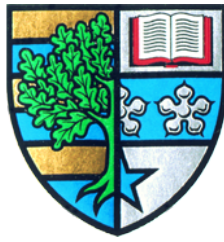


FAT-Pointer based range addresses



Akilan Selvacoumar

Mathematics and Computer Sciences
Heriot Watt University

Year 2 progression report of:
Doctor of Philosophy

June 2024

Abstract

The increasing disparity between application workloads and the capacity of Translation Lookaside Buffers (TLB) has prompted researchers to explore innovative solutions to mitigate this gap. One such approach involves leveraging physically contiguous memory to optimize TLB utilization. Concurrently, advancements in hardware-level system security, exemplified by the Capability Hardware Enhanced RISC Instructions (CHERI) architecture, offer additional opportunities for improving memory management and security.

CHERI introduces capability-based addressing, a novel approach that enhances system security by associating capabilities with memory pointers. These capabilities restrict access to memory regions, thereby fortifying the system against various security threats. Importantly, the mechanisms implemented in CHERI for enforcing memory protection can also serve as accelerators for standard user-space memory allocators. By leveraging capability-based addressing, memory allocators can efficiently manage memory resources while ensuring robust security measures are in place.

Table of contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | TLB based approaches | 2 |
| 1.1.1 | Huge Pages | 2 |
| 1.1.2 | Segment | 3 |
| 1.1.3 | Range Memory Mapping (RMM): | 3 |
| 1.1.4 | FlexPointer: | 5 |
| 1.2 | Capability | 7 |
| 1.2.1 | 1970 Cambridge CAP compute | 8 |
| 1.2.2 | IBM's System/38 | 9 |
| 1.2.3 | M-Machine | 9 |
| 1.2.4 | Low-fat pointers | 9 |
| 1.3 | Capability based operating systems | 9 |
| 1.3.1 | Hydra system | 10 |
| 1.3.2 | KeyKOS | 10 |
| 1.3.3 | Mach microkernel | 10 |
| 1.3.4 | Barrelfish | 11 |
| 1.3.5 | Capsicum | 11 |
| 1.3.6 | CheriBSD | 11 |
| 1.4 | CHERI | 12 |
| 1.4.1 | CHERI compression | 13 |
| 2 | Fat Pointer Based Range Addresses | 15 |
| 2.1 | Range creation and huge pages | 16 |
| 2.2 | Software Stack | 18 |
| 2.2.1 | Contigmem driver from DPDK | 19 |
| 2.2.2 | Sample memory allocator design | 21 |

| | | |
|----------|--|-----------|
| 3 | Evaluation | 25 |
| 3.1 | Benchmarks used | 25 |
| 3.1.1 | DTLB L1 reads | 26 |
| 3.1.2 | DTLB L2 reads | 26 |
| 3.1.3 | DTLB walks | 29 |
| 3.1.4 | L1 cache miss | 29 |
| 3.2 | Analysis | 31 |
| 4 | Future work | 33 |
| 5 | Research Timeline | 35 |
| 5.1 | FAT-Pointer based range addresses | 35 |
| 5.1.1 | July 1th - July 15th (2024) | 35 |
| 5.1.2 | July 15th - July 30th (2024) | 36 |
| 5.1.3 | August 1st - August 30th (2024) | 37 |
| 5.1.4 | September 1st - September 30th (2024) | 37 |
| 5.2 | FAT-Pointer based range addresses with RISC-V | 38 |
| 5.2.1 | October 15th - December 18th (2024) | 38 |
| 5.2.2 | January 1st - February 1st (2025) | 38 |
| 5.2.3 | February 1st - March 1st (2025) | 38 |
| 5.2.4 | March 1st - May 1st (2025) | 39 |
| 5.3 | FAT-Pointer based range addresses with Uni-kernels | 39 |
| 5.3.1 | May 1st - September 1st (2025) | 39 |
| 5.3.2 | September 1st - December 30th (2025) | 39 |
| 5.3.3 | January 1st - September 30th (2026) | 40 |
| 6 | Conclusion | 41 |
| | References | 43 |

Chapter 1

Introduction

In the dynamic landscape of computing, the pursuit of optimal performance is a constant endeavor, especially as applications evolve to handle increasingly complex workloads. One critical aspect influencing performance is memory management, where efficient utilization of resources is paramount. Translation Lookaside Buffers (TLBs) play a pivotal role in this regard, expediting memory access by storing recently accessed memory translations. However, as applications grow in size and complexity, the capacity of TLBs often struggles to keep pace, leading to performance bottlenecks. To address this challenge, researchers have turned to innovative solutions, one of which involves harnessing the benefits of huge pages. Huge pages, also known as large pages, allow for the allocation of memory in significantly larger chunks compared to traditional small pages. By reducing the number of TLB entries needed to access a given amount of memory, huge pages offer a potential avenue for optimizing TLB utilization and thereby enhancing overall system performance.

Simultaneously, advancements in hardware-level security, such as the Capability Hardware Enhanced RISC Instructions (CHERI) architecture, present additional opportunities for performance enhancement. CHERI's capability-based addressing approach not only strengthens system security by tightly controlling memory access but also provides avenues for accelerating memory management operations.

In this context, the integration of huge pages into memory management strategies alongside capability-based addressing in architectures like CHERI offers a compelling synergy. By optimizing TLB utilization through the utilization of huge pages and leveraging the security features of capability-based addressing, significant performance improvements can be realized. This approach not only enhances system security but also accelerates memory

access. The following below are research questions we are addressing:

1. How does the utilization of bounds for tracking memory allocations, in addition to security purposes, affect the run times and Translation Lookaside Buffer (TLB) miss rates in modern computing systems?
2. How does the implementation of bounds for seeking through physically contiguous memory influence the complexity and efficiency of standard memory allocators, particularly those with advanced features such as transparent huge pages, and what are the implications for system performance in terms of execution speed, memory access latency, and resource utilization?

1.1 TLB based approaches

Efficient memory management, particularly in the context of Translation Lookaside Buffer (TLB) optimization, has been a focal point of research and development within computer architecture. Various techniques have been proposed to mitigate TLB-related bottlenecks and improve overall system performance.

1.1.1 Huge Pages

Increasing TLB reach can be achieved by using larger page sizes, such as huge pages[Panwar et al.], which are common in modern computer systems. The x86-64 architecture supports huge pages of 2 MB and 1 GB, backed by OS mechanisms like Transparent Huge Pages (THP) and HugeTLBFS[8] in Linux. However, available page sizes in x86-64 are limited, leading to internal fragmentation issues. For instance, allocating 1 MB with 4 KB base pages requires 256 PTEs, but using a 2 MB huge page would waste half of the memory space. Some architectures offer more page size choices, such as Intel Itanium, which allows different areas of the address space to have their own page sizes. Itanium uses a hash page table to organize huge pages, but without significant changes to the conventional page table, it only helps reduce page walk overheads. HP Tunable Base Page Size permits the OS to adjust the base page size, but still faces internal fragmentation problems, with HP recommending a base page size of no more than 16 KB. Shadow Superpage introduces a new translation level in the memory controller to merge non-contiguous physical pages into a huge page in a shadow memory space, extending TLB coverage. However, this approach requires all memory traffic to be translated again in the memory controller, resulting in additional latency for memory accesses.

1.1.2 Segment

Early processors often used segments[Basu et al.] to manage virtual memory, where a segment essentially mapped contiguous virtual memory to contiguous physical memory. Unlike pages, which are relatively small, segments can be much larger, offering the potential for more efficient memory management in certain scenarios. This concept of segmentation has seen a resurgence in some modern approaches that aim to enhance translation coverage by designating specific areas in the virtual address space.

This method allows programmers to explicitly define a single segment for applications requiring significant memory. It introduces two new registers to the system, which indicate the start and end of this segment. Virtual addresses within this segment are translated by calculating the offset from the virtual start address and applying this offset to the physical start address. This straightforward method simplifies the translation process for large memory areas but requires significant modifications to the source code of applications.

Another approach, Do-it-yourself Virtual Memory Translation (DVMT)[6], similarly introduces two registers to define a special area within the virtual address space. However, DVMT supports more complex translation mechanisms. When an address within this designated area is accessed, DVMT initiates a dedicated thread to handle the translation. This flexibility allows for more sophisticated memory management techniques, but like Direct Segment, it necessitates extensive changes to the application's source code. Additionally, both Direct Segment and DVMT bypass the conventional page table for the special virtual area, operating independently of the standard paging mechanism.

In Figure 1.1, each data memory reference involves presenting the data virtual address V to both the new direct-segment hardware and the Data Translation Lookaside Buffer (D-TLB). If the virtual address V falls within the specified range defined by the direct segment's base and limit register values, the new hardware translates the address to a physical address by calculating $V + \text{OFFSET}$. This translation process bypasses the D-TLB, meaning that addresses handled by the direct-segment hardware do not experience TLB misses. It's important to note that the direct-segment hardware only allows read-write access.

