocols such as AXI, AXI-lite, Wishbone, or other pre-existing IP, some of which require 32-bit data buses to transmit and receive data. Additionally, cache designs with no banking shuttle cache lines around which as stated earlier was sub-optimal. To accommodate for the IP requirements/banking strategy while still reaping benefits of iterative filling, sub-banking is introduced.

The iterative filling strategy is extended with minor tweaks to implement sub-bank filling and eviction. The semantics of the *fill_index* field in the data memory packet was modified to indicate a sub-bank index. The cache can then use this index value and the pre-determined fill size to calculate the correct masks to write into the right data memory bank.

BlackParrot's caches currently support the fill widths and bank widths shown in Table 9.

	Widths supported
Bank	64, 128, 256, 512
Fill	64, 128, 256, 512

Table 9: Bank and Fill Widths supported

Supporting a fill width of 32 bits in BlackParrot required some more careful thought. Firstly, BlackParrot's uncached load and store operations need a maximum of 64 bits. Secondly, many of the peripherals supported by BlackParrot communicate using 64-bit data. Adding Serial-In-Parallel Out (SIPO) modules on the request path and Parallel-In-Serial-Out (PISO) modules on the response path can resolve both the above issues when the fill width is 32-bits. Because almost all of BlackParrot and its peripherals communicate in 64-bits, this would add a reasonably significant overhead to the entire system to support a single bus/accelerator.

Therefore to reconcile the two, BlackParrot will support only a minimum of 64-bit fills, and in order to support 32-bit protocols like AXI-lite, the BP $\leftrightarrow$ IP transducer should have PISOs on the request path and SIPOs on the response path. BlackParrot has three Link Protocols - BP Lite (Similar to AXI-lite), BP Stream (Similar to AXI-Stream), and BP Burst (Similar to AXI-4) to address this. A brief overview of these protocols can be found in [21] but a detailed overview is out of scope for this thesis.

5 Cache Controllers

BlackParrot's new standardized cache interface can support different cache controllers for different use cases provided they adhere to the interface specification. Currently, there are three controllers for three different use cases.

The Unified Cache Engine (UCE) is used with BlackParrot's unicore configuration and is the default controller used in BlackParrot. The Local Cache Engine (LCE) is used with BlackParrot's multicore configuration to communicate with the Cache Coherence Engine (CCE), the directory controller. The P-Mesh Cache Engine (PCE) used with ParrotPiton (BlackParrot with OpenPiton) transduces BlackParrot's requests (responses) into the Transaction Response Interface (TRI) requests (responses). A deep dive into each of these controllers is presented in the following sections.

5.1 Unified Cache Engine (UCE)

The Unified Cache Engine or UCE is the default L1 cache controller used with a single core BlackParrot. It is a lightweight FSM that:

- Handles load and store misses in the L1 cache via multicycle fills and evictions.
- Handles uncached loads and stores.
- Handles invalidations to cache lines requested in uncached mode.
- Forwards atomic memory operations to the next level of the memory system.
- Supports both write-back and write-through protocols.
- Supports credit-based flow control to handle fences.

Figure 9 shows the controller with all its states currently. Each of the controller's states is explained briefly below. The states' names follow the Bespoke Silicon Group, SystemVerilog coding guidelines [22] -

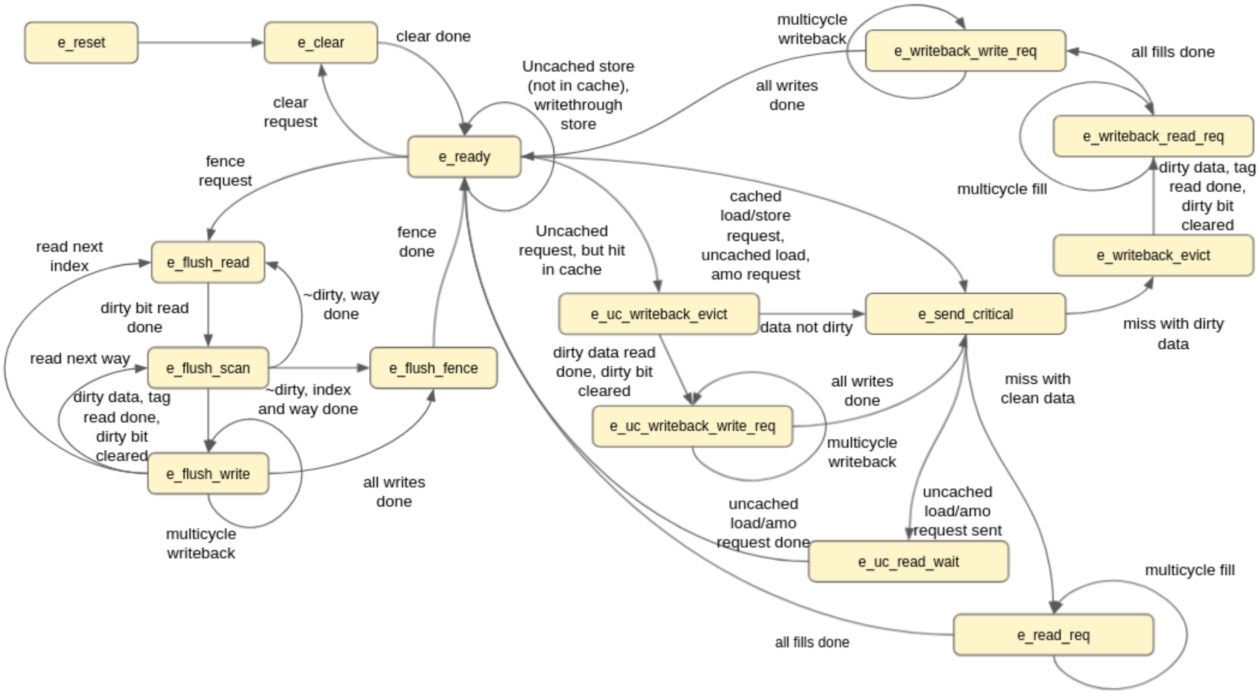


Figure 9: Unified Cache Engine (UCE)

- Reset (*e_reset*): Initial state of the controller
- Clear (*e_clear*): Clears the cache tag and stat memories
- Ready (*e_ready*): The controller waits in this state to receive requests.
- Flush Read (*e_flush_read*): The controller moves to this state on a fence request. In this state, the controller reads the stat memory to check the dirty bits for the cache line.
- Flush Scan (*e_flush_scan*): The controller reads the data and tag memories if the cache line is dirty in the Flush Read state. It also clears the dirty bit. If the line was not dirty, nothing needs to be done, and the controller moves to the next cache line.
- Flush Write (*e_flush_write*): The dirty lines read from the cache are written back to memory iteratively.
- Flush Fence (*e_flush_fence*): After reading all the lines and ways and writing the dirty lines to memory, this state signals the cache that the controller operation is complete and the controller moves back to the ready state.

- Uncached Writeback Evict (*e_uc_writeback_evict*): If there was an uncached request to the controller and the line already existed in the cache, then it must be evicted from the cache before handling the uncached request. In this state, the controller reads the stat memory to check if the line is dirty.
- Uncached Write Request (*e_uc_writeback_write_req*): In this state, the controller iteratively writes the cache line back to memory.
- Send Critical (*e_send_critical*): The controller moves to this state for any other request - cached load/store, uncached load, and atomic requests (uses the same datapath as uncached requests to the cache). It fetches the critical word for cached requests or sends the uncached load/atomic request to the memory.
- Read Request (*e_read_req*): The controller moves to this state if the line to be evicted is clean; the controller iteratively fills the cache line.
- Writeback Evict (*e_writeback_evict*): If the line was dirty, the controller reads the data and tag memory for the dirty data and tag.
- Writeback Read Request (*e_uc_writeback_read_req*): The controller iteratively reads the memory to fill the cache line.
- Writeback Write Request (*e_uc_writeback_write_req*): The controller iteratively writes the evicted line to memory.

5.2 Local Cache Engine (LCE)

The Local Cache Engine or the LCE is the default L1 cache controller in the BlackParrot multicore. It can also function as the controller for a coherent accelerator cache. The LCE is responsible for initiating the coherence requests and responding to coherence commands from the Cache Coherence Engine (CCE) through the Bedrock cache coherence system. The CCE tracks the coherence state of all the blocks maintained by all the LCEs in the system. Each CCE is responsible for a portion of the total physical address space. Figure 10 gives a simplistic view of the BlackParrot multicore system.

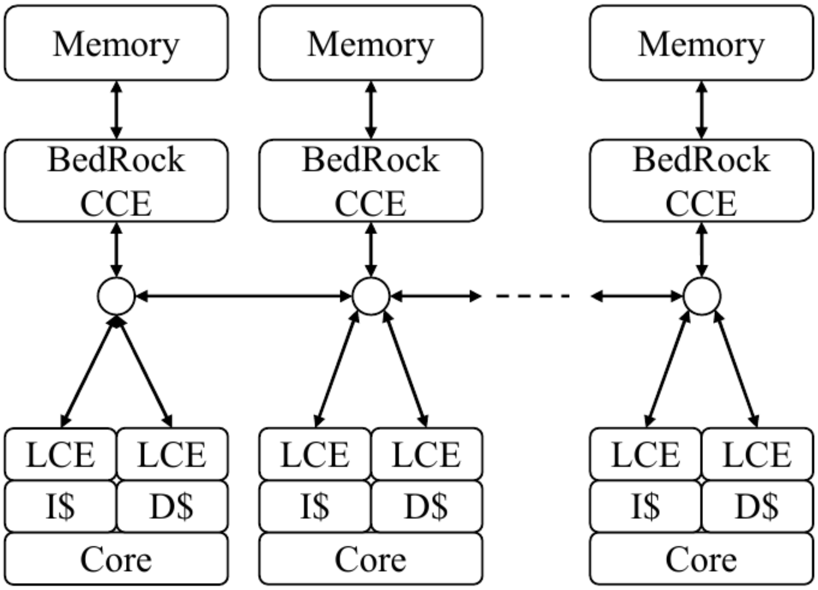


Figure 10: Bedrock Network [15]

Bedrock consists of three networks that allow point-to-point communication between tiles on the Bedrock network. The three networks are namely, Request, Command, and Response. The LCE initiates a coherence request using the Request network and services any command from the CCE through the Command network. Any responses to the CCE commands go via the Response network. The three networks have the following priority

$$Response > Command > Request$$

to prevent deadlock (formally proven to be correct via CMurphi [23], a model checking algorithm built on top of Murphi from Stanford). A message from a low priority network can trigger a message on a high priority network but not the other way around. The Bedrock LCE-CCE interface fully specifies the packet and command types supported by the interface.

