©Copyright 2021

Shashank Vijaya Ranga

ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor

Shashank Vijaya Ranga

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

ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor

Shashank Vijaya Ranga

Chair of the Supervisory Committee:

Michael Taylor

Department of Computer Science and Engineering

Hardware accelerators are an active field of research in computer architecture as one solution to overcome hurdles such as dark silicon. A host processor interfaces with an accelerator and offloads the time-consuming computations onto it. BlackParrot is one such accelerator host being developed at the Bespoke Silicon Group. The initial performance comparisons of BlackParrot with other similar RISC-V processors are promising, and it is validated in silicon through a tapeout in 12 nm technology. BlackParrot is currently an industrial-strength ASIC Design, but ASIC flows are not readily available to individual users outside research groups that want to use BlackParrot in their designs. Open-source ASIC flows mitigate this to a certain extent, but taping out a chip is a costly exercise. An alternative to ASIC design is to use simulation only to validate the design, but this does not provide silicon validation and is an issue for long-running benchmarks which could run for days. In order to provide users with a silicon option for their designs using BlackParrot, the most viable option is an FPGA system. FPGAs allow rapid design iterations and provide an opportunity to prototype the system in real hardware. This thesis examines two tracks of building out BlackParrot's FPGA environment: i) by integrating BlackParrot into the OpenPiton memory system and validating it on an FPGA, and ii) by creating an in-house system using the Zynq-7000 SoC available on specific FPGAs and supporting hardware cosimulation for the BlackParrot design.

Contents

List of Figures

List of Tables

1 Introduction

Since the advent of the RISC-V [1] open-source instruction set architecture (ISA), there has been a steady increase in the number of processors [2–6] built using this ISA across academia and industry. At the same time, hardware accelerators are an active field of research since specialization is a viable solution to overcome the dark silicon problem [7], with accelerators designed for various applications [8–12]. The BlackParrot processor aims to be an accelerator host multicore system.

As shown in [2], the per-core performance of BlackParrot is very competitive compared to other similar RISC-V processors. BlackParrot is silicon-validated in 12 nm technology, and ASIC Design continues to be a significant focus. However, while these chips could be used within the research group, supporting only an ASIC flow does not attract many users because tapeouts are prohibitively expensive. Another option is to support a simulation-only setup for the validation of designs using BlackParrot. While simulation is the first step towards ensuring correct functionality, it is an issue for long-running benchmarks, some of which could run for days. These issues require further exploration into other alternatives for extending the community reach of BlackParrot, and Field Programmable Gate Arrays (FPGAs) are an attractive option due to their reprogrammability, capacity for fast design iterations, and reasonable costs.

This thesis describes these two tracks, encompassing my major contributions to the research group, namely:

- Integration of BlackParrot into OpenPiton (ParrotPiton) that involves designing a transducer between the two systems and bringing up Linux on the integrated system. This track helps BlackParrot to update its integration infrastructure and undergo testing with another system, thereby uncovering any issues previously not seen.

- Creation of a setup for BlackParrot with hardware cosimulation on boards containing the Zynq-7000 SoC (ZynqParrot), allowing users to simulate the design and run it on the board using the same testbench written in C++.

2 Background

This section describes the major components used in this thesis.

2.1 BlackParrot

BlackParrot [2, 13] is an open-source, Linux-capable, 64-bit RISC-V multicore processor. The principles behind the core design are Be Tiny, Be Modular, and Be Friendly. Area minimization is one of the main focuses of this core. A tiny core is beneficial in ASIC design and low-cost FPGAs since it fits in a smaller area, leaving more room for custom logic and accelerators. The second principle places emphasis on modularity and latency insensitive interfaces for connecting different modules. Using latency insensitive interfaces allows the designer to not rely on specific timing handshakes, avoid potential bugs and reduce the verification scope. Being friendly and open-source offers external contributors an opportunity to help develop the system.

The BlackParrot core is an 8-stage in-order pipeline, shown in Figure 1 (reproduced from original work [2]). There are three parts to the design: Front End (FE), Back End (BE), and Memory End (ME).

The FE is responsible for PC generation and instruction access through the Instruction Cache (I-Cache). All architectural state is speculative here and commands from the BE control PC generation (apart from sequential fetching). The FE is also responsible for performing branch prediction.

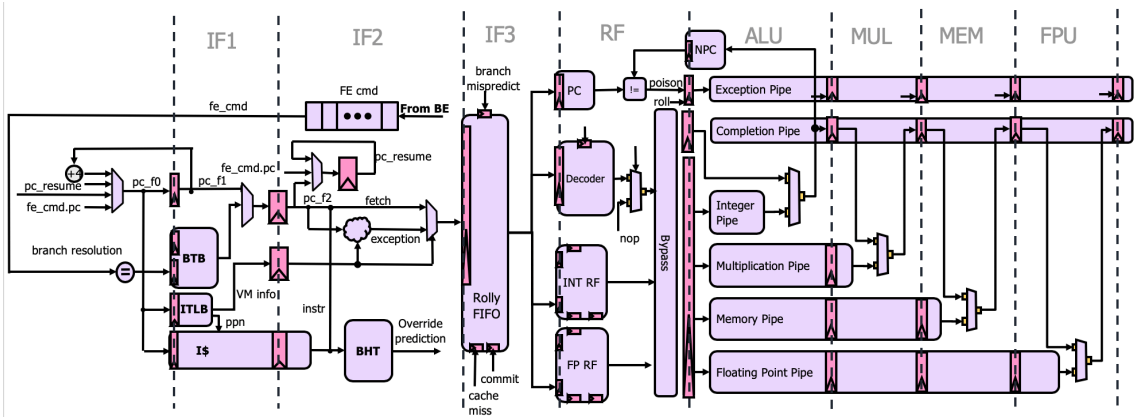

Figure 1: BlackParrot Pipeline [2]

The BE contains modules that execute the different RISC-V instructions, structured as multiple functional units. It includes the integer ALU, multiplication unit, floating-point unit, a system unit containing the different Contol and Status Registers (CSRs), and the memory unit containing the Data Cache (D-Cache).

The FE communicates instructions to the BE using an issue queue, and the BE issues commands such as PC redirections to the FE using a command queue. Both these interfaces are latency insensitive.

The ME contains the cache coherence logic for multicore BlackParrot and a lightweight state machine handling L1 requests for the single-core configuration. The BedRock Programmable Microcode Cache Coherence Engine (CCE) maintains cache coherence by implementing a directory-based, race-free design, and supporting the MOESIF family of protocols. The ME can optionally connect to a distributed and shared L2 Cache.

The multicore BlackParrot configuration is a 2-D mesh connection of routers, each containing one core, one CCE and a slice of the L2 cache. Each CCE controls part of a globally partitioned address space. There are three networks connected between routers for the coherence operations called the Request, Command, and Response networks, as shown in Figure 2 (reproduced from the BlackParrot 2020 RISC-V Summit Slides available at [13]).

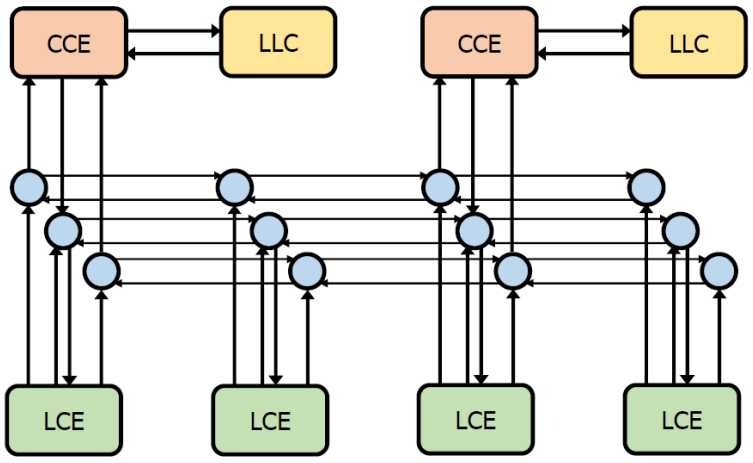

Figure 2: Coherence Networks [13]

Figure 3 (reproduced from the BlackParrot 2020 RISC-V Summit Slides available at [13]) shows a single BlackParrot tile with the coherence router. The L1 cache design is split such that the datapath and SRAM control signals are in a single module (the cache itself), and the control path consisting of the L2/DRAM request and response logic is in another module called the Cache Engine (CE). The figure shows a Local Cache Engine (LCE) which is used in conjunction with the CCE. The LCE requests a given cacheline through the request network, and the request gets routed to the CCE handling this cacheline. The CCE sends a command to any of the controllers based on the cacheline status in the directory. Once an LCE receives a command, it performs the necessary operations and responds through the response network. The complex cache coherence logic is contained within the CCE, whereas the LCE acts as a transducer between the cache and the CCE. A concentrator is responsible for arbitrating between the two LCEs and the CCE. The Local Cache Engine is explained in detail in Section 3.3.

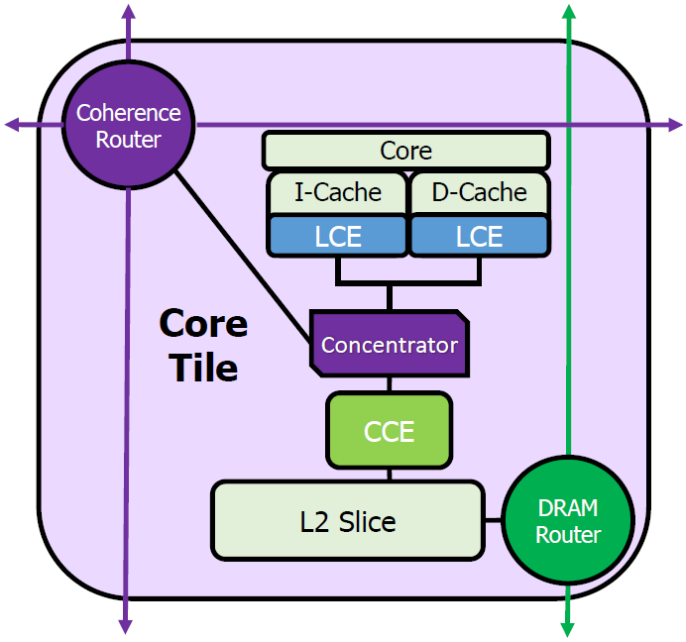

Figure 3: BlackParrot Tile [13]

2.2 OpenPiton

OpenPiton [14, 15] is an open-source multithreaded manycore processor. It is a tiled design that uses the OpenSPARC T1 core with custom uncore components (such as the caches, cache coherence protocol, 2-D mesh Network-on-Chip NoC). The 2-D mesh design and Coherence Domain Restriction [16] allow OpenPiton to scale up to 500 million cores. The OpenPiton design is shown in Figure 4 (reproduced from original work [14]).

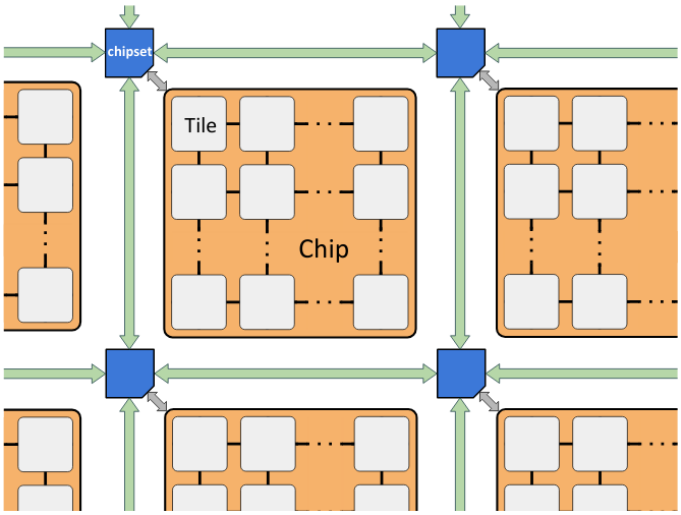

Figure 4: OpenPiton Architecture [14]

Multiple units, each containing a 2-D mesh of tiles, connect to chipset logic through a network. The chipset contains off-chip components such as DDR memory, Ethernet, SD Card, and UART. Each tile contains an OpenSPARC T1 core, an L1.5 Cache, an L2 Cache, a Floating Point Unit (FPU), arbiters, and routers. The L1.5 cache is a write-back cache between the core's L1 cache and the shared L2 cache of OpenPiton Network. There were two reasons for the development of this cache. Firstly, the OpenSPARC core utilized a write-through L1 cache tightly integrated with its pipeline. Write-through caches are a source of bottlenecks in distributed cache frameworks since they have a very high bandwidth requirement to the next level in the cache hierarchy. Secondly, the cache requests and communication had to be compliant with the OpenPiton cache coherence network interface. Rather than modify the L1 cache in the core to overcome the above two problems, the OpenPiton developers decided to include a private L1.5 cache for each core. This write-back

cache avoided overwhelming the memory system with messages and allowed OpenPiton to connect the core easily with the cache coherence network.

The L2 cache is a distributed cache shared across all tiles and is inclusive of both private caches (L1 and L1.5). The L2 is the point of coherence in the memory system, and coherence is maintained using a directory-based MESI protocol.

2.3 Bring Your Own Core (BYOC)

BYOC [17, 18] is an extension of the OpenPiton framework, which allows different cores to connect to the OpenPiton memory system. This enables heterogeneous ISA research and provides a platform for direct comparison of cores using different ISAs. Figure 5 shows an example BYOC system (reproduced from original work [17]).

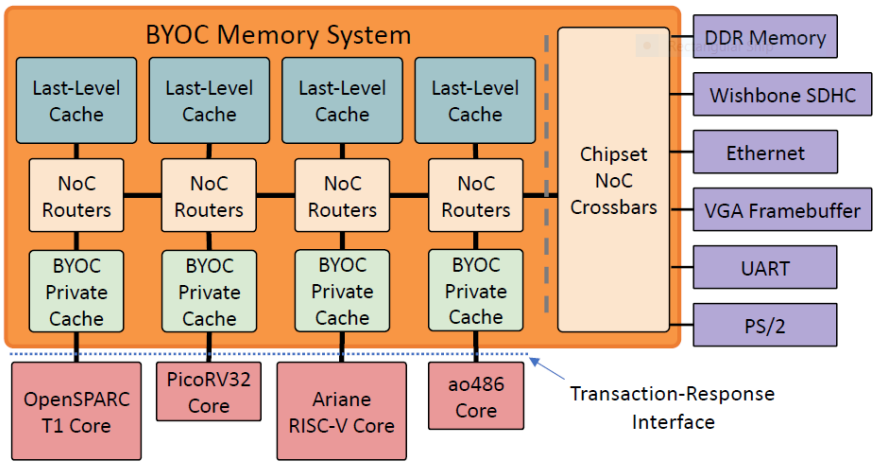

Figure 5: BYOC System Architecture [17]

The standard cache coherence system provided to all the connected cores is called P-Mesh. The memory system contains the BYOC Private Cache (BPC), three NoC networks, and a portion of the distributed LLC. The chipset allows connections to peripherals like DDR Memory, Ethernet, UART, SD card, and accelerators.

Since this system should cater to different cores connecting to it, one question is where to create the interface point. With the assumption that the BYOC framework provides a distributed LLC,

the developers compare three different interface points. The first option is to provide the entire cache hierarchy, including the L1 cache, to the cores. This option is impractical since the L1 cache provided would have to account for all the variants, such as virtual or physical tagging and indexing. Further, since L1 caches are usually tightly integrated with the core, requiring the core to interface with another (general) cache could have performance implications. The second option is to provide only the LLC to the cores. This option places a burden on the user connecting a core to the system by requiring them to understand many details of the coherence protocol and incorporate them in their private caches. The third option is to provide a private cache, local to the framework, and expose an interface to the user. In this case, the connecting core does not need to know details about the coherence protocol, and the framework does not need to (itself) support different types of caches. The developers chose to implement this option for its benefits over the other two. Figure 6 illustrates the three options (reproduced from original work [17]).

