Contents

1 Fat-pointer Address Translations

Fat-pointer Address Translations, 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 implementation, 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.

Figure 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.

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. Here we explore how using Huge pages with CHERI bounds can reduce the number of TLB entries required.

The functionality of ranges encompasses several key aspects:

1.1 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. In this implementation, memory ranges are established using bounds encoded within the FAT-pointer, adhering to the CHERI 128-bit bounds compression scheme[1].

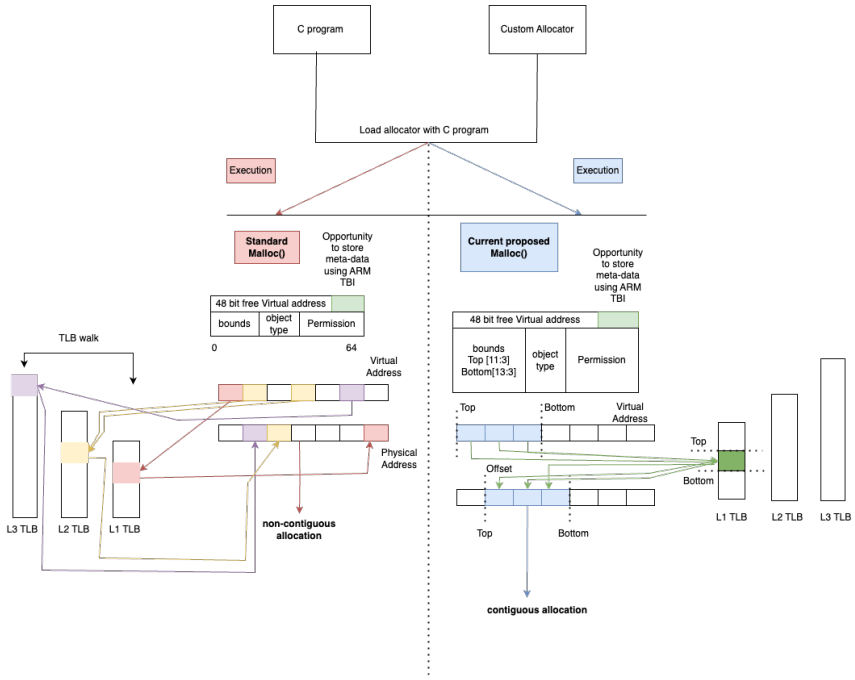

Figure 1: High overview architecture

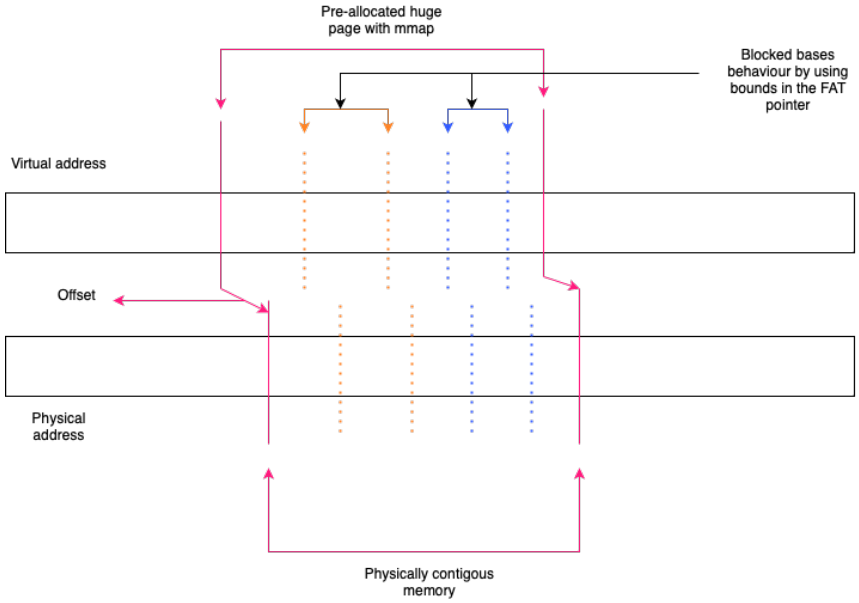

Figure 2: Range of memory

Figure 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.

