Contents

1 FAT Allocator without the TLB

1.1 Abstract

This document explores an extension of the FAT allocator approach to memory management in RISC-V Toooba[1]. CHERI introduces a fine-grained memory protection mechanism by embedding bounds and permissions directly within pointers. Leveraging this model, we propose a system in which offsets are stored within the pointer itself, enabling direct memory access without reliance on traditional address translation mechanisms such as the TLB. This method facilitates the design of a block-based memory allocator within physically contiguous memory. The sections expanded below are the technique expected to be used and the evaluation criteria for the following experiment.

1.2 Research questions

1. How can embedding offsets within a FAT pointer (i.e CHERI pointer) improve memory accesses for a block-based allocator for the RISC-V CHERI modified Toooba architecture ?

2. To what extent does eliminating TLB impact the reduction in CPU clock cycles and memory access latency ?

1.3 Proposed approach

FAT-Pointers based range addresses, combined with the capabilities of the CHERI architecture, introduce bypassing the TLB hierarchy by incorporat-

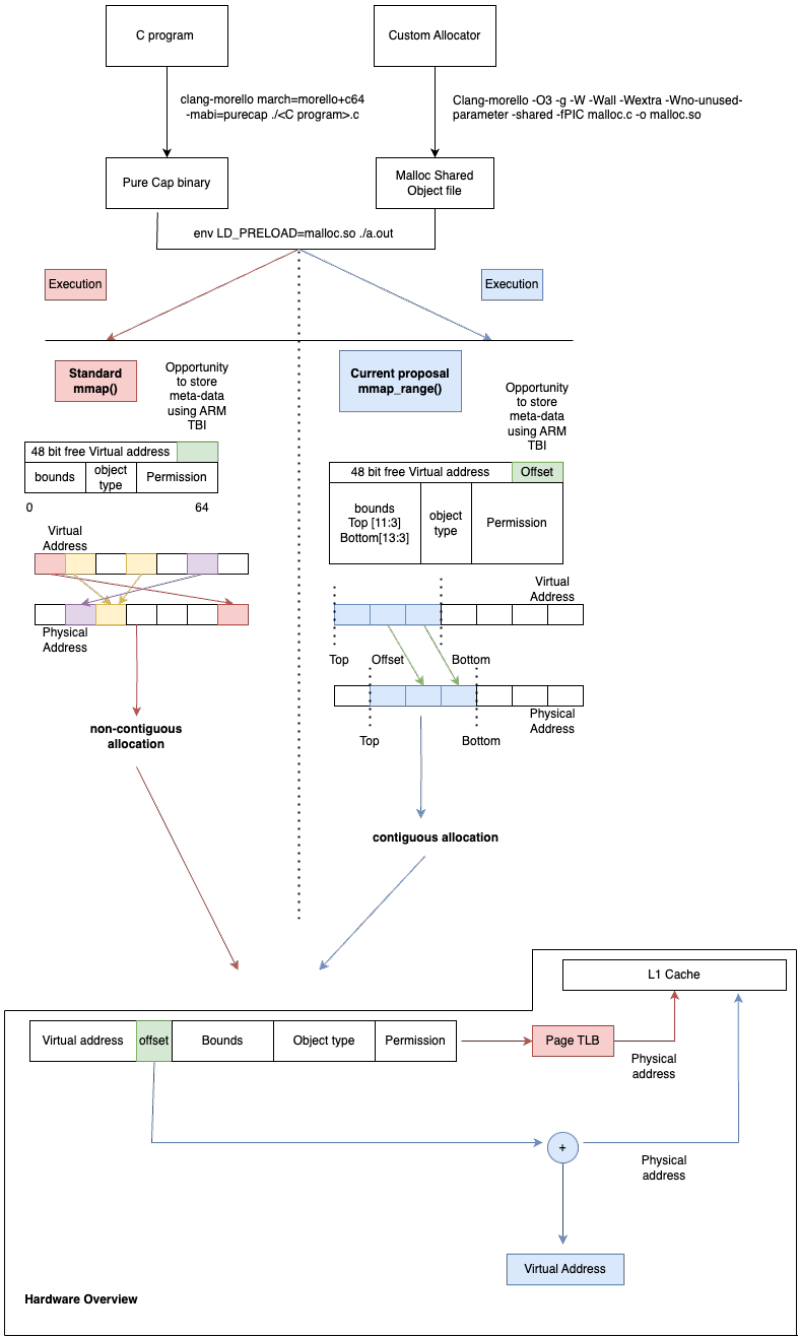


Figure 1: FAT pointer implementation with RISCV CHERI Toooba to strip the requirement of requiring a TLB.

ing additional metadata with memory pointers. This enhanced architecture utilises concepts such as FlexPointer[2], Range Memory Mapping (RMM)[3] to manage memory as customised sizes rather than groups of fixed page sizes. Range addresses play a pivotal role within this framework, defining memory regions bounded by a starting address (Upper) and an ending address (Lower). These range addresses are encoded within FAT-pointers, allowing for precise control over memory regions. The functionality of ranges encompasses several key aspects:

- Creation of Physically Contiguous Memory Ranges.
- Encoding Ranges as Bounds to the Pointer.
- Instrumenting Block-Based Allocators with the FAT Pointer.

