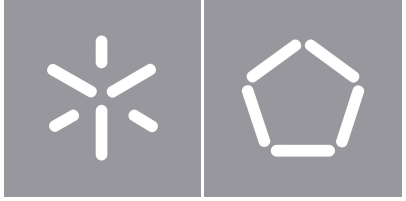
University of Minho
School of Engineering

Rui André Santos Ribeiro

Concurrency hot spot optimisation in transactional memory

september 2023



University of Minho
School of Engineering

Rui André Santos Ribeiro

Concurrency hot spot optimisation in transactional memory

Master's Dissertation
Master's in Informatics Engineering

Dissertation supervised by
José Orlando Roque Nascimento Pereira

september 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

First, I would like to express my heartfelt gratitude to my supervisor, José Orlando Pereira, for his insights, guidance, and invaluable feedback. His support was instrumental in shaping the course of this work. Moreover, his dedication to the research pushed me to do better and take on new opportunities. In addition, I would like to acknowledge the contribution of Nuno Faria, for following the development of this project and helping revise several portions of this dissertation.

The success of this endeavour owes much to the generous support and resources provided by INESC TEC. Their provision of computing resources and fostering of a collaborative work environment has been pivotal in bringing this project to fruition. Additionally, financial support was graciously granted by FCT - Fundação para a Ciência e a Tecnologia, through National Funds, as part of project LA/P/0063/2020.

Reflecting on my five-year journey at the University of Minho, I am indebted to the friends I have made along the way. I will miss all those long late-night study sessions and tech troubleshooting calls.

Lastly, I would be remiss in not mentioning my family, whose unwavering support, understanding, and encouragement have been the driving force behind my achievements.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, september 2023

Rui André Santos Ribeiro

Abstract

Database management systems have a long history of development and research, with systems like PostgreSQL and languages like SQL already being well-established in the industry. Transactional memory emerges as a new concurrency control mechanism for concurrent programming, inspired by ideas and concepts from the database world. As is the case with database transactions, transactions in transactional memory can (and will) conflict when multiple transactions try to modify the same data. This can lead to the appearance of hot spots in contended memory regions, quickly degrading the performance of an application.

In this dissertation, we propose new optimisation techniques for transactional memory hot spots, based on previous research on splitting techniques for numeric database records. We implement the optimisations on an existing transactional memory system and measure their impact on performance, using custom-made and reference benchmarks.

Keywords transactional memory, concurrent programming, hot spot optimisation

Resumo

Sistemas de gestão de bases de dados possuem uma longa história de desenvolvimento e investigação, estando sistemas como PostgreSQL e linguagens como SQL já bem estabelecidos na indústria. Memória transacional surge como um novo mecanismo de controlo de concorrência para a programação concorrente, inspirada por ideias e conceitos do mundo das bases de dados. Tal como se sucede nas transações em bases de dados, as transações em memória transacional podem (e irão) entrar em conflito quando múltiplas transações tentarem modificar os mesmos dados. Isto leva ao surgimento de *hot spots* em regiões contendidas de memória, rapidamente degradando a performance de uma aplicação.

Nesta dissertação, propõem-se novas otimizações para *hot spots* em memória transacional, baseadas em investigação prévia de técnicas de divisão de registos numéricos em bases de dados. As otimizações foram implementadas sobre um sistema de memória transicional já existente, sendo o seu impacto na performance medido recorrendo tanto a um *benchmark* feito à medida como a um de referência.

Palavras-chave memória transacional, programação concorrente, otimização de *hot spots*

Contents

List of Figures

List of Tables

Listings

Acronyms

ACID Atomicity, Consistency, Isolation, and Durability.

API Application Programming Interface.

ASF Advanced Synchronization Facility.

CPU Central Processing Unit.

DDL Data Definition Language.

DSTM Dynamic Software Transactional Memory.

DTM Durable Transactional Memory.

