Coyote: A peek into the future of RISC-V Super-computers

Regina Mumbi Gachomba

Handling sparse workloads in HPC architectures can be a bummer, but Coyote, a performance modelling system and emulation platform, is here to the rescue! It proposes and analyses the viability of a decoupled architecture (ACME) in which the memory operations are separated from the computation.

Picture this: you want to revamp your old desktop computer to include new or updated features that will optimize its performance. Lucky enough, your system has a PCI Express bus; an expansion slot to which you can add graphic cards, drives etc. However, there is a threshold to how much you can reconfigure your system because some components are heavily dependent on the software. Furthermore, these components that you will add to your system will probably cost a lot, and if you don't know how to go about the reconfiguration, you will need to pay for the service as well. Now imagine doing this for a supercomputer (a system that usually fits a room. . . or two). It would take a while. . . and cost a fortune!

Nowadays, the flexibility of software-hardware co-design is a highly sought-after feature in most computing systems. The need for this flexibility spans across speed, scale, cost, innovation and so much more. To address this, Mare Nostrum Experimental Exascale Platform (MEEP) proposes a flexible FPGA based emulation platform, designed to explore hardware-software co-designs for future RISC-V supercomputers.[1] It would be easier to think of MEEP as a prototype that can be used to test the viability of a certain framework or architecture.

To demonstrate MEEP's emulation capabilities, an accelerator architecture called the Accelerated Compute and Memory Engine (ACME) will be incorporated into MEEP. ACME employs a decoupled architecture, which offers a clean-cut approach to how sparse and dense workloads are handled in computation. While dense HPC workloads are compute-bound, sparse workloads are memory-bound, as the vector elements need to be gathered/scattered using multiple requests. Data or workloads are considered sparse when certain expected values in a dataset are missing, which is a common phenomenon in the domain of High-Performance Computing (HPC) and High-Performance Data Analytics (HPDA). In addition, this data needs to arrive in time to the required destination. To achieve this in MEEP, a tool that can run and test various data movement and marshalling policies to decide on the optimal one, based on performance, is needed. Enter Coyote.

Coyote is a new open source, execution-driven simulation tool, that is capable of performance modelling and analysis of the ACME architecture before it can be cast onto silicon.[1] Coyote is founded on existing simulators (Spike and Sparta) and is being improved to cater for their shortcomings, especially in the High Performance Computing domain where the number of resources to be simulated is high, hence making it a powerful modelling tool. MEEP and coyote is a reference to the cartoon, Wile E. Coyote and the roadrunner, where Coyote aspires to match the roadrunner's using his wits. This

is precisely the goal of our simulator: To identify the best data movement and reordering policies that ensure a better leveraged access pattern for applications that will ensure faster and more efficient computing.

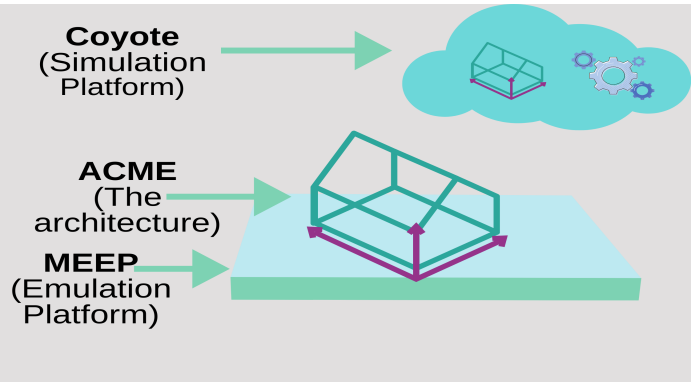

Figure 1: The fantastic three

Introduction – The Memory Tile

During the tenure of this internship, my work has been focused on setting up the basic functionality of the memory tile on the Coyote simulator. The memory tile houses the MCPU (Memory CPU), which can be loosely described as the 'intelligence' of the tile, responsible for organizing resources that are needed to perform the different memory operations.