In figure 1, the green-highlighted section marks the unused space between the 48th and 64th bits within the FAT-pointer. This area of unused bits presents an opportunity to store additional metadata, potentially enhancing the capabilities of the memory management system.

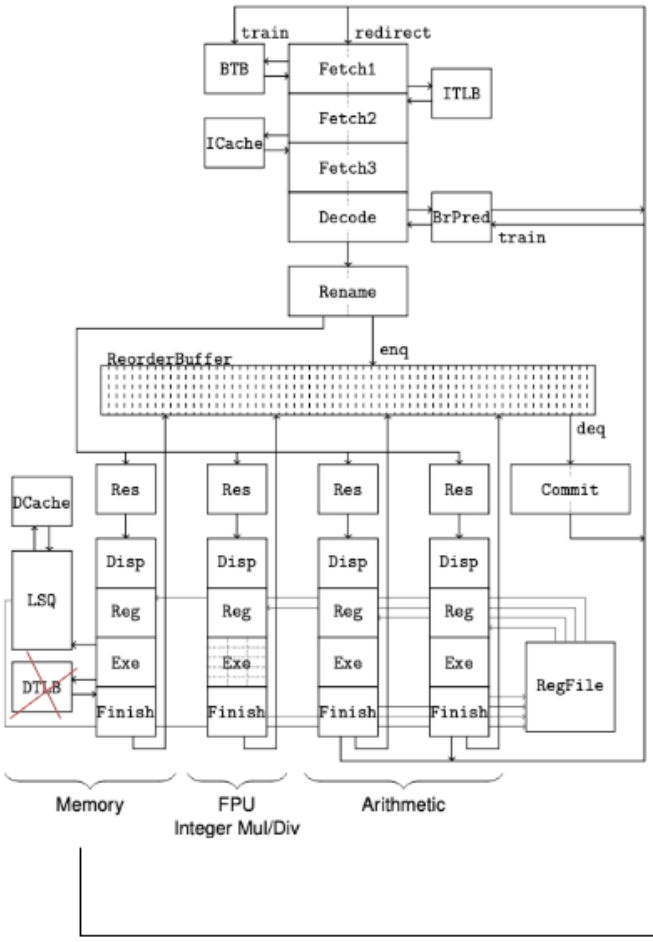
1.3.1 Implementation

The figure 2 above illustrates the Toooba processor, showcasing all available pipelines to provide a broad overview of the architecture. On the right-hand side, the pseudo-code of the BlueSpec[4] implementation highlights the modifications that we will need to do to the memory pipeline in order to bypass the use of the Data TLB and instead utilise the offset from the pointer. The primary focus will be on the memory pipeline.

- Each pipeline will contain a reservation station that will accept relevant instructions from the rename stage and buffer them until their register dependencies have been resolved.

- Following this, the dispatch stage will update the necessary state in the reservation station and forward the instruction to the rest of the pipeline.

- The register read stage will then latch the required values from the physical register file, or from forwarding paths if those values are already available.



Algorithm 1 Modified Memory Pipeline Stages

```
1:  procedure doDispatchMem
2:      Enqueue into regToExeQ:
3:        mem_func ← x.mem_func
4:        imm ← x.imm
5:        tag ← x.tag
6:      ...
7:        rVal1, rVal2, vaddr, cap_checks, origBE, shiftBEData
8:      ...
9:        spec_bits ← dispToReg.spec_bits
10: end procedure

11: procedure doExeMem
12:     regToExe ← regToExeQ.first
13:     regToExeQ.deq()
14:     ...
15:     shiftBE ← DataMemAccess(x.shiftBEData)
16:     if x.origBE == TagMemAccess then
17:         shiftBE ← TagMemAccess
18:     end if
19:     ddc ← cast(inIfc.scaprf_rd(scrAddrDDC))
20:     Compute accessByteCount
21:     if x.origBE == TagMemAccess then
22:         accessByteCount ← CLineNumMemDataBytes
23:     end if
24:     ...
25:     Call procPAddr(x.vaddr)
26: end procedure

27: procedure doFinishMem
28:     ...
29:     PAddrResp ← procPAddr.procResp
30:     x ← PAddrResp.inst
31:     {paddr, expCause, allowCapPTE} ← PAddrResp.resp
32:     cause ← Invalid
33:     if expCause matches Valid c then
34:         cause ← Valid(Exception(c))
35:     end if
36:     ...
37: end procedure
```

Figure 2: Toooba processor with pseudo code change to bypass DataTLB.