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

HLE Hardware Lock Elision.

HTM Hardware Transactional Memory.

HyTM Hybrid Transactional Memory.

MRV Multi-Record Value.

NUMA Non-Uniform Memory Access.

NVML NVIDIA Management Library.

OCC Optimistic Concurrency Control.

OS Operating System.

PhTM Phased Transactional Memory.

PM Persistent Memory.

PR Phase Reconciliation.

PTM Persistent Transactional Memory.

RAPL Running Average Power Limit.

RNG Random Number Generator.

RRB-Tree Relaxed Radix Balanced Tree.

RTM Restricted Transactional Memory.

SMT Simultaneous Multithreading.

SQL Structured Query Language.

STAMP Stanford Transactional Applications for Multi-Processing.

STL Standard Template Library.

STM Software Transactional Memory.

STO Software Transactional Objects.

TM Transactional Memory.

TME Transactional Memory Extension.

TSX Transactional Synchronization Extensions.

UTM Unbounded Transactional Memory.

Chapter 1

Introduction

1.1 Context and motivation

Computing hardware has evolved exponentially since the days of the first microprocessors. In 1965, Gordon Moore—who would later found Intel—predicted that the number of transistors in a chip would double every year over the following ten years. This statement, despite being merely a forecast based on his observations, has since then been known as the "Moore's law" [Britannica] and ended up being exceeded long after 1975.

At the turn of the 21st century, microprocessors were hitting physical limits, restricting further advances in computing power. Manufacturers sidestepped this problem with the introduction of multiprocessor architectures.

However, this created a new problem, as programs written in a typical sequential fashion are not able to exploit the power of multiple processors. Instead, parallel programming techniques must be employed, such as the usage of threads. These, in turn, require additional coordination, so multiple threads do not concurrently modify the same data and cause inconsistent results. Mutual exclusion (or locking) is one such synchronisation mechanism, in which a thread must acquire a lock before accessing its respective data. This approach has some well-known limitations, such as being susceptible to deadlocking, not being composable, and requiring additional effort to implement correctly. Thus, **Transactional Memory (TM)** emerges as a programming paradigm well-suited to multicore systems, avoiding the complexity and error-proneness of locking mechanisms [Harris et al., 2010].

Despite the ease-of-use benefits, **TM** has not seen wide adoption with developers [Zardoshti et al., 2019]. There are two main factors usually attributed to this: the non-existence of generally agreed-upon transaction syntax/semantics and performance concerns. The latter point, particularly, is one of great importance; after all, concurrent programming is usually employed to improve the performance of a system.

In database systems, *hot spots* arise in workloads where just a few popular data items are updated

by a significant number of transactions. Value-splitting is a strategy that aims to mitigate this problem, resorting to the division of a numeric value into multiple variables to reduce contention in transactional accesses and updates.

1.2 Main aims and results

Improving the performance of **TM** systems is the main goal of this work, specifically in workloads that are prone to high levels of contention on numeric values. To achieve this, we have adapted to **TM** two value-splitting techniques that have been previously proposed for databases, namely **Multi-Record Values (MRVs)** [Faria and Pereira, 2023] and **Phase Reconciliation (PR)** [Narula et al., 2014]. These techniques allow the splitting of an integer value into multiple partial chunks, enabling threads to progress independently and reduce conflicts in contended items. We evaluate our work within a single target **TM** system, using the **STAMP** benchmark suite [Minh et al., 2008] and a custom-made microbenchmark. We consider both performance and energy efficiency metrics.

Our contribution includes a publication made in the context of this dissertation, based on earlier results:

- Rui Ribeiro, José Pereira and Nuno Faria. An Experimental Evaluation of Value Splitting in Transactional Memory Systems. INForum 2023.

1.3 Structure of the document

The rest of this document is organised as follows:

Chapter 2 Introduction of the essential concepts that are needed to understand the core of this dissertation. We focus on the main ideas behind **TM** and how it works.

