SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of the Performance of the Memory Tagging Extensions

Raphael Dichler

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of the Performance of the Memory Tagging Extensions

Evaluation der Performance der Memory Tagging Extensions

Author:	Raphael Dichler
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Martin Fink and Ilya Meignan--Masson
Submission Date:	28.03.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 28.03.2025 Raphael Dichler

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisors, Martin Fink and Ilya Meigan--Masson, for their exceptional guidance throughout my thesis. Their invaluable feedback, support, and the opportunity to develop both personally and professionally were instrumental in shaping the course of my research.

I also wish to thank Prof. Dr. Pramod Bhatotia and the TUM Systems Research Group for allowing me to work on such an intriguing and challenging topic.

Lastly, I am immensely grateful to my friends and family. Their unwavering support has been crucial to my journey, and I would not be where I am today without them.

A special thanks goes to Tony Wang, whose time and thoughtful suggestions were invaluable in helping me improve my work.

Abstract

Memory safety violations, such as buffer overflows and use-after-free errors, pose significant security risks in software systems written in low-level languages like C and C++. ARM introduced the Memory Tagging Extension (MTE) to mitigate these risks as part of the ARMv8.5-A architecture.

This thesis evaluates the performance overhead introduced by MTE in scenarios critical to database systems, including contiguous and non-contiguous access patterns, memory allocation and deallocation, and multi-threaded environments. Using controlled benchmarks executed on a Google Pixel 8 with an ARM Cortex-A715 processor, we measure the effects of MTE.

Our results show that MTE introduces negligible memory access overhead if the data is cached or prefetchable. If neither of these is confirmed, an overhead of 5-15% is introduced. Moreover, tagging memory is about 7× more costly than standard `load` and `store` operations. In multi-threaded environments, reading shared data has no measurable overhead, but writing to shared data with MTE introduces an overhead of 5-15%, regardless of the number of concurrent accesses.

Zusammenfassung

Speichersicherheitsverletzungen wie Buffer-Overflows und Use-after-free stellen erhebliche Risiken für Softwaresysteme dar, insbesondere für solche, die in Low-Level-Sprachen wie C und C++ entwickelt wurden. Um diese Gefahren zu verringern, hat ARM mit der ARMv8.5-A Architektur die Memory Tagging Extension (MTE) eingeführt.

Diese Arbeit untersucht den durch MTE verursachten Leistungs-Overhead in Szenarien, die für Datenbanksysteme relevant sind, wie etwa zusammenhängende und nicht zusammenhängende Speicherzugriffe, Speicherallokationen und -freigaben sowie Multithreading, und analysiert dessen Auswirkungen. Mithilfe kontrollierter Benchmarks auf einem Google Pixel 8 mit ARM Cortex-A715 wird der Leistungs-Overhead von MTE analysiert.

Die Ergebnisse zeigen, dass MTE beim Zugriff auf im Cache gespeicherte oder vom Prefetcher vorgeladene Daten einen vernachlässigbaren Overhead verursacht. In anderen Fällen liegt der Overhead im Bereich von 5 bis 15%. Zudem ist das Tagging von Speicher etwa siebenmal teurer als reguläre Load- und Store-Operationen. In Multithreading-Szenarien bleibt das Lesen von gemeinsam genutzten Daten ohne messbaren Overhead, während das Schreiben mit MTE unabhängig von der Anzahl paralleler Zugriffe einen Overhead von 5 bis 15% verursacht.

Contents

1. Introduction

Databases often rely on memory-unsafe languages like C and C++ to meet their performance-critical requirements. While these languages offer the highest possible performance, they also impose the burden of manual memory management, significantly increasing the risk of memory safety. This challenge is not unique to databases — other large-scale software projects written in unsafe languages face similar issues [13]. These memory safety vulnerabilities can result in exploitable security flaws. Various software-based techniques, such as address sanitizers [12], have been developed to mitigate such risks. Complementing these efforts at the hardware level, the ARM architecture introduced the Memory Tagging Extension (MTE) [1] as part of the ARMv8.5 specification [8], providing an alternative approach to detecting and preventing memory-related issues.

MTE operates on the principle of tagging pointers and their associated memory regions. When a pointer accesses memory, its tag is compared with that of the target memory region. If tags match, execution usually proceeds; otherwise, the program terminates, preventing unsafe memory access.

Several studies have highlighted the potential of MTE in enhancing memory safety. Jung et al. [3] investigate the usability and security guarantees of memory safety sanitizers using a novel LLVM-based instrumentation framework. In contrast, Fink et al. [2] introduce Cage, a hardware-accelerated WebAssembly toolchain that uses MTE to implement memory safety and sandboxing for WebAssembly. The paper also includes experiments on MTE, such as instruction latencies, throughput, and other related metrics.

Research has also explored improving MTE 's security. For instance, Partap et al. [6] propose advanced techniques for managing tag storage. Additionally, studies focused on performance by Liljestrand et al. [5] examine MTE in an emulated environment, and Zhang et al. [4] estimate a tag-checking overhead of about 7%.

Despite these insights, a detailed analysis of MTE 's impact on fundamental performance aspects in database systems remains unexplored. In particular, there is little to no research into how MTE affects execution time, cache behavior, tagging overhead, multi-threading performance, and memory access latency in database-related and other performance-critical workloads.

We propose a comprehensive evaluation of MTE 's performance characteristics

through targeted experiments that isolate its effects on low-level interactions. We provide insights into how programs can be constructed to work with MTE with minimal overhead. We investigate how MTE influences cache utilization, prefetching efficiency, and concurrency overhead by designing benchmarks that stress different aspects. Our approach aims to provide a comprehensive understanding of MTE 's behavior in performance-critical workloads, offering insights into its feasibility for high-performance systems like databases.

2. Memory Tagging Extension

MTE aims to detect invalid memory accesses by associating metadata with pointers and memory regions through tagging. This process assigns a small identifier, a tag, to the pointer and the corresponding memory region. When a memory access occurs, MTE verifies whether the pointer's tag matches the tag of the accessed memory region. If they do not match, the program crashes. Moreover, the tag is a 4-bit value, which can hold values between 0 and 15 — resulting in a limited range, which results in a 6.25% probability that a randomly assigned tag will match by coincidence. We differentiate between types of tags, the *logical tag* and the *allocation tag*. Figure 2.1 illustrates how MTE affects the system, including changes to the pointer, allocated memory, and other internal elements.

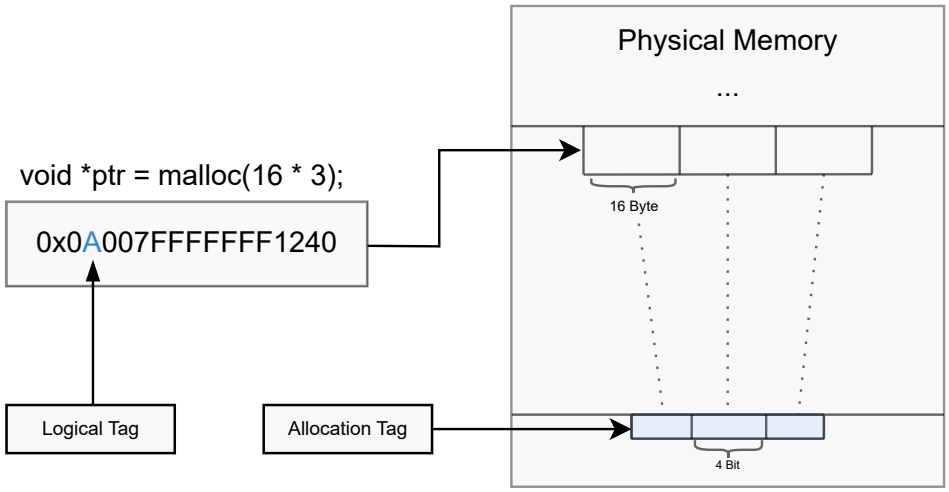

Figure 2.1.: Structure of memory allocation with MTE, showing the interaction between the memory allocator, tagged heap memory, and MTE components.