1.1.3 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

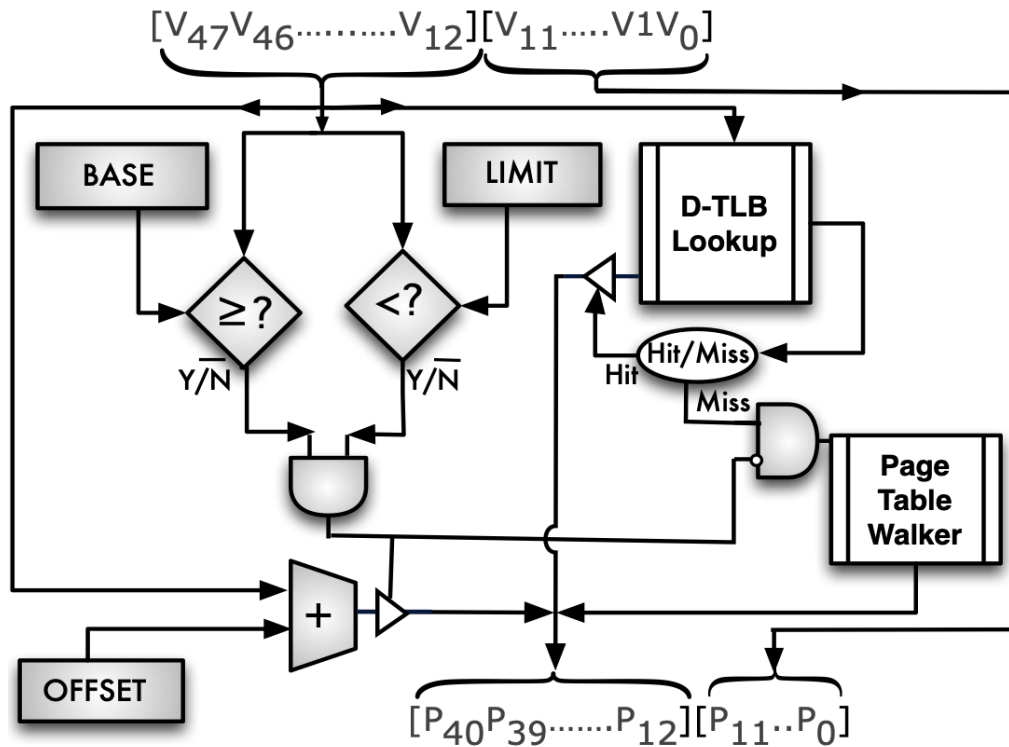


Fig. 1.1 Logical view of address translation with direct segment[Basu et al.].

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 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 RTLTLB, 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.

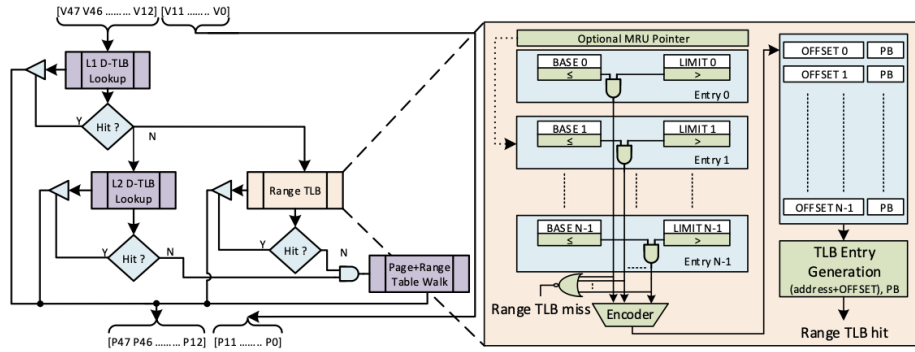


Fig. 1.2 RMM[Karakostas et al.] hardware support consists primarily of a range TLB that is accessed in parallel with the last-level page TLB.

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 $BASE_i$ and $LIMIT_i$ values, defining the boundaries of the virtual address range. The translation part holds the $OFFSET_i$, which is the difference between the starting point of the range in physical memory and $BASE_i$, 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 $BASE_i \leq \text{virtual page number} < LIMIT_i$. On a hit, the range TLB returns the $OFFSET_i$ 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 $OFFSET_i$ 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.

1.1.4 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 (physical_address - virtual_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.

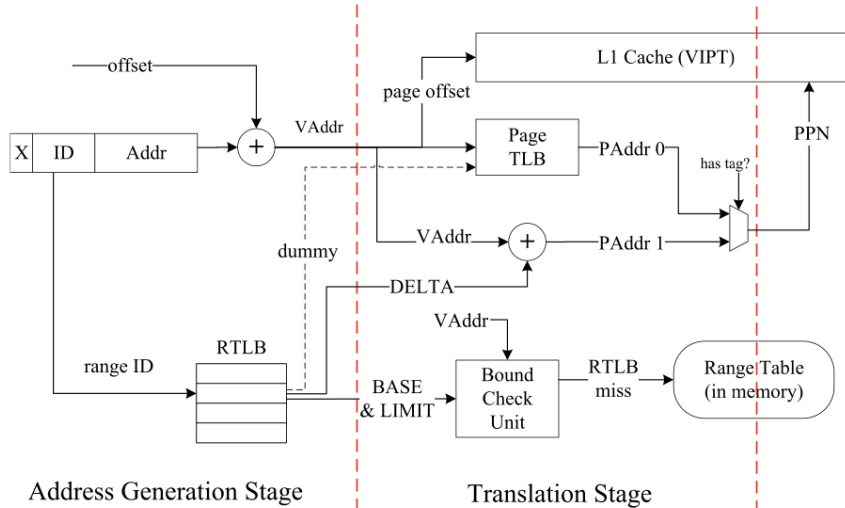


Fig. 1.3 FlexPointer[Chen et al.] hardware overview.

In Figure 1.4 FlexPointer utilizes specialized hardware components and a streamlined workflow for efficient memory management. During address generation, the processor deter-

mines 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 $(ID \ll 5)$ 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.

1.2 Capability

A capability[4] is an unforgeable token that names an object and grants the bearer the authority to act upon it. Capabilities can be delegated between principals to both identify an object and grant authority over it, in contrast to other methods that use separate channels for naming and authority sharing. For instance, on UNIX, a file name is shared as a path, and the authority to open it is granted by adjusting permissions separately. In a capability system, a 'path capability' would not just represent a path but also the authority to open it in specific modes.

Every access to an object in a capability system requires authorization by a capability, making it clear which authority should be used for each operation as it is part of the reference. In some systems like the CAP architecture, dynamic capability lookup is performed. The CHERI work introduces the concept of intentional access, where the invoked authority is explicitly stated if a set of authorities is available. Systems that lack explicit identification of which capability to use or do not utilize capabilities at all are vulnerable to confused-deputy attacks, unlike capability systems that follow the intentional access principle, such as CHERI.

Capabilities also separate policy from enforcement, simplifying the parts of the system that need to be trusted. They can be shared in a decentralized manner based on principals, leading to a proliferation of capabilities throughout the system. Revoking a capability involves globally destroying or invalidating it and potentially any derived capabilities. The trade-off between the complexity of use and revocation exists in capability systems, with some schemes using tracking structures for revocation maintenance. CHERI, however, does not require such tracking and opts for a more intricate revocation process to enhance capability usage.

Capability machines implement and enforce capabilities in hardware, offering benefits such as improved performance and a reduced trusted computing base. While currently not as popular, there have been many capability machines in academia and industry. One example is the *Burroughs B5000*[1] stack machine from the 1960s, which featured a descriptor table for each program segment. When accessing an array, an index in the descriptor table was calculated first, followed by an indirect operation through it. Descriptors in the table had bounds, and hardware was responsible for performing access checks. Although predating the modern concept of a capability machine, the *Burroughs B5000* was one of the first to use tagged memory to differentiate segment descriptors (similar to capabilities) from other memory types, enabling entries to be directly loaded onto the stack.

1.2.1 1970 Cambridge CAP compute

The 1970 Cambridge CAP computer[Wilkes and Needham] was an early register-based capability machine that utilized explicit capability registers. These registers could be loaded and stored from dedicated capability segments, which did not require tags but limited the flexibility in mixing capabilities and data.

1.2.2 IBM's System/38

On the commercial front, IBM's System/38 and its successor, the AS/400, were successful systems that, like the Burroughs B5000, incorporated tagged memory to distinguish capabilities from data. Intel's iAPX 432 faced challenges due to poor performance, particularly attributed to its inefficient ADA compiler. The architecture supported both hardware- and software-defined capabilities, dividing segments to differentiate between data and capabilities, similar to the CAP computer.

1.2.3 M-Machine

The M-Machine[Fillo et al.] utilized guarded pointers for capabilities, with support for power-of-two bounds and alignments. A fat pointer encoded bounds, permissions, and an address for the capabilities. The M-Machine introduced two distinct capability variants: a 'key' capability and an 'entry' capability, both unmodifiable and non-forgable. Entry capabilities could be executed, transforming into executable capabilities, while key capabilities functioned as identifiers. A supervisor mode bit in executable capabilities facilitated secure transitions to privileged code using normal jumps at predefined entry points.

1.2.4 Low-fat pointers

In 2013, Low-fat pointers[Kwon et al.] implemented hardware support for bounds checking fat pointers compressed into 64 bits, with non-power-of-two alignment support. While similar to constructs seen in M-Machine and CHERI, these pointers lack some essential capabilities such as authority, unforgeability, and compartmentalization features.

1.3 Capability based operating systems

Capabilities are beneficial not only for controlling access at the hardware level but also have applications in operating systems and software applications.

1.3.1 Hydra system

The Hydra system[Wulf et al.], developed on the PDP-11, incorporated capabilities for both system-level and user-defined objects. Due to security concerns, all capability operations necessitated syscalls as only the kernel was deemed trustworthy to handle capabilities reliably. While this approach restricted the frequency of operations and the types of objects that could be effectively represented by capabilities, Hydra successfully showcased the practical implementation of the object-capability model through interprocess communication (IPC).