Chapter 3 Presentation of the state-of-the-art research directly related to this work's core, such as the aforementioned value-splitting techniques, other related **TM** optimisations, different profiling applications, immutable data structures, and the production-ready **Software Transactional Memory (STM)** system that has been picked as our target.

Chapter 4 Specification of our contribution, namely the value-splitting techniques and benchmarks created/adapted to test them. We detail the reasoning behind the main decisions made during the devel-

opment process.

Chapter 5 Evaluation of the value-splitting techniques with the aforementioned benchmarks. We target performance and energy efficiency metrics.

Chapter 6 Final remarks and prospects for future work.

Chapter 2

Background

This chapter introduces the concepts and terminology needed to understand the core of this dissertation. We mainly focus on **Transactional Memory**, as it is the target of optimisation of our work. We present an overall description of what it is, why it is important, and how it works. We also give special attention to a **TM** system used in production, as it allowed the evaluation of our work in a more realistic setting. Finally, we introduce a library of immutable data structures for C++, as it was the foundation of the implementation we describe in the following chapters.

2.1 Transactional memory

A transaction, as first defined by Gray [1981], is a transformation of state. It is *atomic* (either all or none of the changes are made), *durable* (its effects are able to withstand failures), and *consistent* (the state keeps being correct after the transformation, assuming it was before). These three properties were later used by Haerder and Reuter [1983] when coining the **Atomicity, Consistency, Isolation, and Durability (ACID)** principles, with the added *isolation* (effects of concurrent transactions are not visible to each other).

Transactional Memory (**TM**) is a concurrency control mechanism that allows the coordination of threads without the explicit use of locks or other manual synchronisation mechanisms; the **TM** system automatically handles any concurrency issues and conflicts.

The concept of **TM** first appeared as an alternative to locked data structures [Herlihy and Moss, 1993]. Its ideas were based on the transactions used in database systems, applying the same **ACID** properties.[1] **TM** has the main goal of making the development of programs for multiprocessor systems easier while avoiding the pitfalls usually associated with locks, with some benefits being:

- **Easier to write:** Programmers do not need to worry about keeping track of locks and can write

[1] The *D* of *durability* is ignored here since the transactions do not leave the volatile domain.

code as usual.

- **Deadlock elimination:** Transactions can be aborted to make others able to proceed.
- **Composability:** Transactions can be combined to form larger transactions.
- **Easier to maintain data consistency:** Changes are only applied upon a commit; intermediate states are discarded and not visible outside the transaction.

Listing 2.1: A pseudocode comparison between locking and TM.

```
// External locking.
queue1_lock.lock();
queue2_lock.lock();
var item = queue1.pop();
queue2.push(item);
queue1_lock.unlock();
queue2_lock.unlock();
```

```
// Internal locking.
var item = queue1.pop();
queue2.push(item);

LockQueue::push(item) {
  this.lock.lock();
  this.queue.push(item);
  this.lock.unlock();
}
```

```
// Transactional version.
atomic {
  var item = queue1.pop();
  queue2.push(item);
}
```

Listing 2.1 shows a comparison between a usual locking mechanism and a generic lexically-scoped **TM** transaction, when removing an item from a queue to insert it in another. The first locking version is susceptible to deadlocks if another thread tries to lock the queues in the reverse order. The second locking version avoids this problem by having the data structure itself lock its accesses, but in doing so no longer allows for an atomic modification of both queues. This makes the intermediate state exposed to other threads, when the item has been popped off `queue1` but not pushed to `queue2`. The transactional version wraps the critical section inside an `atomic` block, both simplifying and reducing the footprint of the code, while solving the problems of both locking implementations.

It is worth pointing out that, while the programmer using the **TM** constructs has an easier time programming, the complexity of concurrent memory accesses does not simply disappear; this responsibility is instead passed on to the libraries/compilers that provide the underlying **TM** implementation.