Logical Tag. The tag associated with a pointer is referred to as the logical tag. It is stored between bits 56 and 59 of a 64-bit pointer [11]. This design is possible because, under Linux with Armv8, a 64-bit pointer uses only 48 bits to address memory. The

upper bits (bit 48 to bit 63) are either all zeros or ones [7], depending on whether the pointer is in user or kernel space.

Allocation Tag. The tag associated with a memory region is called the allocation tag. Unlike the logical tag embedded in the pointer, the allocation tag is stored in a dedicated area of physical memory [11]. Physical memory is divided into 16-byte blocks, called the granularity of MTE. Each block has a corresponding allocation tag in the aforementioned dedicated region. Due to this granularity, a single allocation tag applies to an entire 16-byte chunk of memory, meaning different bytes within the same chunk cannot have distinct tags. Additionally, if a memory allocation spans multiple 16-byte chunks, the exact allocation tag must be set for each chunk within the allocated range.

Application. To utilize MTE in a program with a custom allocator, additional steps are required to manage both logical and allocation tags. The process begins by assigning a random tag to a pointer, which can be done using the `irg` instruction. Once the pointer contains a logical tag, the corresponding memory region must be tagged accordingly. This can be achieved using the `stg` instructions. The first operand determines the allocation tag to use, which is derived from the logical tag. In contrast, the second operand specifies the 16-byte memory chunk that should be tagged with the assigned value. Additionally, the instruction in the `st2g` allows tagging 32 contiguous bytes in a single operation, using the same operands as `stg` [9].

3. Motivation

While MTE offers strong security mechanisms (as described in Chapter 2), its impact on performance, particularly in heavily memory-bound and multi-threaded workloads, remains unclear.

Given this uncertainty, it is not well understood how MTE would integrate into environments where performance, concurrency, and efficiency are paramount. This raises fundamental questions about the feasibility of MTE in database workloads.

The additional allocation tag fetching required during data access and potentially complex caching behavior may introduce overhead that could render MTE unsuitable for database systems. Additionally, memory tagging adds extra steps in memory allocation and deallocation, and whether this overhead is negligible or prohibitive in a database setting is still an open question. Moreover, database systems coordinate a complex interplay of threads, requiring precise synchronization and efficient inter-thread communication. The introduction of MTE could impose additional constraints on these aspects, potentially affecting performance or necessitating adaptations in system design.

Currently, MTE is a promising solution, offering potential advantages for addressing some of the most pressing issues in memory unsafe languages. However, our understanding of the limitations and trade-offs introduced by MTE is still incomplete. Through this thesis, we aim to investigate these challenges and evaluate the viability of MTE in high-performance database environments.

4. Research Objectives

4.1. RQ1: Memory Access

Efficient memory access is crucial in database systems, which are often memory-bound due to the need to handle vast amounts of data. These systems rely on predictable data retrieval across the memory hierarchy. Understanding their interactions with MTE is essential, given the fundamental role of memory operations in performance.

In this section, we outline the scope of our investigation into the impact of MTE on memory access performance, focusing on the overhead it introduces. To guide this study, we define the following research questions:

- **RQ1.1:** How does MTE impact the performance of *non-contiguous* memory accesses across different cache levels and main memory? [Takeaway 1.1]
- **RQ1.2:** How does MTE impact the performance of *contiguous* memory accesses across different cache levels and main memory? [Takeaway 1.2]
- **RQ1.3:** Under what conditions does MTE introduce measurable performance overhead in memory-bound workloads, and how does this overhead vary based on access patterns and data locality? [Takeaway 1.3]

To address these questions, we conduct two experiments: one examining non-contiguous memory accesses and another focusing on contiguous accesses, additionally, under *rationale* in Subsection 5.2.1 (non-contiguous) and Subsection 5.2.2 (contiguous) we present the implication and insights of the experiment and why they answer the research question.

4.2. RQ2: Tagging Memory

When using MTE, memory tagging is a fundamental requirement for leveraging its features. Allocated memory regions must be assigned a tag, which introduces additional processing overhead. Therefore, understanding the impact of this tagging mechanism on performance is essential for evaluating the practicality of MTE in applications.

In this section, we outline the scope of our investigation into the impact of memory tagging and define the following research questions:

- **RQ2.1:** How does the overhead of memory tagging operations compare to the cost of `load` and `store` operations? [Takeaway 2.1]
- **RQ2.2:** What is the impact of MTE on `malloc` performance, including allocation and deallocation times? [Takeaway 2.2]

To address these questions, we conduct two experiments: one analyzing the overhead of contiguous memory tagging and tag loading, and another examining the impact of MTE on `malloc` performance, additionally, under *rationale*, in Subsection 5.3.1 (contiguous tagging) and Subsection 5.3.2 (`malloc`), we present the implication and insights of the experiment and why they answer the research question.

4.3. RQ3: Multi-Threaded-Environments

In highly multi-threaded database systems, using MTE requires synchronization mechanisms that rely on atomic operations and shared memory access. Understanding the overhead introduced by these mechanisms is crucial, especially in scenarios where multiple threads concurrently access and modify shared data.

This section outlines the scope of our investigation into the effects of MTE in multi-threaded execution and presents the following research questions:

- **RQ3.1:** Does using MTE for synchronization via Compare-And-Swap (CAS) lead to a measurable difference in performance? [Takeaway 3.1]
- **RQ3.2:** Does enabling MTE introduce significant overhead when multiple threads read or write shared data? [Takeaway 3.2]

To address these questions, we conduct two experiments: one analyzing the overhead of CAS, and another examining parallel non-contiguous memory accesses, additionally, under *rationale*, in Subsection 5.4.1 (CAS) and Subsection 5.4.2 (parallel non-contiguous), we present the implication and insights of the experiment and why they answer the research question.

5. Experiments

This chapter discusses the environment in which the benchmarks are conducted (see Section 5.1), as well as the structure and implementation of the experiments. Each experiment section presents the implementation of the benchmark and its design to address the corresponding research questions outlined in Chapter 4. Each benchmark begins with an *objective*, which outlines the primary focus of the investigation. This is followed by an *overview*, providing a high-level summary of the approach employed. The *structure* section then presents the methodology and implementation in detail, while the *rationale* clarifies how the design contributes to addressing the research question.

5.1. Benchmarking Framework and Environment

All benchmarks are implemented in C and Assembly, compiled using `clang` (version 17.0.6) with `-O3` and `-march=armv8.5-a+memtag`. We execute the benchmarks on the *Google Pixel 8*, which, along with the *Google Pixel 9*, is one of the only commercially available hardware platforms supporting MTE at the time of this study. The device features a Tensor G3 chip, which contains 4 $\times$ Cortex-A510, 4 $\times$ Cortex-A715, and 1 $\times$ Cortex-X3. All benchmarks run on the Cortex-A715, which memory system contains an `L1`, `L2`, and an optional `L3` data cache (summarized in Table 5.1). Furthermore, the Cortex-A715 processor operates at a maximum frequency of 2.367 GHz and a minimum frequency of 402 MHz.