These resources are obtained from the microengine, the vector address generator (VAG) and within the MCPU itself. The microengine is responsible for generating transactions for the instructions, whereas the vector address generator generates the memory requests. Another impressive feature of this memory tile is that it allows the re-usability of some already implemented functionalities. For example, a scalar load operation is handled like a unit stride vector load with a loop iteration of 1.

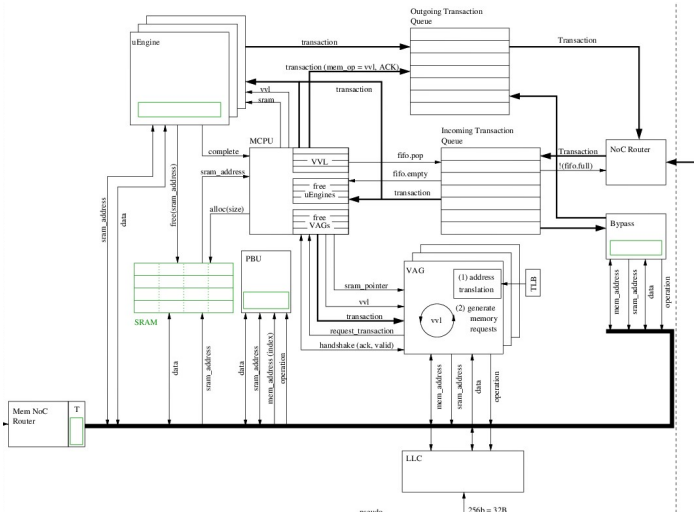

Figure 2: The architecture of the memory tile

Objectives

The primary objective was to understand how instructions, commands and data packets are to be received into the memory tile. Coyote allows us to create endless possibilities, even unrealistic ones, so it was important to also define the hardware constraints in the beginning so that we could obtain realistic results.

Once an overall understanding of the architecture was established, our goal was to simulate the different load and store operations and analyse the output and performance.[3] The types of scalar and vector operations to be simulated are as follows:

Unit stride: for vector elements that are located/stored next to each other in memory

Non-unit stride: for vector elements that are accessed at regular intervals, e.g., every second or third element

Indexed: for vector elements that are accessed by their indexed address. Quite similar to non-unit stride