2.1.1 Fundamental concepts/terminology

Read/write sets In order to detect conflicts, there needs to be a mechanism in place to track which memory regions have been accessed and modified by a transaction. Addresses that have only been read and not modified are part of the transaction's *read set*, while those that have been modified are part of the *write set*.

Conflict detection Conflicts occur when a transaction's read set overlaps with another transaction's write set. The detection of conflicts usually happens on one of two occasions: at the time that a transaction tries to commit, which known as a *lazy* approach, or at the time that a memory address is accessed, which known as an *eager* approach.

Version management A version management mechanism is responsible for handling the storage of both new and old versions of data, when writes occur in a transaction. Both need to be stored since the outcome of a transaction is unknown: it can either commit, in which case the new version replaces the original data, or abort, in which case the new changes are discarded and the original version remains.

As with the detection of conflicts, versions of modified data can either be managed lazily or eagerly. A *lazy* strategy avoids writing to the target location until the transaction commits, instead storing the data on the side. An *eager* strategy performs in-place changes, at the moment that the targeted memory location is modified, and stores the old data on the side.

This "on the side" where data is buffered can be implemented as a log: either as a *redo log*, where changes are stored so that they can be applied upon a commit (lazy), or an *undo log*, where the original data is stored so that it can be restored upon an abort (eager).

Granularity/metadata In software, **TM** generally admits two distinct levels of granularity: it can be *object-based* or *word-based*. As the naming implies, word-based systems offer finer control over object-based systems since they associate metadata to individual memory addresses, instead of whole objects. This reduces the probability of false positives in regard to conflict detection, which can happen on object-based designs when two distinct transactions modify different fields of the same object. However, there can be a bigger overhead on word-based designs due to the need for overall more metadata.

Concurrency control Concurrency control is a mechanism that ensures data integrity and correctness when being accessed and modified by concurrent transactions. It is usually divided into two distinct categories, depending on the approach: *pessimistic* and *optimistic*.

Pessimistic concurrency control assumes the worst by default, that there will always be more than one transaction trying to modify the same data at any given time, regardless of the probability of conflicts. Therefore, it employs some sort of locking mechanism before accessing the data, be it a shared or exclusive lock. On the other hand, **Optimistic Concurrency Control (OCC)** assumes that conflicts are rare and lets the transactions run freely. In the case of conflict, only one transaction is committed and the rest are discarded.

Disjoint-access parallelism An implementation of word-based **TM** can be described as disjoint-access parallel [Israeli and Rappoport, 1994] if transactions that act on disjoint sets of memory words are able to progress concurrently without conflicting with each other.

2.1.2 Implementation types and examples

There are two major types of **TM** implementation: in hardware, known as **Hardware Transactional Memory** (**HTM**), or in software, known as **Software Transactional Memory** (**STM**).

The first **TM** design was proposed in hardware, by Herlihy and Moss [1993]. It worked by extending cache coherence protocols and adding a dedicated cache for holding uncommitted data. However, it had two major limitations: long transactions were prone to being aborted due to interrupts or synchronisation conflicts and the transactional cache was not able to handle large data sets, if their size exceeded its capacity. This meant that programmers needed to be aware of such restrictions in order to develop programs effectively, undermining the desired **TM** benefits. Implementations with these limitations were later labelled as *best-effort*.

In contrast, *unbounded* hardware systems are able to process both large and long-running transactions. Ananian et al. [2005] proposed the first design of this kind, called **Unbounded Transactional Memory (UTM)**. As a trade-off, it has a more complex implementation than best-effort systems.

Also aiming to solve the best-effort hardware limitations, Shavit and Touitou [1995] introduced the first **STM**. By being based on software, it did not rely on specialised hardware features and was more portable and resilient, in regard to timing anomalies and processor failures. It did not, however, strive to achieve the same level of performance. The proposed implementation also only supported static transactions, which can only access a pre-determined set of memory locations, although later implementations allowed for dynamic transactions, like the one described by Herlihy et al. [2003].