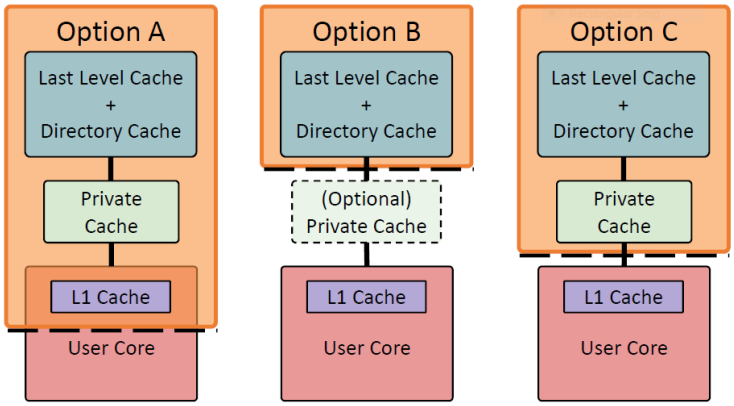

Figure 6: Options for interfacing the memory system with the core [17]

The Transaction Response Interface (TRI) provides a uniform memory system interface to all the cores. The different cores need to design a transducer that converts the core-specific signals to those understood by the interface to connect to the system. However, the core must have a write-through data cache, resulting from the choice to implement a simple TRI that avoids the extra write-back operation upon invalidation of a dirty line in the core's L1 cache. The L1 cache thus becomes transparent to the BPC.

The BPC is a write-back cache to avoid excessive amounts of data flow to the next level. The BPC stores only data from the L1 D-Cache, with the L1 I-Cache requests bypassed to the next level. It has to be inclusive of the D-Cache so that all lines in L1 are accessible for invalidation. The BPC tracks the L1 way information using a Way Map Table (WMT). The WMT allows flexibility in the BPC eviction policy while sending invalidations to the correct way in the L1 cache. The BPC currently uses a fixed instruction cacheline size of 32 bytes and a data cacheline size of 16 bytes to support the OpenSparc T1 core. Configurable cacheline sizes for the BPC are a work in progress.

3 ParrotPiton

ParrotPiton is the integration of the BlackParrot core into the BYOC framework. The motivations for this integration were three-fold and mutually beneficial to the two systems. Firstly, this would extend BlackParrot's community reach. The OpenPiton system has a larger community accumulated over the years, which would allow more people to discover the BlackParrot core and use it. Secondly, it would add a validated core with best-in-class performance and energy efficiency to the BYOC framework. Thirdly, we could leverage the BYOC framework to compare BlackParrot's performance with other cores. This comparison could be with other RISC-V cores or with cores of other ISAs.

Figure 7: ParrotPiton logo

Some of the TRI specifications for the connecting cores are as follows:

- L1 D-Cache (if any) should have a 16-byte cacheline width and 4-way set associativity and should be smaller or equal in size to the BPC
- L1 I-Cache (if any) should have a 32-byte cacheline width and 4-way set associativity
- L1 D-Cache should be write-through
- The core should handle remote invalidations and interrupts and support L2 atomics

Remote invalidations and L2 atomics are explained in Section 3.4. Keeping these requirements in mind, we take a look at the BlackParrot L1 cache organization.

3.1 BlackParrot L1 Caches

This section describes the BlackParrot L1 caches as of this commit (https://github.com/black-parrot/black-parrot/tree/414747b05833f429c418af6502f13a327ef4df67). Section 3.2 describes the original caches at the start of the ParrotPiton integration (at the commit https://github.com/black-parrot/black-parrot/tree/900ce4ff5fef907eaa3a5506503513814e810b91) and the changes made to allow the integration.

The I-Cache (I$) and D-Cache (D$) have similar pipelines and basic design philosophies. Both caches are virtually indexed, physically tagged (VIPT) set-associative caches. The set associativity is configurable as 1, 2, 4, or 8, and the cacheline block width is configurable as 8, 16, 32, and 64 bytes. Both caches contain support for iterative fills and evictions in the single-core configuration with a constraint on fill width between double word width (64 bits) and the cacheline width.

There are three SRAMs in each cache: one each for storing the i) data, ii) tag and coherence state, and iii) Least Recently Used (LRU) and dirty information. The data memory is banked with as many banks as the associativity, with the data width within each bank not being less than 64-bits. An index in each bank stores the corresponding data in a single way. Writes to the banks are interleaved as given in the following equation (word offset denotes the specific 64-bit segment in the cacheline block):

$$\text{Bank ID} = \text{Word offset} + \text{Way ID (modular arithmetic)}$$

This design speeds up a load and a cacheline write. A read operation of the same index in each bank provides the same 64-bit data in the cachelines stored in each way. The tag matching is straightforward at this point and points to the correct data (if available). A write operation in the interleaved manner allows a cacheline write to complete in one cycle. Figure 8 illustrates this design for a 4-way data memory.

The tag memory stores the physical tag and coherence state of the data in each way of an index. The status memory stores the LRU and dirty bit information for each index. The caches are fetch-on-write and write-allocate caches, meaning that the store misses fetch the data from the next level

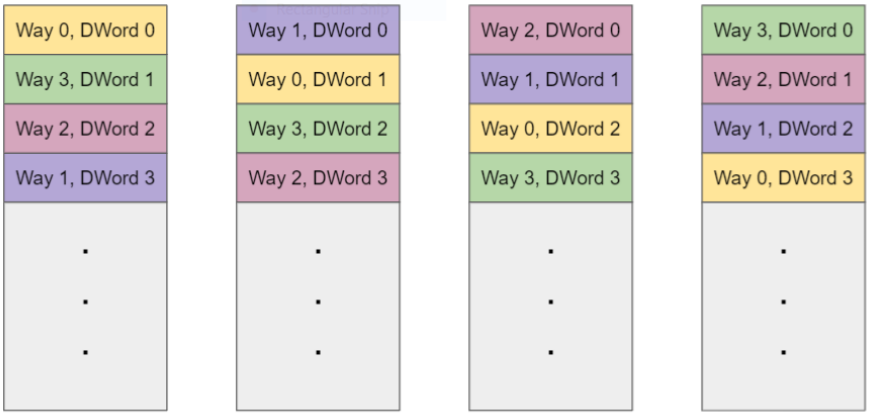

Figure 8: Data memory organization

before the actual write occurs. BlackParrot handles cache misses by replaying the operation once the cache indicates that the requested data has been filled.

There are some differences between the I$ and D$, which are explained briefly below.

I-Cache

The I$ has a two-stage pipeline. Before the first stage, the address information is latched on the positive edge of the clock. The first stage is called Tag Lookup (TL). In this stage, the tag memory and all banks of the data memory for the specific index are read. By reading the data memory in parallel with the tag memory, we can save a cycle in the event of a hit. The tags read across all the ways for that element are compared with the physical tag from the Translation Lookaside Buffer (TLB). This information is latched on the positive edge of the clock. The second stage is called Tag Verify (TV). In this stage, the data in the tag match way is muxed out and sent to the core pipeline. If a miss is detected, a request is sent to the Cache Engine (CE). The CE functionality is explained in Section 3.3.

D-Cache

The D$ has a three-stage pipeline. Before the first stage, the address and opcode information is latched on the clock's negative edge. The first stage is TL, the same as the I$. However, the information is latched on the negative edge of the clock. The second stage is TV, the same as the I$. The data for a 32-bit or 64-bit load hit is sent to the core pipeline as "early_data" and latched

on the next positive edge. For sub-word operations and floating-point loads, there is a third stage called Data Mux (DM). This stage contains a multiplexer that picks out the correct byte(s) for integer loads and has a recoding unit for floating-point loads.

The D$ uses the negative edge of the clock for the TL and TV stages so that a 32-bit or 64-bit load can complete within two cycles of the request rather than three. The initial half cycle is sufficient time for the index calculation before reading the memories in the TL stage. The final half cycle is sufficient time for the hit data to be latched on the next positive edge since it is a smaller multiplexer only picking between 32-bit or 64-bit data, rather than including 8-bit and 16-bit data.

The D$ has a write buffer that holds the write data from the TV stage until the data memory is free from incoming loads. This buffer supports store forwarding to prevent data hazards. The D$ also supports executing atomic operations at the L1 level. There are three types of atomic operations: Load Reserved/Store Conditional (LR/SC), Fetch-and-Logic, and Fetch-and-Arithmetic Operations. A reservation register in the cache stores the requested address and its validity after the LR operation. This register provides the valid reserved address for the SC operation if the coherence system did not clear it. In order to aid SC progress, a backoff mechanism blocks any invalidations for a specified number of cycles to allow the SC operation to complete following the reservation. A small ALU inside the cache executes the other atomic operations.

3.2 Flexibility in Cache and CE Design

Originally, BlackParrot I$ and D$ were fixed at 32 KB size, with a 64-byte cacheline width and 8-way set associativity. Further, the LCE was tightly coupled to the CCE and was the only CE option available.

There were a few problems with this setup, both for single-core and integration purposes. As shown earlier, a BlackParrot core, a CCE, and the coherence router existed within a tile. A single core BlackParrot configuration was still constrained to use the LCE and CCE with the coherence routers connected in a loopback fashion such that all the messages would return to the same tile. This setup added significant hardware and performance overhead.

3.2.1 Motivation for single core optimization

An initial experiment dealt with making BlackParrot optimized for single-core operation. The CCE essentially became an FSM that forwarded memory requests from the LCE to the next level by removing the directory and the routers. This change resulted in an area reduction of about 29%, indicating that optimizing for a single core would be a beneficial task. However, this design still used the LCE before the FSM, which added an unnecessary step between the two levels of memory. In order to avoid the LCEs altogether and design an FSM that delivers cache requests to the next level, supplies the response data back to the cache, and handles the write-back cases, the cache and the CE had to be decoupled with a standard interface that allowed connecting different CEs for different configurations.

3.2.2 Allowing plug-and-play operation

The original interface contained latency insensitive request and response paths. The cache misses would communicate misses to the CE through the request path, and the controller would request the next level for the data, which was then supplied back to the cache through the response path. The original request interface contained individual signals indicating load, store, miss, uncached operation, among others, connected from the cache to the CE. This interface was unwieldy to use since any additional functionality (such as write-through or L2 atomic support) would add more signals to the interface.

A second issue was that the cache miss tracking and cacheline locking logic were present in the LCE. Cache miss tracking allows the cache to indicate when it is ready to service requests from the core. It was not ideal to place this logic in the LCE since every CE would have to implement the miss tracking and communicate it to the cache.

Cacheline locking is used on an LR hit, giving the core some additional cycles to complete the following SC operation. By locking a particular cacheline, any invalidation command to that line would be stalled until a specified number of cycles, thereby avoiding a scenario where the cacheline bounces from core to core and is evicted before the SC succeeds. This logic is only relevant if the

L1 cache executes LR/SC instructions. Again, it is not ideal for this logic to be present in the LCE since every other CE would need to implement the locking logic.

These issues and a need for a standard interface across different configurations prompted some changes in the request interface as described below.

The request interface was converted into two structures for the cache request and its metadata, containing the information as illustrated in Tables 1 and 2.

Field	Description
Message type	Indicates the type of request.
Hit	Indicates whether this address was a hit. Used for uncached requests
Data	Store data Used for uncached and writethrough stores.
Size	Indicates the size of the request. Can be between 1 byte and 64 bytes.
Address	Address of the request
Subop	Indicates the sub operation. Used for L2 atomics

Table 1: Cache request structure

Field	Description
Hit or replacement way	Indicates the way that was hit or the way to be replaced
Dirty	Indicates whether the way was dirty

Table 2: Cache request metadata structure

Uncached requests are for addresses that are not cached, as indicated by the system address map. In the BlackParrot address space, all addresses outside the DRAM address range are uncached, meaning that any data loaded from or written to these addresses do not enter the cache data memory. However, the requests still move through the cache request datapath. L2 atomics are atomic instructions executed in an ALU at the L2 level.

The miss tracking logic and locking logic was moved to the cache module so that the CE would only be responsible for

- sending the request to the next level

- taking the correct action upon receipt of a response (such as a write to the data memory, invalidation, write-back of dirty data)

The response path contained three packets talking to the data memory, tag memory, and status memory.

- The data memory packet contained the index, the way ID, data for a cacheline fill, and the opcode indicating a read, write, or uncached operation.
- The tag memory packet contained the index, way ID, tag, and the opcode indicating a set, invalidate, or clear operation.
- The status memory packet contained the index, way ID, and the opcode indicating a read, clear, or dirty bit clear operation.

All the three response paths also had separate buses communicating their corresponding data to the CE. These paths were not altered during the interface changes.

These changes set the stage for a plug-and-play type of design since any CE could connect to the BlackParrot cache, and any cache could be connected to the BlackParrot CE as long as they met the interface specifications. It was also an essential milestone in the integration process since we now had the tools to design a transducer that bridges BlackParrot and OpenPiton. The logic required for meeting the other requirements of the TRI is described next.

3.2.3 Multiple widths and associativity combinations

An ideal cache supports parameterizable cacheline width and associativity. Different configurations could then be achieved without manually changing the RTL for each one, thereby improving the adaptability of the cache to different situations. For example, BlackParrot in an ASIC Design could probably afford to use the largest cache size along with the largest width/associativity combination. However, in a different setting, such as a smaller, low-cost FPGA, a smaller cache would be a better option to allow more logic utilization for the accelerators connected to BlackParrot. In addition,

BYOC integration required specific cacheline widths and associativities that BlackParrot did not support at that stage. The solution was to add parameterization to the existing cache design with minimal RTL changes. Version 1 added support for the widths and associativities as shown in Table 3. Version 2 added support for a direct-mapped cache with an 8-byte cacheline width.

Configuration	Cacheline Width (in bytes)	Associativity	Cache Size
Baseline	64	8	32 KB
Medium	32	4	16 KB
Small	16	2	8 KB

Table 3: Width and associativity combinations

The number of combinations was constrained because a bank in the data memory of the cache had to be at least 8 bytes wide. This allowed a double word (64-bit data) to fit in a single bank, thereby avoiding a multi-index read or write for a double word instruction.

3.2.4 Write-through option enabled in the same cache

Writethrough caches are relatively uncommon in shared-memory multicore configurations since they add significant overhead to memory bandwidth. Each store to such a cache would trigger a write to the next level. Since the write-through use case was not required previously, BlackParrot did not support it. However, as explained earlier, BYOC requires a write-through L1 cache so that all the cache operations are transparent to the BPC. Therefore, the next step was to add this capability to the D$ so that the requirement could be satisfied with minimal RTL changes and an added message type in the cache request structure.

A design decision was the choice between using the fetch-on-write, write-allocate policy (loading the cacheline from memory on a store miss and allocating the address written to by the store in the cache) and adding a no-fetch-on-write policy for the write-through use case. In order to reduce the traffic to the next level, a no-fetch-on-write policy seems better, but a deeper analysis is required to understand if it is a change worth making in the context of systems that use the BlackParrot write-through cache.

There are three different policies within no-fetch-on-write: write-validate, write-around, and

write-invalidate [19]. Write-validate implies that the data write into the corresponding cacheline occurs with only the corresponding write valid bits turned on. By doing this, we are achieving no-fetch-on-write with write allocate functionality. However, there are a few issues with this design in the BlackParrot setting. The main issue would be the requirement of sub cacheline valid (or coherence) bits in the tag memory, which would add significant area overhead, going against BlackParrot's Be Tiny motto. Theoretically, byte store misses could use fetch on write, and word/double word store misses could use the write validate policy, reducing the number of extra bits required per cacheline. However, this would still be an invasive change in the cache design for an uncommon use case, so it does not seem like a viable option.

