Fat Address Translations

Akilan Selvacoumar*
as251@hw.ac.uk
Heriot Watt
Edinburgh, Scotland, United Kingdom

Abstract

The increasing gap between workload memory requirements and the capacity of translation lookaside buffers (TLBs) means TLB misses are more frequent, costing additional clock cycles and impacting runtime performance. One solution is to use physically contiguous memory in conjunction with huge pages. We propose an alternative approach, by exploiting capability-based addressing in the CHERI architecture. This paper presents a new memory allocator. It associates capabilities with memory pointers. Integrating block-based allocations within huge pages. Our allocator reduces TLB misses by up to 90%, which leads to decreasing runtimes for memory-intensive applications.

ACM Reference Format:
Akilan Selvacoumar. 2018. Fat Address Translations. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In computing, achieving high performance is an ongoing challenge, especially as applications handle increasingly memory intensive workloads. Memory management is a key factor in reducing the time it takes to access a memory region. Translation Lookaside Buffers (TLBs) are crucial in speeding up memory access by caching recent memory address translations. A TLB is a specialised cache in the memory management unit (MMU), It reduces the time required to convert virtual addresses to physical addresses. When a program accesses data in memory, the MMU first checks the TLB for a matching entry and avoids the slow process of accessing the page tables situated in memory. However, as applications grow more complex, the fixed size of TLBs often cannot keep up, leading to more TLB misses and performance slowdowns [1]. To tackle this issue, researchers have explored new solutions, including the use of huge pages [2].

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 optimising TLBs which are used by reducing the number of entries needed to map large memory regions. This not only decreases the frequency of TLB misses but also lowers the overhead associated with address translation. By minimising these bottlenecks, huge pages can improve system performance in aspects such as speeding up memory-intensive applications, reducing latency in data access, and enhancing throughput for workloads that rely heavily on large datasets.

Simultaneously, advancements in hardware-level security, such as the Capability Hardware Enhanced RISC Instructions (CHERI) [3] architecture presents additional opportunities for performance enhancement. CHERI's capability-based addressing approach not only strengthens system security by tightly controlling memory access but also opens avenues for optimising memory management operations. By integrating CHERI's compressed encoded bounds [4] with the use of huge pages, We have shown it is possible to track and manage large, physically contiguous memory blocks without requiring numerous TLB entries. This combination reduces the TLB pressure by minimising the number of entries required to map extensive memory regions, thereby decreasing TLB misses and improving address translation performance. Furthermore, we introduce FAT which accelerates memory-intensive tasks by reducing the overhead associated with managing non-contiguous memory allocations. The contributions for the following paper are as follows:

- **FAT Addresses Translations**: Introduces FAT that include memory bounds, allowing efficient tracking and management of physically contiguous memory regions (Section 3).
- **CHERI's Capability-based Optimisation**: Demonstrates how CHERI's architecture can be used to optimise memory allocation by encoding memory bounds directly within pointers, reducing TLB reliance (Section 3.2).
- **Memory Allocation Algorithms**: Provides an algorithms for allocating, freeing physically contiguous memory , and integrating huge pages with CHERI's capability-based bounds for enhanced memory management (Section 4).

Through comprehensive evaluation including micro and macro benchmarks, we demonstrate the allocators ability to reduce TLB misses by up to 90% which yields in significant improvements in wall clock runtimes for memory-intensive applications. While its impact on larger and computation-heavy workloads is less pronounced. The proposed allocator shows strong potential for advancing memory management in scenarios requiring high memory throughput by reducing the address translation overhead.

2 Related work

2.1 Huge Pages

Increasing TLB reach[5] can be achieved by using larger page sizes such as huge pages [2] which are common in modern computer systems. The x86-64 architecture supports huge pages of 2 MB

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

and 1 GB which are backed by OS mechanisms like Transparent Huge Pages (THP) [6] . 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 (Page Table Entries) and in contrast 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 organise huge pages and without significant changes to the conventional page table. It only helps reduce page walk overheads. Huge page tunable base page size permits the OS to adjust the base page, but still faces internal fragmentation problems with huge page recommending a base page size of no more than 16 KB. Shadow Superpage [7] introduces a new translation level in the memory controller to merge non-contiguous physical pages into a huge page in a shadow memory space by 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.

2.2 Direct Segment

Early processors often used segments to manage virtual memory, where a segment [8] 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. 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.