The LCE does not currently support the iterative filling mechanism. However, downstream logic serializes the data into packets before sending them over the network.

Multicycle fill support in the LCE requires additional modifications to the CCE, a work in progress.

Figures 11 and 12 show the FSMs that handle requests and commands respectively. The command FSM also triggers the required responses or other commands (e.g., cache to cache transfers).

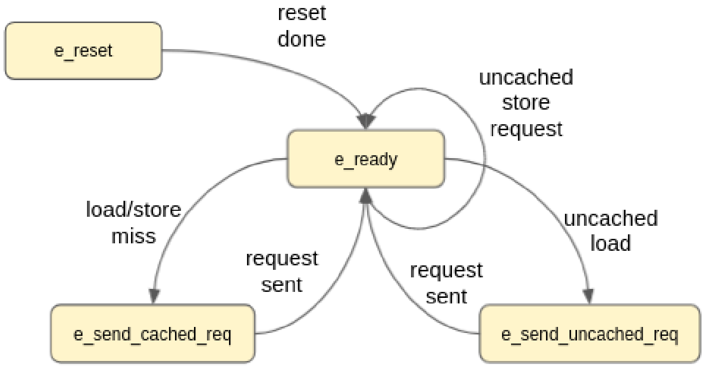


Figure 11: LCE Request Controller

Each of the states of the request and command FSMs is explained briefly below. For the request FSM -

- Reset (*e_reset*): Initial state of the controller
- Ready (*e_ready*): The controller waits in this state to receive requests. The requests can be cached loads/stores and uncached loads. Uncached stores are immediately sent out and acknowledged.
- Send Cached Request (*e_send_cached_req*): The controller sends a request on the LCE request network for a cache line. While doing so, it also specifies if it requires exclusive access to the line or otherwise.
- Send Uncached Request (*e_send_uncached_req*): The controller sends a request to load data in uncached mode.

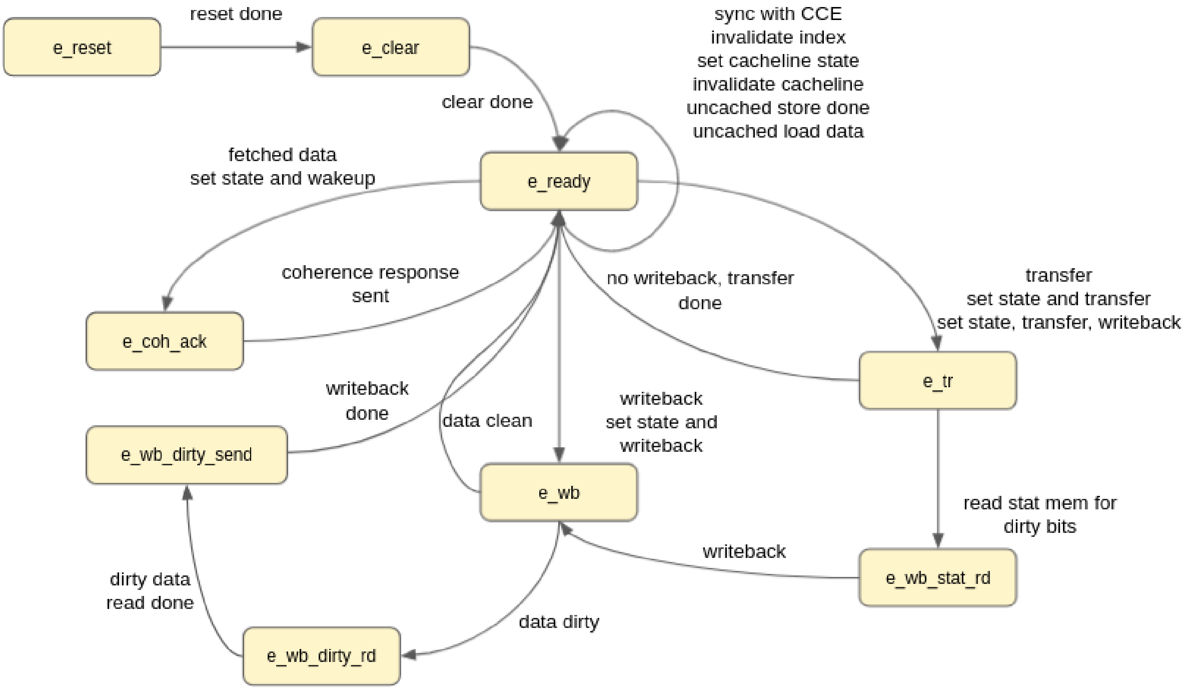


Figure 12: LCE Command and Response Controller

For the command FSM -

- Reset (*e_reset*): Initial state of the controller
- Clear (*e_clear*): Clears the cache tag and stat memories
- Ready (*e_ready*): The controller waits in this state for commands from the CCE. Table 10 shows the commands that could be received by the controller.
- Coherence Acknowledge (*e_coh_ack*): Send acknowledgements after receiving load data or wakeup command.
- Cache to Cache Transfer (*e_tr*): Send an LCE command to another cache to transfer a cache line. If the block was dirty and the state of the line is updated to 'Invalid', then transfer ownership to the other cache. Else, write the data back to the memory.
- Writeback Stat Mem Read (*e_wb_stat_rd*): Read the stat memory to check if the line is dirty after cache to cache transfer.

- Writeback (*e_wb*): If the data is clean, send a null response; otherwise, read the dirty data.
- Writeback Dirty Data Read (*e_wb_dirty_rd*): Reads the dirty data and tag from the cache memories.
- Writeback Dirty Data Send (*e_wb_dirty_send*): Send the dirty data to the memory.

Command	Description
Sync	Register existence of LCE with CCE
Invalidate	Invalidate a cache line
Set state	Set cache line state
Load data	Data in response to a load
Writeback	Writes dirty data back to the memory while setting the state
Transfer	Transfer data to another cache with optionally setting state and writeback

Table 10: LCE Command Types

For a more detailed description of the interface, its capabilities, and the Bedrock system in general, refer to the Interface Specification and Bedrock Guide for BlackParrot [15].

5.3 P-Mesh Cache Engine (PCE)

OpenPiton's Bring Your Own Core (BYOC) [24–26] is a cache-coherent manycore framework designed to interact with processor cores of different ISAs and microarchitectures. In order to provide a common medium of communication between the cores and decouple the core from the memory system, the framework provides a standard interface called the Transaction Response Interface (TRI). Figure 13 illustrates the interface and the memory system. This interface handles loads (both instruction and data), stores and atomic operation requests while forwarding invalidations and responses from the coherence system to the cores. A core that integrates with this system is only required to have an optional write-through L1 cache and a transducer to convert between the core's and the framework's message and data types.

The P-Mesh Cache Engine (PCE) is the default L1 cache controller used for ParrotPiton, a system that integrates BlackParrot into OpenPiton's Bring Your Own Core framework. The PCE is respon-

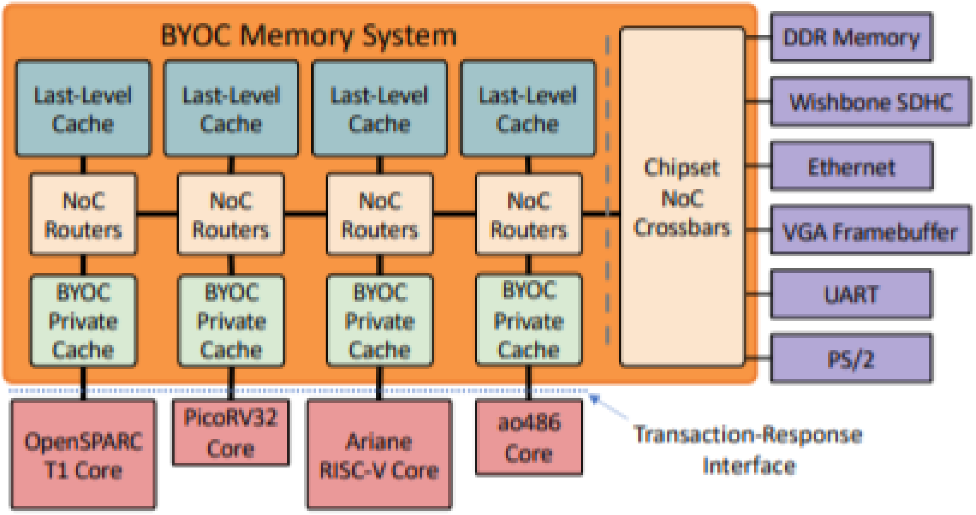


Figure 13: The Bring Your Own Core (BYOC) system with supported cores and peripherals [25]

sible for converting the BlackParrot standard interface messages into messages understood by the TRI.

The BYOC framework uses an L1.5 inclusive write-back cache as the initial point of coherence. This cache is a 4-way set associative and 8 KB big and has 16-byte cache lines. The L1.5 maintains the cache lines from the L1 D$ and forwards the L1 I$ lines to the L2 memory. The L1.5 contains a way map table that keeps track of the ways used in the L1 D$ and maps them to the ways occupied in the L1.5, thereby presenting a uniform interface to the OpenPiton coherence system.

The OpenPiton memory system expects the core's L1 D$ to be write-through and have the exact specifications as the L1.5, i.e., 4-way, 8KB cache with 16-byte cache lines. The core's L1 I$ is expected to be 4-way, 16KB with 32-byte cache lines. The I$ organization has an exact match in BlackParrot's table of supported cache configurations (Table 1), but the D$ organization does not. The next best D$ organization that partially resolves this mismatch is the 2-way, 8KB cache with 16-byte cache lines. This cache has twice as many indices and half as many ways as the L1.5. To present the illusion of a 4-way cache, we divide the cache into four logical ways, mapped as shown in Figure 14.