Table 5.1.: Cortex-A715 Memory System [10]

Cache Level	**Size**
L1 I-Cache and D-Cache	32 KB or 64 KB
L2 Cache	128 KB, 256 KB, or 512 KB
L3 Cache	256 KB to 16 MB

5.1.1. Execution Setup

To ensure accurate and consistent results, the benchmarks execute under controlled conditions:

- CPU pinning is applied using `taskset` to reduce scheduling overhead and ensure that all executions run on the same core.
- Each run, defined as a combination of one or more input parameters (such as data length and stepping size), executes 10 times. Increasing the number of executions does not significantly improve result stability; more runs often lead to higher mean deviations.
- Each experiment is repeated multiple times to ensure consistency. When comparing MTE-enabled and MTE-disabled version, additional control runs compare multiple MTE-disabled versions against each other. To ensure that any observed differences result from MTE rather than normal execution variability.
- A fixed seed ensures reproducibility if a benchmark setup relies on randomization. This allows the exact benchmark to be rerun under identical conditions.
- If a benchmark setup relies on randomization, a fixed seed is set to ensure reproducibility, allowing the exact benchmark to be rerun under identical conditions.
- The memory usage of all experiments is limited to a maximum of 512 MiB. Exceeding this threshold causes unpredictable remote connection losses, most likely due to excessive workload.
- Depending on the scenario, benchmarks are written in Assembly when achieving precise low-level control is necessary, whereas C is preferred for general benchmarks.
- The benchmarks are executed within `Termux`, an Android terminal emulator and Linux environment app.

This setup provides a reliable foundation for analyzing the performance impact of MTE across different memory access patterns and workloads.

5.2. RQ1: Memory Access

This section discusses the implementation of each benchmark which correlates to "RQ1: Memory Access" (Section 4.1). In Subsection 5.2.1 and Subsection 5.2.2, we describe

the structure, runs, and reasoning of the benchmarks used to measure the impact of non-contiguous and contiguous memory access, respectively.

5.2.1. Non-Contiguous Memory Access

Objective. We design this benchmark to analyze the overhead of memory access in the presence of MTE across different caches and main memory levels. Moreover, the benchmark investigates how MTE influences memory access latency when traversing memory in a non-contiguous access pattern. The results and findings of this benchmark are presented in Subsection 6.1.1.

Overview. The benchmark iterates over a linked list that forms a loop, measuring the time required to make $100,000,000$ jumps from one node to another. Furthermore, the benchmark is executed with varying nodes to assess the effect on the execution time of an increasing number of nodes participating in a single loop.

Structure. As part of the setup, excluded from measurement, the benchmark allocates a given size of memory using `mmap` and applies memory tags if MTE is enabled. After allocation, it ensures all memory is mapped to physical memory to eliminate page faults during measurement. The allocated region is then divided into 16-byte chunks, aligning with the granularity of memory tags.

These chunks form an array-like structure, where each element represents a node in a linked list. Based on a fixed seed, the nodes are linked together in a pseudo-random order (see Figure 5.1). Although the nodes are physically adjacent in memory, their logical linkage creates a non-contiguous traversal pattern, leading to irregular memory access behavior. As a result, the process forms a circular linked list within the same memory block.

After initializing, the benchmark evaluates the total execution time for $100,000,000$ pointer traversals through the list.

We run the benchmark with varying numbers of nodes participating in a single cycle. The total memory occupied by all nodes starts at 1 KiB and doubles per run, reaching up to 512 MiB, with additional data points introduced in regions where execution time exhibits a steep increase.

Rationale. By structuring the benchmark this way, we ensure that after the first iteration, a proportion of nodes resides in the `L1` cache, while others may be in `L2`, `L3`, or main memory, depending on the workload size. As a result, subsequent accesses may lead to cache hits, improving execution time.

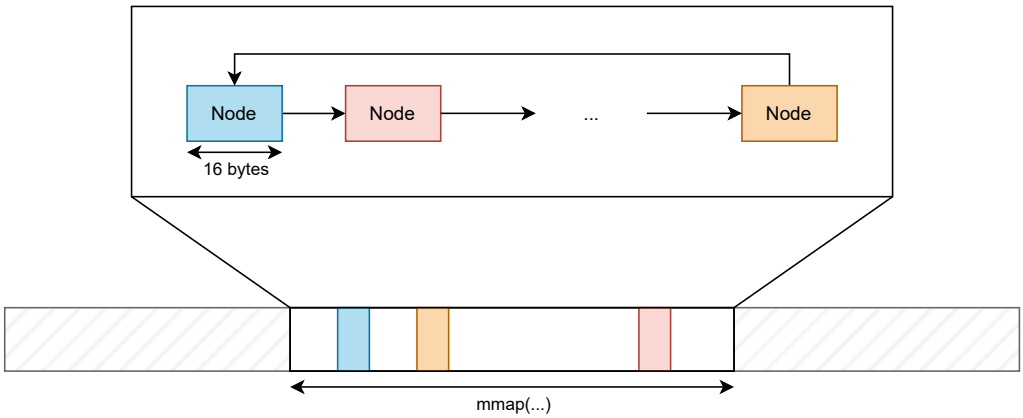

Figure 5.1.: Benchmark structure: non-contiguous memory access

When the node count is small, all nodes fit within L1, minimizing memory access latency. As the count increases, L1 capacity is exceeded, causing cache misses and requiring data retrieval from L2, L3, or main memory. As the number of nodes increases, the L2 and L3 capacity also exceeds, further deteriorating the execution time.

Furthermore, due to the benchmark's structure and the next node's unpredictability, the prefetcher has a negligible impact on performance. As a result, the execution time primarily depends on the efficiency of data retrieval from cache or main memory.

Based on the partition of the nodes across the caches and main memory, as well as the negligible impact of the prefetcher, we can analyze the effect of MTE on memory access depending on where data is located by choosing the number of nodes participating in a single cycle. This means if we select the number of nodes small enough that all nodes fit into the L1 cache and compare it with the MTE-enabled and MTE-disabled versions, we can compare the difference in execution time and measure the overhead associated with MTE when data is located in the L1 cache.

If we increase the number of nodes, where most nodes reside inside the L2 cache, most jumps result in a L2 fetch. Using the same logical reasoning as before and comparing the difference in execution time of the two versions (MTE-enabled and MTE-disabled), we can measure the overhead of MTE when data is located in the L2 cache. By further increasing the nodes, we can measure the overhead for the L3 cache and main memory, respectively.

We chose our workload sizes based on a doubling strategy to measure the latencies across different cache levels and the main memory. By doubling the workload size in each run, we ensure that most data fits within a single cache level in one run. In the next run, with double the nodes, the data is distributed across both cache levels.

After another doubling, most of the data resides in the higher-level cache. With his approach, we ensure that most of the data stays within a single cache level in most runs. Additionally, we include extra runs — those that don't follow the exponential growth pattern — at points where the workload size lies between two cache levels, helping to smooth out the latency curve.

Given the constant number of iterations performed by the benchmark at each run, the different workloads can be compared since the execution time is not dependent on the number of nodes.

5.2.2. Contiguous Memory Access

Objective. We design this benchmark to analyze the overhead of memory access when accessing memory contiguously in the presence of MTE. The results and findings of this benchmark are presented in Subsection 6.1.2.

Overview. The benchmark sequentially traverses an allocated memory region, measuring the time required to iterate over the entire region. The benchmark is executed with varying memory region sizes to assess performance changes as the accessed memory footprint increases.

