FAT allocator

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) in hardware cache management of modern processors means that TLB misses are more frequent, costing additional clock cycles and impacting runtime performance. One solution that has been explored is to use physically contiguous memory in conjunction with huge pages.

The contribution is an alternative approach by exploiting capability-based addressing in the CHERI architecture. This paper presents a new memory allocator called Fat Address Translations (FAT) which associates capabilities with memory pointers by integrating block-based allocations within huge pages. The FAT allocator when ran independently and embedded inside Jemalloc reduces walking the TLB hierarchy by upto 90%, which leads to decreasing runtimes for memory read and write intensive applications.

ACM Reference Format:
Akilan Selvacoumar. 2018. FAT allocator. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 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. Performance referring to memory access rather than compute bound. Memory management is a key factor in reducing the time it takes to access a memory region. A Translation Lookaside Buffer (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. TLBs are crucial in speeding up memory access by caching recent memory address translations. The TLB hierarchy [1] comprises of a smaller, faster L1 TLB situated close to the processor core and a larger slightly slower L2 TLB, both of which are usually private to each core thereby reducing the latency associated with page table lookups. However, as applications grow more complex the TLBs often cannot keep up with the number of translation entries leading to more TLB misses which requires walking the TLB hierarchy and this leads to performance slowdowns [2]. To tackle this issue, researchers have explored new solutions, including the use of huge pages [3].

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 and therefore minimising these bottlenecks.

Simultaneously, advancements in hardware-level security, such as the Capability Hardware Enhanced RISC Instructions (CHERI) [4] which are pointers that are replaced with capabilities with 128-bit or 256-bit encoding scheme which consists of both the address and metadata. The metadata includes bounds, permissions and validity of the pointer which forms a capability. This transformation enforces strict memory safety as capabilities prevent arbitrary pointer arithmetic and unauthorised memory access. 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 [5] with the use of huge pages, one of the contribution is demonstrating that it is possible to track, manage large and 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, this paper introduces the Fat Address Translations(FAT) allocator which accelerates memory-intensive tasks by reducing the overhead associated with managing non-contiguous memory allocations by emulating block allocations on physically contiguous memory to reduce the TLB pressure. 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 4).
- **Memory Allocation Algorithms (FAT allocator)**: Provides an algorithm for allocating, freeing physically contiguous memory , and integrating huge pages with CHERI's capability-based bounds for enhanced memory management (Section 5).

Conference acronym 'XX, Woodstock, NY

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

- **Modified Jemalloc Memory Mapping with FAT allocator**: Modification to Jemalloc with the FAT allocator to support physically contiguous memory allocation using a block-based strategy with capability-based addressing (Section 6).

Through evaluating micro and macro benchmarks, FAT though the use of CHERI's capabilities and huge pages demonstrates the allocator's ability to reduce TLB misses by up to 90% which yields to improvements of wall clock runtimes by upto 6% 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[6] can be achieved by using larger page sizes such as huge pages [3] which are common in modern computer systems. The x86-64 architecture supports huge pages of 2 MB and 1 GB which are backed by OS mechanisms like Transparent Huge Pages (THP) [7] . 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 [8] 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 [9] 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 [10] 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) [11] 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 FlexPointer

The key insight behind FlexPointer [12] is that large memory objects are relatively uncommon. This allows memory ranges to be constructed around them and assigned unique identifiers. These range IDs are embedded within the unused bits of pointers, enabling direct indexing of the range TLB and simplifying its design. Since the range ID is unaffected by address generation, range lookups can occur earlier in the pipeline, concurrently with address computation. Simulation results show that FlexPointer significantly reduces L1 TLB misses and the need for page walks in a range of memory-intensive workloads. When compared with a traditional 4KB-page system, FlexPointer delivers an average performance improvement of 14%, with peak gains of up to 2.8×, and introduces no performance regressions in less demanding scenarios.

2.5 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.

2.6 CHERI CC

CHERI CC(Concentrate Compressed Capabilities) [5] 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) is our new memory allocator for CHERI. It uses the CHERI architecture to bring about block-based allocations in physically contiguous memory. FAT leverages techniques like FlexPointer [12] and RMM [11] to reduce pressure on the TLB. A key component in this implementation is the use of range addresses with CHERI CC [5].