2.3 Redundant Memory Mapping (RMM)

Redundant Memory Mappings (RMM) [9] enhance memory management by introducing an additional range table that pre-allocates contiguous physical pages for large memory allocations thus creating ranges that are both virtually and physically contiguous. This approach simplifies address translation by adding ranges. RMM is similar to Direct Segment with the support of multiple ranges and operates transparently to programmers. No source code modifications is required on the programmers side. The range table which is 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. Since this process is computationally expensive hence it is therefore performed only after an L1 TLB miss. To optimise this, RMM uses a Range TLB (RTLB) to quickly identify if an address falls within any pre-allocated range. Range mapping works alongside the paging system by generating TLB entries on TLB misses and still performs TLB lookups for each virtual address translation. Unlike traditional segmentation mechanisms, range mapping activates the RTLB located with the last level TLB upon a miss. The hardware TLB miss handler then searches the RTLB for the missed address. If found, generates a new TLB entry with the physical address derived from the base virtual address and range offset along with the 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 on the background. This avoids delays for in-memory operations. The RTLB functions as a fully associative search structure ensuring that most last-level TLB misses are handled efficiently by range mapping which reduces the need for costly page table walks.

2.4 CHERI

CHERI extends conventional processor Instruction-Set Architectures (ISAs) with architectural capabilities to enable fine-grained memory protection and highly scalable software compartmentalisation. It is a hybrid capability architecture that can combine capabilities with conventional MMU (Memory Management Unit) based systems. The contributions of CHERI includes ISA changes to introduce architectural capabilities and is a new microarchitecture which shows that capabilities can be implemented efficiently in hardware. CHERI provides support for efficient tagged memory to protect capabilities and compresses them to reduce memory overhead. The CHERI ecosystem provides language and compiler extensions for using capabilities with C and C++. An OS extensions is also provided to support fine-grained memory protection (including spatial, referential, and non-stack temporal memory safety) and abstraction extensions for scalable software compartmentalisation.

2.5 CHERI CC

CHERI Concentrate: Practical Compressed Capabilities[4] introduces a compression scheme for CHERI that aims to address the performance and compatibility challenges associated with capability pointers. Capability pointers enhance memory safety by embedding bounds and permissions directly within pointers. Traditional implementations of CHERI double their size—leading in bounds to increased memory usage. CHERI CC proposes a compression strategy that preserves security while reducing size and inefficiencies. Key contributions include a floating-point bound encoding technique with an internal exponent mechanism that offers greater precision for smaller objects and optimised space usage for larger ones.

3 Fat Address Translations

Fat Address Translations (FAT) uses the CHERI architecture to bring about block-based allocations in physically contiguous memory. FAT leverages techniques like FlexPointer [10] and RMM [9] to reduce pressure on the TLB. A key component in this implementation is the use of range addresses with CHERI CC [4].

Figure 1 illustrates a comparison between standard memory allocation (*malloc()*) and the proposed FAT method. The standard approach involves a C program interacting with a custom allocator which uses 48-bit virtual addresses and a TLB walker (L1, L2 and L3 cache) to achieve non-contiguous allocation in physical memory. This typically results in more TLB entries and increased TLB misses