The conversion between BlackParrot D$'s way and index to the L1.5 way and index are summarised

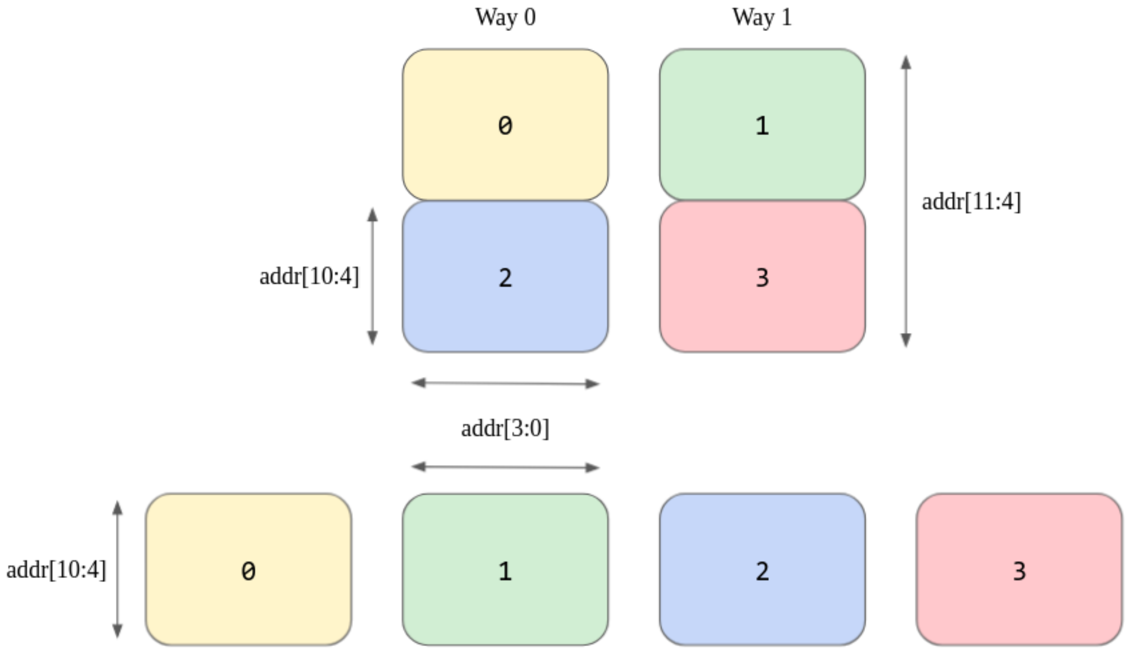


Figure 14: Way mapping in the PCE between L1 D$ (top) and L1.5 (bottom). The address bits shown can be used to index into the logical ways in BlackParrot

in Tables 11 and 12.

Interface	Request Address	Request Way
L1 → PCE PCE → L1.5	Address (40 bits) {Address[39:4], 4'b0000} (40 bits)	Way ID (1 bit) {Address [11], Way ID} (2 bits)

Table 11: Mapping L1 requests to the L1.5

Interface	Response Index	Response Way
L1.5 Responses		
L1.5 → PCE PCE → L1	Request Address[11:4] (8 bits)	Request Way ID (1 bit)
L1.5 Remote Invalidations		
L1.5 → PCE PCE → L1	Inv. Index (7 bits) {Inv. Way ID[1], Inv. Address[11:0]} (8 bits)	Inv. Way ID (2 bits) Inv. Way ID[0] (1 bit)

Table 12: Mapping L1.5 Responses and Invalidations to the L1

OpenPiton is a big-endian system, whereas BlackParrot is a little-endian core. Therefore, any data transfers between the two systems should have their endianness swapped. The PCE is also responsible for swapping the endianness of the data.

The P-Mesh Cache Engine can handle the following operations -

- Loads for both the I$s and D$.
- Write-through stores to the L1.5 D$.
- Forwarding arithmetic and LR/SC atomic operations to the L2 and receive the response.
- Uncached loads and stores.
- Remote invalidations from the OpenPiton coherence system.

Figure 15 shows the PCE's control FSM. Note that the figure does not illustrate the invalidation support because remote invalidations can coincide, and therefore cannot be captured by a dedicated state(s) in the control FSM. The PCE can take invalidations irrespective of its current state. A FIFO also orders the responses, and therefore, there will not be a condition where an invalidation and a normal response arrive simultaneously, thereby losing one of the packets. The FSMs states are briefly described below -

- Reset (*e_reset*): The controller needs to wait for a reset interrupt from the L2 to move out of this state.
- Clear (*e_clear*): Clears the cache tag and stat memories
- Ready (*e_ready*): The controller waits in this state for commands from the cache. Write-through and uncached stores are forwarded to the L1.5 in this state.
- Uncached Store Wait (*e_uc_store_wait*): The controller waits for acknowledgment from the memory system before proceeding further. Such a state is required to ensure non-idempotent operations complete since these operations could be potentially reordered in the memory system. Atomic operations that write to the zero register also behave the same way.
- Send Request (*e_send_req*): The controller sends a uncached load, cached load or an atomic LR, SC or atomic op (AND, OR, XOR, ADD, SWAP, MAX(U), MIN(U)) requests in this state.

- Uncached Read Wait (*e_uc_read_wait*): The controller waits for an uncached load.
- Read Wait (*e_read_wait*): The controller waits for a cacheable load response.
- Atomic LR wait (*e_amo_lr_wait*): The controller waits for the load reserved operation to complete.
- Atomic SC wait (*e_amo_sc_wait*): The controller waits for the store conditional operation to complete.
- Atomic OP wait (*e_amo_op_wait*): The controller waits for the atomic op operation to complete.

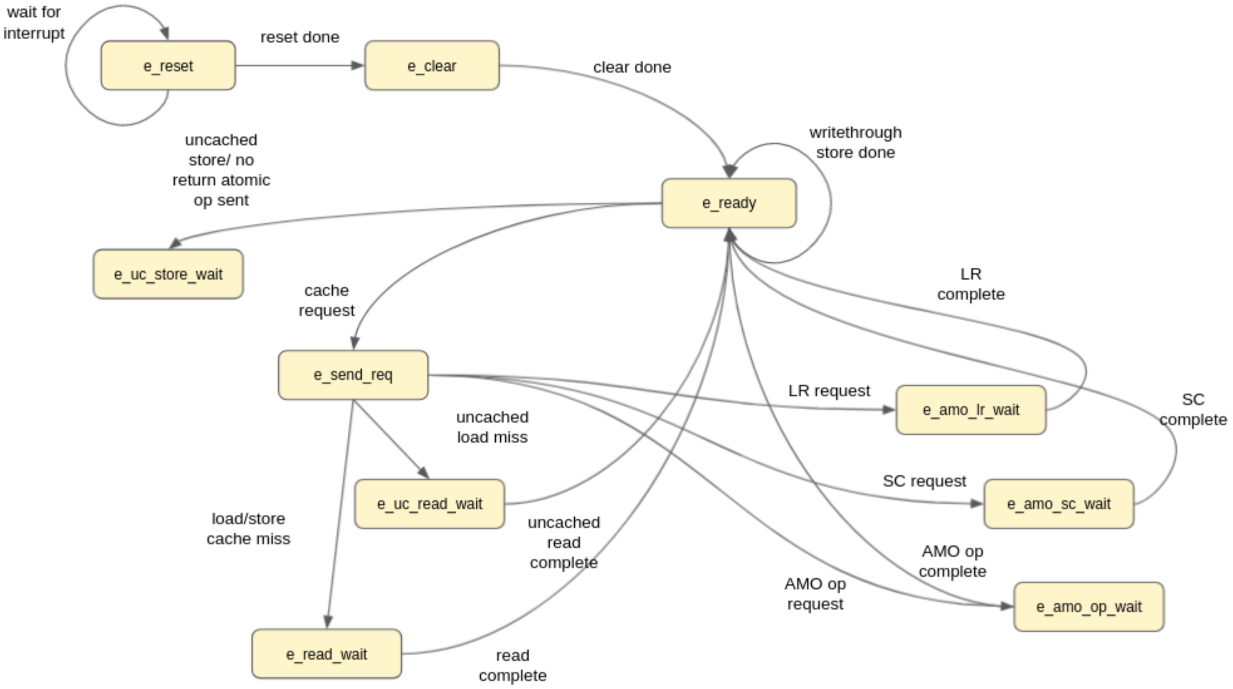


Figure 15: P-Mesh Cache Engine (PCE)

6 Verification Strategy

A SystemVerilog testbench is used to verify the correct functioning of the cache. The testbench uses BaseJumpSTL's verification methodology called trace replay. The trace replay mechanism is a synthesizable verification technique that allows the designer to execute test traces even after taping out the chip.

The methodology uses a ROM that stores test traces. These traces embed in them an opcode that controls the operation of the trace replay module. A trace replay module reads the ROMs' traces and uses the embedded opcode to execute the desired function. Some of the supported operations include -

- SEND: Send the trace as input to the Design Under Test (DUT).
- RECV: Receive output from the DUT and compare against the trace for correctness.
- WAIT: Wait for one cycle.
- CYCLEINIT: Initialize a cycle counter.
- CYCLEDEC: Decremement a cycle counter.
- DONE: End test
- FINISH: Finish simulation by calling $finish.
- NOP: No operation.

The D$ and the I$ have separate testbenches, but the D$ testbench is used to explain all the features. Relevant parts of the core pipeline were emulated in the testbench to ensure the testing proceeds under real operating conditions. Figure 16 shows the unicore testbench setup. Traces are written by hand and loaded into the trace ROM. The traces contain D$ packet information, the physical tag, and a bit to denote if the request is cached/uncached along with the opcode for the trace replay module. The trace replay module fetches the traces from the ROM and enqueues requests into BlackParrot's issue queue. The issue queue outputs one element per cycle unless there is a miss, in which case it issues the trace that caused the miss until the miss resolves. The cache receives the packet and operates as specified in Section 4.1. The physical tag and uncached bit are injected into the cache after a single cycle to mimic the TLB and PMA modules in the core. The output FIFO is dequeued at random time instants to create artificial backpressure. The trace replay module compares the dequeued data against the expected value to check for functional correctness. The fake memory can additionally use Verilog DRAM simulation models, which will be useful to check for timing correctness. The handwritten traces are written to trigger specific conditions and were quite helpful in identifying bugs when writing cache lines back to the memory.