1.3.2 KeyKOS

KeyKOS[Hardy], originally known as GNOSIS, was an early commercial capability operating system designed for the IBM System/370 that could accommodate a UNIX environment. KeyKOS, similar to other capability-based operating systems mentioned, operated as a message-passing system and utilized capabilities termed as keys to represent various system resources. In KeyKOS, the system was divided into domains, with gate keys enabling message transfer between these domains. The OS featured a single-level persistent store that users could interact with using capabilities, and the kernel could seamlessly swap this store with backing storage. KeyKOS paved the way for subsequent research-oriented systems like Eros and Coyotos.

1.3.3 Mach microkernel

The Mach microkernel[Rashid et al.] introduced the concept of tasks, which encompassed collections of threads combined with a virtual address space. In Mach, interprocess communication (IPC) was facilitated through ports, resembling KeyKOS gate keys, where the capability to communicate on a port was represented by a capability. Capabilities could be transmitted over ports for delegation, and messages, dynamically-sized and typed, could be sent synchronously or asynchronously. Similar to Hydra, Mach implemented an object-capability model using IPC, as exemplified by the process of creating a thread, which involved sending a message on a specific port and delegating authority by transmitting the port capability. Mach aimed to offer a kernel compatible with a UNIX environment while delegating maximum functionality outside of the kernel to a set of servers. Various UNIX variants have emerged from Mach or utilized Mach as a microkernel.

Following the Mach microkernel, the L4 microkernel was developed with the aim of reducing complexity and overhead compared to previous microkernels. L4 featured a simplified interface, with an early version supporting only 7 system calls, providing functionality for threads, address spaces, and IPC. The streamlined IPC mechanism in L4 resulted in significantly faster across-address-space message transfers compared to Mach, with the initial L4 implementation being 20 times faster. SeL4, a formally-verified implementation of L4 built from scratch, stores capabilities in a tree structure where each node consists of an array of capabilities. Capabilities serve as fat-pointers to objects in physical memory, starting as untyped and then being retyped to create objects like page-tables. These capabilities can only be modified by the kernel and are accessed from user-space via an index.

1.3.4 Barrelfish

Barrelfish[Baumann et al.] is an operating system developed by ETH Zurich specifically tailored to support large-scale heterogeneous systems. Utilizing capabilities similar to those in seL4, Barrelfish has extensions to manage the increased complexity of operations such as deletion, copying, re-typing, and revoking in a distributed manner. Capabilities in Barrelfish represent physical memory, kernel objects, and communication endpoints, adhering to the principles of capability-based systems.

1.3.5 Capsicum

Capsicum[23] extends the POSIX API with file capabilities and was initially demonstrated on FreeBSD, where capability operations still require system calls due to the conventional hardware environment of FreeBSD. While files in UNIX systems can encompass various entities, Capsicum does not provide memory capabilities, allowing fabricated pointers to be exchanged between the operating system and user space.

1.3.6 CheriBSD

CheriBSD[7] enhances FreeBSD to utilize CHERI hardware-enforced capabilities for protecting memory accesses in both user space compartments and the kernel. Although CheriBSD operates on a FreeBSD base, many aspects of the system interface remain non-capability based. For instance, basic operations like spawning threads in FreeBSD require syscalls,

granting ambient authority to do so, while Mach, in a capability-based system, would utilize a capability for such privileges. In comparison to other capability-based operating systems, CheriBSD's use of memory capabilities is more fine-grained, enabling precise control over memory accesses and operations. Systems like seL4 use memory capabilities for defining privileges like creating page mappings, although the actual use of these mappings during load or store operations may not involve capability checks. Additionally, since enforcement in systems like seL4 relies on the MMU, capabilities in these systems may refer to objects with coarse sizes and alignments.

1.4 CHERI

CHERI[Woodruff et al.] is an instruction set extension that modifies an existing ISA[24] by mandating memory accesses to be conducted via tagged capabilities and introduces instructions for examining and modifying these capabilities. Initially developed as an enhancement to the MIPS ISA, CHERI has gained significant traction within the technology community, with notable implementations such as the Arm's Morello[2] experimental prototype integrating CHERI into a commercial Arm processor.

CHERI enhances pointer protection by incorporating a one-bit out-of-band tag to ensure the integrity of pointers. Each tagged capability in CHERI is required to originate from an initial root capability through a series of permissible operations, with each operation reducing the granted rights in a strictly decreasing manner. This strict provenance of tagged capabilities minimizes the risk of pointer corruption and effectively counters attacks that involve manipulating pointers by substituting them with non-capability data, such as Return-Oriented Programming.

The tags associated with capabilities in CHERI prevent pointer corruption and bolster security by necessitating that every memory access is associated with a quoted tagged capability that encompasses the intended boundaries and permissions. This measure serves to safeguard data from both accidental and malicious tampering, effectively mitigating common attacks like buffer overflows. A report by Microsoft[9] highlighted that approximately 75% of their security vulnerabilities are related to bounds overflows or temporal safety issues, indicating that CHERI's support for temporal safety could significantly address such issues and provide substantial benefits to the industry and users.

Furthermore, the monotonicity of the CHERI ISA[24] has been formally verified[Zaliva et al.], demonstrating that the architecture, if correctly implemented, upholds the intended invariants, offering a high level of security assurance.

The bit pattern[25] of a capability consists of:

- **Capability tag:** A one-bit out-of-band field that records whether the capability is valid, i.e. whether it has been derived only via legal monotonic operations.
- **Cursor:** A field giving the current address referred to by the capability. This is all that is present in a traditional C pointer and the natural integer interpretation of the capability.
- **Bounds information:** The capability only grants access to a single contiguous region of memory, specified by the bounds field. Changes to the capability can only ever grant access to a subset of the bounds without triggering an exception or clearing the tag.
- **Permissions:** Permissions are required to access memory in different ways: most obviously read, write, and execute, but also the permissions to read and write capabilities, and more experimental uses. These can only be modified by a bitwise AND operation, inherently guaranteeing monotonicity.
- **Object type (otype):** CHERI provides a compartmentalisation mechanism for granting protected access to objects, represented as a pair of code and data. Code and data can be sealed with an otype, and primarily only unsealed by the CInvoke instruction, which atomically jumps to the code pointer and installs the unsealed data capability. Sealed capabilities cannot be mutated other than via a legal CInvoke or CUnseal instruction without raising an exception or clearing the tag.

1.4.1 CHERI compression

One of the significant performance challenges of CHERI stems from the larger size of capabilities compared to traditional pointers, leading to increased hardware resource usage and cache pressure. An approach to mitigate this impact is capability compression, as outlined in the CHERI-Concentrate paper[Woodruff et al.]. This method reduces bounds metadata to as few as 39 bits for a 64-bit pointer, albeit with alignment constraints on representable positions for larger objects. Hardware handles compression and decompression, ensuring minimal software impact.

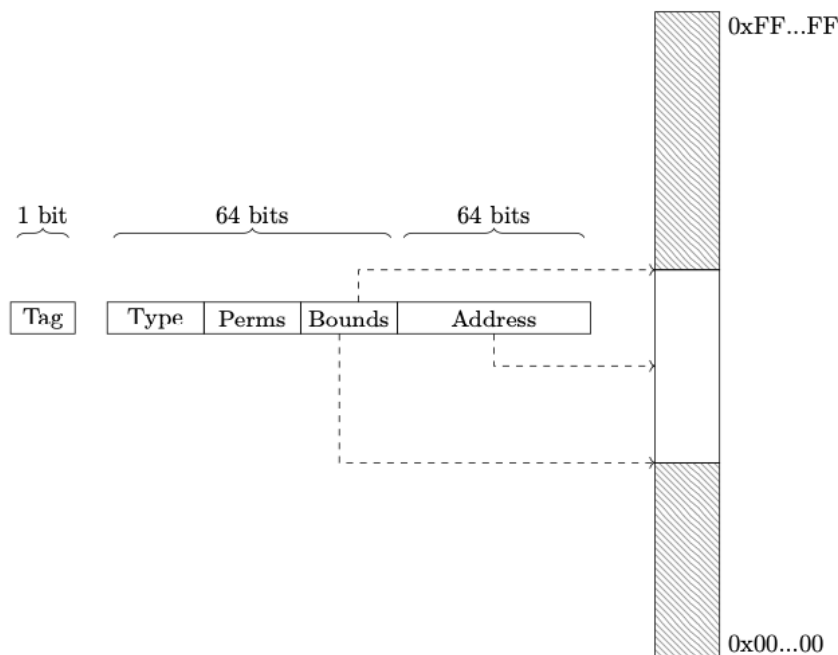


Fig. 1.4 CHERI capability[15].

In capability compression, bounds consist of a base (B) with b bits, a top (T) bits, and an exponent (E) with e bits relative to the cursor of the capability (C). The base is calculated by shifting B left by E and filling remaining upper bits with the corresponding most significant bits of C. The top follows a similar process, with the top two bits of T determined by adding 1 to those of B in the standard case. Bounds are zero-filled in the least significant E bits, enforcing bound-alignment requirements. The internal exponent case stores E only when non-zero, serving as the least significant bits of B and T, thereby allowing more precise representation of smaller objects.