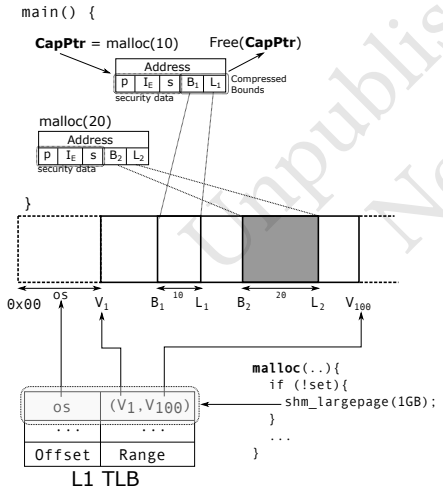


Figure 1: High overview architecture

Figure 1 illustrates a high-level overview of the FAT allocator. The rectangular box represents addresses in memory, and the box from v1 to v100 represents a physically contiguous region of memory. This physically contiguous region is typically allocated using huge pages. When malloc is called, a capability pointer is returned with metadata related to the bounds information encoded within the pointer. The bounds encoded in the pointer are reused for tracking memory blocks within a huge page. As shown, this in turn reduces TLB pressure.

3.1 Encoding Ranges as Bounds to the Pointer

Range is defined from work based on RMM [11] and FlexPointer [12]. In RMM a range refers to a contiguous region of memory defined by a base address and a bound. This information is stored in a hardware managed table called Range Table which is called when there is a L1 TLB miss. FlexPointer builds up on the work of RMM and stores the ID value between 48th bit and 64th bit which is the value to check the range table on parallel to the L1 TLB lookup.

The FAT allocator builds up on the concept of range from RMM and FlexPointer. Instead of using a hardware range table using CHERI, range information can be encoded within a capability pointer. A memory range in FAT has two points to track memory in physically 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 [5] compression scheme.

4 128 bit compressed bounds

FAT uses CHERI CC [5] 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 [5]). 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.

This means that the default behavior of most allocators, such as Jemalloc, would allow precise representation of bounds within a FAT. These pointers can then be repurposed as memory ranges in custom memory allocators, offering a more flexible alternative to fixed-size TLB entries.

4.1 Instrumenting Block-Based Allocators with Physically Contiguous Memory

FAT is able to pre-allocate memory using huge pages and is 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 there are numerous blocks inside a huge page, This

allows for abbreviated block-based memory patterns within physically contiguous memory.

By consolidating address translations into a single TLB entry, this method cuts down on the overhead of managing many entries. It also takes advantage of the bounds encoded within FAT to track and access memory within physically contiguous memory.

5 Memory allocator design

This section presents a straightforward memory allocator design which 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.

Algorithm 1 Malloc implementation

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

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 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

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

The memory deallocation (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 Init alloc function to create a initial 1 GB huge page

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

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

6 Embedding FAT allocator inside Jemalloc

This section describes about the FAT allocator implementation (Section 5) embedded inside Jemalloc. The objective here is to describe the changes needed for a block based allocator to use physically contigous memory with a block based strategy with the help of capability based addresses. In the case of Jemalloc the only changes required was to replace the mmap with the *malloc* function (Algorithm 1) and for munmap the *free* function (Algorithm 2).

6.1 Mmap replaced with MALLOC

The os_pages_map function (Algorithm 4) simulates jemalloc's low-level page mapping routine. It first checks if a specific address is requested in case relevant to CheriABI where such behavior is disallowed and returns NULL. It then allocates memory using the custom MALLOC(size) (Algorithm 1) function and validates whether the returned pointer matches the requested address if one was provided. If there's a mismatch, it unmaps calling FREE() (Algorithm 2) the memory and returns NULL; otherwise, it returns the allocated pointer.

This approach mentioned above is embedded inside jemalloc's strategy of managing memory through arenas and size classes. In jemalloc, memory is divided into chunks, which are further

Algorithm 4 Modified Jemalloc Memory Mapping Routines

Function: OS_PAGES_MAP(ADDR, SIZE, ALIGNMENT, COMMIT)
Require: addr aligned to os_page, size aligned to os_page, size ≠ 0
Ensure: If addr is non-NULL (CheriABI), return NULL
1: **if** addr ≠ NULL **then**
2: **return** NULL
3: **end if**
4: **if** os_overcommits **then**
5: commit ← true
6: **end if**
7: ret ← **MALLOC**(size)
8: **assert**(ret ≠ NULL)
9: **if** addr ≠ NULL **and** ret ≠ addr **then**
10: os_pages_unmap(ret, size)
11: ret ← NULL
12: **end if**
13: **return** ret

Function: PAGES_MAP(ADDR, SIZE, ALIGNMENT, COMMIT)
Require: alignment ≥ PAGE, addr aligned to alignment
14: ret ← **MALLOC**(size)
15: **return** ret

Function: PAGES_COMMIT_IMPL(ADDR, SIZE, COMMIT)
Require: addr aligned to PAGE, size aligned to PAGE
16: result ← **MALLOC**(size)
17: **if** result ≠ addr **then**
18: os_pages_unmap(result, size)
19: **return** true
20: **end if**
21: **return** false

subdivided into runs and regions to handle allocations of various sizes efficiently. By aligning sizes and managing allocations within predefined structures, jemalloc minimizes fragmentation [13].