Write-invalidate implies that the data memory write occurs in the same cycle as the tag memory read, and the cacheline is invalidated if the tag did not match. This policy is helpful in direct-mapped write-through caches because a tag mismatch implies that the cacheline contains data from two different addresses, so the invalidation removes the potential of the core loading incorrect data. Further, the old cacheline data is already consistent with the lower levels in the memory hierarchy, and the new data is written to the next level (since the cache is write-through), resulting in consistent memory values. However, this policy is not suited to BlackParrot because of two reasons. Firstly, it provides a benefit only in a specific use case (direct mapped write-through cache), and it would not work for any higher associativity (since we do not know in which way to write the new data). Secondly, the BlackParrot cache reads the data memory and tag memory in the same cycle since it is a VIPT cache. Supporting a data memory write in conjunction with the tag memory read would require an invasive change in the cache design, which seems unnecessary to support a specific use case.

Write-around implies that the store data on a miss bypasses the L1 cache completely and is written to the next level. This option would be the easiest to integrate into the BlackParrot cache because it would be a matter of parameterizing the store miss logic and modifying the request message type accordingly. However, for the BYOC integration, we decided to continue with the fetch-on-write policy to get the system running initially. A future experiment could involve check-

ing if the write-around policy offers a performance benefit for ParrotPiton.

A cache testbench tested the write-through functionality with directed tests that stressed the D$ pipeline in different ways after the cache accepted a write-through store. An appendix at the end of this thesis explains the cache testbench.

3.3 Cache Engines for BlackParrot

This subsection describes the two cache engines used in BlackParrot, the single-core optimized Unified Cache Engine (UCE) and the Local Cache Engine (LCE) used in the multicore design. These cache engines can be used in the system, as shown in Figure 9. A description of the UCE and LCE follows.

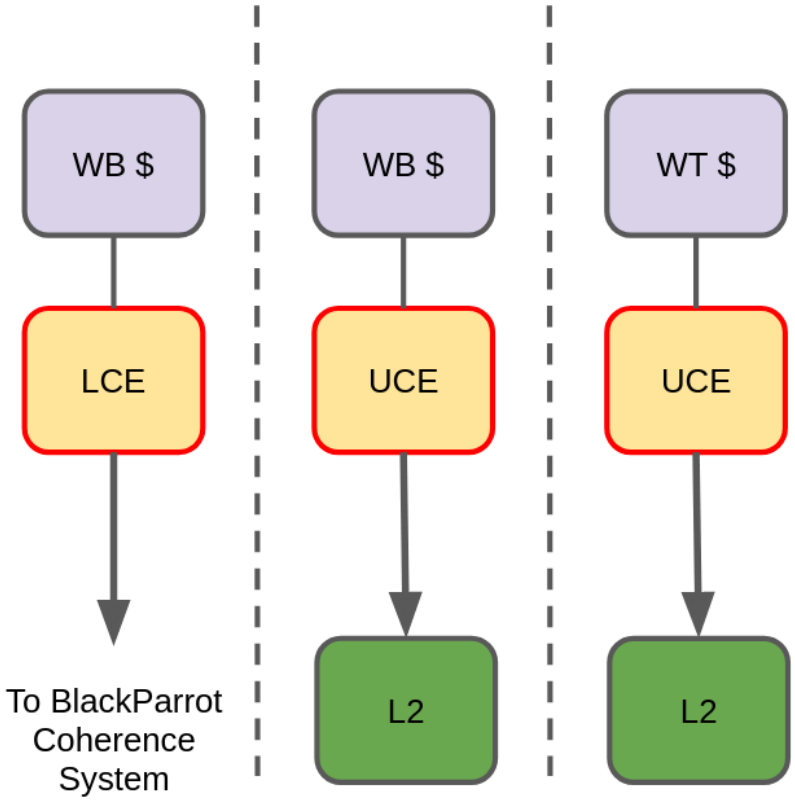

Figure 9: Different BlackParrot configurations

3.3.1 Unified Cache Engine (UCE)

The UCE is one of the cache controllers used in BlackParrot. Optimized for single-core operation, this controller is responsible for sending the requests from the I$ and D$ to the next level and supplying back the responses. The UCE FSM implements the following states (as shown in Figure 10):

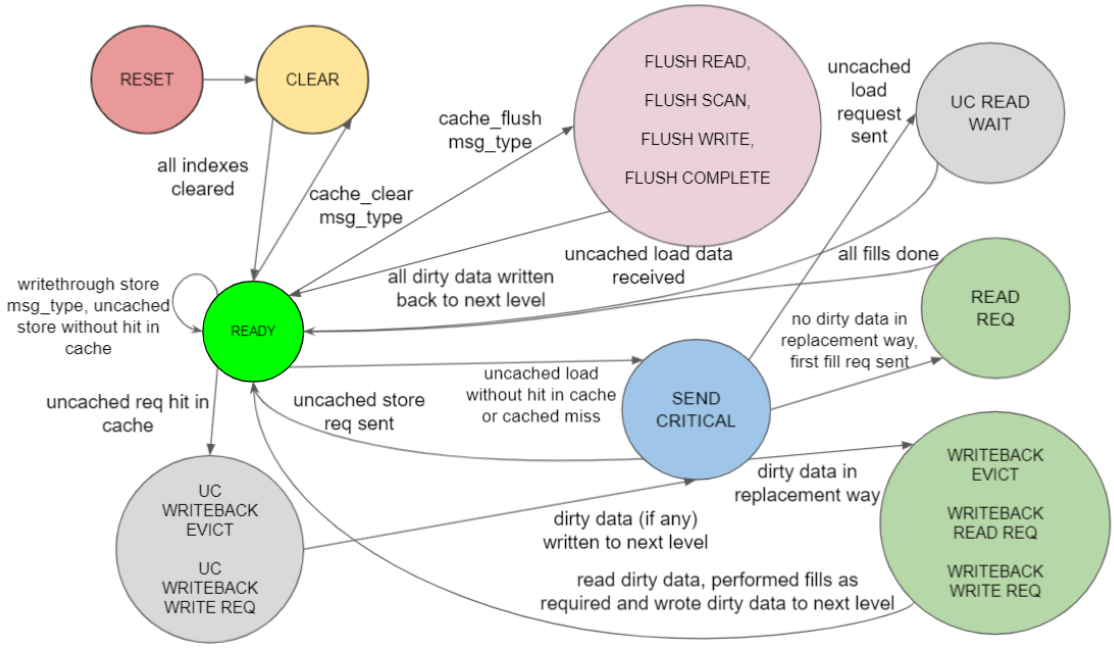

Figure 10: UCE FSM functionality

- **reset:** The FSM starts in this state during reset. Upon exiting, it starts clearing the cache.
- **clear:** This state clears the cache tag and status memory by iterating over all the indexes.
- **ready:** This state receives the request from the cache and decides the action required based on the type of request. If the request was an uncached store for an address not present in the cache or a write-through store, the data is sent to the next level, and the UCE is ready to accept the subsequent request in the next cycle since the cache is not expecting a response. When the response does arrive, a silent acknowledgment occurs. For any other request, different states are triggered, as explained below.

- **flush_read, flush_scan, flush_write, flush_fence:** These states are triggered when the controller receives a cache flush request. This request indicates that the cache has received a fence command, which requires the L1 and its next level (L2/DRAM) to contain consistent data. The controller proceeds to **read** each index of the cache successively, **scan** the dirty bit field to identify which way(s) contain(s) dirty data and obtain this data from the cache, **write** it to the next level, and indicate to the cache once the scanning of all ways and indexes is **complete**.

- **uc_writeback_evict, uc_writeback_write:** These states are triggered when an uncached request (load, store, or L2 atomic) is for an address that is present in the cache. The hit, hit way, and dirty fields mentioned in the structures above are utilized in these states. If the address was a hit in the cache, the data in the hit way is **evicted** (with a **writeback** if it was dirty).

- **send_critical:** This state is triggered if the request is a load or a store miss, an uncached load, or after an uncached request goes through the evict and writeback states. Here, the request is sent to the next level. For an uncached load or store request, the corresponding request is sent, and the FSM moves to wait for the data (load) or gets ready to accept the subsequent request (store). For a cached load or store miss, the request for a load from the critical address is sent first, employing the ”critical word first” optimization.

  The UCE contains support for a multi-cycle fill of data into the cache and multiple requests to the next level following the ”critical word first” policy. An address counter is initialized with the specific ”block” location requested within the cacheline, and the first request is sent in this state. The address is incremented appropriately for subsequent requests and wraps around to the start of the block if required.

- **writeback_evict, writeback_read_req, writeback_write_req:**

  These states are triggered if the first request was sent, but the cacheline was dirty. The cacheline is read, and the dirty bit is reset in the **evict state**, and the data from the cache is registered

for later use. The following addresses are requested, and the responses are sent to the cache in the **writeback_read** state. Once all the sub-blocks are filled, the dirty data previously registered is sent to the next level in the **writeback_write** state.

- **read_req:** This state is triggered if the first request was sent and the cacheline was not dirty. The following addresses are requested, and the responses are sent to the cache. Once all the sub-blocks are filled, the UCE becomes **ready** to accept requests again.

- **uc_read_wait:** This state is triggered after an uncached load request is sent to the next level. The response data is sent to the cache when available, and the UCE becomes **ready** to accept requests again.

3.3.2 Local Cache Engine (LCE)

The LCE is a cache controller used in BlackParrot's multicore configuration. It contains two modules, the LCE Request and the LCE Command. The LCE Request module is responsible for handling the request from the cache and sending the appropriate packet to the CCE. The LCE Command module is responsible for handling the commands from the CCE or other LCEs, taking the appropriate action on the cache, and sending back the responses to the originating module. The two state machines are described below.

LCE Request

Figure 11 shows the states in the LCE Request FSM. These perform the following operations:

- **reset:** The FSM starts in this state during reset. Upon exiting, it becomes **ready** to accept new requests from the cache.

- **ready:** This state receives the request from the cache and decides the action required based on the type of request. If the request was an uncached store, the data is sent to the CCE, and the UCE is ready to accept the subsequent request in the next cycle since the cache is not expecting a response. When the response does arrive, the credit counter (which tracks the

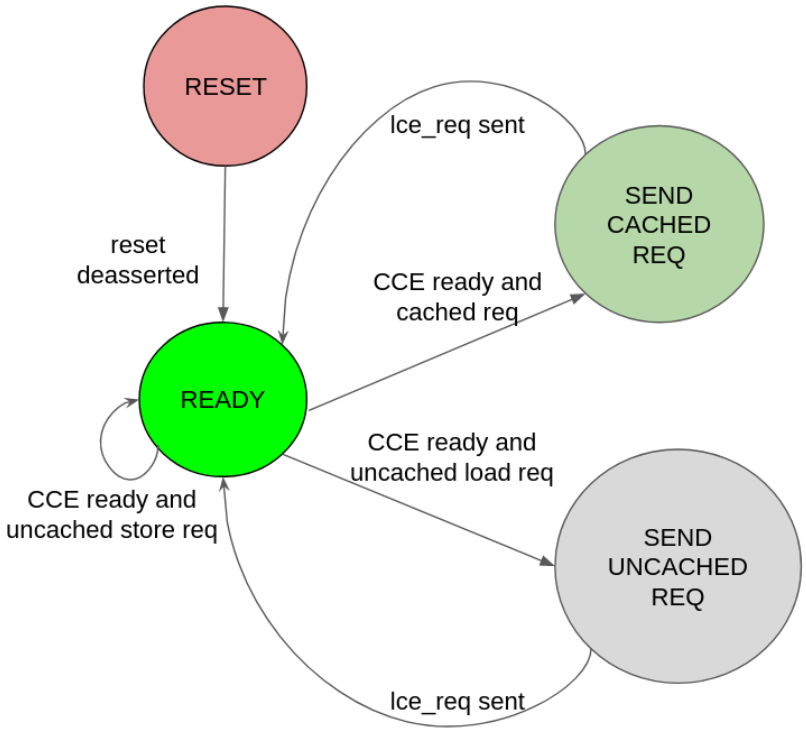

Figure 11: LCE Request FSM functionality

number of outstanding requests) is silently updated. For any other request, different states are triggered, as explained below.

- **cached_req:** This state is triggered when the controller receives a cached load or store miss. The controller proceeds to send the request in the LCE request packet format to the CCE and moves back to **ready** when it is accepted.
- **uncached_req:** This state is triggered when the controller receives an uncached load request. The controller proceeds to send the request in the LCE request packet format to the CCE and moves back to **ready** when it is accepted.

LCE Command

Figure 12 shows the states in the LCE Command FSM. These perform the following operations:

- **reset:** The FSM starts in this state during reset. Upon exiting, it moves to **clear** the tag and status memory in the cache.

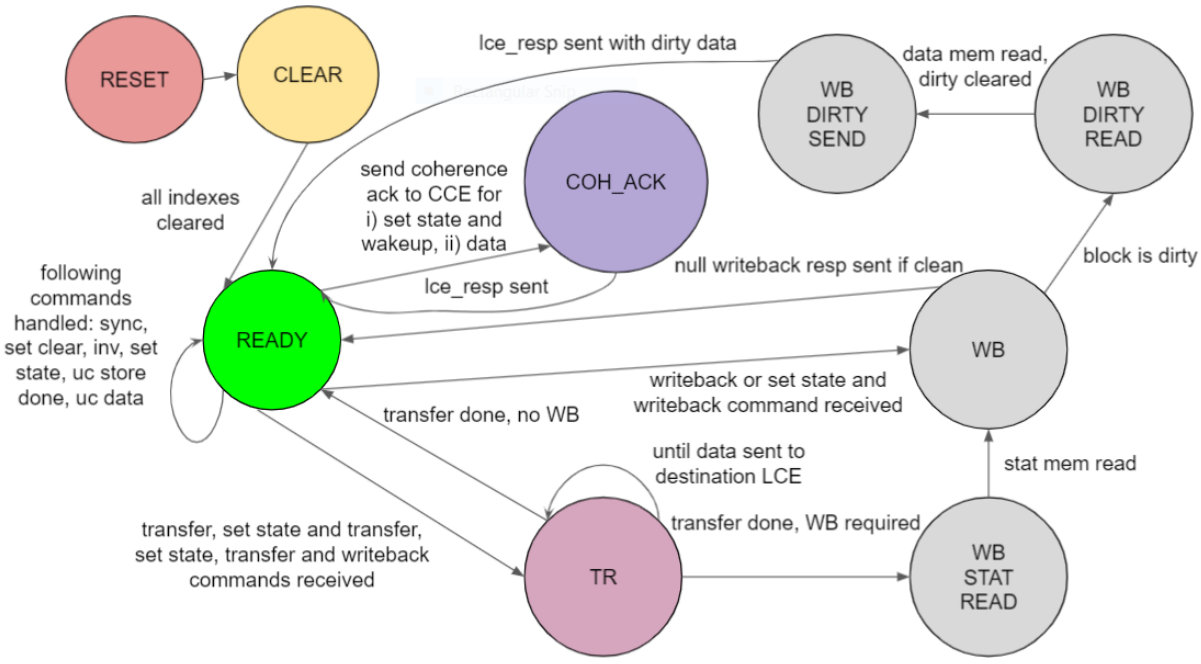

Figure 12: LCE Command FSM functionality

- **clear:** This state clears the tag and status memory for each index of the cache and moves to the **ready** state once all indexes are cleared.

- **ready:** This state handles the commands shown in the Figure upon receipt as follows and remains in the same state:

  – *sync:* Sends back a sync acknowledgment to the CCE

  – *set_clear:* Clears the tag and status memory in the given index

  – *inv:* Invalidates the tag memory at a given index and way by setting the coherence state as invalid and sends back an invalidate acknowledgment once it is completed.