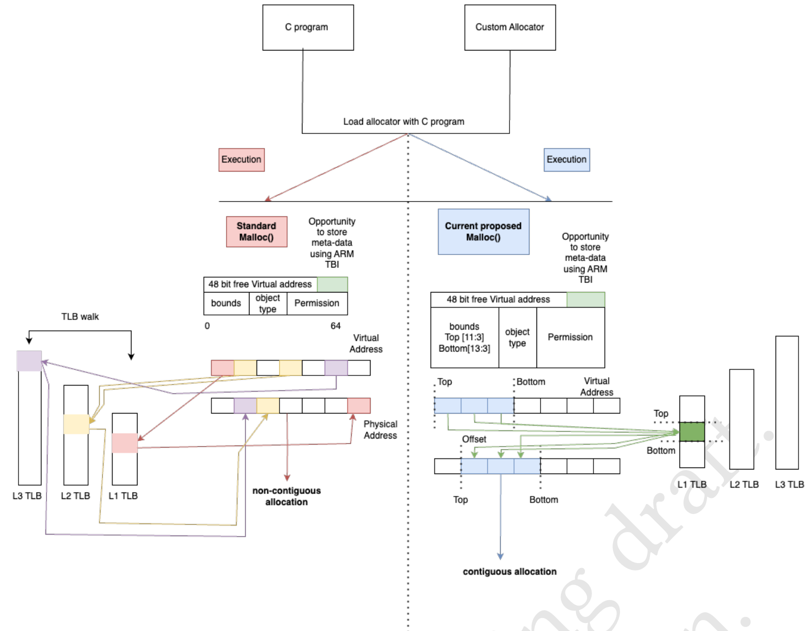

Figure 1: High overview architecture

increasing the reasoning to have more TLB walks. In contrast, the FAT Address Translations method employs a custom allocator leveraging physically contiguous memory by using CHERI to encode bounds within the pointers and as shown in the figure 1 (there is almost no reliance on walking the TLB hierarchy).

3.1 Encoding Ranges as Bounds to the Pointer

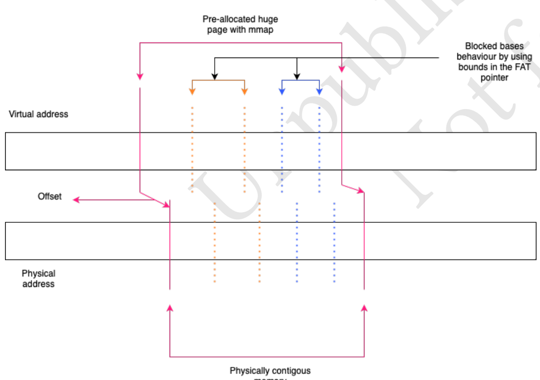

Figure 2: Range of memory

A memory range in FAT has two points to track memory in physical contiguous space which is the top and bottom. These two points are two virtual addresses and the range consists of addresses that lie within this and refers to addresses allocated by invoking *malloc*. FAT memory ranges are established using bounds encoded within the pointer, adhering to CHERI CC [4].

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 a huge page. The allocations use the bounds encoded in FAT which ensures tracking of the allocated memory regions. By using the CHERI bounds, this method maintains the contiguity of the allocated blocks within the huge page.

3.2 128 bit compressed bounds

We use CHERI CC [4] to track regions of memory in physically contiguous space. CHERI CC consists of compressed bounds that represent a 128-bit pointer within a 64-bit virtual address system. Our approach uses the work of CHERI CC by using a single cycle to decode bounds from the capability register and the bounds that are decoded are repurposed for tracking memory which is eagerly allocated.

Allocators like Jemalloc typically allocate objects under 512 bytes. When an object's bounds cannot be precisely represented, padding is required to ensure memory safety. However, it has been observed that Jemalloc rarely needs more than 6 bits to store the exponent values within compressed bounds (as shown in [4]). This means that the default behavior of allocators such as Jemalloc, would allow precise representation of bounds within CHERI CC.

Instead of relying on fixed-size TLB entries with set page sizes (such as 4KB, 2MB, or 1GB), FAT uses CHERI CC to define dynamic bounds based on the size requested at allocation time (e.g., during a *malloc* call). This approach offers a more flexible alternative to the traditional fixed-size TLB model.

3.3 Instrumenting Block-Based Allocators with Physically Contiguous Memory

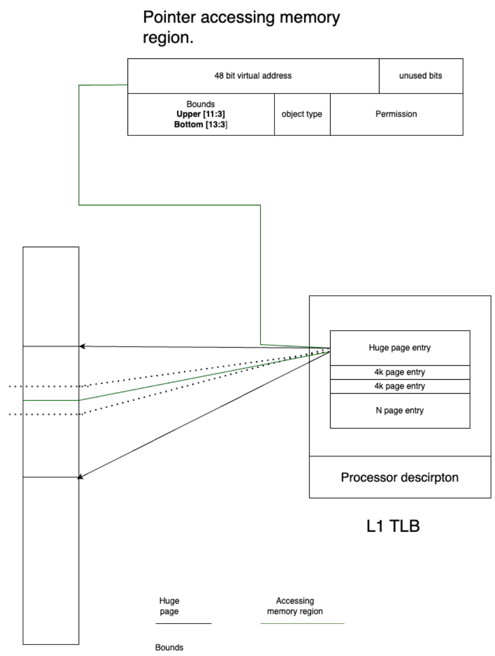

Figure 3: FAT Address Translations using huge pages

We are able to pre-allocate memory using huge pages and are able to mark smaller allocations using ranges by storing them as bounds within the pointer. Each of these memory ranges can be called a block. Since we have numerous blocks inside a huge page we allow block-based memory patterns within physically contiguous memory.