The *CSetBounds* instruction in CHERI offers two variations: one triggers an exception for unrepresentable bounds, while the other returns the closest larger representable bounds. To ensure spatial safety, software must include padding (up to 0.2% or 12.5% of the object size for 64-bit and 32-bit address spaces, respectively) at the beginning or end of large objects to prevent memory usage conflicts resulting from rounding. Instructions altering a capability cursor may impact bounds interpretation if they modify cursor bits determining the base and top. Careful design in the ISA and implementations ensures these cases clear the capability tag or raise exceptions where necessary.

Chapter 2

Fat Pointer Based Range Addresses

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.

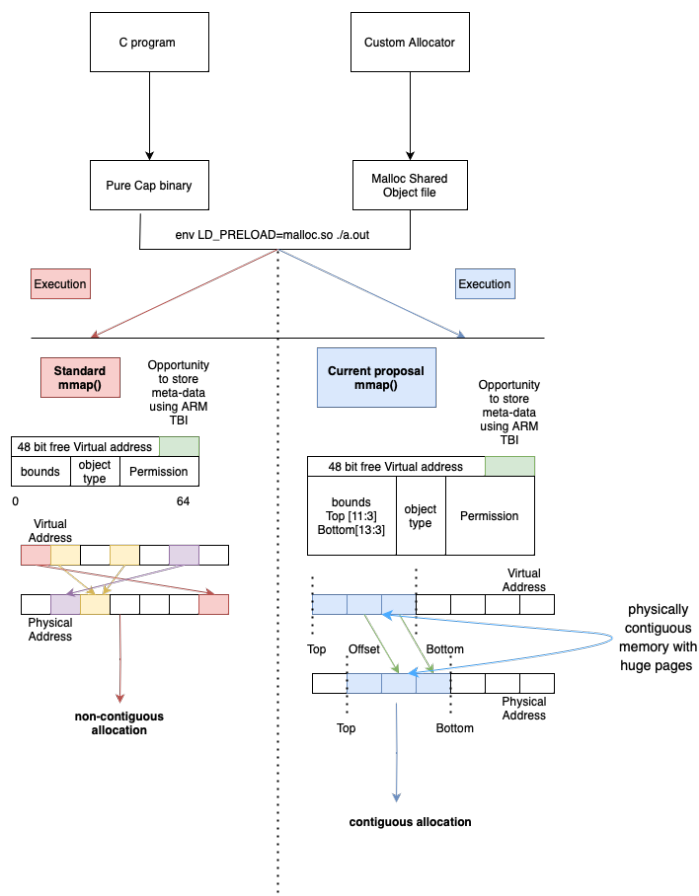


Fig. 2.1 High overview architecture

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.

2.1 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

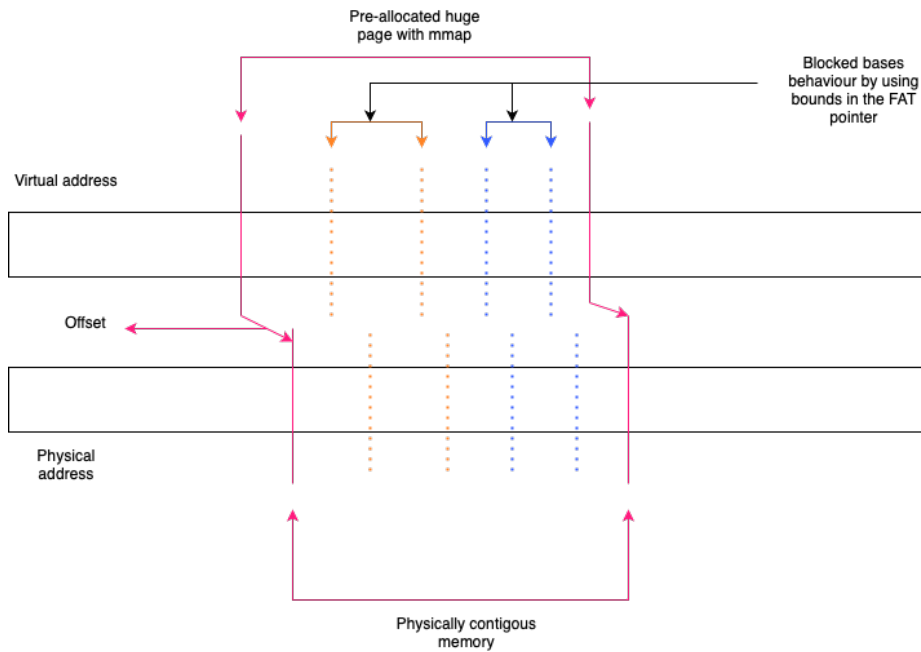


Fig. 2.2 Range of memory

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 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.

2.2 Software Stack

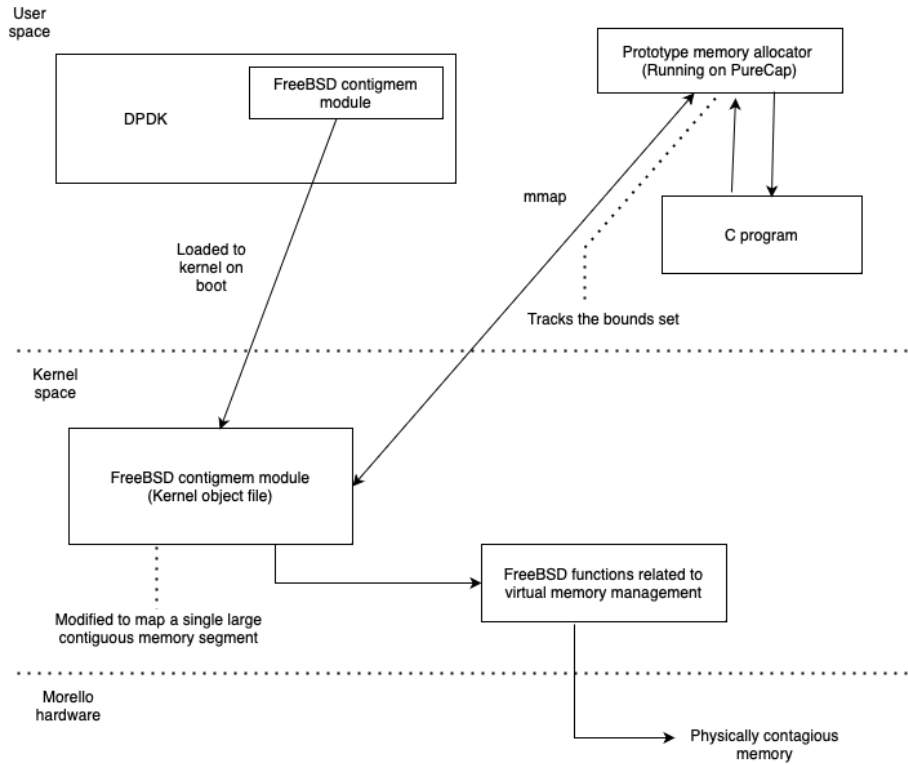


Fig. 2.3 Overview of the software stack

The software stack is based on CHERIBSD[7], selected because ARM officially supports Morello's performance counters[2] on this operating system. As illustrated in the figure 2.3, the setup includes a C program that is linked to the prototype memory allocator or to various memory allocators being benchmarked, as described in section 3. This linkage can occur in two ways: either as a shared object file during compile time for larger allocators, or as a header file for smaller allocators, ensuring flexibility and efficiency in memory management.

This integration ensures that the memory allocation process is optimized for performance, leveraging the contiguity of memory blocks and the capabilities provided by the CHERI architecture and the Morello platform. By using the contigmem driver and the custom mmap function, the system achieves efficient memory allocation and tracking, crucial for the high-performance needs of the application.

2.2.1 Contigmem driver from DPDK

The custom mmap function, tailored to ensure physically contiguous memory allocation, is a key component of this system. This function is linked to the contigmem driver, which has been modified from the DPDK[Bi and Wang] library to meet the specific needs of this implementation. The contigmem driver is essential for managing large contiguous memory blocks and is loaded during the system boot process. It reserves a huge page of arbitrary size, with the size parameter set based on the requirements of the conducted experiments.

Listing 2.1 Contigmem driver

```

MALLOC_DEFINE(M_CONTIGMEM, "contigmem",
"contigmem(4)_allocations");

static int contigmem_modevent(module_t mod,
int type, void *arg)
{
    int error = 0;

    switch (type) {
    case MOD_LOAD:
        error = contigmem_load();
        break;
    case MOD_UNLOAD:
        error = contigmem_unload();
        break;
    default:
        break;
    }

    return error;
}

....

DECLARE_MODULE(contigmem, contigmem_mod,
SI_SUB_DRIVERS, SI_ORDER_ANY);

```

```
MODULE_VERSION(contigmem, 1);
```

```
static struct cdevsw contigmem_ops = {
    .d_name      = "contigmem",
    .d_version    = D_VERSION,
    .d_flags      = D_TRACKCLOSE,
    .d_mmap_single = contigmem_mmap_single,
    .d_open       = contigmem_open,
    .d_close      = contigmem_close,
};
```

```
static int
```

```
contigmem_load()
```

```
{
    ....

    for (i = 0; i < contigmem_num_buffers; i++) {
        addr = contigmalloc(contigmem_buffer_size,
            M_CONTIGMEM, M_ZERO,
            0, BUS_SPACE_MAXADDR,
            contigmem_buffer_size, 0);
        ....
    }

    ....

error:
    for (i = 0; i < contigmem_num_buffers; i++) {
        if (contigmem_buffers[i].addr != NULL) {
            contigfree(contigmem_buffers[i].addr,
                contigmem_buffer_size, M_CONTIGMEM);
            contigmem_buffers[i].addr = NULL;
        }
        ....
    }
}
```

```

        return error;
    }

```

When the `contigmem_load` 2.1 function is called, either during boot or when the Kernel module is loaded, it pre-allocates a segment of physically contiguous memory. This approach differs from FlexPointer[Chen et al.], which allocates physically contiguous memory eagerly. The `contigmem_load` function allocates memory using `contigmalloc`, which allocates physically contiguous memory initialized to zero. The `cdevsw` struct refers to function calls which would be overwritten on loading the driver. In the code snippet 2.1 above the `mmap` function would be overwritten with `contigmem_mmap_single` if the following driver is opened and truncated as shown in Code snippet 2.3.