Structure. As part of the setup, excluded from measurement, the benchmark allocates a given size of memory using `mmap` and applies memory tags if MTE is enabled. After allocation, it ensures all memory is mapped to physical memory to eliminate page faults during measurement. The allocated region is then divided into 64-byte chunks.

After initializing, the benchmark evaluates the total execution time to iterate over the allocated memory region in 64-byte segments.

We run the benchmark with varying array sizes. The total memory occupied — by the array — starts at 1KiB and doubles per run, reaching up to 512MiB.

Rationale By structuring the benchmark this way, we create conditions where the prefetcher may recognize the access pattern and predict subsequent memory accesses (strided prefetching). If the pattern — fetching every 64th byte — is identified, mispredictions should be minimal.

Given the nature of contiguous memory access, execution time will benefit from efficient prefetching, as data can be loaded into the cache before it is required. Moreover, we can analyze the impact of MTE on contiguous memory access, particularly when the processor attempts to prefetch both data and its associated allocation tag.

When working with a large dataset, the prefetcher should be able to load data in advance. In an MTE-disabled environment, this process should operate without

interference, serving as a baseline for evaluating its performance under MTE. However, in an MTE-enabled environment, the prefetcher must not only fetch the data but also retrieve the corresponding allocation tag, which could introduce additional overhead.

By gradually reducing the dataset size, we modify the workload characteristics and increase the reliance on prefetching. Comparing the execution times between MTE-enabled and MTE-disabled environments allows us to assess how contiguous memory access is affected by MTE.

With access occurring every 64 bytes, this benchmark setup places increasing demands on the prefetcher, providing insight into its efficiency when handling data and allocation tags.

5.3. RQ2: Tagging Memory

This section discusses the implementation of each benchmark which correlates to "RQ2: Tagging Memory" (Section 4.2). In Subsection 5.3.2, and Subsection 5.2.2, we describe the structure, runs, and reasoning of the benchmarks with which we measure the impact tagging.

5.3.1. Contiguous Tagging

Objective. This experiment analyzes the behavior of memory tagging when applied to contiguous memory regions. The results and findings of this benchmark are presented in Subsection 6.2.1.

Overview. This benchmark measures the performance of memory tagging by applying different tagging strategies (`stg`, `st2g`, `malloc`, and `ldg`) to a contiguous 512MiB memory region. The execution time for tagging is recorded to assess efficiency. Additionally, we compare the tagging performance against a baseline operation that performs `load` and `store` on every 4-byte segment within the region.

Structure. As part of the setup, excluded from measurement, the benchmark allocates a memory region using `mmap`. To eliminate potential page faults during the measurement phase, the benchmark ensures that the allocated memory is mapped to physical memory.

Once the memory is prepared, the benchmark sequentially sets allocation tags to the entire region, processing either 16-byte or 32-byte segments, depending on the instruction. The execution time is recorded for the following tagging strategies:

1. `stg` version: Tags the entire memory region using the `stg` assembly instruction.

2. `st2g` version: Tags the entire memory region using the `st2g` assembly instruction.
3. `malloc` version: Uses the tagging implementation within glibc's `malloc`.

In addition to tagging, we incorporate loading tags using `ldg`. Additionally, we include a variant that performs a `load` and `store` operation every 64 bytes within the contiguous memory region, serving as a baseline for comparison.

Rationale. By structuring the benchmark in this way, we aim to measure the time required for each tagging operation (`stg`, `st2g`, `malloc`, and `ldg`) and compare it to basic load and store operations. This allows us to quantify the overhead introduced by tagging and establish expectations for memory tagging performance across different approaches.

5.3.2. Malloc

Objective. This benchmark is designed to analyze the behavior of `malloc` in the presence of MTE. The results and findings of this benchmark are presented in Subsection 6.2.2.

Overview. The benchmark evaluates the impact of MTE on memory allocation and deallocation by measuring the execution time required to allocate and optionally free memory under different configurations.

Structure. As part of the setup, excluded from measurement, the benchmark creates a structure that manages all allocations using `malloc` and ensures that memory is tagged when MTE is enabled.

Once the setup is complete, the benchmark measures the time required to perform memory allocations in batches, where the batch size is a user-defined parameter. The measurement optionally includes deallocation time.

The benchmark runs with different batch sizes while maintaining a total allocation size of 512 MiB. The batch sizes are 16, 128, 256, 1024, 2048, and 8192 bytes. For each batch size, we allocate a total of 512 MiB of memory.

Rationale. This benchmark is structured to determine the typical overhead of `malloc` when MTE is enabled. The distinction between allocation-only and allocation-with-deallocation allows us to assess whether the overhead primarily stems from the allocation process or if allocation and deallocation exhibit approximately the same overhead.

The selected batch sizes of 16, 128, and 256 bytes represent typical small object allocations. Including these larger batches mitigates distortions in the measurements caused by `malloc` triggering an internal `mmap` allocation, which can introduce page faults and affect execution time.

5.4. Impact in Multi-Threaded-Environments

5.4.1. Synchronization Operations

Objective. We design this experiment to analyze the overhead that is introduced on CAS operations in multi-threaded environments in the presence of MTE. The results and findings of this benchmark are presented in Subsection 6.3.1.

Overview. This benchmark measures the execution time of CAS operations across multiple threads, evaluating how performance scales with different thread counts. Each thread attempts to increment a shared value numerous times, allowing us to assess synchronization overhead and contention effects.

Structure. As part of the setup, excluded from measurement, the benchmark first creates a specified number of threads, pins each to a dedicated core, and ensures all threads start simultaneously by a barrier. Moreover, a shared 16-byte value is allocated and tagged if MTE is enabled.

After initializing, the benchmark evaluates the total execution time for each thread to perform $100,000,000$ CAS operations, incrementing the shared value by one.

We run the benchmark with thread counts ranging from 1 to 4 to evaluate how CAS performance is affected at different levels of concurrency.

Rationale. By structuring the benchmark this way, we ensure that synchronization overhead and contention effects are systematically evaluated of levels concurrency. Since each thread performs a fixed number of CAS operations, the total execution time reflects the impact of contention and memory access latency.

In a single-threaded scenario, contention is absent, and execution time is primarily determined by the cost of performing CAS on a shared variable. As threads increase, multiple cores compete to update the shared value, increasing contention and potential cache coherence traffic. Measuring execution time across different thread counts allows us to analyze how contention scales and whether additional delays emerge due to failed CAS attempts.

To minimize variability, the benchmark ensures all threads start execution simultaneously and are pinned to dedicated cores, preventing unintended interference from task scheduling. Additionally, we allocate a 16-byte shared variable, aligning with the typical cache-line size, to examine whether cache-line contention contributes to performance differences.

By evaluating execution time across different levels of concurrency, we gain insights into how MTE interacts with synchronization operations, particularly in scenarios where contention and cache coherence effects dominate execution costs.

5.4.2. Concurrent Read and Write Benchmark

Objective. We designed this benchmark to analyze the overhead introduced when multiple threads interact through concurrent reads and writes in the presence of MTE. The results and findings of this benchmark are presented in Subsection 6.3.2.

Overview. The benchmark measures execution time as multiple threads traverse a circular linked list by performing only pointer jumps or combining pointer jumps with additional write operations. Each thread completes 100,000,000 jumps, and the total execution time is recorded.

Structure. As part of the setup, excluded from measurement, the benchmark follows the linked list construction described in Subsection 5.2.1, forming a single circular linked list. This circular linked list is assigned to all threads; each thread operates on the same list during the benchmark. The setup also includes creating the specified number of threads, pinning them to dedicated cores, and synchronizing them using barriers.

