Contents

1 Abstract

The future approach on RISC-V Tooba[**?**] involves storing the offset directly within the pointer. This is possible due to CHERIs capability model, which supports fine-grained memory protection and can encode bounds within pointers. Utilizing Bounds in CHERI for Block-Based Allocation: CHERI capabilities allow pointers to carry metadata about memory bounds, providing hardware-enforced memory safety. By encoding the offset and bounds within the pointer, the system can directly access memory without needing intermediate translations via the TLB. This enables the implementation of a block-based allocator that can efficiently manage memory allocations and deallocations within defined bounds. Bypassing the TLB in RISC-V Tooba.

2 Bypassing the TLB (Literature)

2.1 Range Memory Mapping (RMM)

Redundant Memory Mappings (RMM)[Karakostas et al.] enhance memory management by introducing an additional range table that pre-allocates contiguous physical pages for large memory allocations, creating ranges that are both virtually and physically contiguous. This approach simplifies address translation within these ranges by adding an offset, similar to Direct Segment, but RMM supports multiple ranges and operates transparently to programmers, requiring no source code modifications. The range table, separate from the conventional page table, holds the mappings for these large allocations. To determine which range an address belongs to, RMM compares the address against all range boundaries, a process that is computationally expensive and therefore performed only after an L1 TLB miss. To optimize this, RMM uses a range TLB (RTLB) to quickly identify if

an address falls within any pre-allocated range, facilitating efficient translation and reducing overhead. Range mapping works alongside the paging system by generating TLB entries on TLB misses and still performing TLB lookups for each virtual address translation. Unlike traditional segmentation mechanisms, range mapping activates a range lookaside buffer (RTLB) located with the last level TLB upon a miss. The hardware TLB miss handler then searches the RTLB with the last level TLB upon a miss. The hardware TLB miss handler then searches the RTLB for the miss address and, if found, generates a new TLB entry with the physical address derived from the base virtual address and range offset, along with permission bits. If the RTLB also misses, the system defaults to a standard page walk while a range table walker simultaneously loads the range into the RTLB in the background, avoiding delays in memory operations. The RTLB, functioning as a fully associative search structure, ensures that most last level TLB misses are handled efficiently by range mapping, reducing the need for costly page table walks.

<Figure>

In Figure 5.1 illustrates the structure and logic of the range TLB, which comprises N entries (e.g., 32). Each entry in the range TLB includes a virtual range and a corresponding translation. The virtual range contains the BASEi and LIMITi values, defining the boundaries of the virtual address range. The translation part holds the OFFSETi, which is the difference between the starting point of the range in physical memory and BASEi, as well as the protection bits (PB). Each range TLB entry is equipped with two comparators to facilitate lookup operations. When accessing the range TLB in parallel with the L2 TLB, after a miss at the L1 TLB, the hardware compares the virtual page number that missed in the L1 TLB against all ranges in the range TLB, checking if BASEi <= virtual page number < LIMITi. On a hit, the range TLB returns the OFFSETi and protection bits for the corresponding range translation, and calculates the corresponding page table entry for the L1 TLB. It does this by adding the requested virtual page number to the hit OFFSETi value to produce the physical page number, and copying the protection bits from the range translation. If there is a miss, the hardware fetches the corresponding range translation if it exists from the range table.

2.2 FlexPointer

FlexPointer[Chen et al.] builds upon the range translation concepts found in RMM and Direct Segment. A range consists of contiguous virtual pages

mapped to contiguous physical pages, with uniform protection bits, such as read, write, or execute. Defined by two addresses, BASE and LIMIT, a range is base-page-aligned and can have an arbitrary number of pages.Due to the contiguous nature of these pages, all addresses within a range share a common DELTA, calculated as ($\text{physical}_{\text{address}}$ - $\text{virtual}_{\text{address}}$). To translate a virtual address within a range, the processor simply adds DELTA to it.

A system employing range translations has three main components: (i) the creation of memory ranges, (ii) the management of range information, and (iii) the hardware that efficiently utilizes range translations. FlexPointer creates a range and assigns it a unique ID upon receiving a request for a large allocation. To ensure physical contiguity, it uses an eager paging strategy, which allocates physical pages at the time of the allocation request rather than on first access. This involves modifying memory management functions, such as malloc() and mmap(), to support eager allocation. Additionally, a kernel range table is used to record range translations, with mappings also maintained in the page table to ensure compatibility with other memory subsystems.

Similar to RMM, FlexPointer uses a range TLB to facilitate range translations in hardware. It can pass the range ID through the pointer tag, allowing the range TLB to operate in parallel with address generation. During a memory access, the processor uses the pointer tag to determine whether to search the range TLB or the page TLB, and the tag also guides which table to consult in the event of a TLB miss.