In the code snippet 2.2 the `cdev_pager_ops` refers to the operations which will be overwritten when called with `mmap` such as overwriting page faults.

Listing 2.2 Contigmem driver mmap

```

static struct cdev_pager_ops contigmem_cdev_pager_ops = {
    .cdev_pg_ctor = contigmem_cdev_pager_ctor ,
    .cdev_pg_dtor = contigmem_cdev_pager_dtor ,
    .cdev_pg_fault = contigmem_cdev_pager_fault ,
};

static int
contigmem_mmap_single(struct cdev *cdev, vm_ooffset_t
*offset, vm_size_t size, struct vm_object **obj, int nprot)
{
    ....
    *obj = cdev_pager_allocate(vmh, OBJT_DEVICE,
        &contigmem_cdev_pager_ops, size, nprot,
        *offset, curthread->td_ucred);

    return 0;
}

```

2.2.2 Sample memory allocator design

Listing 2.3 Contigmem driver mmap

```

#define FILENAME "/dev/contigmem"
void *ptr;
int MallocCounter;

size_t sizeUsed;

...

INITAlloc(void) {

    size_t sz;
    sz = 100000000;

    int fd = open(FILENAME, O_RDWR, 0600);

    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    off_t offset = 0; // offset to seek to.

    if (ftruncate(fd, sz) < 0) {
        perror("ftruncate");
        close(fd);
        exit(EXIT_FAILURE);
    }

    ptr = mmap(NULL, sz,
    PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

// Added error handling
    if (ptr == MAP_FAILED)
    {
        perror("mmap");
    }

```

```

        exit(EXIT_FAILURE);
    }
    MallocCounter = (int)sz;
}

void* malloc(size_t sz)
{
    sz = __builtin_align_up(sz, _Alignof(max_align_t));

    // printf("%d \n", sz);
    // printf("%d Malloc counter\n", MallocCounter);

    MallocCounter -= sz;
    void *ptrLink = &ptr[MallocCounter];
    ptrLink = cheri_setbounds(ptrLink, sz);

    return ptrLink;
}

void FREECHERI(void *ptr) {

    // get length of free from bounds
    // in the pointer
    int len = cheri_getlen(ptr);

    munmap(ptr, len);
}

```

The code snippet 2.3 below shows a sample memory allocator which is a really simple implementation that initially in the example allocates 1GB of memory using the mmap call which calls the mmap function from /dev/contigmem driver. This ensures memory allocated to the physically contiguous memory allocated in the contigmem_load() function using the contigmem_mmap_single() function call in the kernel module, uses malloc and free to allocate within this memory chunk. The consideration of this is to ensure that a C program needs minor changes to use the benefit using physically contiguous memory with bounds within a segment of memory.

Chapter 3

Evaluation

To evaluate the performance of FAT-Pointer based range addresses a sample implementation we used the morello board with CheriBSD's Benchmark ABI[3] compilation mode for accurate performance recordings. The evaluation is to identify CheriBSD's default memory allocator SnMalloc against the prototype memory allocator using the contribution of this paper in terms of: To assess the performance of FAT-Pointer-based range addressing in the sample implementation, we utilized the Morello board running CheriBSD in Benchmark ABI compilation mode to ensure precise performance measurements. The evaluation focuses on comparing CheriBSD's default memory allocator, SnMalloc, against the prototype memory allocator developed in this study, with particular attention to the following aspects:

| Metric name | type of graph | tool used | x axis | y axis |
|---------------------|---------------|-----------|------------|-----------------------------|
| DTLB L1 read | line graph | Pmcstat | Time | DTLB L1 reads (each second) |
| DTLB L2 read | line graph | Pmcstat | Time | DTLB L2 reads (each second) |
| DTLB walk | line graph | Pmcstat | Time | DTLB Walks (each second) |
| L1 cache miss | line graph | Pmcstat | Time | L1 cache miss (each second) |
| Wall clock run time | bar graph | time | Benchmarks | Time |

Table 3.1 Metrics evaluated

3.1 Benchmarks used

To conduct the evaluations, we utilized the COZ[Curtsinger and Berger] benchmark suite, a well-regarded tool specifically designed to measure and analyze performance improvements in concurrent programs. The COZ benchmark suite provides a robust framework for identifying bottlenecks and evaluating the performance impact of various optimization techniques.

By leveraging COZ, developers can gain precise insights into the efficiency and scalability of their concurrent code, making it an ideal choice for rigorous performance analysis.

From the extensive set of benchmarks provided by COZ, we selected three representative C programs. These programs were chosen based on their relevance to common concurrent programming patterns and their ability to effectively demonstrate the strengths and weaknesses of different optimization strategies. The selected programs cover a range of concurrency scenarios, ensuring a comprehensive evaluation of performance improvements.

By implementing these modifications, we ensured that the selected C programs not only adhered to CHERI's security model but also maintained their functional and performance characteristics. This allowed us to effectively use the COZ benchmark suite to analyze the performance in a CHERI-enhanced environment.

| Benchmark name | Benchmark metrics extracted | Sizes tried against | Comparators |
|-----------------------|--|--|---|
| Kmeans (Coz) | <ul style="list-style-type: none"> - L1 DTLB reads - L2 DTLB reads - DTLB walks - L1 cache misses - Wall clock run time | <ul style="list-style-type: none"> - 3 Dimensions - 6 Dimension - 40 Dimensions | <ul style="list-style-type: none"> - Physically contiguous allocator with bounds - System allocator |
| Histogram (Coz) | <ul style="list-style-type: none"> - L1 DTLB reads - L2 DTLB reads - DTLB walks - L1 cache misses | <ul style="list-style-type: none"> - Small - Medium - Large | <ul style="list-style-type: none"> - Physically contiguous allocator with bounds - System allocator |
| Matrix multiply (Coz) | <ul style="list-style-type: none"> - L1 DTLB reads - L2 DTLB reads - DTLB walks - L1 cache misses | <ul style="list-style-type: none"> - 200 - 1000 - 5000 | <ul style="list-style-type: none"> - Physically contiguous allocator with bounds - System allocator |

Fig. 3.1 C programs evaluated

3.1.1 DTLB L1 reads

The Graphs above?? represent the DTLB L1 reads which is a Performance counter from the ARM specs. The counter increments for every Memory-read or Memory-write operation that necessitates an access to the Level 1 data or unified Translation Lookaside Buffer (TLB). Each access to a TLB entry is counted including multiple accesses caused by single instructions.

3.1.2 DTLB L2 reads

Similar to how L1 TLB reads are counted, DTLB L2?? counts every read operation that accesses the Level 2 data or unified TLB. Each time there is a read to an entry in the Level 2 TLB, it is counted by the ARM performance counter.