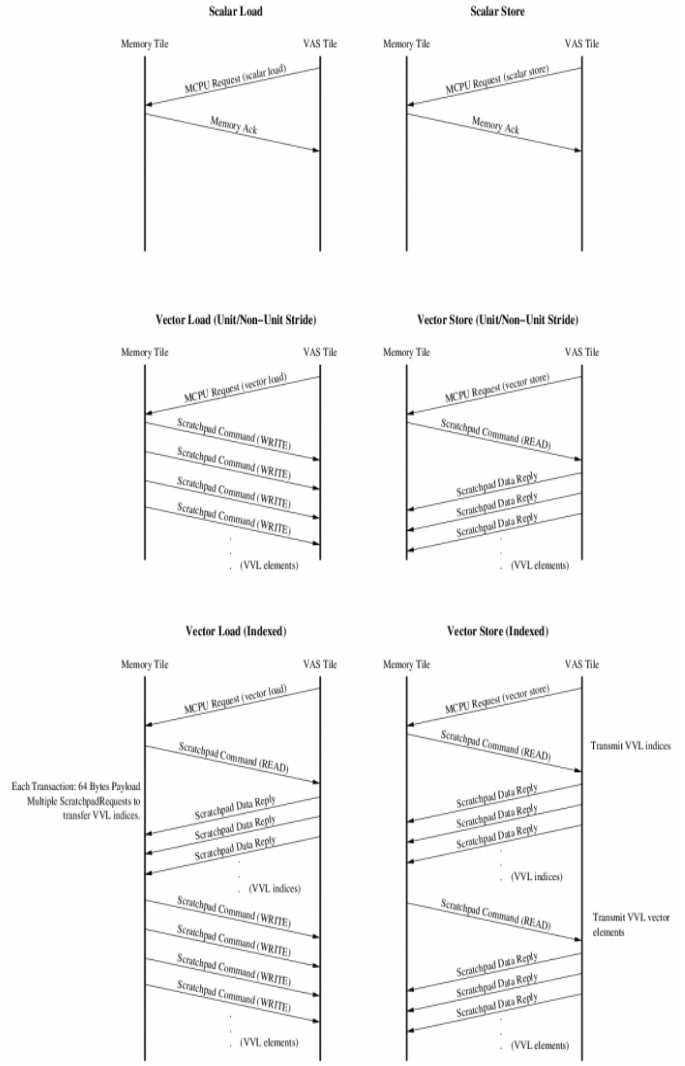

Figure 3: Communication sequences for various scalar and vector operations

Technical Work and Timeline

The first task involved setting up the bypass for scalar memory operations that do not need any resources from the MCPU. The scalar memory operations are handled as cache requests that are forwarded to the memory controller though the bus queue.

The MCPU functionalities that would cater for the load and store of vector operations were then set up. This entailed the handle function, controller cycles and the queues.

The Memory Tile and the VAS tile communicate using NoC messages. When these messages arrive at the memory tile, they can have either of the four payloads namely,

MCPU Instruction: vector memory operations

MCPUsetVVL: instruction to set virtual vector length (VVL) and sends it back to VAS tile

Scratchpad Requests: Commands such as free, allocate, read, write for the scratchpad

Cache Requests scalar memory operations and memory requests going to the MC

Each of these payloads are handled differently. For that reason, we use an overloaded function called `handle` in the source code. The handle function determines the queue and the controller cycle that will schedule the operation.

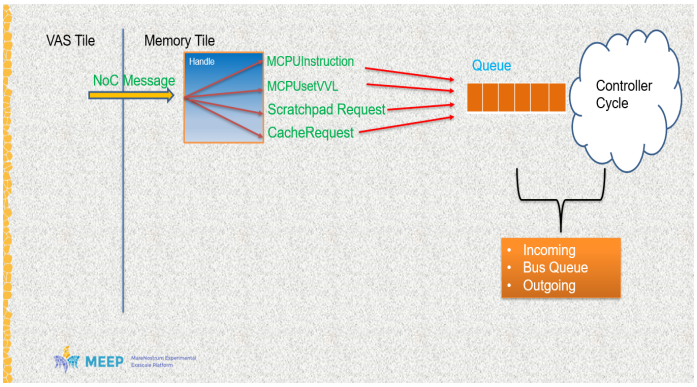

Figure 4: An illustration of how the overloaded function `handle` works

The controller cycle for incoming transactions essentially describes what happens in the MCPU. If an MCPU instruction is peeked from the queue, we first determine whether it is a load or a store. If it is a load, a scratchpad request to allocate some space in the scratchpad is created and sent back to tile (refer to figure 3). If it is a store, then a scratchpad command to load from the scratchpad is created. If the memory operation at the front of the incoming happens to be a scratchpad reply, it means that we had earlier sent a request to the scratchpad , and it is an indication that the subsequent computations can now be carried out.

The controller cycle for memory requests going through the bus queue schedules memory requests in the form of cache requests that will be forwarded to the Memory Controller, while the controller cycle for outgoing transactions schedules outgoing transactions from the MCPU, bypass and microengine.