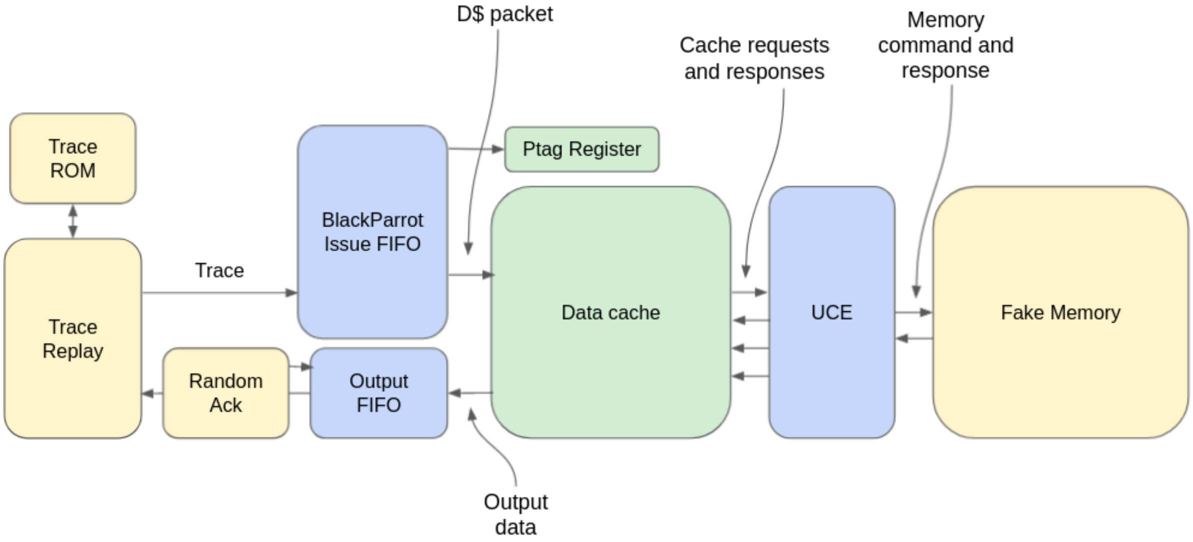


Figure 16: Single core cache testbench

The same testbench was also extended to test the caches in a multicore environment. Multiple D$s are arranged in a manner as shown in Figure 17. Each unit in this testbench (A unit consists of an issue FIFO, the cache, its controller, along with the trace replay modules) runs the same predefined, handwritten traces to trigger specific conditions.

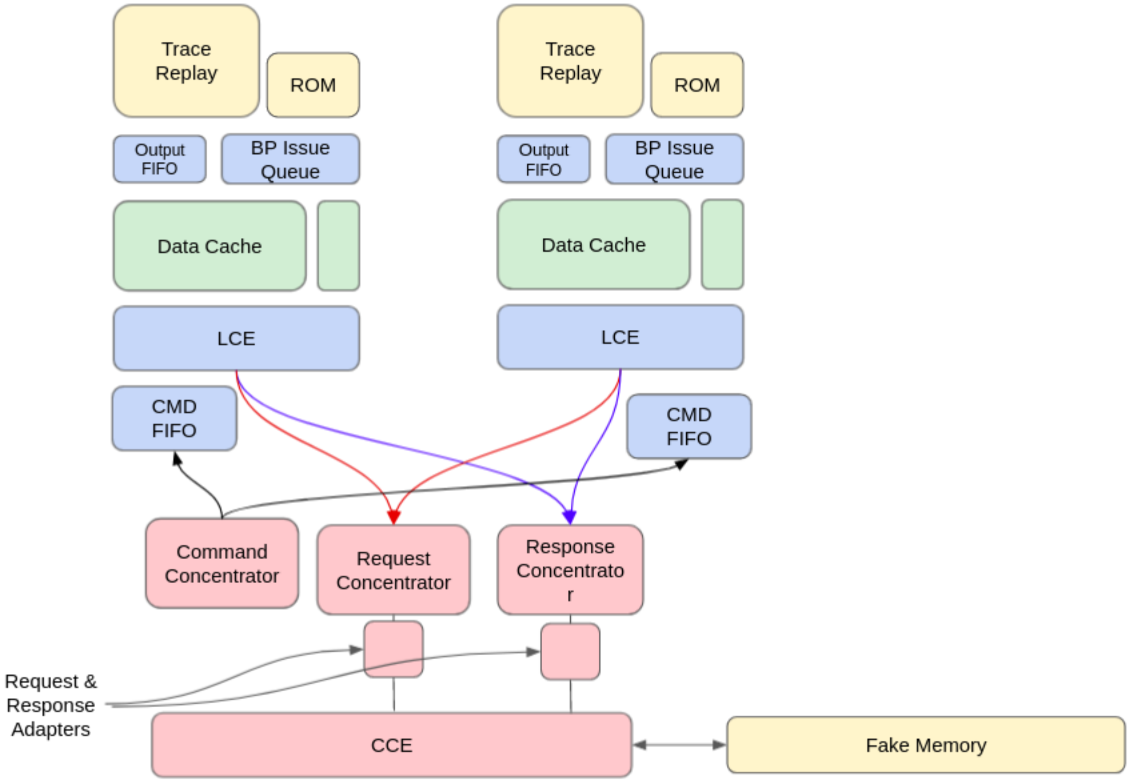


Figure 17: Multicore cache testbench

7 Case Study: TinyParrot

The focus of this case study addresses the issue of having an accelerator core for a system despite many pre-existing physical limitations. The efficient unicore strips away unnecessary multicore components and saves area. A study was conducted to investigate the area occupied by the different components of the core. BlackParrot was synthesized for an ASIC process node and an FPGA. The area reports and utilization numbers revealed that the cache SRAMs and the downstream buffering in the default BlackParrot configuration contributed significantly to the area.

This problem was addressed with the contributions made in this thesis by switching to smaller cache organizations and exploiting the iterative filling/eviction with sub-banking. With these area optimizations in place, BlackParrot can scale to smaller sizes, thereby expanding the range of systems that can use BlackParrot as a host core. The following sections present the synthesis results for a 45nm ASIC process node and an FPGA, along with any challenges that had to be overcome in this exercise.

7.1 ASIC Synthesis

Bespoke Silicon Group has a well-established ASIC tool flow to convert SystemVerilog/Verilog RTL to GDSII layout. BlackParrot with different cache organizations was synthesized using this tool flow as a backbone and an open-source 45nm library, FreePDK45 [27]. The core BlackParrot RTL along with the necessary changes (a combination of commit: c945954e7c and commit: e5723c681a) for the different cache optimizations was merged and synthesized using a pre-existing physical design methodology.

7.1.1 Challenge(s)

Without any intervention, the synthesis tool converts memories into a 2-D array of flip flops. Flip flops are huge gates requiring almost 20 transistors to realize them. The synthesis process that uses flip flops as memory bit cells would also take a long time to finish with large memories. Therefore, memory macros or black boxes generally replace flip-flop-based memories to enable faster synthesis and occupy lower die area. A RAM generator is a tool that converts a memory specification (depth of memory, width of memory) into a macro, made up of 6 transistor SRAM cells packed together tightly along with any interface logic. The synthesis compiler takes as input these pre-generated memory macros and other RTL to compile the design. The synthesis tool views these memory macros as black boxes, thereby significantly saving compilation time. Additional optimizations from BaseJumpSTL, such as width-banking and depth-banking, helped create smaller, square memories optimized for area. For this case study, a RAM generator from the Bespoke Silicon Group, **bsg_fakeram** was used to generate the different RAMs in the design. However, one limitation of this RAM generator was that it could generate only single-ported memories. BlackParrot's integer register file requires 1 write port and 2 read ports, while its floating-point register file requires 1 write port and 3 read ports. BaseJumpSTL modules that realized n-read and 1-write memories using 1-read and 1-write memories, partially solve the mismatch in the number of ports required. A new BaseJumpSTL module was created to convert the 1R1W (dual port) memories to a 1RW (single port) memory that this RAM generator could generate. Figure 18 gives an illustration of this design.

The key idea here is to use two 1RW memories of the exact specification as the 1R1W memory and maintain an additional register for each element in the memory that tells where the next write should go.