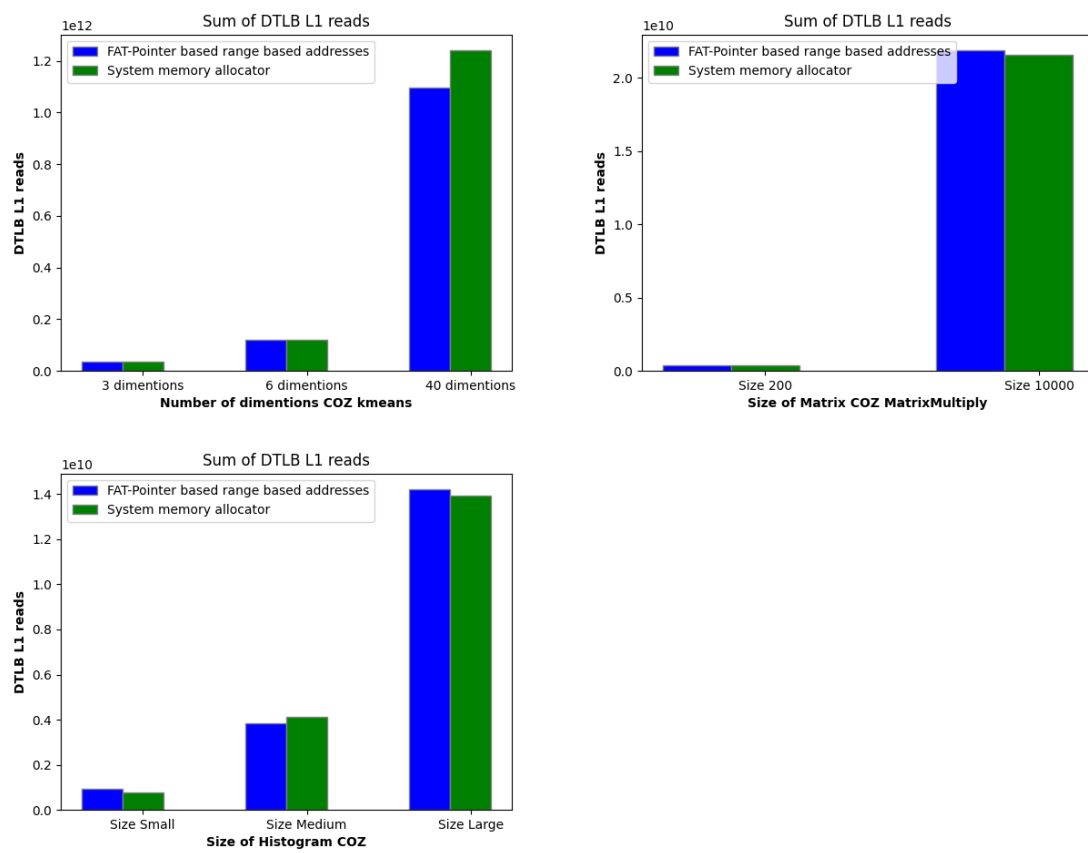


Fig. 3.2 DTLB L1 reads

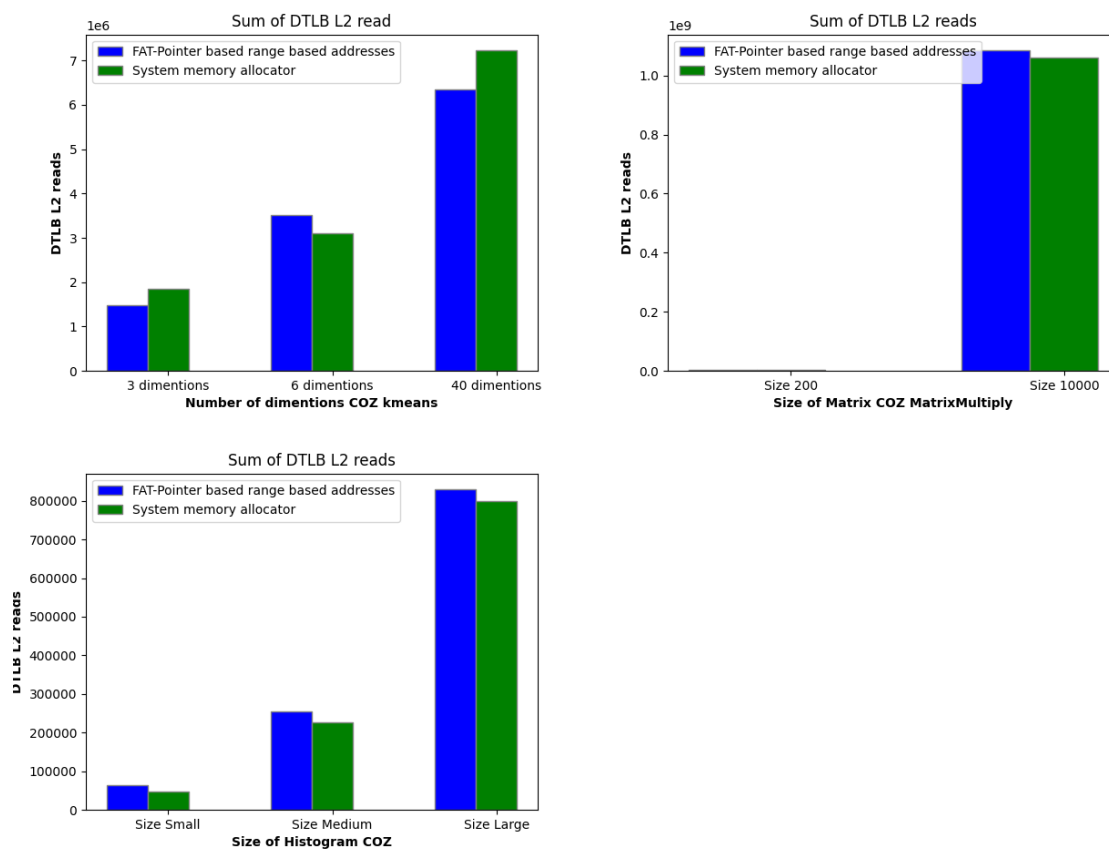


Fig. 3.3 DTLB L2 reads

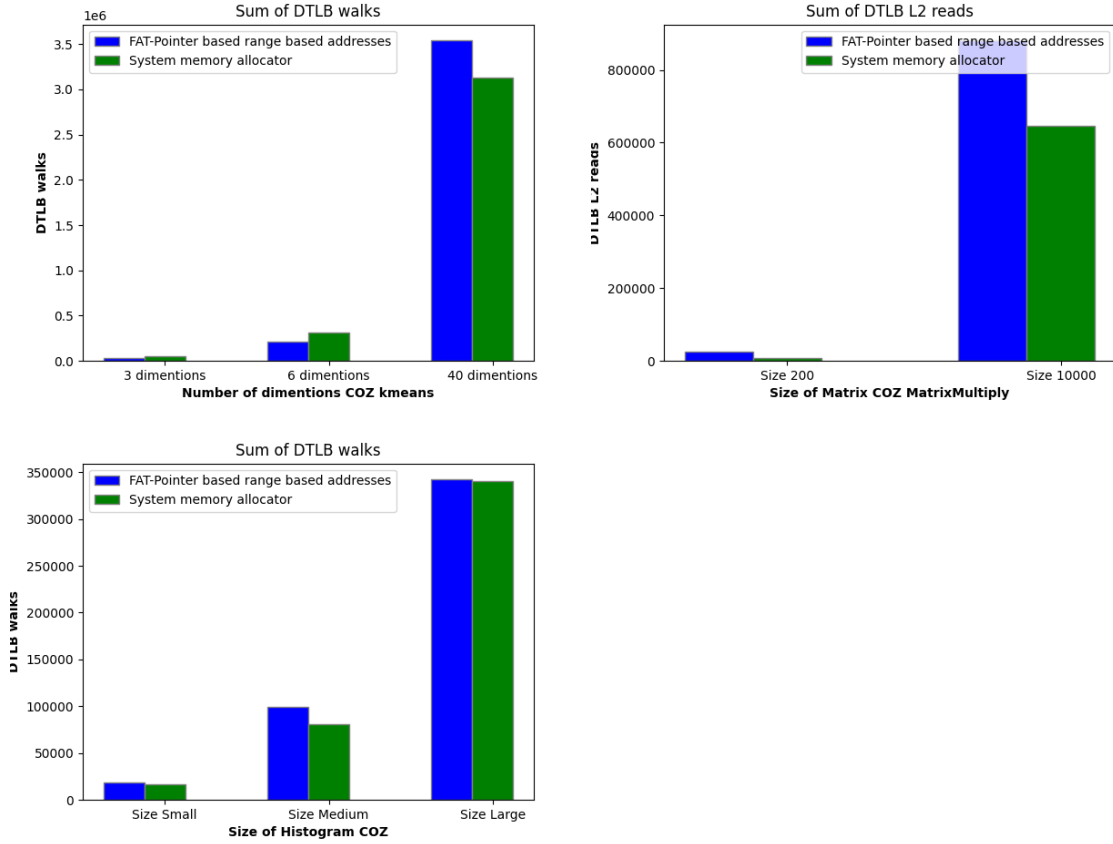


Fig. 3.4 DTLB Walks

3.1.3 DTLB walks

The DTLB walk?? counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 2 data or unified TLB. Each access to a TLB entry is counted including refills of Level 1 TLBs.

3.1.4 L1 cache miss

L1 cache miss?? counter counts each Memory-read operation to the Level 1 data or unified cache counted by L1 cache miss counter that incurs additional latency because it returns data from outside of the Level 1 data or unified cache of this PE. The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 data or unified cache.