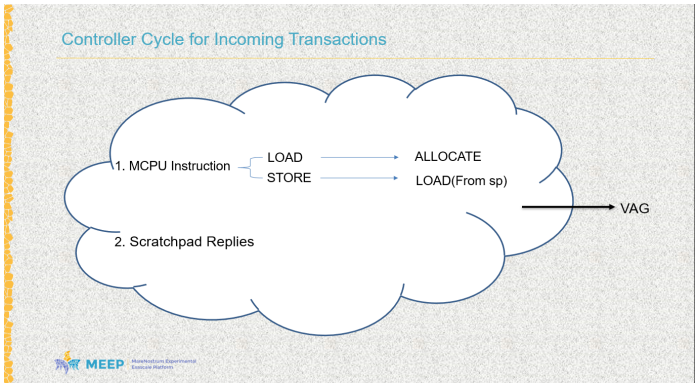

Figure 5: Controller cycle for incoming transactions

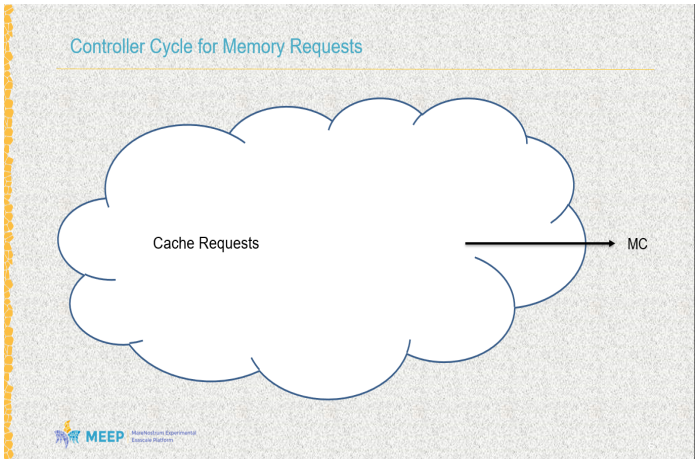

Figure 6: Controller cycle for memory requests generated by the VAG and sent to the memory controller

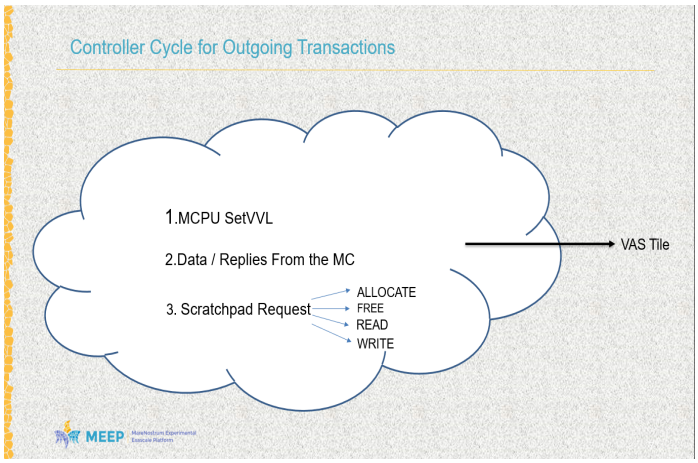

Figure 7: Controller cycle for outgoing transactions

A template class with the basic methods of a queue and boolean values that check the availability of the controller cycles was created. All the queues implement this template class, which is in the form of a header file (`Bus.hpp`). The reason for this was to reduce code replication of the push and pop functions of all the queues in the memory tile.

In addition, The MCPUwrapper is initialised with a hash map (unordered map) to keep track of the cache requests and scratchpad requests that are generated from each MCPU Instruction. The MCPU instructions are consequently initialised with an ID parameter that we use as the key in the hash map.