In addition to both **Hardware Transactional Memory (HTM)** and **STM**, there are more two noteworthy alternative designs that exploit the advantages of both hardware and software and try to mitigate their inherent problems and limitations.

The first one is known as **Hybrid Transactional Memory (HyTM)**, with the first architectures being designed in 2006 by both researchers from Sun Microsystems [Damron et al., 2006] and Intel [Kumar et al., 2006]. These hybrid approaches aim to make use of best-effort **HTM** when available, to improve performance, and fallback to **STM** otherwise. Kumar et al. [2006] built upon the **Dynamic Software Transactional Memory (DSTM)** of Herlihy et al. [2003], an **STM** implementation with support for dynamic-sized data structures, by proposing an **HTM** architecture to work alongside it. In contrast, the

approach of Damron et al. [2006] did not require new hardware to function, enabling developers to release programs that would be able to run fully in software in the systems of the time.

The second one is **Phased Transactional Memory (PhTM)** [Lev et al., 2007], a different kind of hybrid approach that alternates between hardware and software modes (or phases). In contrast with typical **HyTM** implementations, **PhTM** does not need to handle the concurrent execution of hardware and software transactions, since it only runs transactions in one mode at a time. This avoids the need for conflict detection between hardware and software and allows for a more streamlined design. A more refined version of this architecture was proposed by de Carvalho et al. [2019], called **PhTM***. It uses a more sophisticated algorithm to decide when to switch modes, as well as a third mode which executes transactions in sequential order, using a single global lock.

2.1.3 Language support

Built-in support for transactions in programming languages is rare, being mostly limited to the functional paradigm. For example, Haskell has the **STM** monad [HaskellWiki] and Clojure has Refs [Clojure].

Outside the functional realm, our research indicates that C++ is the language with the most developments in regard to **TM**, with ongoing work to add official support in the standard. There is a published draft of a technical specification for C++ [ISO, 2015] that is supported by the **GNU Compiler Collection (GCC)** [FSF] since version 6.1. The specification proposes a new type of transactional block, called atomic (exemplified in Listing 2.2). It comes in three variants, each with different approaches in regard to what happens when an exception is thrown. Only functions that are transaction-safe can be called inside an atomic block.

Listing 2.2: An `atomic` block from the C++ draft specification.

```
auto get_next_ticket() -> int {
    // Static variable: only initialised on first call to the function.
    static int counter = 0;
    atomic_noexcept {
        counter++;
        return counter; // An unique number is returned every time.
    }
}
```

Zardoshti et al. [2019] proposed a different design for **TM** support in C++, due to low adoption both by developers and compiler vendors. Instead of using lexically scoped transactions, the programmer would instead define a transaction inside a lambda expression: `tm_exec([&]() { /* ... */ })`. The

transition to lambdas implies other considerations, such as changes in control flow (e.g. if used in the example from Listing 2.2, the return statement would only return from the lambda and not from the `get_next_ticket()` function, as it currently does).

Following the previous design, Spear et al. [2021] proposed a new lightweight version of **TM** support, in hopes of generating interest in the technology at the cost of some features. It has similar syntax to the one from ISO [2015], simply adding a single `atomic do { /* ... */ }` construct. However, the proposal is still in its early stages and does not contemplate an implementation.

2.1.4 Hardware support

Intel introduced **Transactional Synchronization Extensions (TSX)** [Reinders, 2012] by adding **HTM** support for its **CPU**s and providing two software interfaces, **Hardware Lock Elision (HLE)** and **Restricted Transactional Memory (RTM)**. It was released in 2013, included in the 4th generation Intel Core product line. Since then, several issues have been discovered, from bugs in the implementation [Intel, 2015] to memory ordering problems [Intel, 2021]. These issues have led to the deprecation of **TSX** across several families of microprocessors. In addition, **CPU**s since the 10th generation are being released without support for the technology.