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

4 Memory allocator design

This section presents a straightforward memory allocator designed and is implemented based on the principles outlined in FAT (Section 3). The allocator consists of three core functions: *InitAlloc*, *malloc*, and *free*. The *InitAlloc* function initialises 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 specified size. When the *free* function deallocates memory it is returned to the pool for future use.

When the *malloc* function (Algorithm 1) is invoked, the algorithm employs an eager allocation strategy for physical memory. This is achieved through the use of the SetBounds mechanism. This constructs a FAT-specialised pointer that encodes both the start and

Algorithm 1 Malloc implementation

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

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 method of memory tracking, where the bounds of the allocated region is explicitly encoded in the address which enables efficient monitoring and management of memory usage.

Furthermore, this design uses shared huge page TLB 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 reducing the risk of out-of-bounds errors. This use of FAT and shared TLB entries not only align with the principles of efficient memory management but also demonstrate a practical use case of huge pages in CHERI.

Algorithm 2 Free implementation

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

The memory deallocation (Algorithm 2) mechanism in the proposed allocator is facilitated by the FAT structure introduced in the *malloc* algorithm. When the *free* function is invoked, it uses the metadata embedded within FAT to determine the range and size of the allocated memory region. Specifically, FAT encodes the start and end addresses of each allocation and provides the information needed to identify the memory block to be deallocated. This enables the allocator to accurately unmap the corresponding memory region from the address space.

By extracting the bounds and size directly from FAT, the *free* function eliminates the need for additional metadata lookups or complex data structures.

Algorithm 3 describes the initialisation of physically contiguous memory through the use of huge pages which is a mechanism supported by modern architectures to optimise memory management. The algorithm begins by allocating a fixed block of 1 GB 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. which

Algorithm 3 Init alloc function to create a initial 1 GB huge page

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

is critical for performance-sensitive applications and kernel-level operations.

5 Evaluation

We conducted tests of the FAT memory allocator against Jemalloc [11]. Jemalloc is the default memory allocator for CHERIBSD [12]. We evaluated the reduction in TLB walks and misses and its impact on wall clock runtime.

To comprehensively analyse the proposed allocator, we categorised benchmarks into two classes which are micro and macro benchmarks. Micro benchmarks comprise smaller C programs designed to target specific allocator patterns which enables us to evaluate detailed aspects of the allocators behavior. Macro benchmarks, on the other hand, encompass larger real-world C programs allowing us to assess the allocators performance in a more practical and real-world scenarios.

5.1 Experiment setup

The CHERI Morello [13] board was used to evaluate the proposed memory allocator. Morello implements the ARM A76 with enhanced server-class memory, featuring a quad-core ARM CPU with capability extensions. The L1 and L2 caches were modified to proliferate the capability bit which ensures compatibility with CHERI's capability-based memory model. When compiling the C programs for benchmarking, the Benchmark ABI was used as recommended by the CHERI community. This compilation mode was enabled using the Clang compiler.

The Benchmark ABI [14] was specifically designed because the Morello branch predictor was not expanded to predict bounds. Consequently, a capability-based jump introduces stalls in later PCC-dependent instructions until bounds are established. This issue is particularly significant during dynamically linked calls and returns between libraries where bounds are changed to cover the called or returned-to library. Such stalls can negatively affect performance, making the Benchmark ABI an essential consideration for this evaluation.

Each C program was executed using two different memory allocators. The first was the modified C allocator which is imported as a header file. This approach was necessary because the Benchmark ABI shared object file exhibited unexpected behavior by failing to overwrite the C program at runtime with the intended *malloc* functions. The second allocator was the standard OS memory allocator, which in the case of CHERIBSD is Jemalloc.

Performance measurements were carried out using ARM performance counters [15] to ensure accurate evaluation. These counters provided detailed metrics allowing us to compare the performance of the two allocators and assess the impact of the proposed changes.

5.2 Benchmarks

We elaborate here on the two classes of benchmarks [16]. Micro benchmarks (Section 5.2.1). focused on particular allocation and deallocation patterns such as sequential and random memory accesses. This is to stress-test the allocator under controlled conditions. Macro benchmarks involves real-world applications offering insights into how the allocator performs with complex memory allocation demands such as large datasets with varying execution contexts.