Results

The Coyote simulator can now carry out both scalar vector load and store operations in the memory tile. Address, data, and control packets can be sent to the accelerator tile and are received in the memory tile. However, there is still some debugging to be done. For instance, when scratchpad requests return from the memory controller, they do not arrive in the same order they were sent. Although the simulator has control of how memory operations are scheduled in the queues, it is still not well-defined how the operations are ordered when they are in the MC, MCPU or the microengine.[5]

```
93489   Core 0 simulated 72968 instructions
93490   Core 1 simulated 70447 instructions
93491   Core 2 simulated 70473 instructions
93492   Core 3 simulated 70456 instructions
93493   Total simulated instructions 284344
93494   The monitored section took 0 nanoseconds
```

Figure 8: This is the number of simulations that were run when coyote was run with four cores

```
The cmd is ../../apps/mt-matmul-vec/matmul
1
Calling sim with smart mcpu 1
Created
2
3
48: memory_cpu: receiveMessage_noc_: Received from NoC: Src: 0, Dest: 0, Type: 1
[0;36m48: memory_cpu: handle CacheRequest: 0x1000 @ 1, coreID: 0x0[0;0m
31: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x1000 @ 1, coreID: 0x0
109: memory_cpu: Returned from MC: instrID: 0 - CacheRequest using the Bypass: 0x1000 @ 1, coreID: 0x0
[0;32m10a: memory_cpu: controllerCycle_outgoing_transaction: Sending to NoC: Src: 0, Dest: 0, Type: 5[0;0m
```

Figure 9: Cache request transaction

```
325  Issuing MCPU VVL from core 0 and tile 0
326  Issuing MCPU VVL from core 1 and tile 0
327  Issuing MCPU VVL from core 2 and tile 0
328  Issuing MCPU VVL from core 3 and tile 0
329  1406: memory_cpu: receiveMessage_noc_: Received from NoC: Src: 0, Dest: 0, Type: 6
330  [1;36m1406: memory_cpu: handle MCPUSetVVL: AVL: 20, VVL: 20, lmul: 1, w: 8, coreID: 0[0;0m
331  1407: memory_cpu: receiveMessage_noc_: Received from NoC: Src: 0, Dest: 0, Type: 6
332  [1;36m1407: memory_cpu: handle MCPUSetVVL: AVL: 20, VVL: 20, lmul: 1, w: 8, coreID: 1[0;0m
333  [0;32m1407: memory_cpu: controllerCycle_outgoing_transaction: Sending to NoC: Src: 0, Dest: 0, Type: 6[0;0m
334  1408: memory_cpu: receiveMessage_noc_: Received from NoC: Src: 0, Dest: 0, Type: 6
335  [1;36m1408: memory_cpu: handle MCPUSetVVL: AVL: 20, VVL: 20, lmul: 1, w: 8, coreID: 2[0;0m
336  [0;32m1408: memory_cpu: controllerCycle_outgoing_transaction: Sending to NoC: Src: 0, Dest: 0, Type: 6[0;0m
337  1409: memory_cpu: receiveMessage_noc_: Received from NoC: Src: 0, Dest: 0, Type: 6
338  [1;36m1409: memory_cpu: handle MCPUSetVVL: AVL: 20, VVL: 20, lmul: 1, w: 8, coreID: 3[0;0m
339  [0;32m1409: memory_cpu: controllerCycle_outgoing_transaction: Sending to NoC: Src: 0, Dest: 0, Type: 6[0;0m
340  [0;32m140a: memory_cpu: controllerCycle_outgoing_transaction: Sending to NoC: Src: 0, Dest: 0, Type: 6[0;0m
```