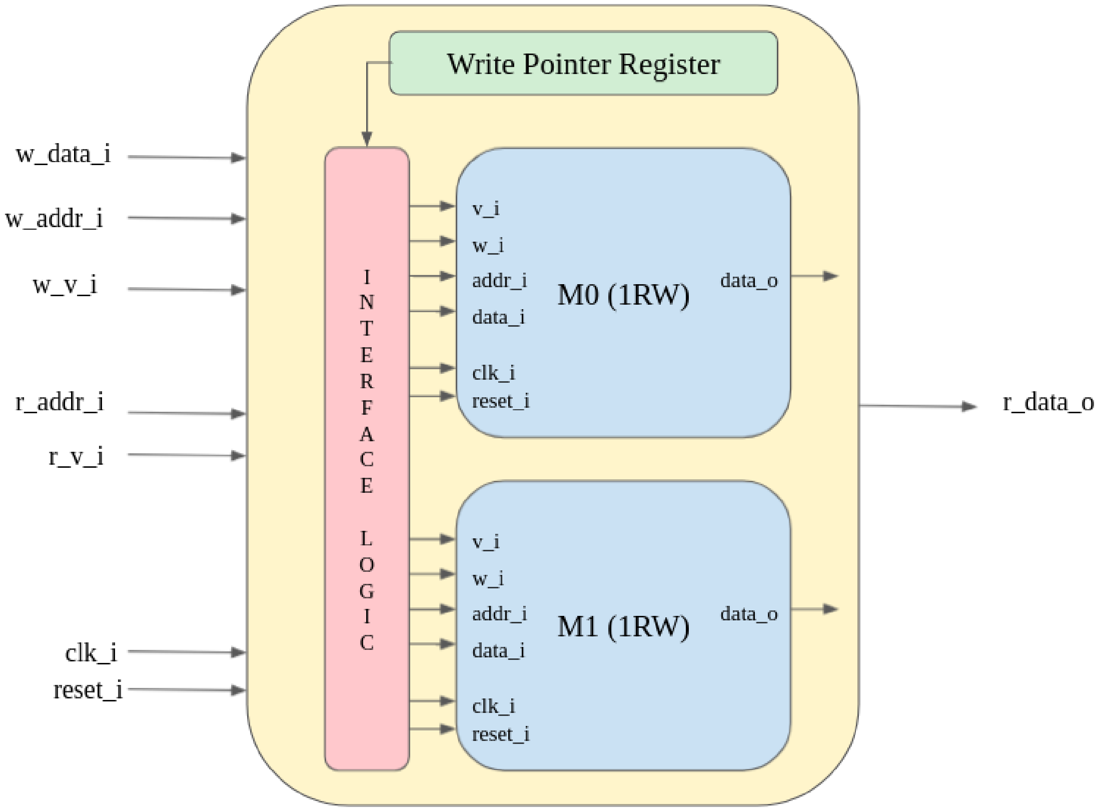


Figure 18: Dual-ported memories using single-ported memories

7.1.2 Synthesis Results

With the necessary design optimizations in place and minor modifications to the synthesis flow, area runs were conducted, and the results summarised. Figure 19 shows the area breakdown of a single core BlackParrot, with different cache organizations. For each case, 64-byte, 32-byte, and 16-byte cache lines, respectively, and an 8-byte fill width were used.

7.2 FPGA Synthesis

A tool flow to synthesize BlackParrot using Xilinx Vivado for a specific FPGA target and simulate the synthesized netlist using Synopsys®VCS was created. For the work presented in this thesis, the FPGA utilization reports for BlackParrot with different cache organizations are generated using the Vivado synthesis tool flow. The target FPGA was a Xilinx Artix-7 FPGA (XC7A200T) [28].

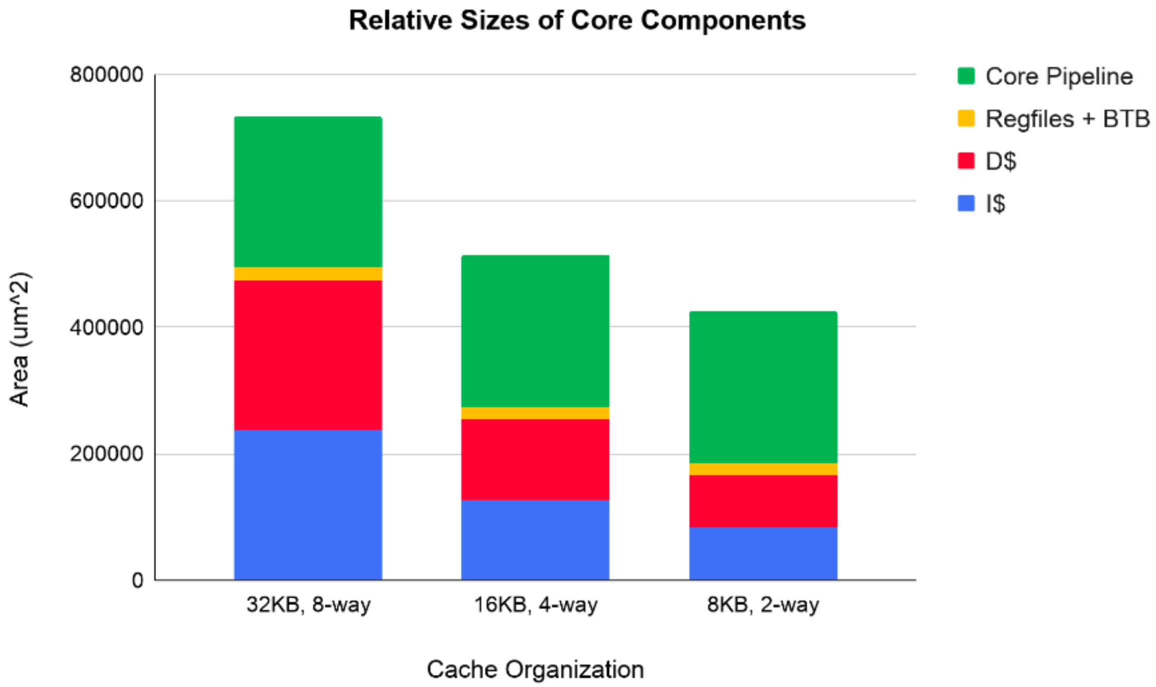


Figure 19: Area breakdown of BlackParrot with different cache organizations in the FreePDK 45nm process node. Note: BlackParrot is a **continuously evolving** processor core and the version used for this assessment was accessed on Dec 2, 2020.

Table 13 summarises the utilization for BlackParrot with different cache configurations (commit: 708ae8fb76). For each case, 64-byte, 32-byte, and 16-byte cache lines, respectively, and an 8-byte fill width were used.

Cache Organization	LUTs	Block RAMs (36Kb)	DSP48	Flip Flops
32KB, 8-way	24752 (18.39%)	46.5 (12.74%)	11 (1.49%)	10402 (3.86%)
16KB, 4-way	22335 (16.59%)	26.5 (7.26%)	11 (1.49%)	8594 (3.19%)
8KB, 2-way	21231 (15.77%)	16.5 (4.52%)	11 (1.49%)	7681 (2.85%)

Table 13: FPGA Utilization breakdown of BlackParrot with different cache organizations. Note: BlackParrot is a **continuously evolving** processor core and the version used for this assessment was accessed on May 12, 2021.

BlackParrot is ASIC-optimized, and the utilization numbers shown in Table 13 is this ASIC-optimized version of the core synthesized for an FPGA. By devoting more attention to mapping the core more efficiently onto the FPGA, the utilization can be decreased further.

Moreover, the area utilization reports and the cost of Xilinx FPGAs indicated that users of the core would only have to spend $100 to buy an Artix-7 (XC7A50T) FPGA of which TinyParrot would occupy approximately 50%, the rest of which could be used to fit in their own accelerator.

8 Case Study: HammerParrot

Traditionally, CPUs and GPUs (or similar manycore architectures) are separate chips on the motherboard that communicate over sophisticated bus protocols like PCIe. This setup is advantageous for the reasons listed below -

- Integrated CPU and GPU chips usually favor the CPU and use the GPU as a tinier co-processor typically weaker than a dedicated GPU. Dedicated chips have more space for more processor cores or functionality.
- Dedicated GPU cards need high bandwidth channels to DRAM, which is not quite possible on an integrated chip.

However, the problem with this approach is,

- Communication at the board level requires a sophisticated bus protocol which is typically harder to set up and verify. PCIe is a popular choice for such a bus architecture which warrant dedicated teams in large companies to ensure functionality. The number of PCIe experts is even lower for open source and academic research projects.
- The bus protocol increases the latency of communication between the host CPU and the GPU.
- Dedicated chips for CPUs and GPUs provide higher performance at the expense of higher power consumption.

HammerBlade (referred to as the manycore in short) [29] is a Tiled Manycore designed for machine learning and graph analytics workloads. It builds on the MIT RAW [30] processor and the 511 RISC-V core Celerity architecture [31]. High-level software written in PyTorch/GraphIt maps their operations to the manycore hardware architecture using CUDA-lite and an intermediate representation layer designed for HammerBlade [32]. The programmer uses the SPMD (Single Program

Multiple Data) programming model to program the manycore. More information about the system architecture will follow in later sections.

Like any other traditional host-accelerator setup, the HammerBlade manycore relied on an off-chip x86 host CPU. Figure 20 shows the current setup that connects the host CPU and the HammerBlade manycore.

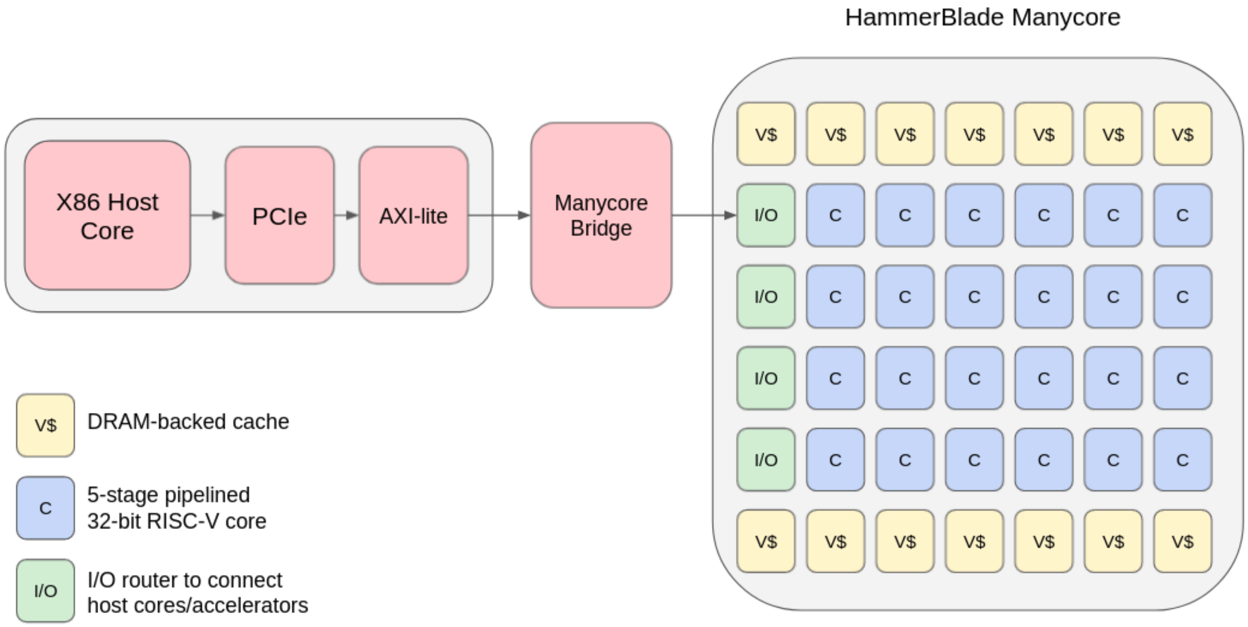

HammerBlade Manycore
X86 Host Core
PCIe
AXI-lite
Manycore Bridge
V$
I/O
C
V$ DRAM-backed cache
C 5-stage pipelined 32-bit RISC-V core
I/O I/O router to connect host cores/accelerators

Figure 20: HammerBlade with off-chip x86 host core

Previous comments on hosting accelerators and CPU and GPU integration concerns can be alleviated by using an open-source, customizable host core like BlackParrot. It offers flexible integration and benefits from being a tiny core that shares the same die area as the HammerBlade manycore. The proximity provides the flexibility of adding scalable, specialized hardware modules that can achieve peak system performance compared to complicated off-chip buses such as PCIe. This resulting system is codenamed HammerParrot.