- The execute stage will perform the specified operation. In the case of the memory pipeline, it will additionally calculate addresses by adding the immediate value to the read register where applicable, and will manage interactions with the Load/Store Queue.

- Originally, the memory pipeline's execute stage also interacted with the Data TLB, but this will be replaced by a translation mechanism encoded in the offset, which will perform the necessary address translations.

- The Load/Store Queue will continue to receive signals from various parts of the processor. A slot will be requested during the rename stage, and the memory pipelines finish stage will commit the access once all exceptions have been resolved.

- Whereas this stage was previously triggered by a TLB response, it will now operate without walking the TLB hierarchy. Instead, the physical address will be computed directly by adding the offset to the virtual address in a single clock cycle, after which any exceptions will be handled appropriately.

Importantly, the finish stage will not require the actual memory access to have taken place for a load instruction, enabling the core to support out-of-order execution. Memory responses will be processed asynchronously, potentially after the commit stage, with the returned data written to the appropriate physical register. Generally, the memory pipeline will remain unchanged, except to support the new access types, incorporate the Data TLB bypass mechanism, and include the necessary capability checks to ensure that accesses are properly authorised.

1.4 Proposed evaluation

The evaluation of the proposed FAT allocator implemented using CHERI-enhanced pointers on the RISC-V Toooba architecture aims to assess both its performance characteristics and architectural implications, particularly in the context of removing the Translation Lookaside Buffer (TLB) from the memory access pathway. The evaluation methodology is designed to address

the research questions with a focus on improvements in memory access and reductions in latency for address translation.

1.4.1 BlueSpec simulator

To evaluate the proposed FAT pointer based memory management architecture, we plan to conduct simulations using the Bluespec SystemVerilog (BSV)[5] framework. Bluespec provides a cycle-accurate hardware simulation environment that allows precise modelling of architectural behaviour, including custom memory pipelines, capability checking, and physical address translation bypass mechanisms.

1.4.2 Performance Metrics

To quantify the performance benefits of the proposed system, the following metrics will be investigated:

- **CPU clock cycles per memory access:** This metric evaluates the computational overhead involved in memory access operations. A comparative analysis will be conducted against conventional TLB-based systems with the hypothesis that the proposed architecture will exhibit reduced cycle counts due to the elimination of virtual-to-physical address translation.
- **Memory access latency:** The latency associated with memory access will be measured under various workload conditions. By embedding offset information directly within pointers and bypassing the TLB, the proposed approach is expected to achieve significantly lower latency compared to traditional systems.
- **Instruction path length:** An analysis of the number of instructions executed during allocation, deallocation and memory access operations will be performed to determine whether the additional logic required for handling FAT pointers introduces meaningful overhead.

1.4.3 System Resource Utilisation

- **Cache behaviour** The proposed system's influence on cache performance will be assessed through performance counters, focusing on metrics such as cache hit and miss rates. It is expected that physically contiguous allocations will result in improved spatial locality and cache efficiency.

- **TLB miss rate (baseline comparison)** Although the system does not utilise a TLB, comparative analysis will be performed with traditional TLB-based systems to quantify the performance cost typically incurred through TLB misses, thereby contextualising the advantage of their removal.

1.4.4 Benchmarking Against Baseline Architectures

A series of micro and macro benchmarks will be employed to compare the FAT allocator with traditional memory allocators that rely on virtual memory and TLBs. Micro benchmarks will include fine-grained tests of memory operations, while macro benchmarks will involve application-level scenarios such as numerical computing and dynamic data structure manipulation, providing a comprehensive assessment of system-level impact.

References

[1] P. Rugg, "Efficient spatial and temporal safety for microcontrollers and application-class processors," 2022. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/353468

[2] D. Chen, D. Tong, C. Yang, J. Yi, and X. Cheng, "Flexpointer: Fast address translation based on range tlb and tagged pointers," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3579854

[3] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 6678. [Online]. Available: https://doi.org/10.1145/2749469.2749471

[4] R. S. Nikhil and Arvind, "What is bluespec?" *SIGDA Newsl.*, vol. 38, no. 23, p. 1, Dec. 2008. [Online]. Available: https://doi.org/10.1145/1862867.1862868

[5] O. Arcas-Abella and N. Sonmez, "Bluespec SystemVerilog," in *FPGAs for Software Programmers*, D. Koch, F. Hannig, and D. Ziener, Eds. Cham: Springer International Publishing, 2016, pp. 165–172. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_9