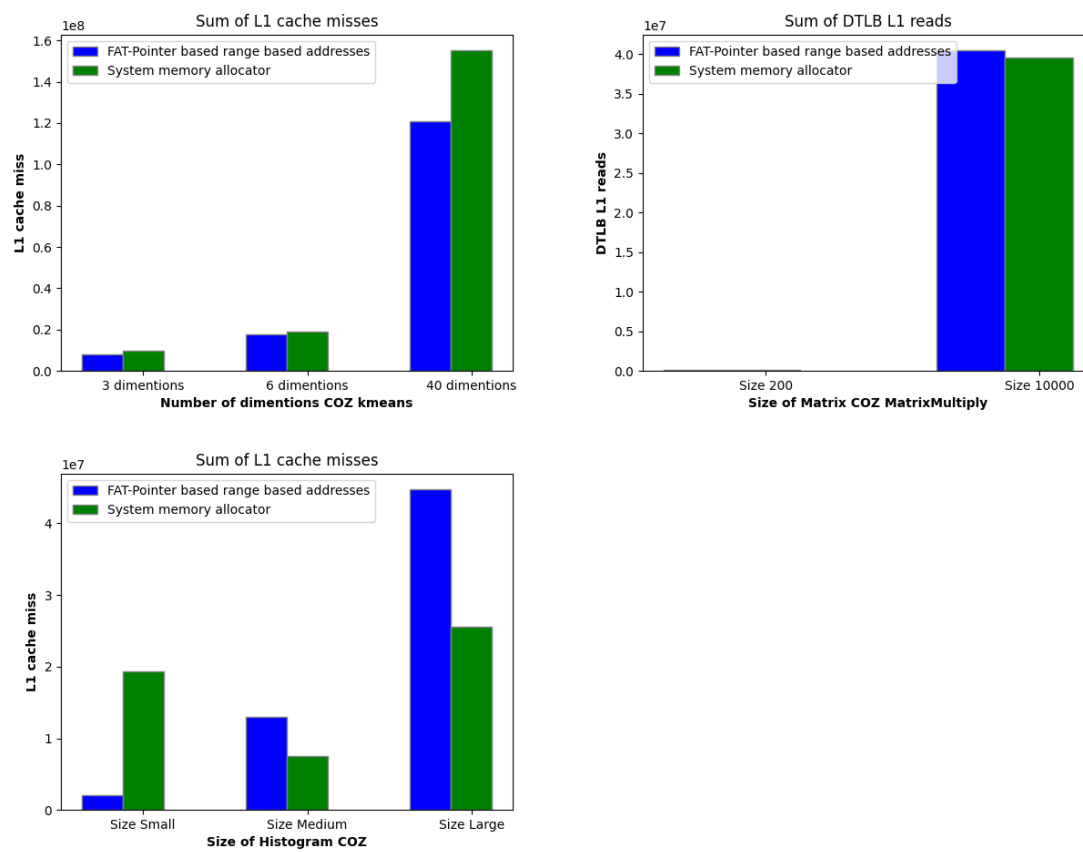


Fig. 3.5 L1 Cache misses

3.2 Analysis

The benchmark results indicate the presence of bugs in the sample FAT-Pointer based range address prototype memory allocator. This evaluation is ongoing, and the results provided here represent a snapshot of the latest findings. Specifically, for the use case of Level 2 Translation Lookaside Buffer (L2 TLB) reads, the FAT-Pointer based range address prototype shows occurrences of reads from the L2 TLB. This indicates the presence of implementation bugs, as all reads should be managed through the Level 1 Translation Lookaside Buffer (L1 TLB), given that the program memory is pre-allocated as a single physically contiguous page.

In a correctly functioning system, the L1 TLB should handle all memory accesses, ensuring efficient translation and minimizing latency. The detection of L2 TLB reads suggests that there are flaws in the current implementation, potentially in the memory allocation or address translation mechanisms. These bugs need to be addressed to achieve the intended performance and correctness of the FAT-Pointer based range address prototype memory allocator. Further investigation and debugging are required to identify the exact causes and rectify these issues to ensure that all memory accesses are correctly routed through the L1 TLB as designed.

The ContigMalloc function in CheriBSD is designed to allocate contiguous blocks of physical memory. In theory, this should result in a single large block of memory that is efficiently handled by the L1 cache, reducing cache misses and improving performance. However, the observed behavior suggests otherwise.

A slab allocator divides memory into small, fixed-size chunks or "slabs" for allocation. While this can be efficient for managing objects of uniform size, it can lead to increased cache misses if objects are not optimally aligned or if there is fragmentation. The slab allocation pattern can cause data to be scattered across different cache lines, leading to inefficient cache usage.

In the k-means C program, which involves intensive computation and frequent memory accesses, L1 cache efficiency is crucial for performance. The presence of significant L1 cache misses indicates that the memory allocation is not as contiguous as expected. Instead, it suggests that the memory may be fragmented or scattered, similar to what happens with a slab allocator.

Given the behavior of ContigMalloc and the observed cache performance, it is speculated that ContigMalloc may be functioning in a manner similar to a slab allocator under the hood. This could involve internal mechanisms that break down the large contiguous allocation request into smaller chunks or slabs for management, inadvertently causing the observed L1 cache misses.

The deviation from truly contiguous allocation impacts the performance of programs like k-means, which rely on efficient memory access patterns. L1 cache misses can significantly slow down computation, as accessing data from higher-level caches or main memory introduces additional latency. To confirm this hypothesis, further investigation is needed. This includes analyzing the internal implementation of ContigMalloc to understand its allocation strategy, profiling the memory allocation patterns and cache usage in detail, and conducting controlled experiments to compare the behavior of ContigMalloc with known slab allocators.

If the hypothesis is confirmed, potential solutions could involve modifying ContigMalloc to ensure truly contiguous physical memory allocation, implementing custom memory allocators tailored to the specific needs of high-performance applications like k-means, and optimizing the existing allocation strategy to minimize fragmentation and improve cache utilization. In summary, while ContigMalloc is intended to provide contiguous memory allocation, its current behavior suggests a resemblance to slab allocation, leading to suboptimal L1 cache performance in the k-means C program. This calls for a deeper dive into the allocator's implementation and strategic adjustments to achieve the desired memory access efficiency.

Chapter 4

Future work

The current experimental setup on the ARM Morello board is constrained by the requirement that all memory reads must pass through the Translation Lookaside Buffer (TLB) for address translation. This necessitates frequent TLB lookups, potentially leading to performance bottlenecks. The planned future work aims to address this by leveraging CHERI (Capability Hardware Enhanced RISC Instructions) extensions on the RISC-V architecture, specifically using the Tooba implementation[5].

Storing Offsets Directly on Pointers

In the current ARM Morello setup, address translations rely on the TLB. The future approach on RISC-V Tooba involves storing the offset directly within the pointer. This is possible due to CHERI's 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.

Hardware Modifications:

The Bluespec design of the RISC-V processor will be modified to allow certain memory operations to bypass the TLB. This means that when a pointer with encoded offset and bounds is used, the system can directly compute the physical address from the capability information. This modification reduces the dependency on the TLB, decreasing latency

and improving performance, especially for frequent memory operations. Transition to a Single-Address-Space Operating System (SASOS)[16].

Concept of SASOS:

In traditional operating systems, there is a clear separation between user space and kernel space. This separation is enforced by memory protection mechanisms and address translation through the TLB. In a Single-Address-Space Operating System, this distinction is removed. Both user applications and the kernel share the same contiguous address space.

Advantages of SASOS with CHERI:

- **Simplified Memory Management :** Without the need to switch between user and kernel spaces, memory management becomes simpler and more efficient. The kernel allocator can be the same as the user space allocator, operating on a single, contiguous chunk of memory.
- **Unified Allocator:** The unified memory allocator can efficiently manage memory for both kernel and user applications, leveraging CHERI's capability-based protection to prevent unauthorized access. This reduces overhead and potential fragmentation issues associated with maintaining separate memory spaces.

Chapter 5

Research Timeline

This PhD plan outlines a comprehensive research project to explore, implement, and evaluate FAT-Pointer based range addressing mechanisms across various systems, including ARM with hugepages, RISC-V, and Uni-Kernels. The research is structured into several phases, each focusing on distinct aspects of the FAT-Pointer based range address mechanism. Each experiment constitutes a chapter in the final thesis.

5.1 FAT-Pointer based range addresses

5.1.1 July 1th - July 15th (2024)

Debug Errors Related to L1 TLB Misses for Contiguous malloc

1. Investigate the cause of L1 TLB misses when using contiguous memory allocation with the FAT-Pointer mechanism.
2. Utilize debugging tools and techniques to identify specific issues within the memory management unit (MMU) and translation lookaside buffer (TLB).
3. Implement and test fixes to ensure efficient TLB usage and reduced misses.

| CPP | Conference |
|-----------|--|
| March | ISMM (International Symposium on Memory Management) |
| April | SOSP (Symposium on Operating Systems Principles) |
| June | ASPLOS (International Conference on Architectural Support for Programming Languages and Operating Systems) |
| September | EuroS&P (European Symposium on Security and Privacy) |
| October | EuroSys |
| December | OSDI (USENIX Symposium on Operating Systems Design and Implementation) |

Table 5.1 List of relavent conferences to submit for the following PhD expirements.

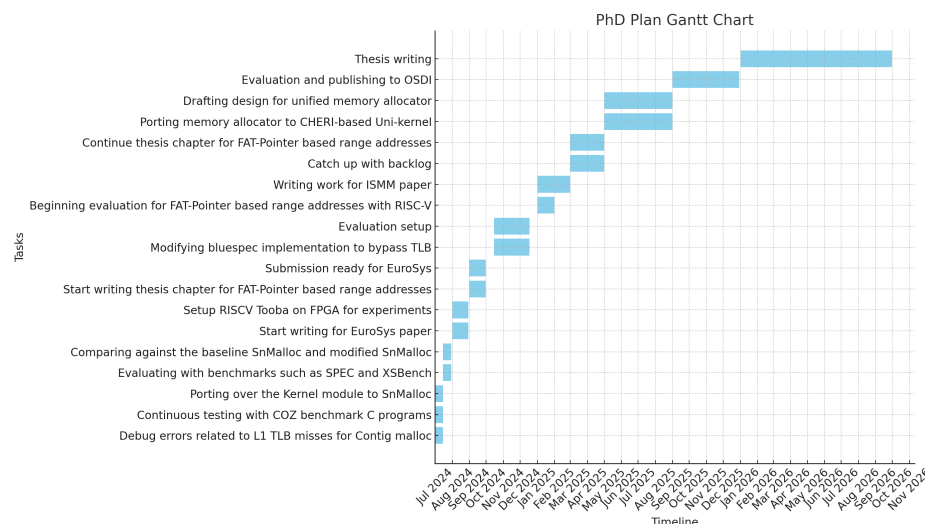


Fig. 5.1 PhD plan Gantt Chart

Continuous testing with COZ benchmark C programs

1. Run COZ benchmark C programs to evaluate the performance and correctness of the FAT-Pointer based range addressing.
2. Analyze the benchmarking results to identify potential performance bottlenecks.
3. Refine the FAT-Pointer implementation based on feedback from the benchmark tests.