AMD also had plans to implement **HTM** in their microprocessor line-up, as early as 2008, upon the first public release of the specification of **Advanced Synchronization Facility (ASF)** [AMD, 2009]. No products implementing the technology ended up being released at the time of writing.

Arm released **Transactional Memory Extension (TME)** [Mann, 2019] in 2021, integrated in their Armv9-A architecture [Arm]. Of the three mentioned companies, it is the only with active support for **HTM**.

2.2 Production-ready STM

Wyatt-STM [Hall] is an object-based **STM** library for C++. It is inspired by the Haskell implementation of **STM**, with additional features to help in handling side effects.

At the core of the library are transactional objects, which encapsulate typical C++ values. These can only be accessed inside of transactions, where the system is able to track concurrent transactions and handle potential conflicts.

Wyatt-STM provides a rich feature set:

- **Nested transactions:** Transactions can run inside other transactions. These nested transactions are self-contained, i.e. a nested transaction can be cancelled without cancelling the outer

transaction.

- **Inconsistent transactions:** Read-only transactions where no validation is done at the end.
- **Explicit retries:** If a given condition is not met, a transaction can call a retry mechanism of Wyatt-STM to cancel the transaction and make a new attempt. This blocks the thread until a variable in the read set of the transaction is changed, avoiding unnecessary wasted work. This feature could be considered akin to waiting on a conditional variable when using locks.
- **Event actions:** Programmer-defined functions that run when a given event is triggered, e.g. after a transaction commits or is cancelled.

Unlike the Haskell implementation, Wyatt-STM does not enforce that the transactions it executes are pure (free from side effects), so the programmer must take extra care to avoid unwanted side effects.

Listing 2.3 shows a direct adaptation to Wyatt-STM of the counter function of Listing 2.2. All the transactional code is explicit in this version, with transactional values being encapsulated in transactional objects. Every read/write must be done inside a transaction, by passing the `WAtomic` object to the `Get()`/`Set()` methods.

Listing 2.3: Example of a Wyatt-STM transaction.

```
auto get_next_ticket() -> int {
    static WSTM::WVar<int> counter(0);

    int value = WSTM::Atomically([&](WSTM::WAtomic& at) -> int {
        int value = counter.Get(at);
        value += 1;
        counter.Set(value, at);
        return value;
    });

    return value;
}
```

A more idiomatic approach is presented in Listing 2.4. The `counter` variable can now be global, since `WVar` will only allow transactional accesses to it—the C++ proposal does not provide any safety guarantees to non-transactional accesses to transactional variables. Moreover, the `get_next_ticket()` function itself can now be used in a composable manner. Since it receives a `WAtomic` object as a parameter, it can be used inside other transactions.

Listing 2.4: Example of an idiomatic approach for Wyatt-STM transactions.

```
WSTM::WVar<int> counter(0);

auto get_next_ticket(WSTM::WAtomic& at) -> int {
    int value = counter.Get(at);
    value += 1;
    counter.Set(value, at);
    return value;
}
```

Overall, Wyatt-STM has a special interest due to its use in a production environment, in an application built by Wyatt Technology. Most of the mentioned **STM** systems in this dissertation have been developed for research purposes, with no immediate real-world applicability.

2.3 Immutable data structures

Thread safety can be achieved with two distinct approaches: by avoiding shared state or enforcing thread synchronisation. Immutable data structures are part of the first strategy, since no thread can modify them after initialisation. Write operations on these structures create a new copy with the applied changes. Conversely, **TM** is a thread synchronisation technique.

Puente [2017] introduced *immer*, a C++ library comprised of immutable data structures based on a variation of **Relaxed Radix Balanced Trees (RRB-Trees)**. It avoids expensive copies by sharing as much as possible of the underlying tree on each update, while still using nodes that are large enough to take advantage of spatial locality.