5.2.1 *Micro benchmark.*

- `GLIBC`: The Glibc benchmark evaluates the performance of *malloc* and *free* functions in single-threaded, multi-threaded, and emulated multi-threading scenarios using various block sizes allocation patterns. It simulates real-world memory usage by partially deallocating blocks in FIFO order and fully deallocating them in LIFO order. Results are gathered across configurations to analyse performance variations.
- `MemAccess`: This benchmark evaluates the performance impact of memory access patterns by constructing and traversing a doubly linked list with varying working set sizes. It supports sequential or randomised structures with optional node operations and multithreaded traversal using pthreads. The program dynamically allocates memory and systematically doubles the working set size to analyse memory hierarchy behavior.

5.2.2 *Macro benchmark.*

- `Kmeans`: Kmeans implements a parallelised K-means clustering algorithm that assigns data points to clusters based on proximity to the centroids. This iteratively updates them until convergence. The computation is distributed across threads using the pthread library, dynamically assigning tasks to optimise performance. Parameters like data size and clusters are configurable and the program ensures efficient memory management and synchronisation.
- `Richards`: Richards is a task scheduling benchmark that simulates a multitasking environment with tasks of varying types and priorities which is communicated through queued packets. The schedule function manages task execution based on the state, priority and tracks processed packets which are held tasks for performance evaluation. Configurable iterations and timing help measure system performance to ensure correctness.
- `BARNES`: Implements the Barnes-Hut algorithm to efficiently simulate the interactions within an N-body system. A comprehensive overview of the Barnes-Hut method is provided by Singh in his doctoral dissertation [17]. The implementation we benchmark extends the original method by permitting multiple particles to be stored within each leaf cell of the spatial decomposition, enhancing performance and scalability. This extension is described by Holt and Singh [18].

5.3 Results

The graph (Figure 4) highlights the performance comparison between the modified memory allocator and Jemalloc, the default memory allocator. The FAT memory allocator is specifically optimised for use with huge pages and demonstrates a clear advantage in scenarios where memory allocation patterns benefit from its design. The results align with expectations, showcasing the impact of its capability to handle memory more efficiently by leveraging huge pages.

- L1 DTLB reads: There was a noticeable reduction of L1 DTLB reads for kmeans of about an average of 4% lesser and for Glibc there was significant reduction of 50% less than Jemalloc.
- L2 DTLB reads: For all the benchmarks on figure 4 there was on average 98% reduction in L2 DTLB reads. This demonstrates that all the TLB translations are read at the L1 TLB cache.
- DTLB walks: Due to most of the TLB entries getting hit at the L1 DTLB, there is no need to walk the TLB cache hierarchy. This is shown by an average of 99% reduction in DTLB walks.
- L1 DTLB refills: Since there are fewer DTLB walks and most reads are done at the L1 DTLB layer there is no need for numerous TLB refills to take place. Our benchmarks show on average a 99% reduction in DTLB refills.

A particularly striking observation is the significant reduction in data TLB walks, L2 data TLB reads and TLB refills-consistently which show a 90% decrease across all benchmarks compared to Jemalloc. This improvement is due to the modified allocators use of a single huge page entry at the L1 TLB layer. By enabling most address translations to be resolved directly at the L1 TLB, the need to walk through the deeper TLB hierarchy is largely eliminated. This reduction in translation overhead is a key factor in the allocators performance for certain types of workloads.

The microbenchmarks which are crafted to emphasise memory read operations, highlight the allocators strengths. These tests simulate frequent and intensive memory access patterns, where the reduction in TLB misses directly translate into measurable performance gains. On average, the FAT allocator achieves a 50% reduction in wall clock runtimes for these workloads. Underscoring its ability to optimise high-throughput memory operations.

Table 1: ARM performance counters

Performance counter	Description
Wall clock	The actual time taken from the start of a computer program to the end.
(p/l1d_tlb_rd) L1 data TLB reads	Level 1 data TLB access, read
(p/l2d_tlb_rd) L2 data TLB reads	Level 2 data TLB access, read
(p/l1d_tlb_refill) L1 data TLB refills	Level 1 data TLB refill. The Level 1 data TLB refill counter tracks each access to the L1D_TLB that results in a refill of the Level 1 data or unified TLB. This includes any access that requires a memory lookup due to a translation table walk or accessing another level of TLB cache.
(p/cpu_cycles) CPU cycles	The CPU CYCLES counter increases with every clock cycle. However, it can be affected by changes in clock frequency, such as when WFI (Wait for Interrupt) or WFE (Wait for Event) instructions pause the clock.
(p/dtlb_walk) Data TLB walks	Data TLB access with at least one translation table walk.
(p/ll_cache_miss_rd) Last level cache miss reads	Last level cache miss, read (This refers to every miss in the Last level cache that occurs during a memory read operation.)