After initializing, the benchmark evaluates the total execution time for either 100,000,000 pointer traversals through the list or pointer traversals with additional write operations on the nodes through the list.

We run the benchmark with varying numbers of threads and nodes participating in a single cycle. The total memory occupied by all nodes ranges from 1KiB up to 512MiB. Moreover, the benchmark is executed with 1, 2, 3, and 4 threads running in parallel.

Rationale. This benchmark is structured to evaluate the potential overhead introduced by MTE in a multi-threaded environment where multiple threads perform concurrent `load` and `store` operations on shared data. By distinguishing between purely read-based traversal and traversal that includes write operations, we can analyze how MTE interacts with contention.

For concurrent read-only access, we do not expect significant contention based on the fundamentals of cache coherence. When multiple threads load data that resides in separate `L1` caches, and no modifications occur, the cache coherence protocol should allow these reads to proceed without triggering cache invalidations or coherence traffic. If the execution time remains nearly identical between the MTE-enabled and MTE-disabled environments, it would suggest that memory tagging imposes little to no overhead on pure read operations. Conversely, if a measurable difference emerges, it could indicate that additional checks or interactions with the tagging mechanism introduce subtle effects even in read-only scenarios.

When writes are introduced, the behavior changes due to cache coherence mechanisms enforcing consistency across cores. In this case, modified data must be propagated across caches, potentially causing additional coherence traffic and invalidation. The presence of MTE might further affect this process if tagged memory updates require additional steps beyond standard cache coherence operations. By comparing execution times with and without memory tagging, we aim to determine whether MTE amplifies these effects.

By varying the total memory footprint from 1 KiB to 512 MiB, we also influence data locality, shifting memory accesses across different cache levels and main memory. This follows the same reasoning outlined in Subsection 5.2.1. If MTE introduces overhead that depends on where data resides in the memory hierarchy, these variations should reveal such effects.

6. Results, Interpretation, and Conclusions

This chapter presents the results of the benchmarks outlined in Chapter 5. It also discusses the key takeaways from the findings and answers the research questions posed in Chapter 4.

6.1. RQ1: Memory Access

This section discusses the results and takeaways based on the benchmarks presented in Section 5.2. Moreover, we answer the research questions presented in Section 4.1.

6.1.1. Non-Contiguous Memory Access

Structure. Figure 6.1.1 shows the execution time for 100,000,000 pointer-to-pointer jumps, illustrating how execution time changes with increasing allocated size. The plot compares the MTE enabled and disabled versions with the percentage difference shown by a red dashed line. If the error bars overlap, the difference is set to 0%.

Observation. The execution time between 1 KiB and 64 KiB remains constant, corresponding to the `L1` cache size. The curve flattens at 128-256 KiB, matching the `L2` cache size, and again at 512 KiB to 4 MiB, corresponding to the `L3` cache. Up to around 16 MiB, there is no significant difference between the MTE versions. Above 16 MiB, the MTE enabled version shows a 5-15% overhead in execution time.

Implication. The key observation is that the difference between MTE enabled and disabled versions becomes noticeable only when data exceeds the cache capacity. With this, we can address RQ1.1, which investigates how MTE impacts the performance of non-contiguous memory accesses across different cache levels and main memory.

> **Takeaway 1.1:** When performing non-contiguous memory accesses, MTE does not introduce significant overhead when the data resides within the cache. However, when non-contiguous memory accesses predominantly involve main memory, MTE incurs an overhead ranging from 5% to 15%.

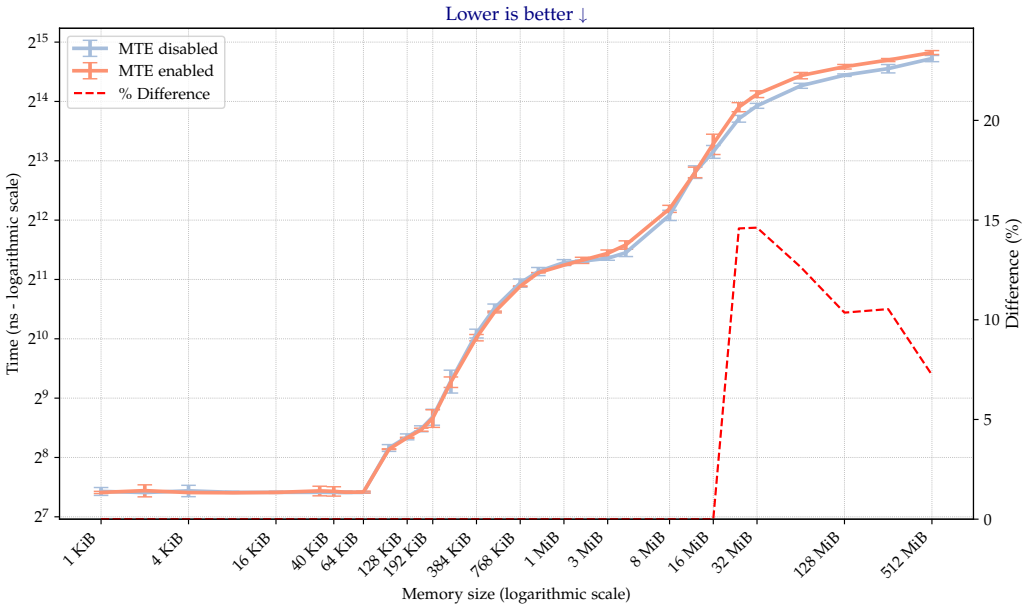

Figure 6.1.: Performance evaluation comparison between MTE enabled and MTE disabled version for non-contiguous memory access.

6.1.2. Contiguous Memory Access

Structure. Figure 6.1.2 shows the execution time for iterating over a contiguous memory region, illustrating how execution time changes with increasing memory region size. The plot compares the MTE enabled and disabled versions.

Observation. The execution time remains consistent across different memory sizes, with no noticeable difference between the MTE enabled and disabled versions. This consistency persists as the memory region increases.

Implication. The key observation is that MTE does not introduce significant overhead when accessing contiguous memory. This allows us to address RQ1.2, which investigates the impact of MTE on the performance of contiguous memory accesses across varying cache levels and main memory.

Takeaway 1.2: When performing contiguous memory accesses, MTE does not introduce significant overhead across all cache layers and main memory.

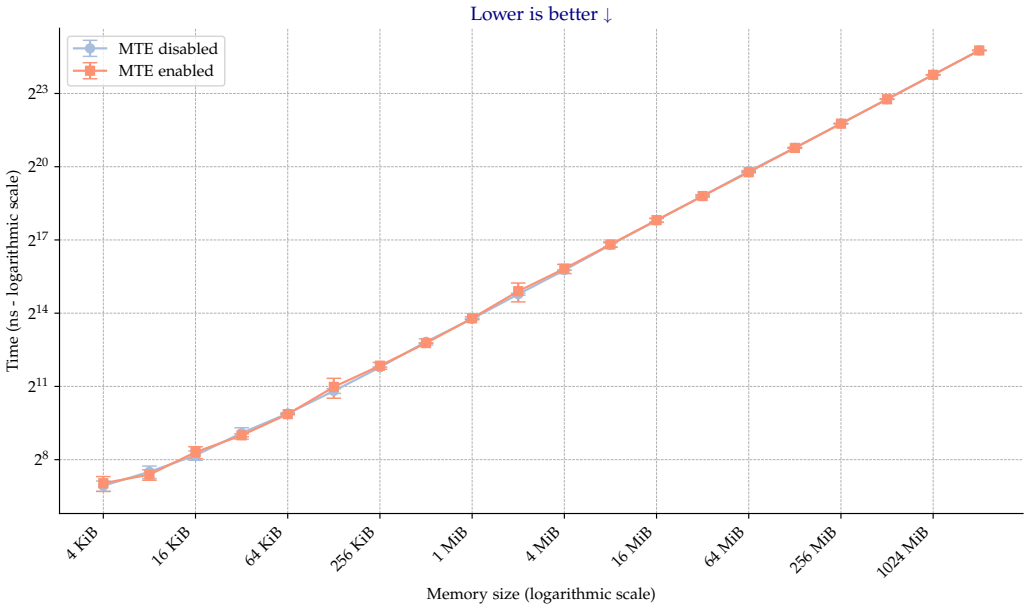

Figure 6.2.: Performance of contiguous memory access with a fixed number of accesses.