  – *set_state:* Writes the given coherence state to the tag memory at the given index and way.

  – *uc_store_done:* Dequeues the command and indicates to the LCE Request module that the uncached store operation is done

  – *uc_cmd_data:* Sends the uncached data to the cache and indicates to the LCE Request

module that the uncached load operation is complete.

This state also handles the following commands and moves to the corresponding state for handling the response:

  - *st_and_wakeup:* Sends the coherence state to the tag memory and moves to the **coh_ack** state.
  - *tr:* Reads the data memory at the given index and way and moves to the **tr** state.
  - *st_tr, st_tr_wb:* Reads the data memory, sets the given coherence state in the tag memory and clears the dirty bit in the status memory at the given index if the command is not write-back. Once all the operations are done, moves to the **tr** state.
  - *wb, st_wb:* Reads the status memory at the given index and way to determine if the block is dirty. Additionally, if the command includes *st*, it sets the coherence state in the tag memory. Once all operations are done, moves to the **wb** state.

- **coh_ack:** This state is triggered when a coherence acknowledgment is to be sent over the response network. Once the CCE receives the response, the FSM is **ready** to accept commands again.

- **tr:** The data is transferred to the target LCE, and the FSM is **ready** to accept commands if no write-back was required. In the event of a write-back, it moves to the **wb_stat_rd** state.

- **wb_stat_rd:** This state reads the status memory to determine if the line at the given index and way is dirty. It moves to the **wb** state when the data from the status memory arrives.

- **wb:** This state checks if the line is dirty and determines the next steps. If the line is not dirty, the FSM is **ready** to accept new commands from the next cycle. If the line is dirty, it moves to **wb_dirty_rd** state.

- **wb_dirty_rd:** This state reads the data memory at the given index and way to write back the data to the next level. It also clears the dirty bit in the corresponding status memory index.

Once both operations are complete, it moves to the **wb_dirty_send** state.

- **wb_dirty_send:** This state sends the write-back response to the CCE and is **ready** to accept new commands from the cycle after the response is received.

The LCE Request and Command modules connect to the CCE, which handles the cache coherence operations using a directory-based approach. The BlackParrot CCE can either be used as a programmable microcode engine or an FSM that handles the coherence operations. The internal details of the CCE are outside the scope of this thesis.

3.4 Setting up for integration

The previous sections described the prior work required to start the process of integrating BlackParrot into OpenPiton. At this stage, there were two design choices to be made before the actual integration.

3.4.1 Way mapping the BlackParrot D$ with the BPC

The BlackParrot I$ could directly connect to the TRI since the required configuration was available. However, for the D$, there were two options - 8 KB size, 16-byte cacheline width, and 2-way set associativity, or 16 KB size, 32-byte cacheline width, and 4-way set associativity. While the set associativity matches in the second option, the cacheline width is larger, which would require multiple fills for every request. Further, the cache is twice the required size, resulting in half the cache being unused to maintain inclusivity with the BPC. The first option offered the same size and cacheline width but smaller set associativity. The way mapping was handled as follows.

The 8 KB BlackParrot D$ contains 256 indexes, whereas the 8 KB BPC contains 128 indexes. This implies that two ways of the BPC would fit into one way of the D$. Figure 13 illustrates this design. The D$ is at the top with an 8-bit index and 4-bit block offset, and the BPC is at the bottom with a 7-bit index and 4-bit block offset. The numbers indicate which way of BPC maps to

which section of the D$. A description of the translation between the BlackParrot way ID and the OpenPiton way ID follows.

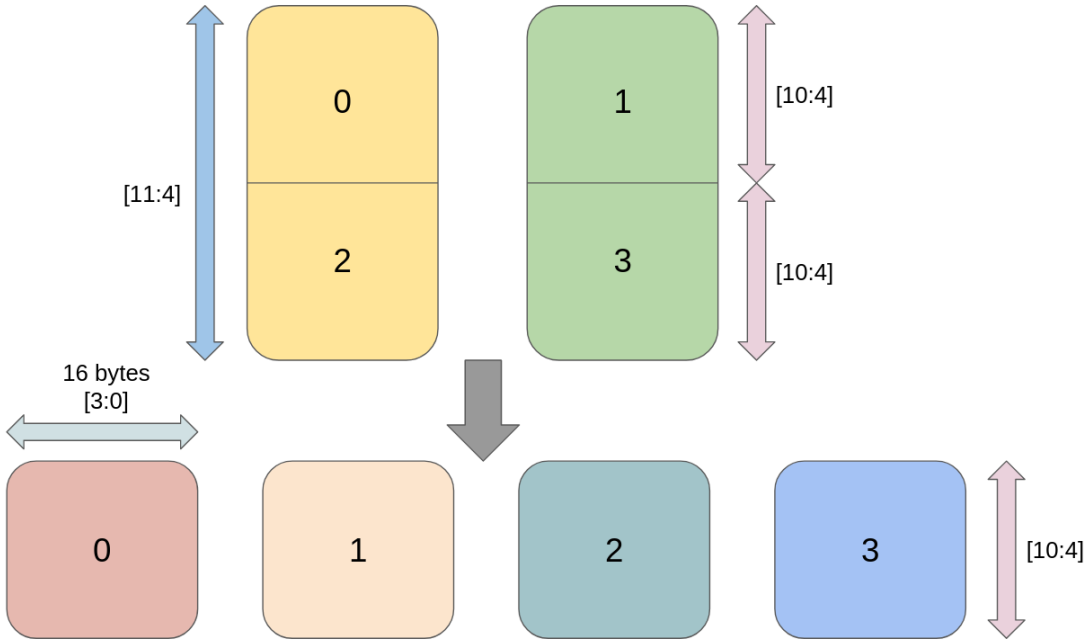

Figure 13: Way Mapping between D$ and BPC

Requests

The address and replacement way ID sent to the BPC are given below:

Address - cache_req.addr

Way ID - {cache_req.addr[11], cache_req_metadata.hit_or_repl_way}

where *cache_req* and *cache_req_metadata* are the request structures of the BlackParrot caches described earlier. By using the top bit of the cache index, each way of the L1 D$ is divided into two ways in OpenPiton. This conversion allows the BPC to be agnostic about the actual orientation of the ways in the L1 cache.

Remote invalidations

The BPC sends a 12-bit index and a 2-bit way ID to the connected core. As the name suggests, the address sent by BPC is bits 15 to 4 of the actual address. A reverse conversion of the above is required to send the invalidation command to the correct cache index and the way. The index and way ID sent to the L1 SRAMs are given below:

Index - {l15_transducer_inval_way[1], l15_transducer_inval_address_15_4[10:4]}

Way ID - l15_transducer_inval_way[0]

3.4.2 L2 Atomics

OpenPiton required LR/SC to execute at the BPC level and other atomics to execute at the L2 level. Table 4 shows the RISC-V atomic instructions. All instructions are of two types - word (32-bit) and doubleword (64-bit).

Atomic type	Operations included
LR/SC	LR
	SC
Swap	swap
Logic	and
	or
	xor
Arithmetic	add
	min
	max
	minu
	maxu

Table 4: RISC-V Atomic instructions

At the time of this integration, the BlackParrot core did not support the swap, logic, and arithmetic atomic operations in hardware. These instructions were instead emulated in software. Support was added for these instructions in the instruction decoder, and the requests were bypassed straight through the D$ to the CE. Further, the LR/SC hardware in the D$ was disabled to allow execution at the BPC level.

No return L2 atomics are optimizations when the atomic operation's destination register is the zero register. This allows the core to send the request and move on to the next instruction since it does not require a response.

3.5 P-Mesh Cache Engine (PCE)

The PCE design includes the above design decisions and is the main integration module. It contains an FSM that handles the different cache requests and BYOC-specific commands such as remote invalidations. It is important to note that P-Mesh is big-endian, whereas BlackParrot is little-endian. This requires a conversion in any data sent from BlackParrot to the BPC and vice versa. Further, the PCE logic performs different operations in the same module if the PCE connects to the I$ or the D$.

The PCE FSM is illustrated in Figure 14 and performs the following operations:

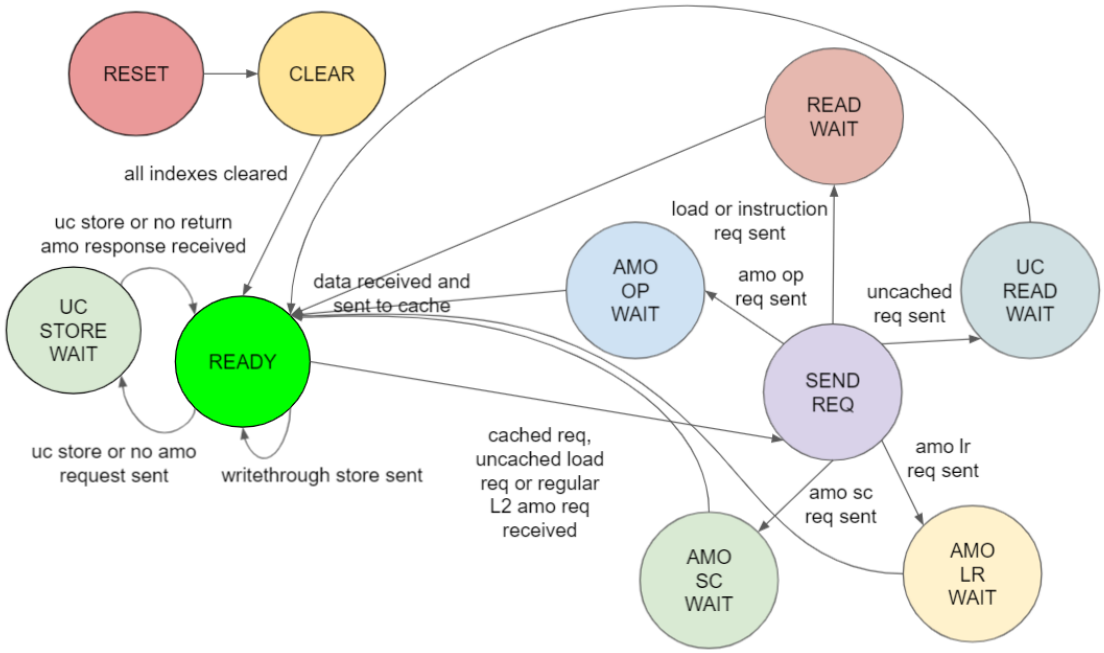

Figure 14: PCE FSM functionality

- **reset:** The FSM starts in this state during reset. Unlike the BlackParrot CEs, an *int_ret* command from the OpenPiton memory system moves the FSM to the **clear** state. OpenPiton is programmed to send this command to each core connected to it.
- **clear:** The FSM clears the tag and status memory for all indexes before moving to the **ready** state.

- **ready:** This state receives the request from the cache and decides the action required based on the type of request.
  - For a write-through store, the FSM sends the data to the BPC, and the PCE is **ready** to accept a request in the next cycle since the cache is not expecting a response. When the response from the BPC does arrive, the module silently acknowledges it.
  - For uncached stores and no return L2 atomics, the FSM sends the corresponding BYOC request and moves to the **uc_store_wait** state.
  - For all other requests, the FSM moves to the **send_req** state.
- **uc_store_wait:** This state is triggered when an uncached store or no return L2 atomic request is sent to the BPC. OpenPiton expects the core to wait until a non-idempotent operation completes, due to the possibility of request reordering in the memory system. In order to adhere to this, all non-idempotent operations wait for the corresponding response in this state. This modification potentially results in a performance reduction in the system since the L1 cache is ready to send subsequent requests, but the memory system is not ready to accept them.
- **send_req:** This state is triggered when the PCE receives any request other than the ones mentioned above, handled as follows:

  For all requests apart from LR (load or store miss, uncached load, SC or atomic logic, or arithmetic operations), the FSM sends the corresponding request to the BPC, with the appropriate changes in the data and way ID. It moves to the respective states as shown in Figure 14 if the next level accepts the request.

  For an LR request, a backoff mechanism is implemented. A backoff mechanism is necessary for a multicore system to avoid cacheline bouncing and ensure forward progress. For example, consider a situation where all the cores request a single cacheline to perform an LR operation. This cacheline could be reserved for a core when requested, but the reservation

could cease before the corresponding SC request can complete due to an LR request from another core. In this situation, the cacheline reservation bounces from one cache to another, with none of the caches completing an SC request successfully.

The backoff mechanism involves a counter in the PCE that is started upon a failed SC request. This counter prevents the core from sending a subsequent LR request until a specified limit is reached. By having such a mechanism, at least one core will successfully complete an SC request, thereby releasing the other cores to follow suit. There are more sophisticated mechanisms such as exponential backoff, which could be a good future exploration.

- **uc_read_wait:** This state waits for the uncached data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_lr_wait:** This state waits for the LR data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_sc_wait:** This state waits for the SC data to arrive from the BPC, checks the data to identify if the linear backoff should start (if the data is not 0), sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_op_wait:** This state waits for the logic or arithmetic atomic operation data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.

Remote invalidations are handled outside the PCE FSM, since the PCE needs to accept these commands regardless of what operation it is currently performing. OpenPiton provides four types of invalidations: *dcache_inval*, *icache_inval_all_way*, *icache_inval*, and *dcache_inval_all_way*. Of these, the first two are currently used in ParrotPiton.

When an invalidation is received, the PCE sends the corresponding command to the cache SRAMs. It clears all ways for the I$ and sets the coherence state of the converted index and way ID for the D$. The transaction is complete when the L1 cache responds with the acknowledgment.

3.6 Full system integration

Figure 15 shows a ParrotPiton tile inside the full OpenPiton system. Both I$ and D$ connect to their own PCEs. On the request path, a fixed priority arbiter chooses the request to send to the BPC, and a FIFO holds back-to-back requests (in the case of write-through stores). A FIFO holds back-to-back responses routed to the correct PCE based on the response type on the response path.

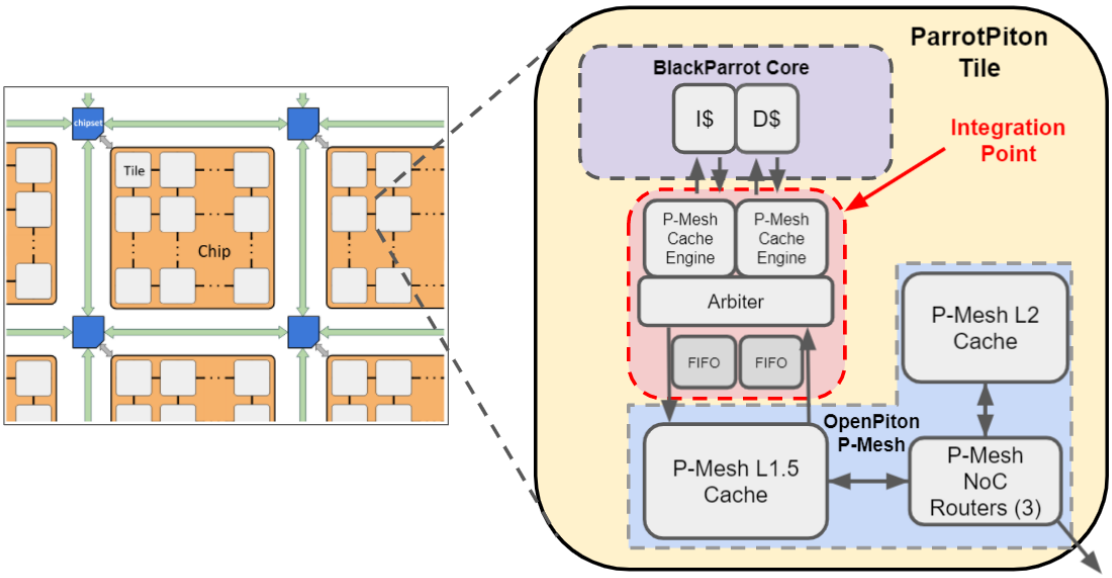

Figure 15: ParrotPiton tile

OpenPiton is integrated with another RISC-V core, Ariane [5]. The memory system uses the Core Local Interrupt Controller (CLINT) and the Platform Level Interrupt Controller (PLIC) of the Ariane core for generating interrupts. BYOC exposes three different interrupt signals, timer interrupt, software interrupt (which is also called Inter Processor Interrupt (IPI)), and external interrupt. All three signals are connected directly to the BlackParrot interrupt handling logic.