8.1 HammerBlade System Architecture

The HammerBlade manycore is a sea of tiles connected by a 2-D mesh network. The tiles on the network can be of three types - Compute, Accelerator, and Memory. Each tile, in addition, contains an endpoint interface that manages communication with other tiles.

8.1.1 Compute Tile

The compute tile in the HammerBlade manycore consists of a 5-stage pipelined, 32-bit RISC-V processor. The core supports integer, atomic, and control instructions, among others, and has a floating-point unit. Refer to the HammerBlade GitHub repository [29] for a full list of supported instructions. Figure 21 shows the compute tile's core pipeline.

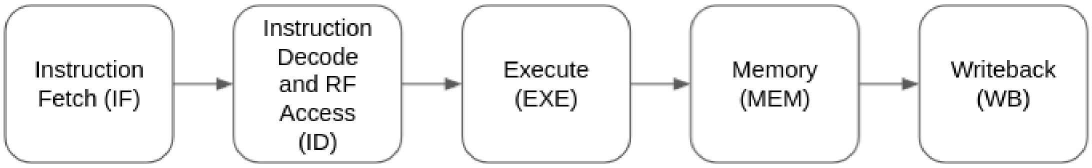


Figure 21: Compute tile core pipeline

Each processor core has an I$ and data memory. The I$ is a 1024 entry, direct-mapped cache, and the data memory is a 4KB, single-ported memory that the core can use as a scratchpad. The data memory maps to a portion of the global address space, which other tiles on the network can access. The core pipeline is a standard 5-stage pipeline with some modifications that allow remote loads and stores to memory locations on other compute tiles or the global memory.

8.1.2 Memory Tile

The memory tile is also called a Victim Cache (V$). It has a pipelined datapath with configurable associativity and cache size, with each cache line being an integer multiple of 32-bit words. The cache follows a write-back, write-allocate, and fetch on write policy. V$s are placed at the north, or south sides of the compute array and are backed by high bandwidth DRAM. The DRAM space is striped across V$s in a cache line granularity. There can be multiple rows of V$s on the north and south side as well. In this case, the cache line is striped from the inner to the outer rows.

8.1.3 Accelerator Tile

Accelerator tiles are tiles that can be used as an accelerator. These tiles are expected to occupy nodes to the east and west of the compute array and connect to the compute array via I/O routers. BlackParrot will act as an "accelerator" to the manycore and be connected to the east (and possibly west) of the manycore array. A later section (Section 8.2) will give more details about how this connection will be made.

8.1.4 Networking and Addressing

Tiles are connected with a 2-D mesh network. There are two physical networks - One network is used to send requests, and the other network is used to receive responses with routing algorithms that avoid deadlock. HammerBlade uses a 32-bit Partitioned Global Address Space (PGAS) with an efficient addressing scheme that clearly defines nodes on the network. The HammerBlade manycore uses a packet mechanism to communicate between tiles and global memory on the network. Each packet contains the necessary information to remotely carry out the desired operation, the address, and the tile coordinates. Response packets contain sufficient information to identify the request that originally generated it. More information on the network and addressing schemes used in HammerBlade can be found in [29].

8.2 HammerParrot Hardware

Figure 20 shows the hardware modules required by an x86 core to host HammerBlade. Since BlackParrot will share the same die as the manycore, complicated bus protocols such as PCIe are no longer required, and the manycore bridge is the only hardware module left to be designed. The x86-HammerBlade bridge consists of FIFOs that the host core uses to send and receive data to/from the manycore. As mentioned earlier, the x86-HammerBlade system is PCIe based, and the PCIe protocol imposes strict response latency requirements, which, if not adhered to, causes the entire system to hang. The FIFOs in the bridge help to adhere to these requirements. The x86-HammerBlade bridge inspired the design of the HammerParrot manycore bridge.

8.2.1 Bridge FIFO Interface

The HammerParrot manycore bridge consists of the traditional host FIFO interface (like the x86-HammerBlade bridge) that can be written to and read from, using memory-mapped stores and loads, respectively. A request packet constructed in software is written to the host request FIFO as a series of 32-bit stores. The host request FIFO is modeled as a SIPO on the BlackParrot side while the HammerBlade response and request FIFOs are PISOs. 4 32-bit stores create a 128-bit packet that can be sent over the network as a manycore request packet. The manycore enqueues the responses to these requests in the host response FIFO and BlackParrot retrieves them using four 32-bit loads. A similar operation is performed when the host chooses to accept manycore requests. The host, however, does not generate responses to requests from the manycore. A manycore packet is sent/retrieved using four 32-bit stores/loads instead of 2 64-bit stores/loads or a single 128-bit store/load because the host software API exposes a packet interface that renders manycore packets as a 4-element array of 32-bit unsigned integers.

8.2.2 Bridge MMIO Interface

One of the downsides of the x86 host interface was the inability to map the manycore into its own address space, thereby not being able to access any location on the manycore directly. The x86 host core, therefore, had to rely on software address translation and packet creation which incurs an overhead of 100s of cycles and an additional overhead of 1000s of cycles for transmitting the packet over PCIe.

BlackParrot overcomes these shortcomings by having the flexibility to parameterize its physical address space to map the whole of the manycore's 32-bit address space, allowing BlackParrot to access everything inside the manycore like any other compute or accelerator tile. Any BlackParrot address (with the high bit set to denote it is a manycore MMIO request) is viewed as an endpoint address and can be translated to an address and tile coordinates that uniquely identify the desired resource. BlackParrot's single-core uses a Bedrock interface to the manycore bridge. The manycore bridge readily converts Bedrock messages into HammerBlade packets. Furthermore, since, BlackParrot is on the same network as the manycore, the overhead of the PCIe bus is eliminated, thereby creating an interface that achieves peak performance. The software only has to execute a single instruction (i.e., load/store) to communicate with the manycore.

Direct access into the manycore address space requires extra care, however. BlackParrot's configurable address space allows it to access any address in the manycore address space. However, performing loads and stores to these addresses can receive responses in a different order because of network latency. Requests to tiles farther away from BlackParrot on the network will take longer to return with a response, while requests to closer tiles might return with a response immediately. A reorder FIFO from BaseJumpSTL was used to handle out-of-order responses from the manycore. Any outgoing request reserves an ID in the reorder FIFO when it is sent to the manycore network. Responses can return in any order and populate the corresponding entry in the FIFO. The reorder FIFO signals that it has a valid packet at its output only when the earliest ID's request returns with a response. BlackParrot currently supports a configurable number of such requests to be in flight at the same time.

8.2.3 System-level Address Maps

Documentation in the HammerBlade Github repository [29] defines in detail the addresses that map to different compute and memory tiles and how to construct them. Since BlackParrot would like to embed the entire manycore address space into its own, BlackParrot's address space is configured to use 42-bits. The manycore is designated as an off-chip ASIC that can communicate with BlackParrot by setting the high bits in BlackParrot's address space. Table 14 shows the full address map.

Address	Use
11_1000_0000_0PPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP	Manycore Bridge FIFO
00_0000_yyxx_1PPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP	Global DRAM banks. yyxx denotes DRAM coordinates.
10_yyy_XXXX_XXXPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP_PP00	Victim cache (V$) address + tags. yyy denotes the V$'s Y coordinate (north/south). XXXX_XXX denotes the X coordinate
11_0000_00YY_YYYY_YXXX_XXXX_PPPP_PPPP_PPPP_PPPP_PP00	Compute Tiles YY_YYYY denotes the Y coordinate XXX_XXXX denotes the X coordinate

Table 14: HammerParrot Address Map

BlackParrot fulfills the traditional host interface by performing memory-mapped stores and loads to the manycore bridge. Table 15 defines the different addresses in the manycore bridge that map to different FIFOs.

8.2.4 Top-level integration

Each BlackParrot tile occupies three nodes on the manycore network. All three nodes together form the manycore bridge. One node contains the host FIFO and MMIO interfaces and used as a means for the BlackParrot core to make/receive requests and receive responses. In the HammerParrot

Address	Use
0x1000	Write to the host request FIFO
0x2000	Check for available host request credits
0x3000	Read from the manycore response FIFO
0x4000	Check for available entries in the manycore response FIFO
0x5000	Read from the manycore request FIFO
0x6000	Check for available entries in the manycore request FIFO

Table 15: HammerParrot Bridge FIFO address map

system, BlackParrot uses a 32KB, 8-way cache with 64-byte cache lines with a fill width of 8 bytes which means every request to DRAM will be a 64-bit load/store. However, the manycore can only support 32-bit fills. Therefore, every request from BlackParrot is split into two requests to the manycore and injected into the network at two different nodes to handle this mismatch. When both the split requests return with their responses, the splitter combines them and sends a single 64-bit data packet back to BlackParrot. The additional two nodes are occupied to accommodate this setup. Figure 22 shows the HammerParrot manycore bridge.