6.1.3. Synthesis

Structure. By synthesizing the results from Subsection 6.1.1 on non-contiguous memory access patterns and Subsection 6.1.2 on contiguous memory access, we can derive a comprehensive understanding of the impact of MTE across different memory access patterns.

Observation. For contiguous memory access, MTE does not introduce significant overhead, regardless of the memory region size. In contrast, for non-contiguous memory access, the difference between MTE enabled and disabled versions becomes noticeable only when the data exceeds the cache capacity.

Implication. Combining these results, we conclude that MTE does not introduce measurable overhead in memory-bound workloads when the data is cache-resident or prefetchable. This addresses RQ1.3, which investigates when MTE introduces performance overhead in memory-bound workloads and how this varies based on access patterns and data locality.

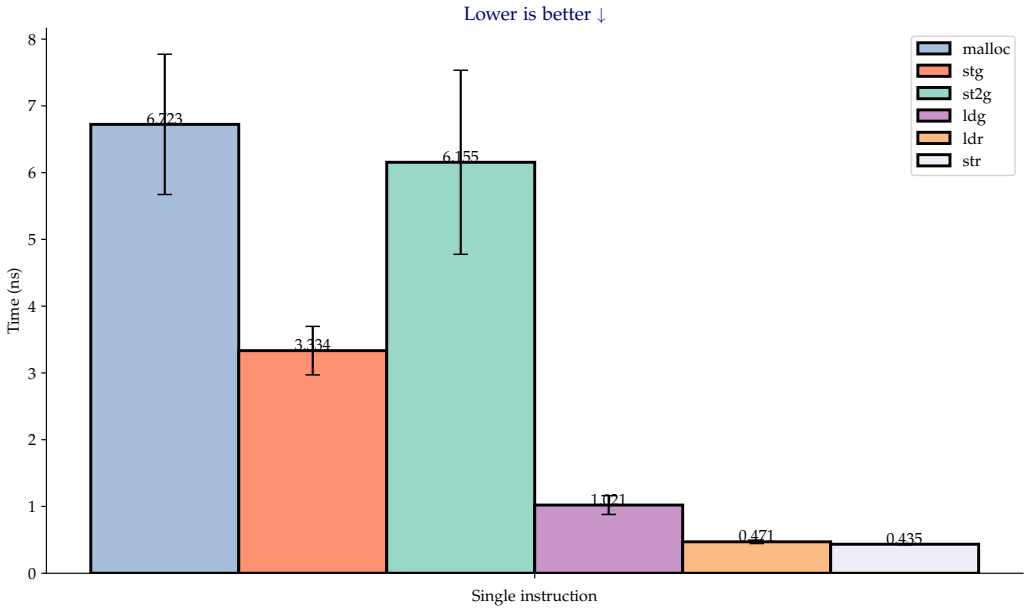

Figure 6.3.: Latency of different MTE allocation tag instructions, including allocation tagging, tagged loads, and standard load/store operations.

Takeaway 1.3: MTE introduces measurable performance overhead in memory-bound workloads primarily when data is not cache-resident and prefetchable.

However, if data resides within the cache or is prefetchable, memory access with MTE does not introduce overhead. For highly memory-bound workloads, the prefetcher can still fetch both the data and the allocation tag in time; therefore, workloads do not experience significant overhead when MTE is enabled.

6.2. RQ2: Tagging Memory

This section discusses the results and takeaways based on the benchmarks presented in Section 5.3. Moreover, we answer the research questions presented in Section 4.2.

6.2.1. Contiguous Tagging

Structure. Figure 6.2.1 illustrates the latency of individual instructions related to memory tagging, comparing the execution time of various MTE operations with standard `ldr` and `str` instructions.

Observation. MTE-related instructions exhibit a notable performance impact compared to standard load and store operations. While the latency of `ldg` is approximately twice that of a standard `ldr` or `str`, the overhead associated with tagging operations ranges from 7× to 14×. Furthermore, although `stg` has only half the latency of `st2g` and `malloc`-based tagging, tagging an equivalent memory region requires two `stg` instructions.

Implication. The results indicate that setting allocation tags imposes a substantial overhead compared to standard memory operations. This addresses RQ2.1, which investigates how the overhead of memory tagging operations compares to the cost of `load` and `store` instructions.

Takeaway 2.1: The tagging process introduces significant overhead, particularly when setting the allocation tag. In an environment where a large contiguous memory region is tagged at once, this overhead becomes pronounced, with an approximate 7× increase in latency compared to standard `load` and `store` instructions. Loading a tag takes roughly twice as long as a standard memory operation.

6.2.2. Malloc

Structure. Figure 6.2.2 and Figure 6.2.2 illustrate the impact of MTE on memory allocation and deallocation performance. Figure 6.2.2 shows the time required for a single `malloc` allocation across different batch sizes, while Figure 6.2.2 presents the corresponding deallocation times. Both figures compare the execution time of MTE-enabled and MTE-disabled versions.

Observation. For small allocations (16 to 1024 bytes), MTE incurs substantial overhead, increasing execution time by 3× to 6×. However, as the batch size increases, the relative overhead decreases. For 2048, 4096, and 8192 bytes allocations, the overhead is reduced from 1.7× to 2.5×. A similar trend is observed for deallocation, where MTE-enabled and MTE-disabled versions exhibit comparable relative differences.

Implication. MTE introduces significant overhead to allocation and deallocation, where tagging operations dominate execution time. This addresses RQ2.2, which investigates the impact of MTE on `malloc` performance, including allocation and deallocation times.

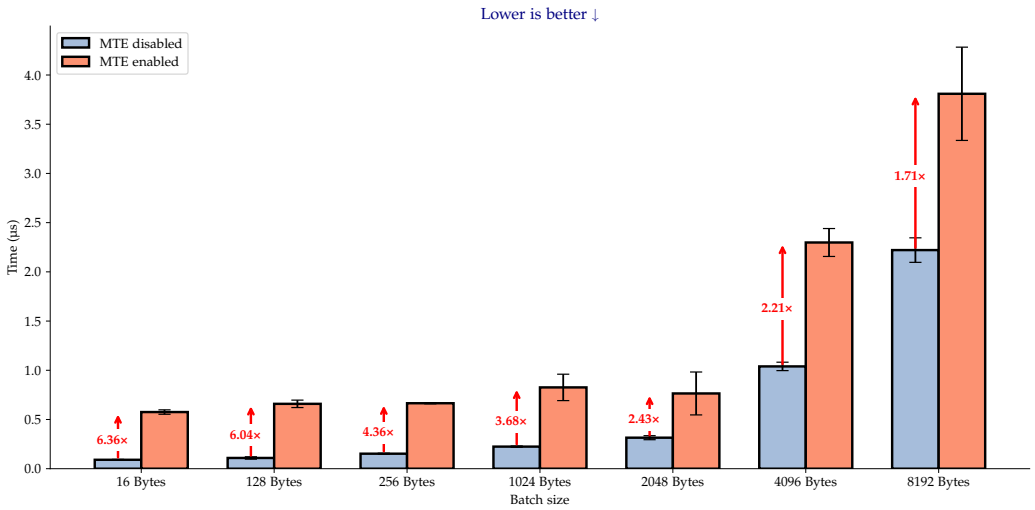

Figure 6.4.: Impact of MTE on the latency of `malloc` calls, highlighting differences in performance with and without MTE enabled.