1.2 Instrumenting Block-Based Allocators with Physically Contiguous Memory

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 stream- lines 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 3 illustrates a use-case of huge pages where the green line represents a sample access to read within a contigous space of physical memory. The dotted lines represents the bounds for that particular pointer access. Using bounds stored on the pointer a block based pattern can be reprecated on physically contigous memory.

1.3 Sample memory allocator Implementation

This section presents a straightforward memory allocator designed and implemented based on the principles outlined in our approach. The allocator consists of three core functions: InitAlloc, malloc, and free. The InitAlloc function initializes the memory pool, setting up the necessary data structures and metadata required for efficient memory management. The malloc function is responsible for allocating a contiguous block of memory of a spec-

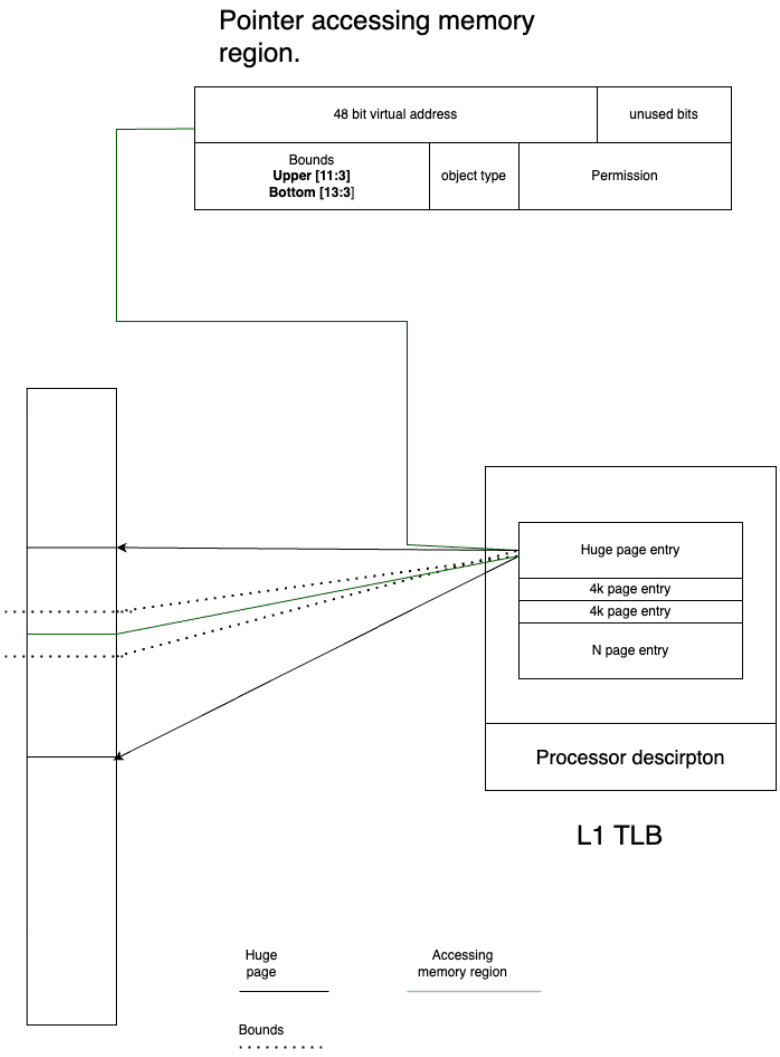

Figure 3: Fat-pointer Address Translations using huge pages

ified size, while the free function deallocates the memory, returning it to the pool for future use.