3.7 Baremetal Testing

ParrotPiton was tested in simulation and FPGA. Initial testing involved the RISC-V test suite (physical, virtual, physical with timer interrupts, atomic) and a suite of C tests provided by OpenPiton that tested operations such as atomics and interrupts in single-core and multi-core configurations.

3.7.1 Simulation

Simulation testing was the initial strategy to identify any bugs during the integration process. All facets of the PCE were tested: LR/SC, other atomics, invalidations (in multi-core configurations), and regular cache operations. The interrupt test enabled the interrupts by writing to appropriate CSRs, waited for the interrupt and cleared it, and repeated this operation for the specified number of times. All the tests passed in simulation leading to the next phase of testing on the FPGA since OpenPiton provides an FPGA infrastructure, and the goal was to boot Linux on ParrotPiton using the FPGA eventually.

3.7.2 FPGA

OpenPiton provides two commands for use with an FPGA. The *protosyn* command builds a bitfile for the specified core, and the *pitonstream* command allows programs to be streamed through UART onto the FPGA. The core, along with the memory system, executes the program and streams the print data through UART onto the screen. The target FPGA for ParrotPiton was Genesys 2.

After adding a floating-point unit (FPU) to BlackParrot, the ParrotPiton setup failed to meet timing at the OpenPiton specified frequency of 66.67 MHz. The problem was insufficient FPU retiming performed during the Vivado [20] implementation. Retiming is a strategy wherein registers inside a design are moved around to improve its critical path. For the FPU, the retiming strategy is to add several registers at the output, and the synthesis tool identifies the spots within the design to move the registers based on the timing constraints supplied. A synthesis tool such as Synopsys Design Compiler® handles this without issues, but Vivado was unable to do the same. Trying different directives did not fix the issue, so the frequency had to be reduced to 30 MHz. This will be fixed to allow all cores connected to the OpenPiton system to run at the same frequency.

Running the *protosyn* command with a single BlackParrot core generated a bitfile, and the same programs used in the simulation were used for testing on the FPGA. All the programs used for simulation ran on the board and generated the expected outputs. Table 5 shows the utilization results for single-core, dual-core and three-core ParrotPiton with this BlackParrot state (https://github.com/black-

parrot/black-parrot/tree/22da9a8e76ce78eb54ddf6f945a6879d0c4baf57) on the Genesys 2 FPGA (*xc7k325tffg900-2*).

Configuration	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
Single-core	69,333	3,242	128	50,465	37	30	12
Dual-core	117,787	4,784	223	72,063	68	56	23
Three-core	156,997	6,326	316	93,880	99	82	34
Single-core	34.02%	5.07%	0.2%	12.38%	8.31%	3.37%	1.43%
Dual-core	55.34%	7.48%	0.35%	17.68%	15.28%	6.29%	2.74%
Three-core	77.03%	9.88%	0.49%	23.03%	22.25%	9.21%	4.05%

Table 5: FPGA Utilization for different ParrotPiton configurations (BlackParrot is **continuously evolving** so this is a snapshot in time)

Table 6 shows the hierarchical utilization for some of the major components of the ParrotPiton system.

Component	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
ParrotPiton	69,333	3,242	128	50,465	37	30	12
BlackParrot core	19,823	822	95	7,105	6	25	11
L2 cache	14,516	0	0	7,372	21	1	0
FMA	2,814	0	71	165	0	0	11
D$	1,801	180	0	980	0	16	0
I$	861	182	0	1,707	0	8	0
Regfiles	763	0	0	492	5	0	0

Table 6: ParrotPiton hierarchical utilization (BlackParrot is **continuously evolving** so this is a snapshot in time)

An additional step was to run six SPEC2000 benchmarks [21, 22] on this system. The benchmarks were *gzip2*, *vpr*, *parser*, *bzip*, *crafty* and *mcf*. These benchmarks worked in BlackParrot as well at this stage, so they were the natural next step. These benchmarks uncovered a bug that was not triggered by previous integration tests, and worked on the board after a fix.

3.8 Linux Capability

Getting Linux to run on the ParrotPiton system was the next step in the process. There were a few steps before testing it on the board:

- ParrotPiton had to run with the bootrom option enabled in BlackParrot to use the bootrom provided by OpenPiton.

- OpenPiton-provided Linux image was given to the OpenSBI [23] framework used by OpenPiton in order to generate a payload written to an SD card. This SD card would be used in the Genesys 2 FPGA setup.

After several iterations, debugging in simulation became challenging because every Linux simulation took about 22 hours, giving a large output waveform file to parse. Litmus tests were helpful at this point to isolate the bugs and fix them with reasonable waveforms and simulation times. "A litmus test is a small parallel program designed to exercise the memory model of a parallel, shared-memory computer" [24]. Litmus tests are generated using the Litmus Tool [24] and CHERI-Litmus [25], a lightweight clone of the Litmus Tool that allows litmus tests to run on baremetal configurations.

The litmus tests isolated a few bugs and all the tests then successfully ran on the FPGA. After identifying a couple more bugs through the Linux simulation waveform, Linux booted on single-core, dual-core, and three-core ParrotPiton. Figure 16 shows the ParrotPiton Linux boot screen for the three-core version.

3.9 Bugs and Conclusions

Some of the major bugs discovered during testing were:

- During the addition of the floating-point unit into BlackParrot, the data sent to the CE through the cache_req structure was not changed to use the recoded floating-point data. Using raw data for cache requests in the write-back cache is sufficient since the only request using the data was the uncached store. Any data write-back would be sent as a response to a CE request over a different path. However, in the ParrotPiton system, every store was write-through, so the floating-point data was incorrectly stored to the next level. After a potential invalidation of the (correct) data in the L1 cache, the value loaded from the same address was incorrect.

Figure 16: Three-core ParrotPiton booting Linux

This was identified using the vpr benchmark in simulation and corrected by sending the recoded data for a write-through store.

- A data problem was discovered while performing endian swapping in the PCE. Before the SC data was sent to the L1 cache, it had to undergo endian swapping. Initially, for a 4-byte request, only the upper half of the big-endian data was swapped, and zeros were concatenated as the remaining bits in the little-endian data. This resulted in a problem when the SC address corresponded to the lower half of the big-endian data since the L1 cache would perceive it to have successfully completed the SC (since the data was 0) even if SC had failed at the L2 level. This was identified using an LRSC litmus test in simulation and fixed by using the

lower bits during endian swapping, even for a 4-byte size.

- BlackParrot used a replay mechanism for all cache operations that did not return when expected. This included load and store misses, uncached loads, and, importantly, L2 atomics. If the atomics execute at the L1 level, the memory state is not changed until a cacheline eviction, and the atomic operation does not complete until the cacheline is brought into the L1 cache, implying idempotency. With remote (L2) atomics, the cache uses the same replay mechanism, but the memory state is changed at the lower level before the L1 cache loads it by performing a replay. One of the interrupt tests exposed a scenario where an atomic swap request by one core was interrupted by a timer interrupt before the result entered the L1 cache. At this stage, the corresponding memory location had changed the value from 0 to 1, potentially indicating that a lock was acquired. However, since the interrupt flushed the pipeline, the core re-issued the atomic swap request upon resumption, but the value received was 1 instead of 0, indicating to the core that the lock was in place. This led the core to try acquiring the lock repeatedly, which never materialized. This was fixed by blocking timer interrupts until the completion of an outstanding cache operation.

- A bug was discovered in the LR implementation of OpenPiton. This was detected using the dual-core Linux simulation waveform of ParrotPiton. The bug was fixed and allowed ParrotPiton to boot Linux on a dual-core and three-core configuration.

The entire integration exercise provided valuable information for BlackParrot:

- It was amenable for integration with different systems after adding a plug-and-play model for the CE.

- It became more robust since many bugs were identified and fixed during the integration.

- L2 atomics were implemented in BlackParrot soon after the integration.

4 Limitations of Previous Methods

The previous section described the integration of BlackParrot into the OpenPiton memory system. Around the same time, BlackParrot was also integrated into another framework called Litex. Litex is an SoC builder that provides the infrastructure to create SoCs with or without CPUs easily. However, both these integrations have a few issues:

- Constant maintenance of the integrated system is required as both the components (BlackParrot and OpenPiton/Litex) might change regularly.
- Users might need to contact two different groups if they run into any problems while using the system.

BlackParrot did have an FPGA infrastructure for running larger benchmarks, but it involved using a PCIe interface. From the research group's experience, PCIe is very hard to get right and very hard to debug if problems arise. Instead of a PCIe-based setup, it would be beneficial to create an in-house FPGA setup using a simpler protocol like AXI [26], that is easy to use in different settings like research groups and students using BlackParrot in a class. The research group could use this FPGA setup for running Linux and spec benchmarks through a continuous integration flow, and students could use this setup for class projects, expanding BlackParrot's community reach. This led to the creation of the ZynqParrot [27] infrastructure, which set up BlackParrot on a Xilinx FPGA containing the Zynq-7000 Processing System.

5 ZynqParrot

This section describes the ZynqParrot system. Any FPGA containing the Zynq Processing System (PS) [28] that runs Linux will be suitable to implement this system. The idea is to implement BlackParrot in the Programmable Logic (PL), connect it to the Zynq PS, and run the tests on the PS Linux, which communicates with the PL via AXI ports. This setup would avoid PCIe altogether since the GitHub repository can be directly cloned in the Linux running on the PS, and the communication to the PL is through the simpler AXI protocol. Another advantage in ZynqParrot is that a desktop is no longer required to run tests on the FPGA since the ARM core on the board can function as the desktop. The bitfiles are sent to the file system of the core over Ethernet. Once the FPGA is programmed with the correct bitstream, the core can run the testbench.

The idea of using the PS to control a processor implemented in the PL has been explored earlier in [29], where the PS communicates with a dual-core RISC-V processor in the PL. Although the structure of this system is similar, there are some key differences between this system and ZynqParrot. Firstly, all communication between the PS and the PL is through shared DRAM addresses. As will be shown in Section 5.2, this is not ideal in the BlackParrot setting since i) messages from BlackParrot can cause race conditions with the PS and ii) BlackParrot requires direct connections with the PS. Further, constant polling of the shared memory locations by the PS could lead to contention and a potential performance implication. Secondly, the RISC-V cores in [29] are not capable of running an operating system on them, whereas the goal in ZynqParrot is to enable a Linux boot using BlackParrot on the FPGA. This requires a different approach to the interfaces between the PS and the PL.

5.1 Choosing the FPGA

The two requirements for choosing the FPGA were the availability of the Zynq PS and a cost between $100 and $150 so that the design could reach more users. The PYNQ Z-2 FPGA [30] is a board designed to support the Python Productivity for Zynq (PYNQ) framework and fit these

requirements well. The PYNQ [31] framework is an open-source Xilinx project that allows users of the Zynq SoC the opportunity to run their programs in Jupyter Notebook and interact with the hardware in the PL. Although the ZynqParrot infrastructure described in this thesis used this FPGA, the underlying design would work for any Zynq-based FPGA. Table 7 shows the BlackParrot utilization results (both raw and percentage) for this part (*xc7z020clg400-1*) for both the default (32 KB L1 caches and a 64 KB L2 cache) and small (8 KB L1 caches and a 64 KB L2 cache) BlackParrot configurations. Both these utilization numbers use the BlackParrot state at this commit (https://github.com/black-parrot/black-parrot/tree/f5cbac361c8099b18b9b665c00bc02f691b654d4). The main difference between these two results is the amount of BRAM used since the L1 caches are four times smaller in the second design.

Configuration	Logic LUTs	LUTRAMs	FFs	RAMB36	RAMB18	DSP48
Default	30,796	1,360	12,945	6	145	11
Small	23,899	1,102	10,234	6	85	11
Default	57.89%	7.82%	12.17%	4.29%	51.79%	5%
Small	44.92%	6.33%	9.62%	4.29%	30.36%	5%

Table 7: FPGA Utilization for two BlackParrot configurations (BlackParrot is **continuously evolving** so this is a snapshot in time)

5.2 Choosing the connections to use

BlackParrot's external interfaces

BlackParrot exposes three interfaces to external logic:

- **Outgoing port:** BlackParrot sends a command to the external logic and expects a response. This port is used when BlackParrot sends a read or write command to the UART or denotes that the program has finished.
- **Incoming port:** BlackParrot receives a command from the external logic and provides a response after processing it. This port is used to set the configuration registers, load the program to the DRAM through the L2 cache, and send user input to BlackParrot for a UART read command.

- **DRAM port:** BlackParrot sends a read or write request to the DRAM and waits for the response.

These ports can be used either with BlackParrot-specific interfaces or as AXI ports. The outgoing and incoming ports connect to I/O <-> AXI converters and become AXI4-Lite ports, and the DRAM output port connects to the L2 cache packet <-> AXI converter and becomes an AXI4 port.

Zynq

Figure 17 shows the Zynq PS. There are two General Purpose AXI ports (denoted in red at the bottom left of the figure) with PS as the master and PL as the slave, and two with PL as the master and PS as the slave. Further, four High-Performance AXI ports (denoted in green at the bottom right) with 64-bit data connect to the DDR3 controller.