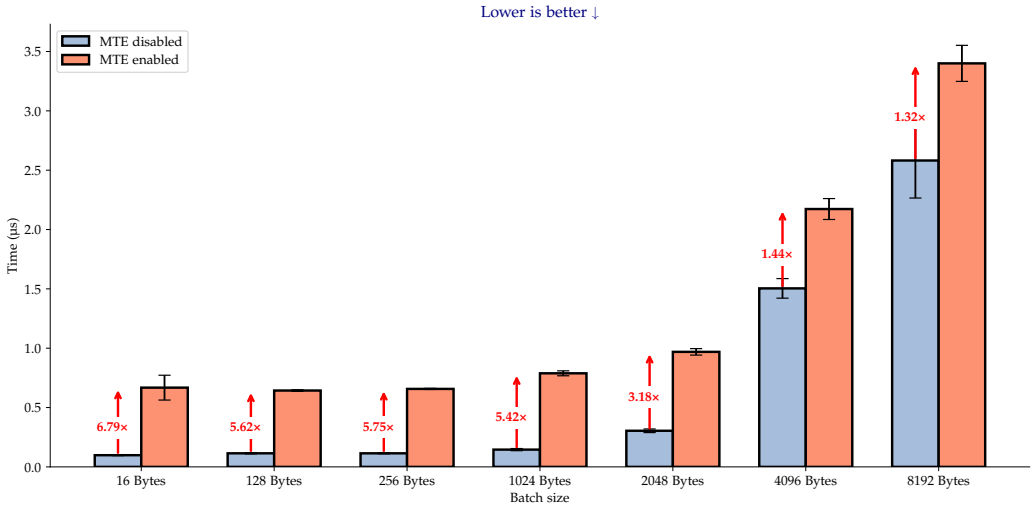

Figure 6.5.: Impact of MTE on the latency of `free` calls, highlighting differences in performance with and without MTE enabled.

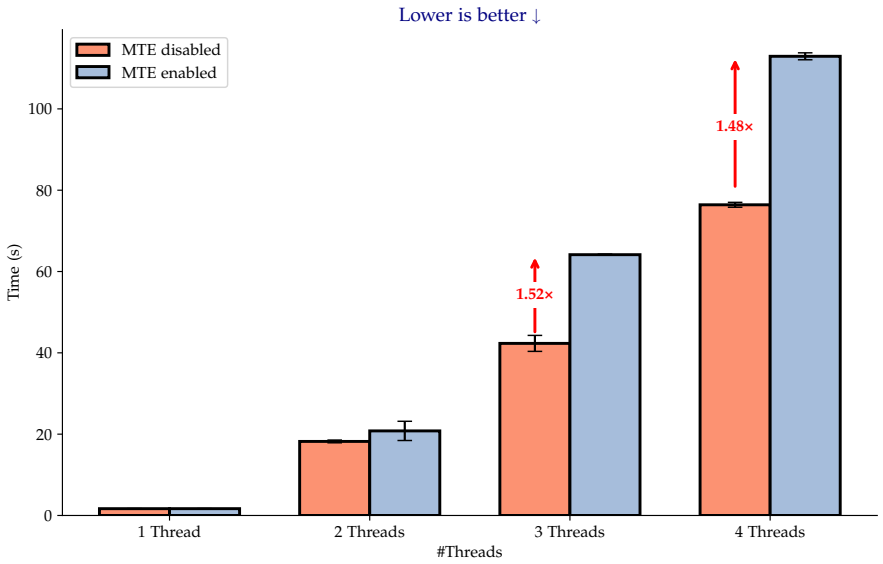

Figure 6.6.: Performance comparison of the CAS operation with MTE enabled versus disabled, illustrating the effect of MTE on operation latency and throughput.

Takeaway 2.2: MTE introduces substantial overhead to `malloc` performance, particularly for small allocations, where execution time increases by 3× to 6×. As allocation grows, this overhead diminishes, ranging from 1.7× to 2.5× for larger batch sizes.

6.3. RQ3: Multi-Threaded-Environments

This section discusses the results and takeaways based on the benchmarks presented in Section 5.4. Moreover, we answer the research question presented in Section 4.3.

6.3.1. Synchronization Operations

Structure. Figure 6.6 illustrates the performance impact of MTE on synchronization using CAS. The figure shows the total execution time required to perform 100,000,000 CAS operations on a shared memory location with multiple threads, comparing MTE-enabled and MTE-disabled versions.

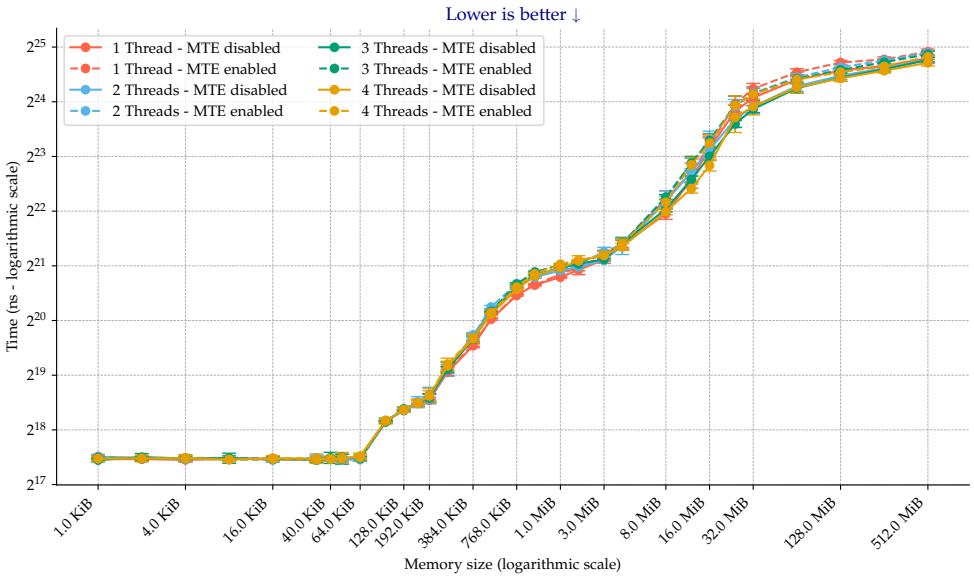

Figure 6.7.: Performance evaluation of parallel read-only access to non-contiguous memory as memory size increases.

Observation. When three or four threads execute concurrently, MTE introduces an overhead of approximately 1.5× compared to execution without MTE.

Implication. MTE affects synchronization performance by increasing execution time, particularly when multiple threads compete for access. This addresses RQ3.1, which investigates whether using MTE for synchronization via CAS leads to a measurable performance difference.

> **Takeaway 3.1:** MTE introduces an overhead of approximately 1.5× when three or four threads execute concurrently compared to non-MTE execution.