Port the Kernel Module to SnMalloc, the Default CHERIBSD Kernel Allocator

1. Modify the existing ported contig kernel module to be compatible with SnMalloc.
2. Test the ported module to verify its functionality and performance.

5.1.2 July 15th - July 30th (2024)

Evaluate Performance Using Benchmarks Such as SPEC and XSBench

1. Set up and run SPEC and XSBench benchmarks to assess the performance of the FAT-Pointer based range addressing.
2. Collect and analyze performance metrics to gauge the impact of the FAT-Pointer mechanism.

Compare Results Against the Baseline SnMalloc and Modified SnMalloc Memory Allocator

1. Perform comparative analysis between the default SnMalloc, modified SnMalloc, and FAT-Pointer based allocator.
2. Document performance improvements or regressions to understand the benefits and drawbacks of the FAT-Pointer based range addresses implementation.

5.1.3 August 1st - August 30th (2024)**Begin Writing the Paper for EuroSys**

1. Start drafting the EuroSys paper focusing on the design, implementation, and performance evaluation of the FAT-Pointer based range addressing.
2. Ensure the setup is capable of running relevant tests and benchmarks for evaluating the FAT-Pointer mechanism on RISC-V architecture.

5.1.4 September 1st - September 30th (2024)**Start Writing the Thesis Chapter on FAT-Pointer Based Range Addresses**

1. Begin compiling and organizing research findings and insights into a comprehensive thesis chapter.
2. Provide detailed explanations of the design, implementation, and performance evaluations conducted.

Prepare the Submission for EuroSys

1. Finalize the EuroSys paper based on feedback and revisions.
2. Ensure all necessary documentation and supplementary materials are ready for submission.

5.2 FAT-Pointer based range addresses with RISC-V

5.2.1 October 15th - December 18th (2024)

Modify the Bluespec Implementation to Bypass the TLB

1. Engineer modifications in the Bluespec implementation to enable bypassing the TLB.
2. Test the changes to ensure they achieve the desired effect and maintain system stability.

Set Up the Evaluation Environment

1. Prepare the necessary hardware and software environment for evaluating the modified Bluespec implementation.
2. Ensure all tools and benchmarks are properly configured for accurate performance measurements.

5.2.2 January 1st - February 1st (2025)

Begin the Evaluation of FAT-Pointer Based Range Addresses with RISC-V

1. Conduct a thorough evaluation of the FAT-Pointer mechanism on the RISC-V architecture.
2. Collect performance data and analyze the results to assess the impact of the FAT-Pointer range address based RISC-V implementation.

Start Writing the Paper for ISMM

1. Begin drafting the ISMM paper, focusing on the technical details and evaluation results of the FAT-Pointer implementation on RISC-V.
2. Highlight the innovations and contributions of this work to the field.

5.2.3 February 1st - March 1st (2025)

Start Writing the Paper for ISMM

1. Refine the ISMM paper based on initial drafts and feedback.
2. Ensure the paper thoroughly covers the research, findings, and implications.

5.2.4 March 1st - May 1st (2025)

Catch Up on Any Backlog Tasks

1. Address any remaining tasks or issues that were postponed earlier.
2. Ensure all aspects of the research project are up to date.

Continue Writing the PhD Thesis Chapter on FAT-Pointer Based Range Addresses

1. Expand on the thesis chapter with detailed explanations of recent findings and evaluations.
2. Ensure the chapter provides a comprehensive overview of the FAT-Pointer research.

5.3 FAT-Pointer based range addresses with Uni-kernels

5.3.1 May 1st - September 1st (2025)

Catch Up on Any Backlog Tasks

1. Adapt the FAT-Pointer based memory allocator for use with a CHERI-based Uni-Kernel.
2. Test the ported allocator to ensure compatibility and performance within the Uni-Kernel environment.

Draft a Design for a Unified Memory Allocator

1. Develop a design for a unified memory allocator that can be used by both the kernel and applications in the Uni-Kernel.
2. Document the design considerations, challenges, and proposed solutions.

5.3.2 September 1st - December 30th (2025)

Perform Evaluation and Prepare a Publication for OSDI

1. Conduct performance and functionality evaluations of the unified memory allocator.
2. Compile results and draft a paper for submission to OSDI, highlighting the key contributions and findings.

5.3.3 January 1st - September 30th (2026)

Focus on Writing the PhD Thesis

1. Dedicate time to writing and refining the PhD thesis.
2. Ensure all research chapters are complete, well-organized, and thoroughly reviewed.

Chapter 6

Conclusion

The report addresses the growing disparity between application workloads and the capacity of Translation Lookaside Buffers (TLBs). To mitigate this gap, it proposes leveraging physically contiguous memory to optimize TLB utilization. Additionally, the report explores advancements in system security, particularly through the Capability Hardware Enhanced RISC Instructions (CHERI) architecture. CHERI's capability-based addressing enhances system security by associating capabilities with memory pointers, restricting access to memory regions, and thus protecting against various security threats. Importantly, these mechanisms can also improve the efficiency of memory allocators by managing memory resources while ensuring robust security measures.

The report highlights the constant pursuit of optimal performance in computing, emphasizing the importance of efficient memory management. TLBs are crucial in expediting memory access by storing recently accessed memory translations. However, as applications grow in size and complexity, TLB capacity often becomes a bottleneck. One innovative solution is the use of huge pages, which allocate memory in larger chunks, thereby reducing the number of TLB entries required and potentially enhancing overall system performance. Advancements in hardware-level security, such as CHERI's capability-based addressing, offer additional performance enhancement opportunities by tightly controlling memory access and accelerating memory management operations. Integrating huge pages into memory management strategies alongside CHERI's capability-based addressing can optimize TLB utilization and leverage security features for significant performance improvements.

This report aims to demonstrate how leveraging physically contiguous memory and advanced security architectures like CHERI can enhance memory management efficiency while ensuring robust security measures. These advancements ultimately contribute to

improved system performance, addressing the challenges posed by the increasing complexity and size of modern application workloads.

References

- [1] The architecture of the burroughs b-5000.
- [2] Arm architecture reference manual for a-profile architecture.
- [3] Benchmark ABI - CheriBSD 23.11 new features tutorial.
- [4] Capability-based computer systems.
- [5] CTSRD-CHERI/DE10pro-cheri-bgas. original-date: 2021-06-12T01:21:35Z.
- [6] Do-it-yourself virtual memory translation | ACM SIGARCH computer architecture news.
- [7] Getting started with CheriBSD 23.11 - getting started with CheriBSD 23.11.
- [8] Hugetlbfs reservation — the linux kernel documentation.
- [9] MSRC-security-research/papers/2020/security analysis of CHERI ISA.pdf at master · microsoft/MSRC-security-research.
- [Basu et al.] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient virtual memory for big memory servers.
- [Baumann et al.] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhanian, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM.
- [Bi and Wang] Bi, H. and Wang, Z.-H. DPDK-based improvement of packet forwarding. 7:01009. Publisher: EDP Sciences.
- [Chen et al.] Chen, D., Tong, D., Yang, C., Yi, J., and Cheng, X. FlexPointer: Fast address translation based on range TLB and tagged pointers. 20(2):1–24.
- [Curtsinger and Berger] Curtsinger, C. and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197.
- [15] Esswood, L. G. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.
- [16] Esswood, L. G. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.

- [Fillo et al.] Fillo, M., Keckler, S. W., Dally, W. J., Carter, N. P., Chang, A., Gurevich, Y., and Lee, W. S. The m-machine multicomputer.
- [Hardy] Hardy, N. KeyKOS architecture. 19(4):8–25.
- [Karakostas et al.] Karakostas, V., Gandhi, J., Ayar, F., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Ünsal, O. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 66–78. ACM.
- [Kwon et al.] Kwon, A., Dhawan, U., Smith, J. M., Knight, T. F., and DeHon, A. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, CCS '13*, pages 721–732. Association for Computing Machinery.
- [Panwar et al.] Panwar, A., Bansal, S., and Gopinath, K. HawkEye: Efficient fine-grained OS support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360. ACM.
- [Rashid et al.] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., and Jones, M. Mach: A system software kernel.
- [23] Watson, R. N. M., Anderson, J., Kennaway, K., and Laurie, B. Capsicum: practical capabilities for UNIX.
- [24] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., Murdoch, S. J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., and Xia, H. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8).
- [25] Watson, R. N. M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S. W., Napierala, E., Sewell, P., and Neumann, P. G. CHERI c/c++ programming guide.
- [Wilkes and Needham] Wilkes, M. V. and Needham, R. M. *The Cambridge CAP computer and its operating system*. Number 6 in The computer science library operating and programming systems series. North Holland.
- [Woodruff et al.] Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R. M., Chisnall, D., Davis, B., Gudka, K., Filardo, N. W., Markettos, A. T., Roe, M., Neumann, P. G., Watson, R. N. M., and Moore, S. W. CHERI concentrate: Practical compressed capabilities. 68(10):1455–1469.
- [Wulf et al.] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: the kernel of a multiprocessor operating system. 17(6):337–345.

- [Zaliva et al.] Zaliva, V., Memarian, K., Almeida, R., Clarke, J., Davis, B., Richardson, A., Chisnall, D., Campbell, B., Stark, I., Watson, R. N. M., and Sewell, P. Formal mechanised semantics of CHERI c: Capabilities, undefined behaviour, and provenance. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 181–196. ACM.