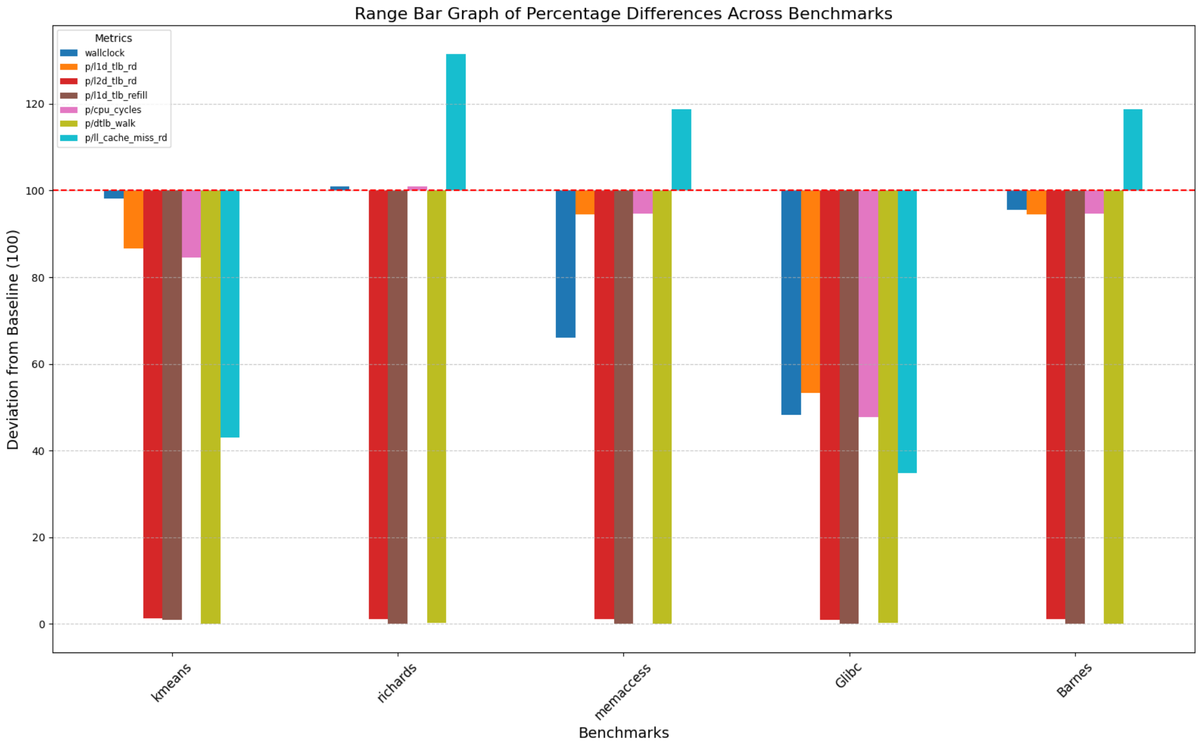

Figure 4: Percentage difference between the modified memory allocator against the default system memory allocator

On the other hand, macro benchmarks which represent larger and more complex real-world applications, exhibit minimal differences in wall clock runtimes when using the FAT allocator. This outcome is expected, as macro benchmarks typically involve a broader range of operations beyond memory allocation. Additionally, the benefits of huge pages may be less pronounced for these workloads, as they are often bottlenecked by factors such as computation or I/O rather than memory translation overhead.