A notable feature of this malloc implementation is its compatibility with kernel modules, where it can be integrated as an alternative to the mmap system call. This integration ensures that memory allocations are physically contiguous, a critical requirement for certain low-level operations and hardware interactions. By providing physically contiguous memory blocks, this allocator can serve as a foundational layer for standard block-based allocators, such as Jemalloc, enabling them to operate efficiently in environments where physical memory contiguity is essential.

Algorithm 1 Sample init alloc function to create a initial 1 GB huge page

```
1: function INIT_ALLOC
2:     sz ← 1 GB                                  ▷ Define pre-allocated memory size
3:     fd ← CREATE_LARGE_PAGE_MEMORY(sz)          ▷ Create shared memory
4:     ptr ← MAP_MEMORY(sz)                       ▷ Map memory region
5:     MallocCounter ← sz                         ▷ Initialize memory counter
6: end function
```

Algorithm 1 describes the initialization of physically contiguous memory through the use of huge pages, a mechanism supported by modern architectures to optimize memory management. The algorithm begins by allocating a fixed block of 1 GB of physically contiguous memory. This decision is driven by the architectural constraints of contemporary systems, particularly ARM-based CPUs, where 1 GB represents the largest supported page size. By leveraging huge pages, the algorithm reduces the overhead associated with page table management and enhances memory access efficiency, which is critical for performance-sensitive applications and kernel-level operations.

When the malloc function is invoked, the algorithm employs an eager allocation strategy for physical memory. This is achieved through the use of the SetBounds mechanism, which constructs a FAT-pointera specialized pointer that encodes both the start and end addresses of the allocated memory region within the pointer itself. The start and end addresses correspond to the size of the memory block requested by malloc. This approach introduces a novel method of memory tracking, where the bounds of the allocated region are explicitly encoded in the address, enabling efficient monitoring and management of memory usage.

Furthermore, this design leverages shared huge page TLB (Translation

Algorithm 2 Sample malloc implementation

```
1: function MALLOC(sz)
2:     sz ← ALIGN_UP(sz, MAX_ALIGNMENT)     ▷ Align size to max alignment
3:     MallocCounter ← MallocCounter − sz ▷ Update remaining memory
4:     ptrLink ← &ptr[MallocCounter]          ▷ Calculate pointer address
5:     ptrLink ← SET_BOUNDS(ptrLink, sz)    ▷ Set bounds for memory safety and to track the length of the pointer
6:     return ptrLink                         ▷ Return allocated memory pointer
7: end function
```

Lookaside Buffer) entries to map and track memory addresses. By encoding bounds directly into the address, the algorithm ensures that memory accesses remain within the allocated region, thereby enhancing safety and reducing the risk of out-of-bounds errors. This innovative use of FAT-pointers and shared TLB entries not only aligns with the principles of efficient memory management but also demonstrates a practical application of huge pages in modern architectures, offering a robust solution for physically contiguous memory allocation.

Algorithm 3 Sample free implementation

```
1: function FREE(ptr)
2:     len ← GET_LENGTH(ptr) ▷ Get length of memory block from the defined bounds
3:     UNMAP(ptr, len)                        ▷ Release memory block
4: end function
```

The memory deallocation mechanism in the proposed allocator is facilitated by the FAT-pointer structure introduced in the malloc algorithm. When the free function is invoked, it utilizes the metadata embedded within the FAT-pointer to determine the range and size of the allocated memory region. Specifically, the start and end addresses encoded in the FAT-pointer provide the necessary information to identify the exact memory block to be deallocated. This allows the allocator to precisely unmapped the corresponding memory region from the address space, ensuring efficient and accurate memory management.

By extracting the bounds and size directly from the FAT-pointer, the free function eliminates the need for additional metadata lookups or complex data structures, streamlining the deallocation process. This approach not

only enhances performance but also reduces the risk of memory leaks or fragmentation.

References

[1] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," vol. 68, no. 10, pp. 1455–1469. [Online]. Available: https://ieeexplore.ieee.org/document/8703061/