6.3.2. Concurrent Reads and Writes

Structure. This subsection presents the benchmark results outlined in Subsection 5.4.2. The benchmark measures the execution time for 100,000,000 pointer-to-pointer operations, with a distinction between read-only operations (Figure 6.3.2) and read-write operations (Figure 6.3.2). These figures show how execution time varies with increasing memory size and the number of threads operating on the same data structure.

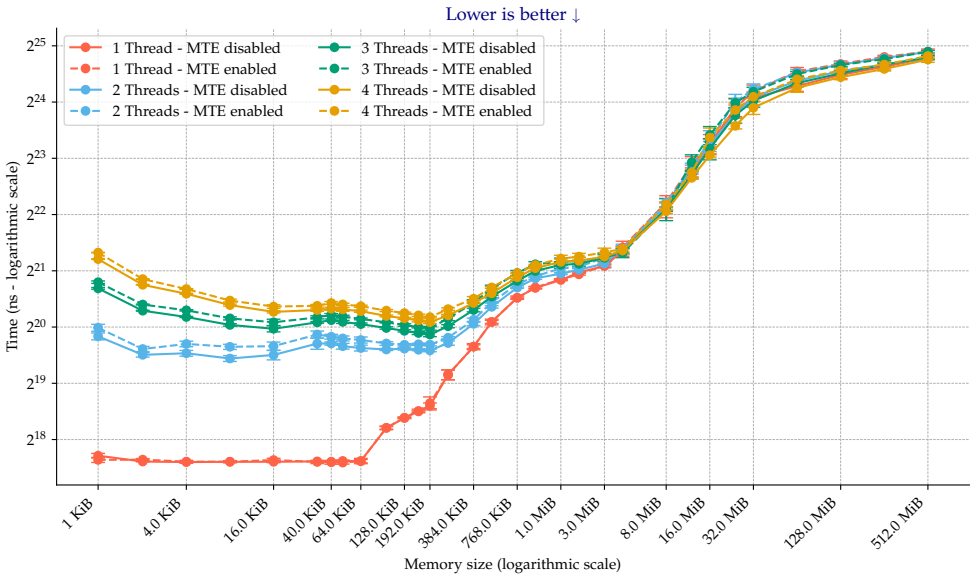

Figure 6.8.: Performance evaluation of parallel read-write access to non-contiguous memory as memory size increases.

Observation. In Figure 6.3.2, we observe that when performing pointer-to-pointer jumps with multiple threads, there is no significant difference in execution time between the MTE-enabled and MTE-disabled versions until the memory size reaches 16 MiB. After this point, a noticeable overhead of 5% to 15% is observed in the MTE-enabled version. On the other hand, in Figure 6.3.2, when multiple threads perform write operations on shared data, the overhead of enabling MTE becomes more apparent. Specifically, when 2, 3, or 4 threads are used, the MTE-enabled version shows an overhead of 5% to 15% compared to the MTE-disabled version.

Implication. The results indicate that MTE primarily introduces performance overhead when multiple threads write to shared data, while read-only operations remain largely unaffected. This addresses RQ3.2, which asks whether enabling MTE causes significant overhead for multi-threaded reads and writes.

Takeaway 3.2: MTE introduces an overhead of 5% to 15% in multi-threaded environments when performing write operations on shared data. However, read operations behave similarly to single-threaded execution and show no significant performance difference when MTE is enabled.

7. Conclusion

In this thesis, we comprehensively evaluated the performance implications of MTE by isolating its impact on low-level interactions, such as cache utilization, prefetching efficiency, and concurrency overhead. We aimed to assess how MTE affects performance-critical workloads and identify scenarios where its overhead is most pronounced.

Our experiments show that when memory is accessed predictably or resides within the cache, MTE does not introduce significant overhead. However, when neither condition holds — meaning the data is not in the cache and cannot be prefetched — MTE incurs an overhead of 5–15%. Additionally, tagging memory, an essential step when using MTE, comes at a notable cost. Our benchmarks indicate that tagging operations result in an overhead of approximately 7× compared to standard `load/store` instructions, making it an expensive but unavoidable step. In multi-threaded environments, MTE further increases performance penalties, particularly in scenarios where multiple threads perform concurrent writes to shared memory, leading to an additional overhead of around 5–15%.

While MTE provides strong memory safety guarantees, its performance impact must be carefully considered in high-performance applications. Although its overhead does not significantly affect ordinary memory operations, workloads with frequent memory allocations and deallocations may experience a noticeable performance degradation.

A. Artifacts

The artifact containing the experimental setup is available on GitHub:

- `https://github.com/raphaeldichler/mte-experiments`.

Each benchmark includes a `Makefile` that uses Clang for compilation. Additionally, every benchmark provides a dedicated script for evaluating the overhead of MTE, which utilizes the compiled source code for execution.

Abbreviations

MTE Memory Tagging Extension

CAS Compare-And-Swap

List of Figures

List of Tables

Bibliography

[1] Arm. *Armv8.5-A Memory Tagging Extension White Paper*. Tech. rep. Accessed: 2025-03-27. Arm, June 2021. URL: https://developer.arm.com/documentation/102925/latest/.

[2] Martin Fink et al. "Cage: Hardware-Accelerated Safe WebAssembly." In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 538–552. ISBN: 9798400712753. DOI: 10.1145/3696443.3708920. URL: https://doi.org/10.1145/3696443.3708920.

[3] Tina Jung, Fabian Ritter, and Sebastian Hack. "Memory Safety Instrumentations in Practice: Usability, Performance, and Security Guarantees." In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 390–404. ISBN: 9798400712753. DOI: 10.1145/3696443.3708926. URL: https://doi.org/10.1145/3696443.3708926.

[4] Michael LeMay et al. "Cryptographic Capability Computing." In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 253–267. ISBN: 9781450385572. DOI: 10.1145/3466752.3480076. URL: https://doi.org/10.1145/3466752.3480076.

[5] Hans Liljestrand et al. *Color My World: Deterministic Tagging for Memory Safety*. 2022. arXiv: 2204.03781 [cs.CR]. URL: https://arxiv.org/abs/2204.03781.

[6] Aditi Partap and Dan Boneh. *Memory Tagging: A Memory Efficient Design*. 2022. arXiv: 2209.00307 [cs.CR]. URL: https://arxiv.org/abs/2209.00307.

[7] Arm team. *Armv8-A Address Translation*. [Online; accessed 2 8-March-2025]. URL: https://documentation-service.arm.com/static/5efa1d23dbdee951c1ccdec5 (visited on 03/28/2025).

[8] Arm team. *ARMv8.5-a memory tagging extensio*. [Online; accessed 24-March-2025]. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf (visited on 03/23/2025).

[9] Arm team. *Base Instructions*. [Online; accessed 2 8-March-2025]. URL: https://developer.arm.com/documentation/ddi0602/2025-03/Base-Instructions (visited on 03/28/2025).

[10] Arm team. *Cortex-A715*. [Online; accessed 2 5-March-2025]. URL: https://developer.arm.com/Processors/Cortex-A715 (visited on 03/25/2025).

[11] Arm team. *Where is the MTE Tag stored and checked?* [Online; accessed 2 8-March-2025]. URL: https://developer.arm.com/documentation/ka005620/latest/ (visited on 03/28/2025).

[12] Clang team. *AddressSanitizer*. [Online; accessed 24-March-2025]. URL: https://clang.llvm.org/docs/AddressSanitizer.html (visited on 03/23/2025).

[13] Google team. *The Chromium Projects*. [Online; accessed 24-March-2025]. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 03/23/2025).