Algorithm 5 os_pages_unmap

Require: addr aligned to os_page, size aligned to os_page
Ensure: Memory region at addr is unmapped
1: **assert**(ALIGNMENT_ADDR2BASE(addr, os_page) = addr)
2: **assert**(ALIGNMENT_CEILING(size, os_page) = size)
3: **FREE**(addr)

6.2 Mumap replaced with FREE

The os_pages_unmap (Algorithm 5) represents a customized abstraction of jemallocs memory unmapping routine, designed to integrate with the previously defined simplified free(ptr) implementation. In conventional jemalloc configurations, os_pages_unmap would invoke low-level system calls such as munmap to release virtual memory pages back to the operating system. However, in this adapted version, the function instead delegates the deallocation to a higher-level FREE(addr) (Algorithm 2).

The function begins by enforcing two invariants through assertions: first, that the input address addr is aligned to the operating system's page size and second that the size of the memory region is also a multiple of os_page. These alignment checks are critical for maintaining consistency with jemalloc's internal page-based memory management semantics and ensuring compatibility with the allocator's expectations. Following these checks the memory at the specified address is deallocated via the FREE(addr) (Algorithm 2) operation.

The following changes done to free is embedded inside jemalloc's deallocation mechanism, where metadata associated with each allocation (such as size and location) is used to efficiently return memory to the appropriate arena or pool. jemalloc maintains separate metadata structures to track allocations, allowing for quick deallocation and reuse of memory blocks without significant overhead [13].

7 Evaluation

Benchmarks of the FAT memory allocator and the FAT allocator embedded within Jemalloc against the standard Jemalloc [14] allocator was conducted. Jemalloc is the default memory allocator for CHERIBSD [15]. The objective was to evaluate the reduction of TLB walks ,misses and its impact on the wall clock runtime.

To comprehensively analyse the proposed allocator, the benchmarks [16] were categorised 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.

7.1 Experiment setup

The CHERI Morello [17] 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 [18] 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 (table 1) were carried out using ARM performance counters [19] 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.

7.2 Benchmarks

Elaborated here are two classes of benchmarks. Micro benchmarks focus on particular allocation, 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.

7.2.1 *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.

7.2.2 *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.

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.)

- `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 [20]. This implementation 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 [21].