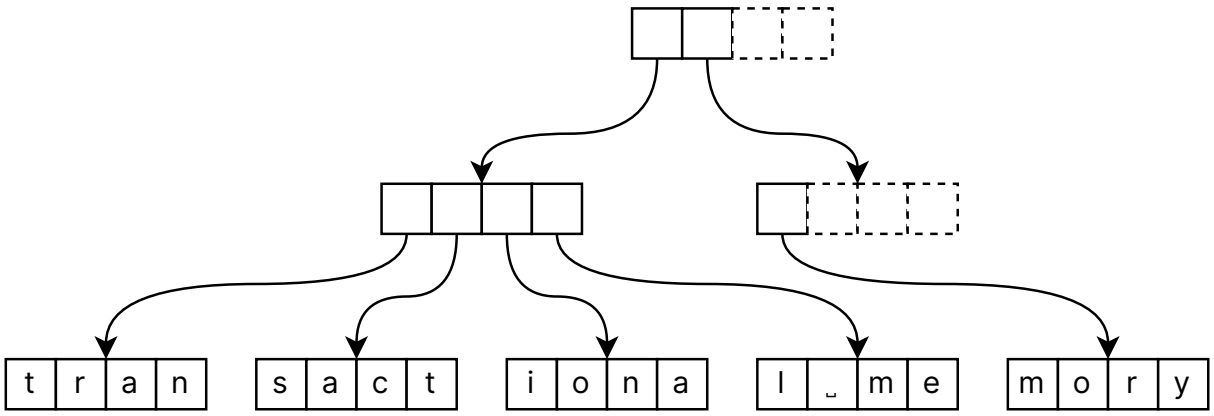


Figure 1: Example of a radix tree.

Figures 1 and 2 demonstrate how an update would occur in a vector of characters, with the change

of a lowercase “o” to an uppercase one. To perform the update, we would first need to look up the target position and create a new node with the updated data. Then, we would recursively update the node’s parents, ending on an updated root node. As we can see, the immutability is preserved, since all the old nodes of the original tree still exist; the newly added nodes for the update (in blue) do not interfere with it.

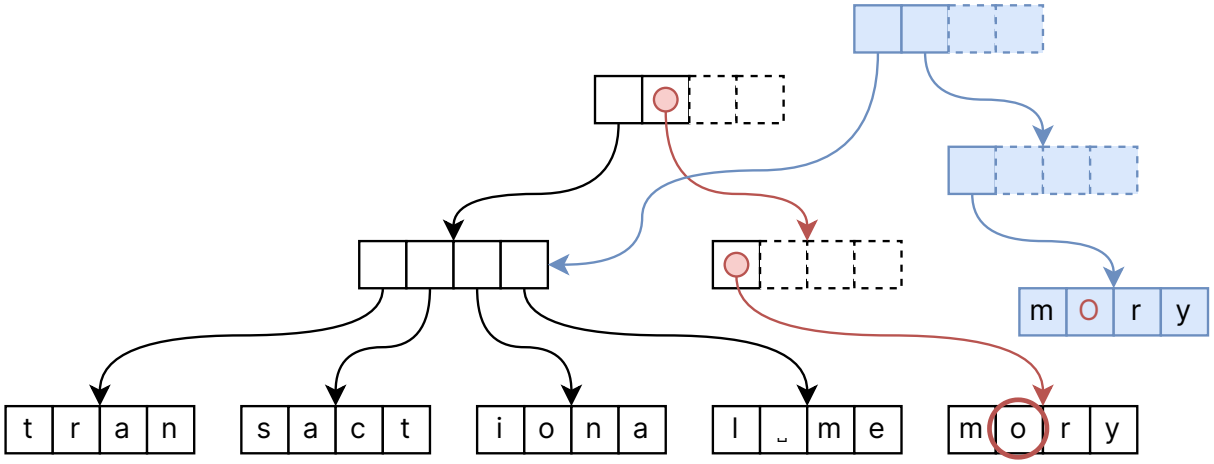


Figure 2: Example of an update to a radix tree.