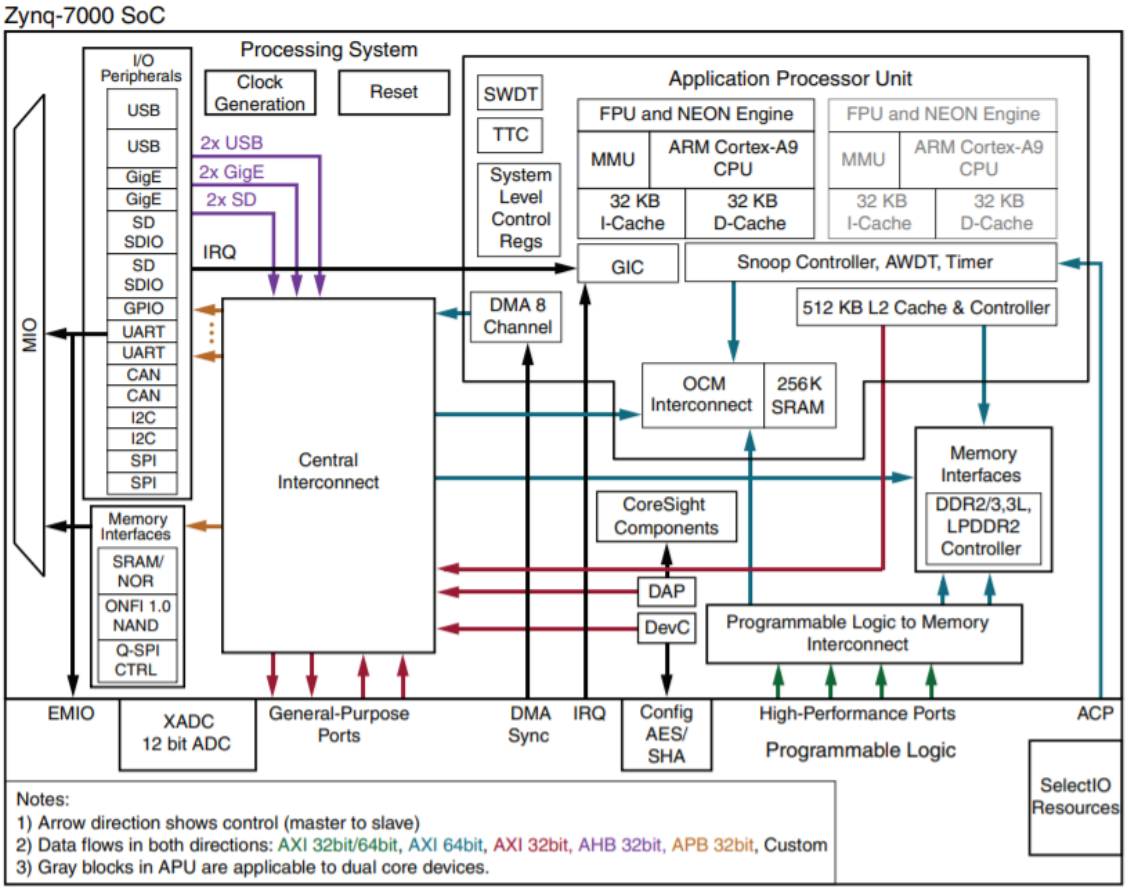

Figure 17: Zynq-7000 SoC Block Diagram [28]

Figure 18 shows the PS address map. Based on the settings used, the Zynq PS ports map to the following physical addresses:

- DDR: 0x00000000 - 0x3FFFFFFF
- M_AXI_GP0 - 0x40000000 - 0x7FFFFFFF
- M_AXI_GP1 - 0x80000000 - 0xBFFFFFFF

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters(1)	Notes
0000_0000 to 0003_FFFF(2)	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU(3)
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF(4)	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF(2)	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 18: Zynq-7000 System-Level Address Map [28]

The PS allows the user to allocate DRAM space for the PL to access. The allocation returns a virtual and physical address corresponding to the start of the allocated region. The PS can use the virtual address for access and send the physical address to the PL for correct address translation.

AXI4-Lite Protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between Manager (master) and Subordinate (slave) components. It has separate read and write address and data channels. The AXI4-Lite protocol is a subset of the full AXI4 protocol, with all transactions having a burst size of 1 and all data accesses using the full width of the data bus (32-bits or 64-bits). The specification [26] contains in-depth information about the AXI4 protocol signals and timing diagrams.

Identifying connectable interfaces

The BlackParrot AXI interface would be the best choice for the ZynqParrot system since the PS communicates using the AXI3 protocol.

The following connections are immediately apparent:

- PS M_AXI_GP1 -> BlackParrot incoming port AXI. The PS port follows AXI3, and BlackParrot follows AXI4-Lite, requiring a converter between them.
- BlackParrot DRAM port AXI4 -> PS S_AXI_HP0. The HP port allows 64-bit data, the minimum data width supported by the BlackParrot L2 cache. BlackParrot follows AXI4, and the PS port follows AXI3, requiring a converter between them.

The converter used in this system is the Xilinx AXI SmartConnect [32]. This IP allows multiple master AXI devices to communicate with multiple slave AXI devices while performing the required protocol conversion. For the first connection mentioned above, only the protocol conversion is necessary since it is a 1:1 connection.

The remaining connection is the outgoing port of BlackParrot. There are two options here:

- Connect the BlackParrot port to the S_AXI_GP0 port and map the connection to the DDR or OCM
- Use a FIFO in the PL that can be written by the BlackParrot port and read by the PS M_AXI port

The first option seems to be the most intuitive since the BlackParrot outgoing port is a master AXI interface, and the PS S_AXI port is a slave interface. However, even if the address mapping is such that the data goes to the DDR or the OCM, there are two issues.

- BlackParrot currently sends a UART write request to a single UART address (0x101000). If the requests go to a single DRAM location, there is a possibility of missed reads by the PS due to race conditions. The program working would depend on the relative latency between BlackParrot sending a request and the PS reading it, which is not acceptable.
- Even if BlackParrot is modified to send UART requests to consecutive locations, it is limited by the DRAM space allocated for this purpose. Further, it has no way of knowing if the PS read the data at older locations before overwriting them.

The second option is not apparent at first but offers an easy way for BlackParrot to guarantee that the PS receives every request. Every request is converted into an AXI write and packed into the 32-bit data, indicating the address and data (for a UART write). If the FIFO is full, BlackParrot stalls until space is available. If the FIFO is empty, there could be additional logic to send garbage data so that the PS AXI request does not stall. This FIFO is designed using the AXI Peripheral creation option provided by Vivado and borrowing the FIFO design from BaseJump STL [33]. It is connected to the M_AXI_GP0 port of the PS and the BlackParrot outgoing port using the AXI SmartConnect.

Additionally, the PS writes a base address register to communicate the physical address of the allocated DRAM to the PL logic. An AXI GPIO [34] module supplies the data written by the PS in its register to the PL.

The connections described above are used in Version 1 of the ZynqParrot design.

5.3 Version 1

Figure 19 shows the Version 1 ZynqParrot design. All the connections mentioned previously appear in the figure along with their protocol type. Before connecting BlackParrot as the Design Under

Test (DUT), a trivial module was added as the DUT to test the interfaces. This DUT contained the following:

- AXI4-Lite BRAM Controller connected to a Block RAM module (to test the direct PS to BlackParrot connection).
- Loopback connection to the DRAM from PS M_AXI_GP1 port to S_AXI_HP0 port (to test the BlackParrot to DRAM connection).
- Simple FSM with AXI4-Lite master interface generating the sentence ”Hello from PL” (to test the BlackParrot to FIFO connection).

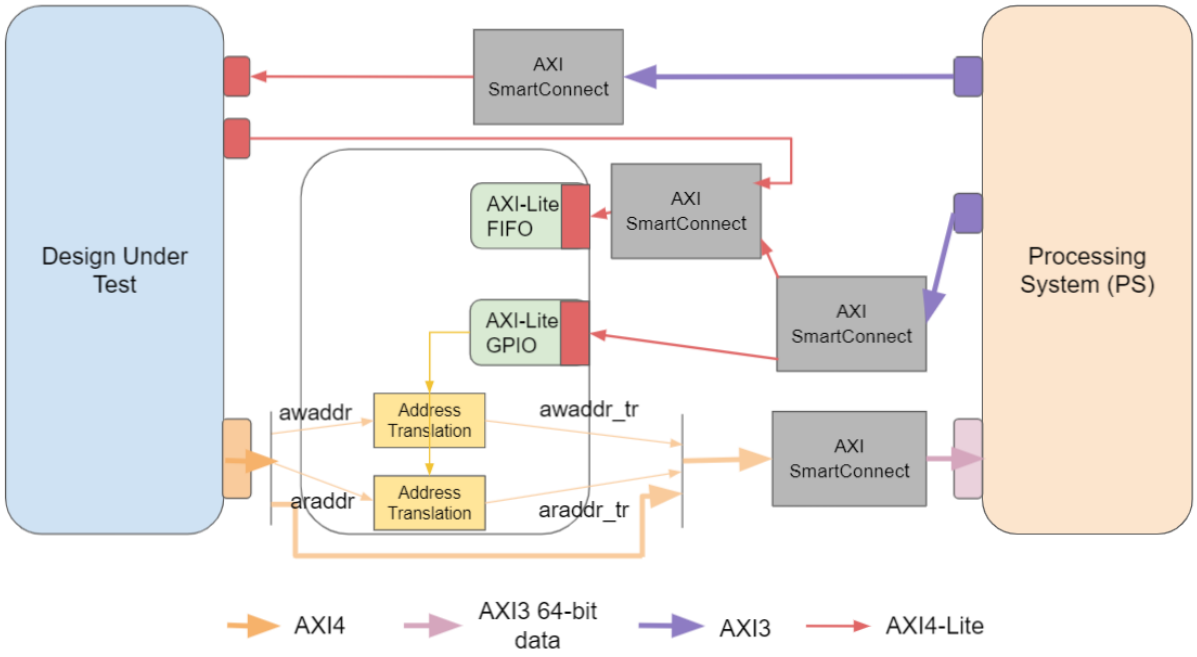

Figure 19: Version 1 ZynqParrot design

A Python code using the PYNQ-provided APIs tested the connections on the FPGA and confirmed their working. However, this design was highly Xilinx IP-centric, potentially making debugging more difficult. The ideal setup would allow for simulation of the DUT using an open-source simulator such as Verilator [35] with an emulated PS AXI port interface before testing the design on the board. The PS AXI port emulation would provide confidence to the users that a design working

in simulation is likely to work on the board, as long as the address mappings are correct. Further, BlackParrot is tested in Verilator so it would be a better option than Vivado simulation.

These issues prompted further experiments to identify a better setup, resulting in Version 2 of the ZynqParrot design.

5.4 Version 2

Figure 20 shows the Version 2 design for ZynqParrot. The PL modules reside in a single Vivado IP Integrator (IPI) block. The changes from the above design are as follows:

- The AXI SmartConnects are limited only for converting the PS AXI3 connections to BlackParrot's AXI4-Lite and AXI4 connections.
- Instead of using Xilinx IPs for GPIO and FIFO modules, a decoder module contains the FIFO with both ends connected to an AXI4-Lite interface, the FIFO occupied count, and offset registers.

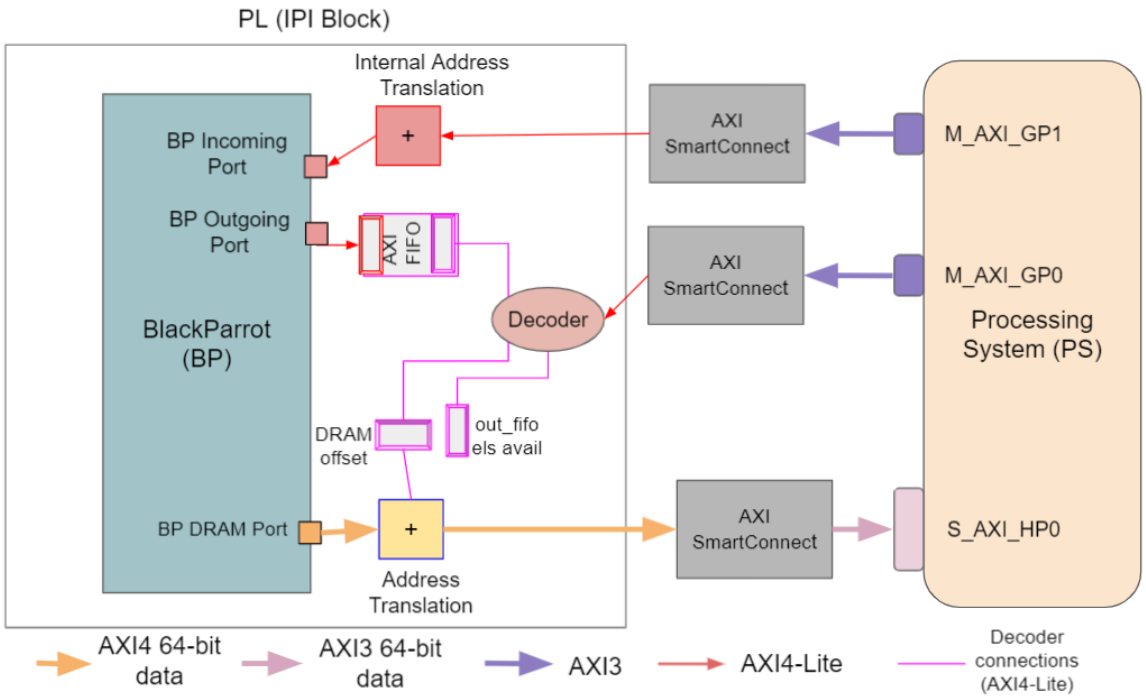

Figure 20: Version 2 ZynqParrot design

The address translations for the DRAM remain the same but are not shown explicitly in the figure. By including the decoder module, the only Xilinx IPs used (apart from the PS) are the three AXI SmartConnects. The fourth SmartConnect, which arbitrated between the PS and the DUT for FIFO access, is replaced by creating a FIFO with AXI4-Lite interfaces on both the input and output sides. This design provides an opportunity to design the PL modules, test them entirely in Verilator simulation before adding the SmartConnect modules and the PS for implementation on the FPGA.

5.4.1 AXI4-Lite Decoder

The decoder module allows users to add a parameterizable number of Control and Status Registers (CSRs) to which the PS can write address offsets, the above-mentioned FIFO, and its corresponding flow counter, which is a BaseJump STL module that provides the number of elements currently in the FIFO. It has one AXI4-Lite interface, and the AXI write and read addresses are decoded into one-hot signals, which are used as the select signals for a one-hot multiplexer, all BaseJump STL modules. The PS can read and write the registers, whereas it can only read the FIFO since the DUT writes to it.

5.4.2 FIFO with AXI4-Lite interfaces on both ends

This FIFO allows an AXI4-Lite master to write to it and an AXI4-Lite master to read from it. The difference between this FIFO and the Xilinx custom IP earlier is that there are two different AXI interfaces in this module. Therefore, arbitration is not required, reducing the number of modules for debugging. On the input side, the AXI master can only write to the FIFO. However, if the master does read, zero is returned as the data without hanging. On the output side, the AXI master can only read from the FIFO. However, if the master does write, the write is dropped without hanging.

The counter works in conjunction with the FIFO. When an element is enqueued, the counter is incremented. Similarly, when an element is dequeued, the counter is decremented. The PS should ideally read the count at regular intervals and only read the number of elements from the FIFO as specified by the count.

5.4.3 BlackParrot Address Space

BlackParrot uses a 40-bit physical address split as follows:

- 0x0000000000 - 0x007FFFFFFF: Local addresses (<7-bit tile ID, 4-bit device ID, 20-bit device address space>. An example device is the UART.
- 0x0080000000 - 0x0FFFFFFFFF: Cached DRAM addresses.
- 0x1000000000 - 0x1FFFFFFFFF: On-chip streaming accelerator address space.
- 0x2000000000 - 0xFFFFFFFFFF: Off-chip addresses

For the ZynqParrot design, the BlackParrot local addresses and cached DRAM addresses are supplied by the same PS M_AXI port to avoid performing an extra MUX operation (required if two different AXI ports supply the data for the two regions). An equal distribution would ideally allow 512 MB for the DRAM address space and 512 MB for the local address space, since a single PS M_AXI port can access 1 GB of address space.

The PS runs PetaLinux [36], which requires 64 MB of DRAM space, and the PYNQ Z-2 board contains a DDR3 memory of size 512 MB. BlackParrot can therefore use 256 MB of DRAM space, which is sufficient for most of the standard benchmarks.

Similarly, a 256 MB address space for the local addresses ensures access to 16 tiles since the highest address accessible in this scenario is 0x000FFFFFFF, which includes 4 bits of the tile ID. Due to hardware constraints of the board, it is improbable to require more than 4 bits for the tile ID.

The two address spaces are combined to map to 512 MB of the PS GP1 port. The address maps are shown in Table 8. An internal address translation is required since the Xilinx tool subtracts the 0x80000000 offset after moving through the GP1 port. This is also indicated in the table.

Region	Address seen by PS	Address inside PL IPI	Address Translation
DRAM	0x80000000 - 0x8FFFFFFF	0x00000000 - 0x0FFFFFFF	Add 0x80000000
Local	0xA0000000 - 0xAFFFFFFF	0x20000000 - 0x2FFFFFFF	Subtract 0x20000000

Table 8: ZynqParrot Address Mapping

5.4.4 BlackParrot UART read issue

The Version 2 design contains all BlackParrot interface logic except for the UART read request logic. The only path for this request in the design is to the FIFO with AXI4-Lite interfaces at both ends. As already mentioned, the input to this FIFO is write-only, but a read will return zero without hanging. This issue is not immediately apparent since most standard tests used in BlackParrot testing do not involve UART reads. However, consider the scenario where BlackParrot is running Linux and requests user input. The user input would always be zero, without any intervention possible by the user.