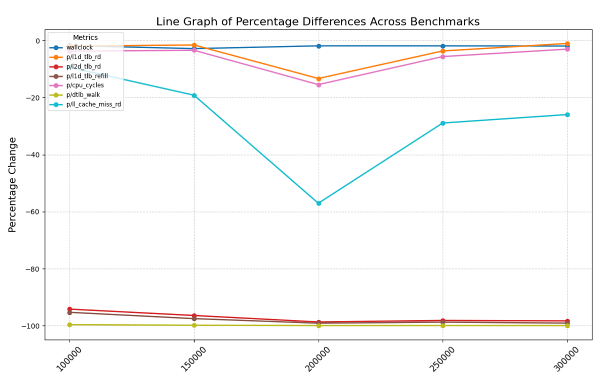

Figure 5: Kmeans COZ benchmark executed against various cluster sizes

The K-means algorithm was executed with varying cluster sizes to evaluate the performance difference between the FAT allocator and Jemalloc as the workload scales. This analysis aims to understand how the allocators optimisations, particularly its ability to manage memory more efficiently with huge pages, impact performance under different workload conditions.

For most cluster sizes tested, the percentage difference in performance remained relatively consistent. This indicates that the allocators efficiency scales predictably with increasing workload sizes. Suggesting a stable and uniform benefit across different configurations. The consistent performance gain is likely due to the allocators ability to minimise TLB misses and efficiently manage memory allocations for the centroid and data point structures used in the K-means algorithm.

However, an anomaly was observed at a cluster size of 2000, where the percentage difference deviated significantly from the trend. At this cluster size, the memory access patterns and allocation behavior may align in a way that temporarily offsets the advantages of the FAT allocator. For example, the memory layout might interact with system-level caching mechanisms or TLB behavior differently leading to an unexpected change in performance. Additionally, the increased complexity of managing a higher number of clusters might introduce computational overhead that overshadows the memory allocators optimisations.

5.4 Analysis

The FAT memory allocator demonstrates significant potential for enhancing memory management in systems that benefit from huge page optimisations. Its design effectively reduces TLB misses, achieving up to 90% fewer data TLB walks, L2 TLB reads, and TLB refills compared to Jemalloc. These improvements lead to noticeable performance gains especially in micro benchmarks, where the allocator reduces wall clock runtimes by an average of 50%.

The allocator integrates seamlessly into memory-intensive workloads, as evidenced by its consistent performance across varying cluster sizes in the K-means benchmark with only minor anomalies observed under specific conditions. These outliers provide valuable insights into the allocators interaction with system-level caching and memory translation mechanisms.

While the allocator excels in scenarios emphasising on high-memory throughput. Its impact on macro benchmarks is less pronounced. This suggests that its benefits are most relevant for applications with frequent and intensive memory operations rather than those constrained by computation or I/O bottlenecks.

6 Conclusion

This paper addresses the growing disparity between application workloads and the capacity of TLBs. To mitigate this gap, we proposed leveraging physically contiguous memory with CHERI bounds to reduce TLB walks. We designed a memory allocator that uses huge pages with the CHERI CC scheme to track allocations within the allocated huge page. This approach reduces the number of TLB entries needed while using bounds to minimise fragmentation.

The benchmarks demonstrate that the allocator reduces TLB misses by up to 90%, leading to substantial performance gains in memory-intensive workloads, though the improvements are less pronounced for larger and computation-heavy applications. These results highlight the allocators potential to advance memory management by repurposing CHERI's capability-based model with the use of huge pages.

References

[1] Sparsh Mittal. A survey of techniques for architecting TLBs. 29(10):e4061. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4061.

[2] Ashish Panwar, Sorav Bansal, and K. Gopinath. 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.

[3] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: revisiting RISC in an age of risk. 42(3):457–468.

[4] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI concentrate: Practical compressed capabilities. 68(10):1455–1469.

[5] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, 2014.

[6] Juan Navarro. Practical, transparent operating system support for superpages.

[7] Cheol Ho Park and Daeyeon Park. Aggressive superpage support with the shadow memory and the partial-subblock tlb. *Microprocessors and Microsystems*, 25(7):329–342, 2001.

[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News*, 41(3):237–248, June 2013.

[9] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 66–78. ACM.

[10] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. FlexPointer: Fast address translation based on range TLB and tagged pointers. 20(2):1–24.

[11] JEMALLOC.

[12] Benchmark ABI - CheriBSD 23.11 new features tutorial.

[13] Department of computer science and technology – CHERI: The arm morello board.

[14] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, September 2023.

[15] Arm architecture reference manual for a-profile architecture.

[16] CHERI-allocator/benchmarks/benchmarks/StressTestMalloc/glibc-bench.c at main · akilan1999/CHERI-allocator.

[17] Jaswinder Pal Singh. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.

[18] C. Holt and Jaswinder Pal Singh. Hierarchical n-body methods on shared address space multiprocessors. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995. To appear.