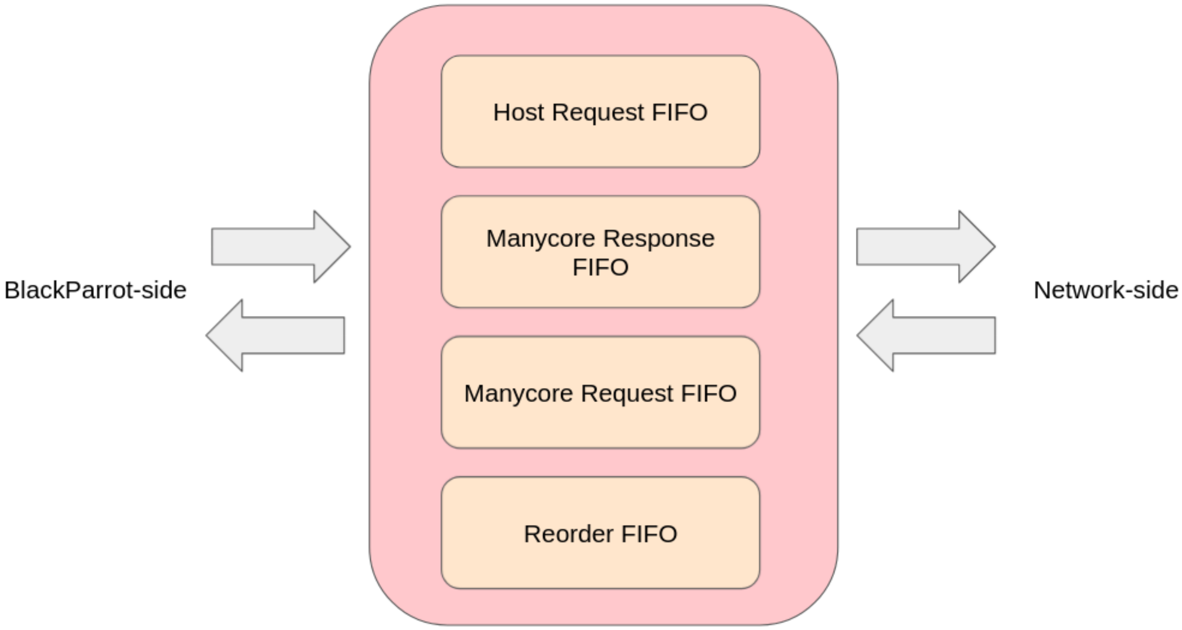


Figure 22: HammerParrot manycore bridge

8.3 HammerParrot Software

8.3.1 HammerBlade-x86 Simulation Infrastructure

The HammerBlade software stack written in C/C++ is based on CUDA-lite hosted at [32]. The library handles many things, including software-level address translation, sending and receiving packets to the manycore hardware, writing and reading from tiles, creating tile groups and synchronizing between tiles within a tile group, and allocating kernels to tile groups, and more. A system would only have to implement a platform-specific API exposed by the library to run the CUDA-lite host code. The API dictates how the platform communicates with the manycore and can be thought of as a hardware abstraction layer that hides the underlying communication mechanism from higher levels of software. However, this complicated software stack has only been compiled and tested with an x86 host core running Linux and an Amazon EC2 F1 instance. Figure 23 shows how the existing system was simulated using Synopsys VCS.

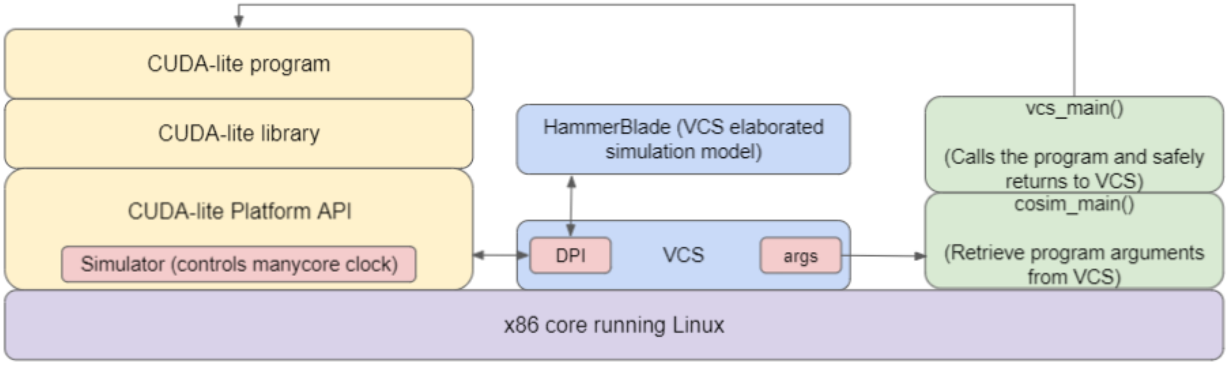

CUDA-lite program
CUDA-lite library
CUDA-lite Platform API
Simulator (controls manycore clock)
HammerBlade (VCS elaborated simulation model)
DPI
VCS
args
vcs_main()
(Calls the program and safely returns to VCS)
cosim_main()
(Retrieve program arguments from VCS)
x86 core running Linux

Figure 23: HammerBlade Simulation Infrastructure in VCS

The x86 host core executes CUDA-lite code until it needs to send/receive data to/from the manycore. To send/receive data, the platform API uses a SystemVerilog Direct Programming Interface (DPI) to communicate with hardware FIFOs. These FIFOs simulate the x86 manycore bridge that connects and communicates with the manycore.

8.3.2 Porting CUDA-lite to RISC-V

Replacing the x86 core with BlackParrot can proceed either by booting up Linux on BlackParrot in simulation and running the CUDA-lite binary on top of Linux or running the binary in bare metal mode. While the Linux route will emulate the final system, it is very time-consuming because it takes many hours to a few days to boot Linux in BlackParrot in simulation. To run in bare metal mode, the CUDA-lite library needed to be compiled using a lightweight implementation of the standard C library called PanicRoom [33]. PanicRoom is a port of Newlib, a lightweight C library for embedded systems, and some system calls for file I/O and a DRAM-based file system called LittleFS. The CUDA-lite library was easy to port to PanicRoom save for a few missing libraries, which were quickly resolved with help from experts. The next step was to then replace the x86 core in the simulation infrastructure with BlackParrot.

8.3.3 Software Validation using Dromajo HammerBlade

BlackParrot and HammerBlade are both continuously evolving machines. A direct integration meant that if something goes wrong, one had to debug either the BlackParrot RTL, HammerBlade RTL, or the software. To reduce this initial verification burden as well as to accelerate the software porting effort, Dromajo [34], an open-source C++ reference model for a 64-bit RISC-V core from Esperanto Technologies, was used. Dromajo is originally a RISC-V reference model used to verify RTL correctness through RTL co-simulation. The BlackParrot project uses Dromajo extensively to verify RTL correctness after every feature change. It works by executing each instruction when the RTL commits an instruction and comparing results with the RTL. However, Dromajo can also be used in a standalone mode which is exploited here.

Dromajo’s source code was modified to create the HammerParrot bridge in software to emulate the HammerParrot set up as closely as possible. The Dromajo-HammerBlade bridge supports only the traditional FIFO interface like the x86 system, and the MMIO support is a work in progress.

8.3.4 Dromajo HammerBlade Simulation Infrastructure

Since Dromajo is still only a RISC-V software model, it cannot directly communicate with the hardware running in the VCS simulation environment. Therefore, the packets from the Dromajo FIFOs are communicated to the manycore hardware via the Direct Programming Interface like in the x86 case, which requires that the Dromajo source code be compiled with the VCS executable and another x86 binary that interfaces between VCS and Dromajo through DPI. The x86 binary, in this case, is called the “Simulator” (note: this is not VCS. Refer to Figure 23) that controls the manycore clock. In this case, the Simulator will also execute N instructions in the Dromajo platform at every manycore clock tick and poll the DPI interface for any request and response packets from the manycore while transmitting any host request packets from Dromajo to the manycore. The platform API also requires implementations for other functions that require polling the hardware directly for data, such as the manycore configuration stored in a ROM (not mapped to the manycore address space) or checking if the manycore has finished cycling through a reset or retrieving arguments before calling main. Such requests are handled by sending and receiving packets to the Simulator, which emulates the BlackParrot host that performs these operations. Figure 24 summarises this whole setup.

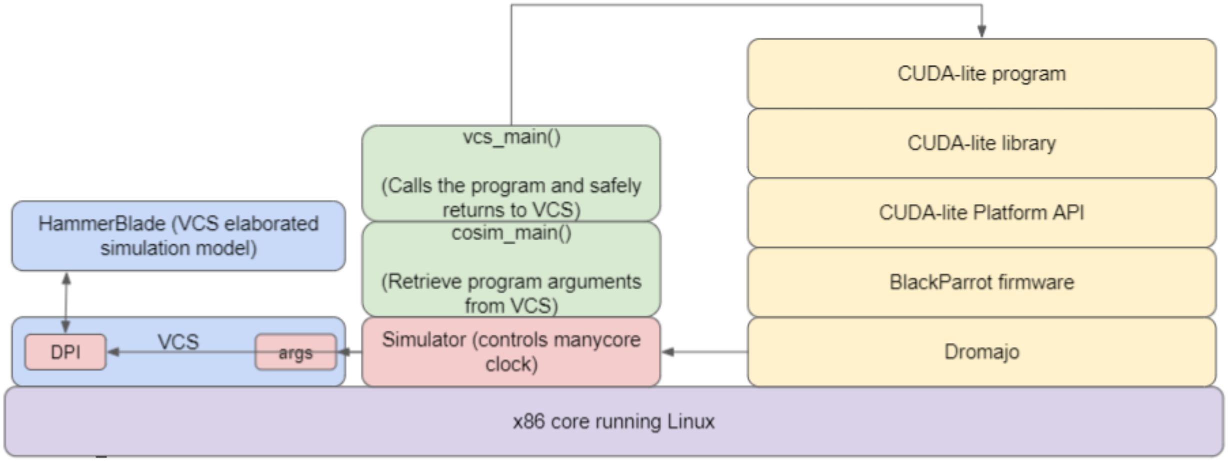

CUDA-lite program
CUDA-lite library
CUDA-lite Platform API
BlackParrot firmware
Dromajo
vcs_main()
(Calls the program and safely returns to VCS)
cosim_main()
(Retrieve program arguments from VCS)
Simulator (controls manycore clock)
HammerBlade (VCS elaborated simulation model)
DPI
VCS
args
x86 core running Linux

Figure 24: Dromajo+HammerBlade Simulation Infrastructure in VCS

This cross-platform integration effort revealed many inconveniences in the HammerBlade simulation infrastructure. Therefore the Dromajo+HammerBlade integration was incredibly beneficial for both projects and has set the stage for HammerParrot's simulation infrastructure.