Version 3 of the ZynqParrot design fixed this issue using a new module and another FIFO.

5.5 Version 3

A store packer converts read requests from the BlackParrot outgoing port into write requests that are then enqueued onto the FIFO. Further, we add another FIFO in the reverse direction that the PS can write, and BlackParrot can read. The PS reads the request in the FIFO, deciphers it, performs the required operation (print to the screen, obtain user input or terminate the program), and sends the user input (if any) to the newly added FIFO. The store packer would contain a small FSM that waits for the user input from the PS if the BlackParrot request was a UART read and responds to BlackParrot once it receives the data.

Earlier, the FIFO enqueueing BlackParrot AXI requests had an AXI4-lite input interface. Now, since all requests are converted into write packets that the PS deciphers, the FIFO can revert to having an output AXI interface and a standard input interface of data and valid.

The entire process is described below. The FIFO enqueuing requests from BlackParrot is called the output FIFO, and the FIFO enqueuing requests from the PS is called the input FIFO.

- BlackParrot sends an I/O request (UART read/write, finish).
- The store packer converts this request into a 32-bit packet with a concatenation of <read/write (1-bit), address (23-bits), data (8-bits)>.
- The 32-bit packet is enqueued onto the output FIFO.
- The PS reads the packet, decodes it in software, performs the necessary operation, and returns data, if any, to the input FIFO.
- In the event of a read, the store packer responds to BlackParrot with the data.

The decoder module (called the *shell* in this version) contains the two FIFOs, a DRAM base offset register and a newly added reset register that controls the BlackParrot reset signal along with the AXI interface reset. Earlier, the only way BlackParrot could be reset was by reloading the bitstream. This register allows the PS to control the BlackParrot reset through the testbench without having to rely on the interface reset.

Figure 21 shows the version 3 ZynqParrot design. The store packer is denoted as BP r/w to write in the figure.

A future version of ZynqParrot would benefit from using the AXI4-Lite outgoing port interface since BlackParrot could easily interface with other accelerators and use a multicore configuration with multiple I/O ports. Figures 22 and 23 show an example design.

5.6 Software setup

The main idea behind cosimulation is to provide users with an option to test their BlackParrot system in simulation by mimicking the PS AXI interface as closely as possible. The strategy is to create a single C++ testbench file and swap the include files based on the environment used. Verilator converts the Verilog modules into a C++ equivalent triggered using the testbench for use in simulation, whereas the same testbench is run on the Linux instance of the ARM core on the FPGA to communicate with the PL.

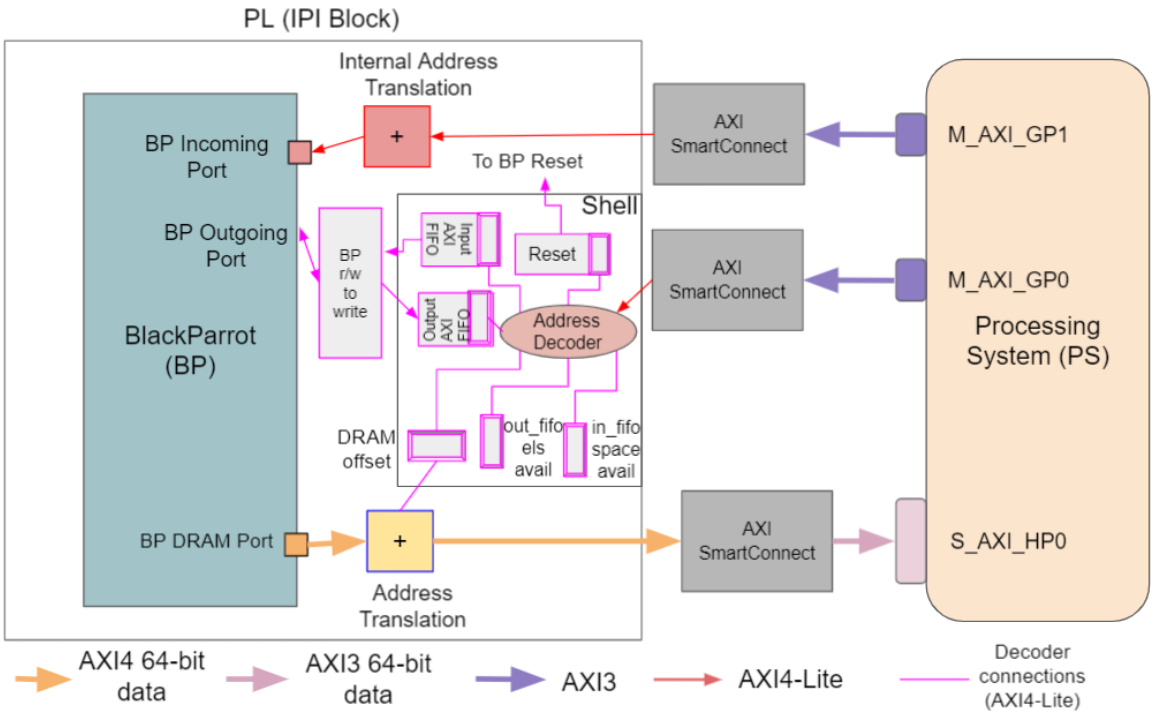

Figure 21: Version 3 ZynqParrot design

5.6.1 Directory structure

Figure 24 shows the directory structure in the ZynqParrot repository. As depicted, the same testbench is reused for both the simulation and the hardware by including different header files for each environment.

5.6.2 Header files

The software API provides two header files, one for simulation and one for the FPGA. These two files contain a class with largely the same methods so that the testbench can swap them based on the environment. The common methods between the two files are as follows:

- *axil_write:* This method performs a write operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a write to a pointer that signifies the specific AXI port.

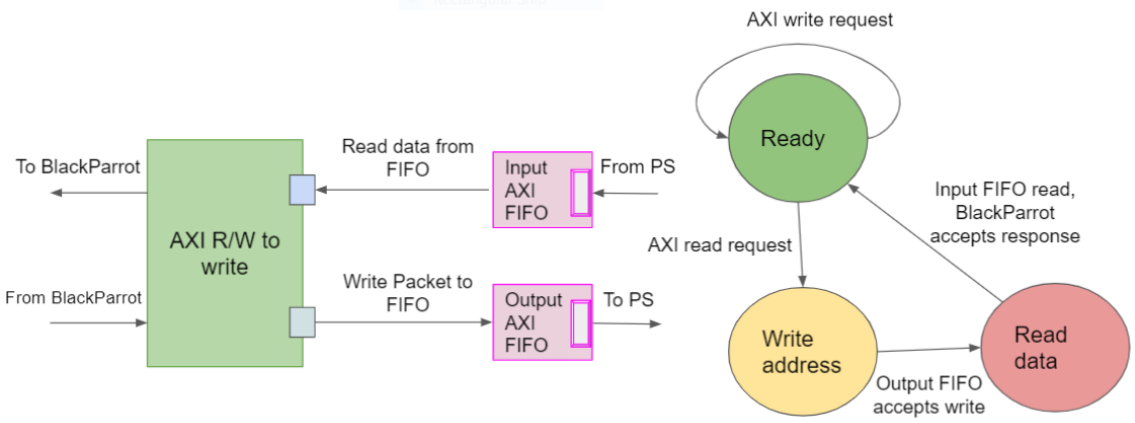

Figure 22: AXI R/W to Write module and FSM

- *axil_read:* This method performs a read operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a read from a pointer that signifies the specific AXI port.

- *done:* This method signifies the end of testing.

There are a few additional methods and operations in the FPGA environment:

- *allocate_dram:* The PS needs to allocate the required amount of DRAM for the PL to access. This method performs the allocation and returns the virtual and physical pointer to the first DRAM location. Currently, we have experimented with different DRAM sizes and found that 64 MB works well. A plan is to try and extend this size since some programs need more than 64 MB of DRAM.

- *free_dram:* This method frees the allocated DRAM and should be called at the end of testing.

5.6.3 Testbench

The testbench for BlackParrot instantiates an object of the class described above and performs two operations: set the configuration registers inside BlackParrot and write the program to DRAM through the BlackParrot L2 cache. Once these operations are complete, the testbench waits for

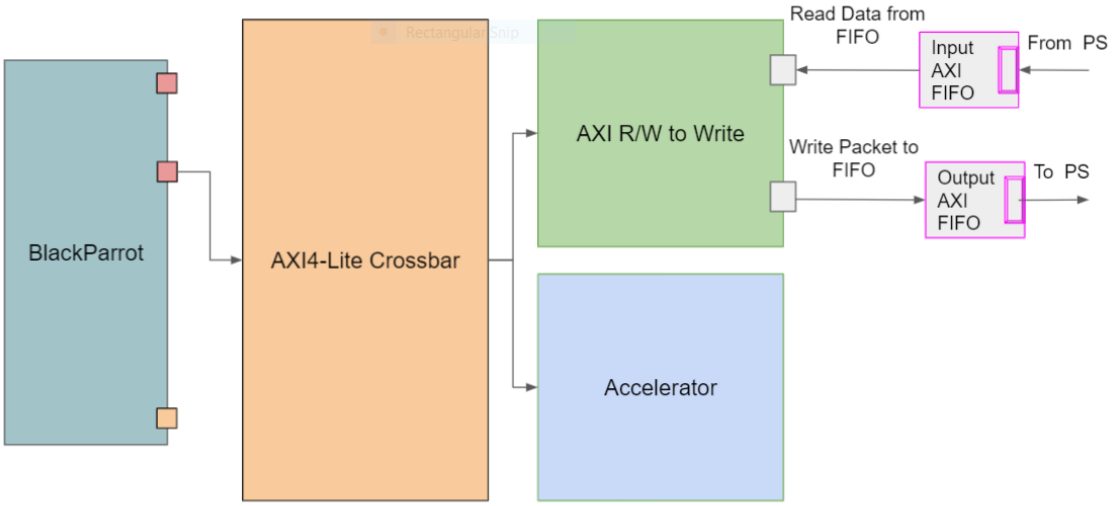

Figure 23: Future extension to ZynqParrot

data to arrive at the output FIFO and decodes the data to perform the correct operation. The two functions specific to the BlackParrot testbench are described below:

- *nbf_load:* This function sets the configuration registers and loads the program into the DRAM by using a .nbf (Network Boot Format) file available in BlackParrot. The nbf file contains a packet with the address and data in each line. This function reads the file line by line, extracts the address and data, and calls the *axil_write* method of the object with these values.

- *decode_bp_output:* This function receives 32-bit data and decodes the contents. As mentioned in the hardware section, a packet sent by BlackParrot is a concatenation of <R/W, 23-bit address, 8-bit data>. If the packet is a UART write (R/W = 1 and address = 0x101000), the data is printed onto the screen. If the packet indicates a finish (R/W=0 and Address = 0x102000), the program is terminated. UART reads are currently not supported in this function but can be added by asking for user input and writing the result to the input FIFO.

The testbench also contains multiple diagnostic tests aimed at testing each interface connection in both simulation and actual hardware. The following tests are performed before BlackParrot is initiated:

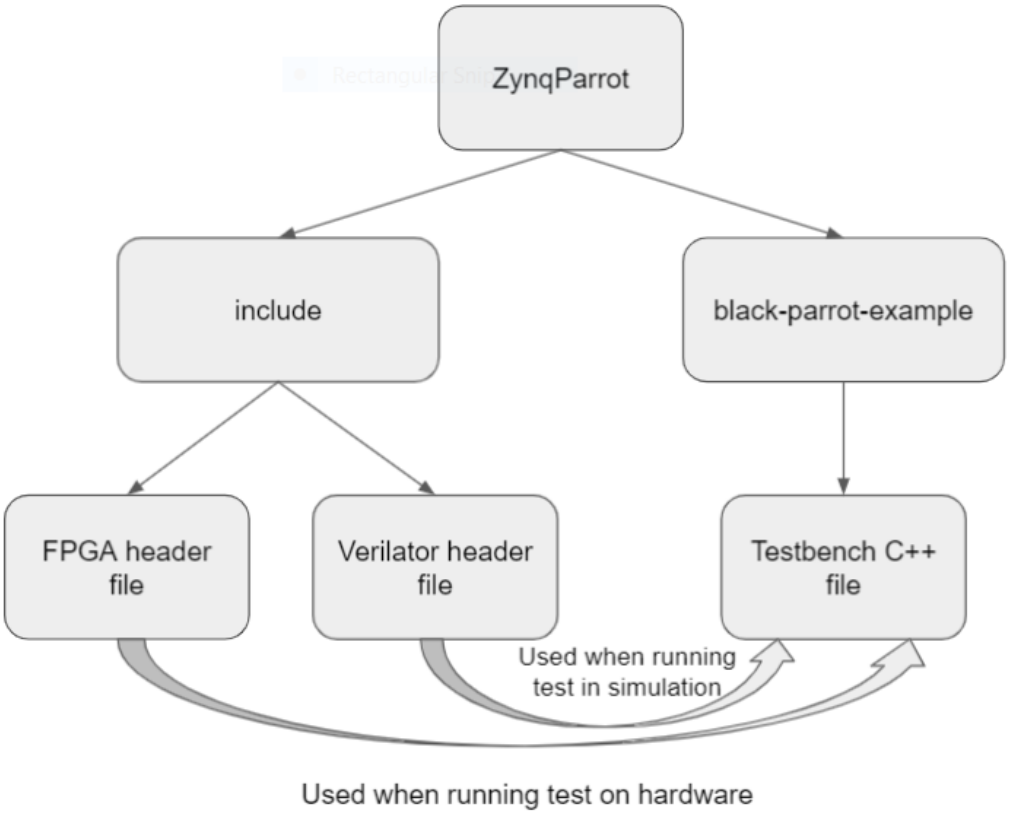

Figure 24: Testing directory setup

- Writing and reading the CSRs in the decoder (tests the GP0 port connection).
- Writing to enough addresses in the DRAM through BlackParrot to trigger evictions from the L2 cache and reading these addresses using the virtual addresses in the PS space (on hardware) or through BlackParrot (in simulation) (tests the GP1 port connection for DRAM addresses and HP0 port connection).
- Reading from the *mtime* and *mtimecmp* registers (machine timer registers) inside BlackParrot (tests the GP1 connection for local addresses).

5.7 Validation and Results

This section provides the numbers and the status as of this state (https://github.com/black-parrot-hdk/zynq-parrot/tree/498d90532c7ca768504cfd4c30dc3b62755dd6d8). It is important to note that ZynqParrot is a continuously evolving system, so these results are valid only at this point in the history. The testing of this system involved both simulations using Verilator and running the test-bench on the PYNQ-Z2 FPGA. Table 9 shows the utilization results on the PYNQ-Z2 for the ZynqParrot system with the default BlackParrot configuration containing 32 KB L1 caches and a 64 KB L2 cache. Table 10 shows the hierarchical utilization of ZynqParrot with some of the major components of the system displayed. The basic "hello_world" program provided the initial confirmation that the system worked as expected. The next step involved running several SPEC2000 and SPEC2006 benchmarks [21] on the FPGA. We also ran the *beebs* benchmarks [37] as a regression on the board (all tests run with a single command) with the help of the reset register in the shell module and confirmed its working.