7.3 Results

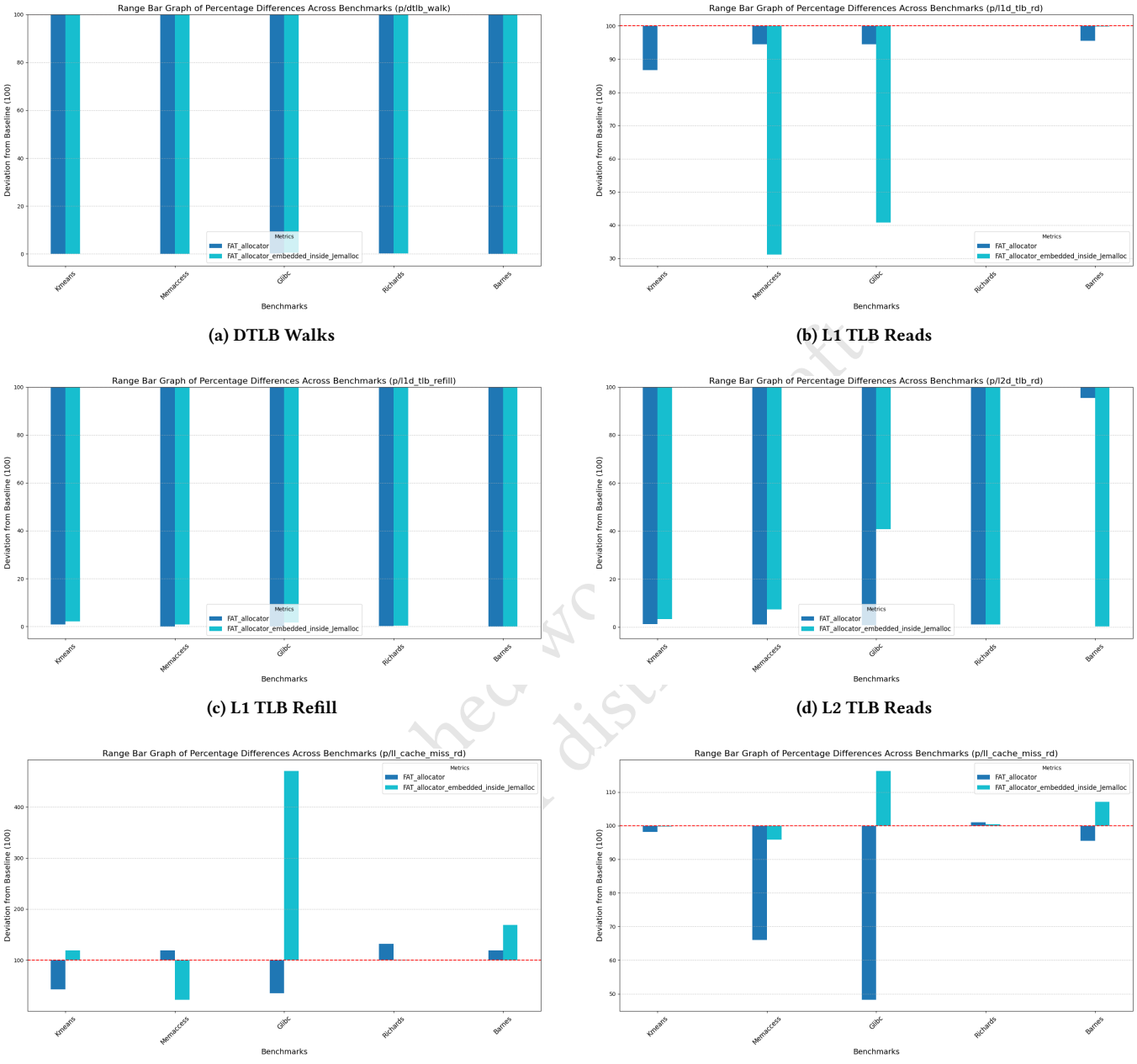
Figure 2 highlights the performance comparison between the modified memory allocator and Jemalloc, the default memory allocator. The FAT memory allocator is specifically optimised for use with huge pages and demonstrates a clear advantage in scenarios where memory allocation patterns benefit from its design. The results align with expectations, showcasing the impact of its capability to handle memory more efficiently by leveraging huge pages.

- L1 DTLB reads: L1 DTLB reads are critical for achieving fast memory access; therefore, a reduction in events that could signify misses or lead to further lookups is generally beneficial. In the Kmeans benchmark, the FAT allocator demonstrated 23% fewer L1 DTLB reads than the baseline allocator. However, when the FAT allocator was embedded within Jemalloc, the L1 DTLB reads were the same as the baseline. For the memaccess benchmark, the embedded FAT allocator resulted in 72% fewer L1 DTLB reads compared to the baseline allocator. A similar pattern was observed with Glibc, showing 62% fewer L1 DTLB reads. In the Richards benchmark, there was no discernible difference for either allocator. For the Barnes benchmark, the FAT allocator exhibited 5% fewer L1 DTLB reads, and the result was the same for the FAT allocator embedded within Jemalloc.
- L2 DTLB reads: L2 Data TLB reads (or lookups) serve as a secondary cache for address translations, and, similar to L1 TLB, lower values are indicative of better performance. FAT allocator consistently performed at the baseline (100%) for this metric across all benchmarks except Barnes. FAT allocator embedded inside Jemalloc also showed no significant change for Kmeans, Memaccess, and Richards. However, for the Glibc benchmark, it achieved a significant reduction of 60% in L2D TLB reads. This mirrors its L1D TLB improvement for Glibc and suggests that its strategy for page locality extends effectively to deeper levels of the TLB hierarchy for this particular benchmark, which is notable given Glibc's high frequency of malloc calls that stress memory management. A minor reduction of 3% was also observed for the Barnes benchmark with FAT allocator embedded inside Jemalloc.
- DTLB walks: which occur when a virtual-to-physical address translation is not found in the DTLB and a page table traversal is necessary, represent a performance cost. thus, fewer walks are preferable. In the observed tests neither FAT allocator nor FAT allocator embedded inside Jemalloc demonstrated any significant deviation from the baseline performance (100%) for this metric. This consistent behavior was noted across all benchmarks evaluated: Kmeans, Memaccess, Glibc, Richards, and Barnes. This outcome suggests that the memory allocation strategies employed by these allocators do not substantially alter the frequency of DTLB misses that necessitate page table walks within these specific workloads. Even for the Memaccess benchmark, which is designed with random list traversals that can stress TLB pressure, the impact on dTLB walks was negligible according to the provided graphs.
- L1 DTLB refills: L1 Data TLB refills are a direct consequence of L1D TLB misses, with fewer refills indicating better performance. Interestingly, despite the variations observed in L1D TLB reads for FAT allocator embedded inside Jemalloc, the L1D TLB refills metric remained at the baseline (100%) for both FAT allocator and FAT allocator embedded inside Jemalloc. This consistent performance at baseline was observed across all tested benchmarks: Kmeans, Memaccess, Glibc, Richards, and Barnes. This outcome presents a point of potential inconsistency; if L1D TLB reads (interpreted as events related to misses or lookups that could lead to misses) are reduced, a corresponding decrease in actual refills would typically be expected.