9 Future Work

BlackParrot's configurability has significantly improved with the work presented in this thesis; however, there is still work pending. Optimizations to the cache such as iterative cache line filling/eviction and sub-banking have certainly created a means for BlackParrot to transfer data in smaller chunks. However, the core continues to remain stalled during the fill/eviction, contributing to an increase in the miss latency of the cache. Techniques such as critical word first can reduce this performance penalty. BlackParrot needs to support true critical word first in its caches, rather than the rudimentary version that is currently used. Some important questions to consider here are

- How to safely restart the core?
- How to handle the contention for the cache memories for an ongoing fill and core access?
- How to handle the case when the core tries to query a portion of the cache line being filled?

TinyParrot allows single-core BlackParrot to scale in size by modifying the cache organization. While this offers area and power benefits, performance takes a hit. An in-depth analysis of the system's performance under real working conditions in its tiny configuration is necessary to understand the magnitude of the penalty. However, given that the tiny system has a smaller cache, analyzing the cache access patterns in real workloads might be sufficient. Some high-impact solutions in this space include making the cache non-blocking and adding the necessary support in the controllers, which requires careful modifications to the core and crucial decisions, such as if the cache will support hit-under-miss or miss-under-miss. A more straightforward solution that might work from the start is increasing the TLB size to get better performance on virtual memory workloads. A simple configuration change in BlackParrot for the TLB size, could provides a significant impact since most of the time is spent walking page tables in an actual VM workload.

With so many modifications required in the cache, a better testbench is also required. The current testbench does not support randomized testing and requires designer intervention to create directed tests and place their results in the trace ROM. An upgrade to the testbench will include constrained random testing, a more sophisticated and automated self-checking mechanism, and coverage analysis to test the cache pipeline more thoroughly. The multicore cache testbench also needs test traces that generate interactions between the different caches to check the memory consistency model.

BlackParrot's overall system performance in the HammerParrot system can be improved further with the cache optimizations mentioned above. A repeat of the TinyParrot case study also needs to be conducted to understand the position of BlackParrot in ASIC and FPGA environments and iterate further to achieve the required area efficiency with a minimal performance penalty.

10 Conclusion

The work presented in this thesis optimizes BlackParrot for integration into accelerator-centric systems by improving its configurability. The standardized cache interface and the iterative filling mechanism jointly contribute towards making the core tinier to fit on low-cost FPGAs or occupy much less silicon area. The standard interface also allows BlackParrot to plug into different environments with minimal overhead. Furthermore, its configurable address space and the standard interface create an opportunity to reuse pre-existing components to achieve a scalable, high-performance connection to a tiled manycore architecture.

BlackParrot is currently being ported to diverse FPGA architectures, and the HammerParrot system will be taped out soon using the Global Foundries 14nm process node. With a few more optimizations in place to achieve the best performance possible, only integrating BlackParrot with more standalone accelerators or into more accelerator SoCs will remain, bringing it one step closer to achieving the goal of becoming the default accelerator host multicore in state-of-the-art SoCs.

Acknowledgements

I would like to thank my parents Sreelatha and Muralitharan, and my sister Nithyasree for their constant support without which, I would not have survived almost two years away from them in a foreign country in graduate school.

I am very grateful to my advisor, Professor Michael Taylor, for having given me the opportunity to work in his research group and funding my master's program. I am grateful to him for putting me in situations that always demanded more from me and allowed me to grow personally and professionally. I am thankful for his guidance and expertise, which has helped me acquire the insight to approach problems the right way.

I am also grateful to Professor Scott Hauck, my initial advisor. His computer architecture class in my first quarter at the University of Washington gave me a strong foundation that helped me acquire the necessary skills to complete this thesis. I was also a TA for him for an introductory course in digital design. During that time, his guidance helped me grow in a professional capacity.

I would like to thank Daniel Petrisko, the lead architect of the BlackParrot processor, for being a fantastic mentor throughout this master's experience. I have been closely working with him for almost two years now, and I have had a lot of fun and learned a lot from him.

I would also like to thank other members from the Bespoke Silicon Group - Farzam Gilani for helping me with issues relating to software and the Dromajo co-simulation framework, Scott Davidson for helping me when I ran into trouble with the ASIC flows, Dai Cheol Jung for answering questions relating to the HammerBlade RTL, Dustin Richmond and Max Ruttenberg for helping me with bringing up the Dromajo + HammerBlade simulation infrastructure and many others. I would also like to thank Jonathan Balkind (Princeton University; now an Assistant Professor at the University of California, Santa Barbara) for guiding me for the brief duration that I worked on the ParrotPiton project.

I would like to thank my roommates - Shashank Vijaya Ranga and Mukund Gupta, for being a constant source of support in my home away from home and being patient with me, especially during the COVID-19 pandemic. I was fortunate to have Shashank trod the same path with me during this master's program. Both of us joined the same research group and have similar academic interests, and this has supported me throughout this program. Our lunch and dinner time discussions on digital design or otherwise were quite stimulating.

I would like to thank all my friends and family members here in the United States and in India and my professors from my undergraduate university, without whom I wouldn't have been able to gain this master's experience.

This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([5, 35–48]), ASIC Clouds ([49–56]), open source hardware ([17, 57, 58]) RISC-V ([1, 31, 59–63]), Network-on-Chips ([20, 64–67]), security ([68–71]), benchmark suites ([72–74]), dark silicon ([3, 47, 48, 75, 75–78]), multicore ([1,30,66,67,79–87]), compiler tools ([88–96]) and FPGAs ([74,97,98]).

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7856, FA8650-18-2-7846 and FA8650-18-2-7863. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This work was partially supported by NSF SaTC Award 1563767, and NSF SaTC Award 1565446.

References

[1] 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.

[2] "Dennard Scaling," https://en.wikipedia.org/wiki/Dennard_scaling.

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

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[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] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 52–59.

[7] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A Reconfigurable Architecture for Parallel Patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[9] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "Hardware Acceleration of the pair-HMM Algorithm for DNA Variant Calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 275–284.

[10] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol *et al.*, "MIAOW-an open source RTL implementation of a GPGPU," in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. IEEE, 2015, pp. 1–3.

[11] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and Benchmarking of Machine Learning Accelerators," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–9.

[12] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The Rocket Chip Generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[13] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[14] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, “SHAKTI Processors: An Open-Source Hardware Initiative,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 7–8.

[15] “BlackParrot,” https://github.com/black-parrot/black-parrot.

[16] “RISC-V Specifications,” https://riscv.org/technical/specifications/.

[17] M. B. Taylor, “BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design,” in *Design Automation Conference*, June 2018.

[18] “BaseJumpSTL,” https://github.com/bespoke-silicon-group/basejump_stl.

[19] “Pseudo LRU,” https://en.wikipedia.org/wiki/Pseudo-LRU.

[20] D. Petrisko, C. Zhao, S. Davidson, and P. Gao, “NoC Symbiosis:(Special Session Paper),” in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2020, pp. 1–8.

[21] “BlackParrot Link Protocols,” https://tinyurl.com/yfg67awm.

[22] “BSG SystemVerilog Coding Guidelines,” https://tinyurl.com/yzl7fqot.

[23] “CMurphi,” http://mclab.di.uniroma1.it/site/index.php/software/18-cmurphi.

[24] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, “OpenPiton: An Open Source Manycore Research Framework,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: ACM, 2016, pp. 217–232. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872414

[25] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba *et al.*, “BYOC: a” bring your own core” framework for heterogeneous-ISA research,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 699–714.

[26] “BYOC: OpenPiton Research Platform,” https://github.com/bring-your-own-core/byoc.

[27] “FreePDK45,” https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[28] Xilinx, "Xilinx 7-Series FPGAs Datasheet," https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[29] "HammerBlade Manycore," https://github.com/bespoke-silicon-group/bsg_manycore.

[30] M. Taylor, "Tiled Microprocessors," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[31] 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.

[32] "BSG Replicant: HammerBlade Manycore Cosimulation and Emulation Infrastructure," https://github.com/bespoke-silicon-group/bsg_replicant.

[33] "Panic Room," https://github.com/bespoke-silicon-group/bsg_newlib_dramfs/tree/dramfs.

[34] "Dromajo," https://github.com/bsg-external/dromajo/tree/blackparrot_mods.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] M. B. Taylor, "Geocomputers and the Commercial Borg," in *SIGARCH Computer Architecture Today*, Dec 2017.

[52] M. Taylor, "The Evolution of Bitcoin Hardware," *Computer, IEEE*, Sept-Oct. 2017.

[53] M. Khazraee, L. Vega, I. Magaki, and M. Taylor, "Specializing a Planet's Computation: ASIC Clouds," *IEEE Micro*, May 2017.

[54] 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.

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

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

[57] ——, “Your agile open source HW stinks (because it is not a system),” in *ICCAD*, 2020.

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

[59] 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.

[60] ——, “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.

[61] 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.

[62] 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.

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

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

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

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

[67] 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.

[68] 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.

[69] B. Hawkins, B. Demsky, and M. B. Taylor, " BlackBox: Lightweight Security Monitoring for COTS Binaries," in *Code Generation and Optimization*, 2016.

[70] ——, "A Runtime Approach to Security and Privacy," in *European Security and Privacy*, 2016.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] M. Taylor, "A Landscape of the New Dark Silicon Design Regime," in *Design Automation and Test in Europe*, April 2014.

[76] ——, "A Landscape of the New Dark Silicon Design Regime," *Micro, IEEE*, Sept-Oct. 2013.

[77] 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.

[78] 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.

[79] 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.

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

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

[82] 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.

[83] 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.

[84] 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.

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

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

[87] 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.

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

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

[96] 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.

[97] 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.

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