Figure 10: setVVL transaction

```
412  198e: memory_cpu: controllerCycle_incoming_transaction: 0x80042e00 @ 1978 Op: 0x0 SubOp: 0x0, width: 0x8, coreID: 0x2,
413  198e: memory_cpu: memOp_unit: noepr: 8, re: 20, address: 80042e00
414  198e: memory_cpu: memOp_unit: noepr: 8, re: 18, address: 80042e40
415  198e: memory_cpu: memOp_unit: noepr: 8, re: 10, address: 80042e80
416  198e: memory_cpu: memOp_unit: noepr: 8, re: 8, address: 80042ec0
417  198f: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042c80 @ 1978, coreID: 0x0
418  198f: memory_cpu: controllerCycle_incoming_transaction: 0x80042f00 @ 1978 Op: 0x0 SubOp: 0x0, width: 0x8, coreID: 0x3,
419  198f: memory_cpu: memOp_unit: noepr: 8, re: 20, address: 80042f00
420  198f: memory_cpu: memOp_unit: noepr: 8, re: 18, address: 80042f40
421  198f: memory_cpu: memOp_unit: noepr: 8, re: 10, address: 80042f80
422  198f: memory_cpu: memOp_unit: noepr: 8, re: 8, address: 80042fc0
423  1990: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042cc0 @ 1978, coreID: 0x0
424  1991: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042d00 @ 1978, coreID: 0x1
425  1992: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042d40 @ 1978, coreID: 0x1
426  1993: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042d80 @ 1978, coreID: 0x1
427  1994: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042dc0 @ 1978, coreID: 0x1
428  1995: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042e00 @ 1978, coreID: 0x2
429  1996: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042e40 @ 1978, coreID: 0x2
430  1997: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042e80 @ 1978, coreID: 0x2
431  1998: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042ec0 @ 1978, coreID: 0x2
432  1999: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042f00 @ 1978, coreID: 0x3
433  199a: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042f40 @ 1978, coreID: 0x3
434  199b: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042f80 @ 1978, coreID: 0x3
435  199c: memory_cpu: controllerCycle_mem_request: Sending to MC: 0x80042fc0 @ 1978, coreID: 0x3
```

Figure 11: Unit-stride vector transaction in which the data for a vector load is returned in multiple NoC transactions.

Future Work

At the moment, the analysis of Coyote's performance is done mostly on the command line. This is expected to shift to the use of a visualization tool, Paraver, which is a flexible data browser developed at BSC, used to capture the behaviour of parallel programs, and give a quantitative analysis of the problem.[4] Due to its flexibility, it perfectly meets the needs of Coyote in testing the large number of data management policies with different workloads used in HPC.

References

[1] Fell, A., Mazure, D. J., Garcia, T. C., Perez, B., Teruel, X., Wilson, P., & Davis, J. D. (2021). The MareNostrum Experimental Exascale Platform (MEEP). Supercomputing Frontiers and Innovations, 8(1), 62-81

[2] Perez, B., Fell, A., & Davis, J. D. (2021, February). Coyote: An Open Source Simulation Tool to Enable RISC-V in HPC. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 130-135).

[3] MEEP Wiki https://wiki.meep-project.eu/index.php/The_Coyote_simulator

[4] Paraver: a flexible performance analysis tool https://tools.bsc.es/paraver

[5] GitLab repository https://gitlab.bsc.es/meep/meep-performance-modelling/coyote-tool/coyote-sim/-/tree/basicVersionMemoryTile

PRACE SoHPCProject Title

Analysis of Data Management Policies in HPC architectures

Regina Mumbi Gachomba

PRACE SoHPCSite

Barcelona Supercomputing Center, Spain

PRACE SoHPCAuthors

Regina Mumbi Gachomba, Ankara Yildirim Beyazit University, Turkey

PRACE SoHPCMentors

Borja Perez, Barcelona Supercomputing Center, Spain
Alexander Fell, Barcelona Supercomputing Center, Spain

PRACE SoHPCContact

Regina Mumbi Gachomba, Ankara Yildirim Beyazit University
Phone: +90 553 7108223
E-mail: mumbigachomba254@gmail.com

PRACE SoHPCSoftware applied

C++, GitLab, Spike, Sparta, Coyote

PRACE SoHPCMore Information

https://github.com/borja-perez/Coyote

PRACE SoHPCAcknowledgement

I would like to express immense gratitude to my mentors Alexander Fell, Rahul Shrivastava, Borja Perez and Teresa Cervero for their valuable guidance through my tasks in the project. I also had the great pleasure of working with my project partner, Aneta Ivaničová and the whole MEEP team at BSC. Thanks to PRACE for organising this valuable programme.

PRACE SoHPCProject ID

2101