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 micro benchmarks 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.

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.z



(a) DTLB Walks

(b) L1 TLB Reads

(c) L1 TLB Refill

(d) L2 TLB Reads

(e) LL Cache Reads

(f) Wall Clock Time

Figure 2: Benchmarks comparing the percentage difference between FAT.

7.4 Analysis

The FAT memory allocator and the modified Jemalloc 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 the system allocator (i.e default Jemalloc). These improvements lead to noticeable performance gains especially in micro benchmarks, where the allocator reduces wall clock runtimes by an average of 50%.

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 are compute-bound workloads.

8 Conclusion

This paper addresses the growing disparity between application workloads and the capacity of TLBs. To mitigate this gap, FAT proposed leveraging physically contiguous memory with CHERI bounds to reduce TLB walks. FAT is 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 demonstrates the FAT allocator and the FAT allocator embedded within Jemalloc which reduces the TLB misses by upto 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] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs. *ACM Trans. Archit. Code Optim.*, 10(1), April 2013. ISSN 1544-3566. doi: 10.1145/2445572.2445574. URL https://doi.org/10.1145/2445572.2445574.

[2] Sparsh Mittal. A survey of techniques for architecting TLBs. 29(10):e4061. ISSN 1532-0634. doi: 10.1002/cpe.4061. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4061. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4061.

[3] 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. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304064. URL https://dl.acm.org/doi/10.1145/3297858.3304064.

[4] 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, . ISSN 0163-5964. doi: 10.1145/2678373.2665740. URL https://doi.org/10.1145/2678373.2665740.

[5] 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, . ISSN 0018-9340, 1557-9956, 2326-3814. doi: 10.1109/TC.2019.2914037. URL https://ieeexplore.ieee.org/document/8703061/.

[6] 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. doi: 10.1109/HPCA.2014.6835964.

[7] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2003. ISSN 0163-5980. doi: 10.1145/844128.844138. URL https://doi.org/10.1145/844128.844138.

[8] Marius Cornea, John Harrison, and Ping Tak Peter Tang. Intel® itanium® floating-point architecture. In *Proceedings of the 2003 Workshop on Computer Architecture Education: Held in Conjunction with the 30th International Symposium on Computer Architecture*, WCAE '03, page 3–es, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 9781450347327. doi: 10.1145/1275521.1275526. URL https://doi.org/10.1145/1275521.1275526.

[9] 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. ISSN 0141-9331. doi: https://doi.org/10.1016/S0141-9331(01)00125-9. URL https://www.sciencedirect.com/science/article/pii/S0141933101001259.

[10] 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. ISSN 0163-5964. doi: 10.1145/2508148.2485943. URL https://doi.org/10.1145/2508148.2485943.

[11] 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. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2749471. URL https://dl.acm.org/doi/10.1145/2749469.2749471.

[12] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. Flexpointer: Fast address translation based on range tlb and tagged pointers. *ACM Trans. Archit. Code Optim.*, 20(2), March 2023. ISSN 1544-3566. doi: 10.1145/3579854. URL https://doi.org/10.1145/3579854.

[13] Jason Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD.

[14] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[15] Benchmark ABI - CheriBSD 23.11 new features tutorial. URL https://www.cheribsd.org/tutorial/23.11/benchmark/index.html.

[16] CHERI-allocator/benchmarks/benchmarks/StressTestMalloc/glibc-bench.c at main · akilan1999/CHERI-allocator. URL https://github.com/Akilan1999/CHERI-Allocator/blob/main/benchmarks/benchmarks/StressTestMalloc/glibc-bench.c.

[17] Department of computer science and technology – CHERI: The arm morello board. URL https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html.

[18] 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.

[19] Arm architecture reference manual for a-profile architecture. URL https://developer.arm.com/documentation/ddi0487/latest.

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

[21] 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.