<figure>

In Figure 1.4 FlexPointer utilizes specialized hardware components and a streamlined workflow for efficient memory management. During address generation, the processor determines whether to query the range TLB or the page TLB based on the high-order bits of the address. If these bits are all 0s or 1s, indicating a regular page TLB operation, the address is translated accordingly post-generation. Alternatively, if the high bits suggest a range TLB operation, FlexPointer uses the range ID embedded in the pointer to directly access the range TLB. Each range ID in the TLB corresponds uniquely to a DELTA, simplifying translation. The processor adds this DELTA to the virtual address and performs a boundary check against the BASE and LIMIT of the range. If the address falls within this range, the sum of DELTA and the virtual address yields the correct physical address. Addresses failing the boundary check are directed to the page TLB for translation.

A range TLB miss occurs under two conditions: either no matching ID exists in the range TLB, or the address fails the boundary check of a dummy

entry. In response to a range TLB miss, the processor initiates a range table walk to retrieve the corresponding range translation. To optimize TLB lookup efficiency, FlexPointer maintains unique IDs within the range TLB. If the miss results from a sub-range mismatch, the updated translation replaces the previous sub-range entry rather than adding a new one. During the range table walk, the processor computes the address of the translation entry by adding (ID ń 5) to the base address of the range table. This base address, akin to storing the page table base in CR3, is part of the program context. If the address to be translated falls within the BASE and LIMIT range of the fetched entry, it is fetched into the range TLB, regardless of whether it is a dummy entry. For a dummy entry, the page table is queried for the correct translation, which is then inserted into the page TLB. If the address falls outside the (BASE, LIMIT) range and the entry is not the last sub-range (determined by the L bit), the processor retrieves the next sub-range entry with the NRID and repeats the process. However, if it is the last sub-range and a violation occurs, it indicates a safety breach.

3 High overview

FAT-Pointers based range addresses, combined with the capabilities of the CHERI (Capability Hardware Enhanced RISC Instructions) architecture, introduce robust memory safety and security features by incorporating additional metadata with memory pointers. This enhanced architecture utilizes concepts such as FlexPointer, Range Memory Mapping (RMM) to manage memory effectively.

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: By defining memory regions

that are physically contiguous, systems can achieve optimal memory access patterns, enhancing performance and efficiency.

- Encoding Ranges as Bounds to the Pointer: Integrating range bounds directly into

FAT-pointers enables the architecture to enforce memory access restrictions at the pointer level thus allowing tracking of memory ranges on a pointer level.

- Instrumenting Block-Based Allocators with Physically Contiguous Memory: The

integration of range-based memory concepts into memory allocation systems, such as block-based allocators, facilitates the efficient management and utilization of physically contiguous memory blocks, mitigating issues related to memory fragmentation. Figure 2.1 illustrates the methodology employed to leverage the CHERI 128-bit FAT Pointer scheme for facilitating block-based memory management on physically contiguous memory, which is depicted on the right side of the figure. This technique contrasts with the conventional mmap approach.

<figure>

In figure 2.1, 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. Here we explore how this additional metadata storage could be used to further optimize memory allocation.

3.1 Range creation and huge pages

In this implementation, memory ranges are established using bounds encoded within the FAT-pointer, adhering to the CHERI 128-bit bounds compression scheme[Woodruff et al.]. The memory chunk defined by the upper and lower bounds is always physically contiguous. Initially, a huge page of arbitrary size is allocated. Within this huge page, custom-sized memory segments are allocated using a custom-designed mmap function, which overrides the existing block-based mmap function. Once the memory is physically allocated through this custom mmap function, bounds are set to track the memory block, eliminating the need for traditional TLB usage for this purpose. Traditional TLB usage involves maintaining numerous TLB entries, often supplemented by an L2 TLB and other hierarchical structures, to translate virtual addresses to physical addresses. This approach requires multiple entries to handle various memory segments, leading to increased overhead and complexity in address translation. Conversely, the current approach streamlines this process by using a single TLB entry to translate

multiple addresses within a contiguous memory range. This reduces the number of required TLB entries, simplifying the translation process and improving efficiency. By consolidating address translations into a single TLB entry, this method minimizes the overhead associated with managing numerous TLB entries and leverages the bounds encoded within the FAT-pointer for efficient memory tracking and access. This approach allows for precise and efficient memory management within the allocated huge page.

<figure>

Figure 2.2 illustrates a straightforward use-case in which the dark pink line represents a single, large contiguous memory area, or huge page. Within this huge page, the orange and blue lines indicate two separate memory allocations equivalent to invoking malloc twice to allocate memory in distinct regions. This scenario simulates a block-based memory allocator operating within the confines of the huge page. The allocations leverage the bounds encoded in the FAT-pointer, ensuring tracking and efficient management of the allocated memory regions. By using the FAT-pointer bounds, this method maintains the integrity and contiguity of the allocated blocks within the huge page.