FPGA Component	Utilization	Percent utilization
Logic LUTs	31,917	59.99%
LUTRAMs	1,652	9.49%
SRLs	433	2.49%
FFs	16,586	15.59%
RAMB36	6	4.29%
RAMB18	145	51.79%
DSP48	11	5.00%

Table 9: FPGA Utilization for the ZynqParrot system with the default BlackParrot configuration

Tables 11 and 12 show the instructions, cycles and Instructions per Cycle (IPC) values for the SPEC2000 and SPEC2006 benchmarks running on ZynqParrot. The FPGA emulation speed is around one billion BlackParrot cycles per minute, whereas the Verilator simulation speed is around 100,000 cycles per minute. The numbers in the table show that the longest running benchmark (464.h264ref) would take about 200 minutes to complete on the FPGA, whereas even the smallest benchmark in terms of cycles (175.vpr) would take around 270 minutes to simulate using Verilator. ZynqParrot enables running this new class of tests that is impossible to do in simulation.

Component	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
ZynqParrot	69,333	3,242	128	50,465	37	30	12
D$	3,751	191	0	2,117	0	64	0
BP-L2 converter	3,340	384	0	614	0	0	0
SmartConnects	2,933	224	338	3,842	0	0	0
L2 cache	1,797	454	0	1,987	0	64	0
I$	1,476	183	0	1,373	0	16	0
FMA	1,256	0	71	165	0	0	11
Regfiles	763	0	0	492	5	0	0

Table 10: ZynqParrot hierarchical utilization

SPEC2000 Benchmark	Instructions Retired	Cycles	IPC
164.gzip	1,234,135,782	1,968,599,120	0.627
175.vpr	23,452,462	27,906,352	0.840
177.mesa	437,214,657	562,967,056	0.777
183.equake	526,995,458	841,464,792	0.626
186.crafty	4,142,052,045	5,172,257,264	0.801
188.ammp	38,813,804	72,238,528	0.537
197.parser	227,585,917	289,582,784	0.786
256.bzip2	1,592,049,924	2,205,371,304	0.722
300.twolf	97,342,911	182,005,544	0.535

Table 11: SPEC2000 benchmarks running on ZynqParrot

SPEC2006 Benchmark	Instructions Retired	Cycles	IPC
401.bzip2	3,175,874,105	4,210,062,456	0.754
410.bwaves	20,684,252,488	48,997,476,110	0.422
444.namd	64,051,683,912	104,703,424,500	0.612
445.gobmk	416,632,992	1,062,763,112	0.392
462.libquantum	176,152,994	203,135,528	0.867
464.h264ref	142,762,531,949	200,478,638,500	0.712
473.astar	23,673,222,279	42,845,912,910	0.553

Table 12: SPEC2006 benchmarks running on ZynqParrot

6 Conclusions and Future Work

This thesis explained the requirement of a single-core configuration with a flexible cache design and an interface between the cache and various cache engines. It also described how this helped integrate BlackParrot into OpenPiton, including the design decisions made along the way. The integration was complete with Linux booting on the integrated system for the one, two, and three core configurations.

Once the ParrotPiton system was functional, we realized that creating an in-house FPGA system would also be highly beneficial for external users trying out BlackParrot as an accelerator host in their designs. The ZynqParrot system would eliminate the need for a desktop computer and the accompanying PCIe connection to connect the FPGA, which is very hard to debug when a problem arises. The ARM PS on the FPGA instead replaces the desktop computer and uses the AXI protocol for the PS-PL connection, which significantly simplifies interfacing with the PS.

We compared different design choices to allow the ZynqParrot system to be cosimulated with Verilator simulation and hardware support using the same testbench. The final design (at this point) contained no Xilinx IPs apart from the SmartConnect modules performing the protocol conversion between the PS and the BlackParrot module, which were not required for Verilator simulation since this step emulated the PS connections and did not actually include the PS.

The creation of the ZynqParrot infrastructure makes BlackParrot more attractive to users due to the ease of use and low-cost nature of the solution. Users can leverage this FPGA system to test their accelerator designs in actual silicon instead of just simulation, leading to faster design times.

Some future steps are given below:

- Explore the write-around policy and its effect on ParrotPiton performance.
- Explore the interplay between non-blocking BlackParrot requests and the OpenPiton memory system.
- Explore more sophisticated backoff mechanisms in the PCE.

- Explore alternatives for retiming the FPU and ensure that the frequency is brought back to 66.67 MHz.
- Convert the BlackParrot outgoing port interface into AXI4-Lite with the necessary changes to the store packer.
- Extend the DRAM allocation size to 256 MB for ZynqParrot.
- Run Linux on BlackParrot in the ZynqParrot framework.

7 Acknowledgments

The past two years have helped me immensely to develop my skills in hardware design. Many people were responsible for making it a knowledgeable and enjoyable ride, and I would like to thank them.

Firstly, I would like to thank my parents, Hemamalini and Vijaya Ranga, for their unwavering support throughout my life. My journey through the last two years would not have been possible without their constant encouragement, even while being many thousand miles away.

I would like to thank my advisor, Professor Michael Taylor, for his support and encouragement throughout my masters. His advice has been invaluable during my time in his research group and also for my professional life after masters. I am very grateful to him for giving me the opportunity to work in his research group, and I have picked up a lot of hardware design skills during my time there. I would also like to thank Professor Scott Hauck, who, in addition to being a part of my thesis committee, was also instrumental in helping me build a solid foundation in computer architecture through his course in my first quarter here. I also enjoyed being a TA for both Prof. Taylor's and Prof. Hauck's courses, and I would like to thank them for these opportunities.

My time with the Bespoke Silicon Group has been one of constant learning and improvement. Daniel Petrisko deserves a special mention since he has been a mentor to me for the past one and a half years and has constantly taught me new things. He showed a lot of patience while I got up to speed with any new task and was available to help me whenever I needed it. Sripathi Muralitharan has been a great teammate for many course projects and research group collaborations over the last two years. I would also like to thank Mark Wyse and Farzam Gilani, among many others in BSG with whom I've been lucky to interact.

I would like to thank Jonathan Balkind (then Princeton University, now an Assistant Professor at the University of California, Santa Barbara) for his support throughout the ParrotPiton integration. He always answered my questions and pointed me in the right direction whenever I needed it.

I would also like to thank my roommates - Sripathi and Mukund Gupta, and all my friends who added in the splashes of fun necessary to make this experience complete. Finally, I want to thank

my whole family for their support.

This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([11,38–51]), ASIC Clouds ([52–59]), open source hardware ([33, 60, 61]) RISC-V ([2, 12, 62–66]), Network-on-Chips ([67–71]), security ([72–75]), benchmark suites ([76–78]), dark silicon ([7,50,51,79,79–82]), multicore ([2,70,71, 83–92]), compiler tools ([93–101]) and FPGAs ([78,102,103]).

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, NSF SaTC Award 1565446, and by the DARPA/SRC JUMP ADA Center.

References

[1] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, 05 2011.

[2] 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.

[3] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[4] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] 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, Nov 2019.

[6] 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.

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

[8] 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.

[9] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of Machine Learning Accelerators," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2020. [Online]. Available: http://dx.doi.org/10.1109/HPEC43674.2020.9286149

[10] 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*, 2015, pp. 52–59.

[11] 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.

[12] 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.

[13] "BlackParrot GitHub Repository," https://github.com/black-parrot/black-parrot.

[14] 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

[15] "OpenPiton GitHub Repository," https://github.com/PrincetonUniversity/openpiton.

[16] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence Domain Restriction on Large Scale Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 686–698. [Online]. Available: https://doi.org/10.1145/2830772.2830832

[17] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 699–714. [Online]. Available: https://doi.org/10.1145/3373376.3378479

[18] "BYOC GitHub Repository," https://github.com/bring-your-own-core/byoc.

[19] "Cache Write Policies and Performance," https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-91-12.pdf.

[20] Xilinx, "Vivado Design Suite," https://www.xilinx.com/products/design-tools/vivado.html.

[21] "SPEC Benchmarks," https://www.spec.org/benchmarks.html.

[22] https://github.com/black-parrot-sdk/spec2000.

[23] https://github.com/Jbalkind/opensbi/tree/byop.

[24] http://diy.inria.fr/.

[25] "CHERI-Litmus," https://github.com/CTSRD-CHERI/CHERI-Litmus.

[26] ARM, "AMBA AXI and ACE Protocol Specification," https://developer.arm.com/documentation/ihi0022/latest.

[27] "ZynqParrot GitHub Repository," https://github.com/black-parrot-hdk/zynq-parrot.

[28] Xilinx, "Zynq-7000 SoC Technical Reference Manual," https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[29] P. Tegegn, "An Implementation of a Predictable Cache-coherent Multi-core System," Master's thesis, University of Waterloo, 2019. [Online]. Available: https://core.ac.uk/download/pdf/200282743.pdf

[30] "TUL," https://www.tul.com.tw/ProductsPYNQ-Z2.html.

[31] "Pynq - Python productivity for Zynq - Home," http://www.pynq.io/.

[32] Xilinx, "SmartConnect v1.0 LogiCORE IP Product Guide," https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf.

[33] M. B. Taylor, "INVITED: BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[34] Xilinx, "AXI GPIO v2.0 LogiCORE IP Product Guide (PG144)," https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf.

[35] "Verilator," https://www.veripool.org/verilator/.

[36] Xilinx, "PetaLinux Tools," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[37] https://github.com/black-parrot-sdk/beebs/tree/blackparrot_mods.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

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

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

[57] 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.

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

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

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

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

[62] 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.

[63] ——, "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.

[64] 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.

[65] 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.

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

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

[68] D. Petrisko, C. Zhao, S. Davidson, P. Gao, D. Richmond, and M. B. Taylor, "NoC Symbiosis," in *NOCS*, 2020.

[69] 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.

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

[71] 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.

[72] 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.

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

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

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

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

[81] 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.

[82] 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.

[83] 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.

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

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

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

[87] 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.

[88] 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.

[89] 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.

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

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

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

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

[101] 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.

[102] 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.

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

Appendix

The cache testbench initially supported single cache testing using the setup shown in Figure 25.

Trace replay is a BaseJump STL module that reads traces from a ROM and sends the data to the system connected to it. This module helps test functional correctness by sending a set of inputs to the system and validating the outputs from the system. There are a small set of instructions that the trace replay can decipher and act accordingly. These are provided in Table 13.

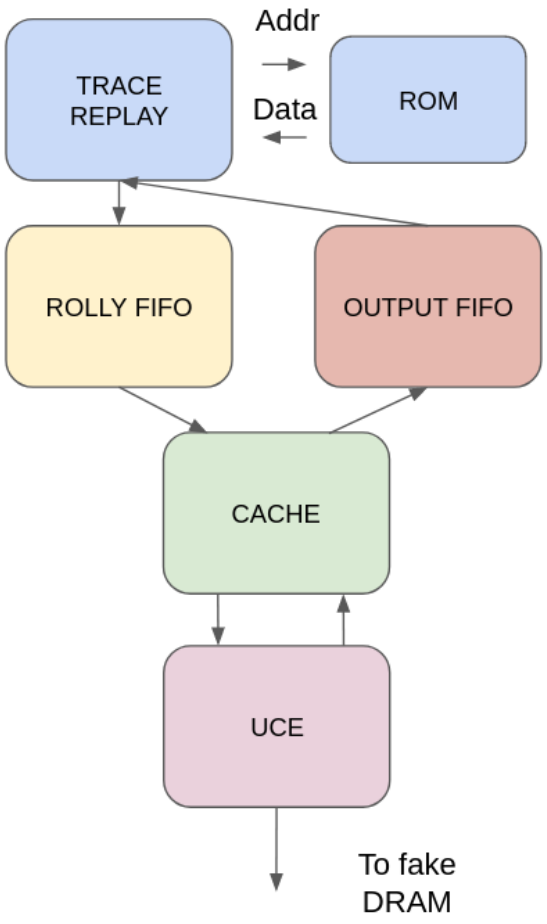

Figure 25: Example single D-Cache testbench

The user can create a trace file with multiple instructions indicating different input combinations. A Python script converts the trace file into a ROM that can be accessed using addresses by the trace replay as shown. The trace replay then decodes the trace data and takes the appropriate

Instruction Code	Description
0000	Wait for one cycle
0001	Send data, that is the rest of the instruction
0010	Receive data, compare with rest of instruction
0011	Trace replay asserts its done signal
0100	Finish simulation
0101	Wait for cycle counter to reach 0
0110	Set a value for the cycle counter, rest of instruction is the 16-bit value (clipped if larger)

Table 13: Trace replay instruction set

action as given by the table.

An issue queue (called rolly FIFO in the figure) stores the commands sent by the trace replay and sends them to the cache. An output FIFO stores the hit data from the cache, and the trace replay module can read this to perform the value comparison. This output FIFO is required because the test could perform a stream of loads, all of which hit in the cache, before moving to the value comparisons. This might be a result of our testing strategy and could be a point of exploration in future work.

The cache is instantiated along with the UCE. The UCE connects to the "fake" (non-synthesizable) DRAM, which can be preloaded with values if necessary.

Version 2 of the testbench added support for testing multiple caches along with the cache coherence engine (CCE), as shown in Figure 26.

Each cache is connected to a trace replay module (with the ROM), along with the two FIFOs as before. Currently, all the trace files are the same, so the output is guaranteed to be validated by the trace replay module if the design is functional. A future extension would be to allow different trace files for each trace replay module, which would complicate the validation step since the order of operations to the same address would depend on the order in which the CCE services the request.

The differences between the single cache and multi-cache configurations are the cache engines used, the concentrators, and the CCE. The LCE-CCE setup is required in the multi-cache configuration to test the coherence between caches. There are three concentrator modules, one each for the request, command, and response networks. The request and response concentrator modules

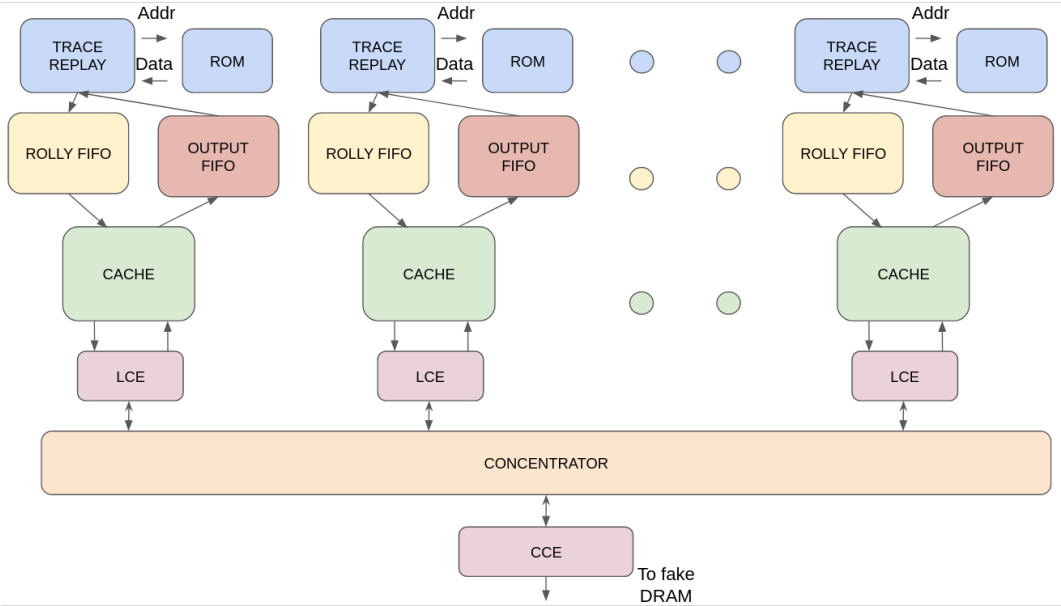

Figure 26: Testbench supporting multiple caches

essentially provide N->1 arbitration between the caches (assuming there are N caches in the design). The command concentrator module provides (N+1)->N arbitration (since the LCEs and the CCE can issue commands). The CCE connects to the "fake" DRAM, which can be preloaded with values if necessary.