There are situations where we might need to perform more than a single update to the data structure, e.g. pushing a range of values to a vector. This could be achieved by inserting all the values one at a time, but there would be unnecessary copies of the vector on every update. To avoid this, *immer* provides a transient **Application Programming Interface (API)** for its data structures, where one can request a transient version of a given immutable object and modify it in a way similar to mutable **Standard Template Library (STL)** objects. Once all the changes are applied, the transient object can be converted back into an immutable one. Other references to the original immutable object are not affected.

Chapter 3

Performance of Transactional Memory

This chapter presents research and concepts that, directly or not, are relevant to the improvement and analysis of **TM** performance. We start with a brief summary of optimisations that target **TM**, with the aim of improving performance and generally reducing conflicts. Then, we detail the value-splitting techniques as they were originally developed for database systems—these are the ones in which we based our work. Finally, we present several profiling tools with different goals, namely for general-purpose performance, specifically for **STM**, and for energy consumption.

3.1 Transactional memory optimisations

3.1.1 Transactional boosting

STM systems do not take into account the semantics of the objects that transactions interact with, sometimes making transactions conflict when, semantically, there should not be a need to. Herlihy and Koskinen [2008] propose *transactional boosting*, a methodology for enabling higher concurrency in contended transactional objects.

As an example, the authors present a set implemented as a sorted linked list. Semantically, in the set $\{1, 7, 30\}$, two transactions that perform the concurrent addition of 3 and 17 should not conflict with each other, since they are different numbers that will be added in different positions. In practice, however, assuming both transactions must read every item from the head of the linked list up until the point where the new number should be added, the read set of one transaction inevitably overlaps with the write set of the other.

In order to boost an object, there needs to be a specification of its semantics, comprised of the inverses of all object operations along with rules that state how the operations are able to commute. The inverses are required for the reversal of aborted transactions, while the commutativity rules are required to let the

STM know when to let concurrent transactions act on the same object without conflicting.

3.1.2 Type-aware transactions

Like transactional boosting, the work of Herman et al. [2016] aims to leverage object semantics to improve concurrency in a **STM** system. Their design, called **Software Transactional Objects (STO)**, works by leaving up to the object the management of its modifications, locking, and version verification. The **STO** system itself only works with reads and writes on an abstract level on these data types.

As an example, the authors present an increment-only counter. In a typical **STM** system, an increment would both read the counter and write to it the updated value, inevitably leading to conflicts. In **STO**, however, since increment is an inherently commutative operation, it can be executed in concurrent transactions without the need to abort any of them.

The **STO** library offers transactional versions of several data structures from the C++ **STL**, for general purpose use; the implementation of other custom types is also available, but it is only targeted at advanced developers, in contrast with the transactional boosting approach.

3.1.3 Delayed actions

Delayed actions [Diegues and Romano, 2015], as the name suggests, delay the execution of certain actions until the transaction commits, where they can be executed sequentially to avoid unnecessary aborts. However, this can only be applied to operations whose output is not read in the transactions where they are executed.

For example, a global counter that logs the number of executed transactions could benefit from such optimisation. The increment of the counter would only happen after the commit of the transaction and this would not change the semantics of the application, as each transaction that would increment the counter would not use the value itself for anything.

3.2 Database splitting techniques

In general terms, value-splitting is the process of dividing a single value into multiple variables, so that multiple processes/workers are able to work in parallel without creating contention or causing conflicts. The two techniques we analyse achieve this in different ways, in regard to their storage/assignment of chunks and their internal adjustment to varying workloads. None of these techniques weakens consistency, as parallel operations are allowed only as long as they commute and read operations always return the

total current value.

It should be noted that we only consider non-negative integer values as our splitting target in this work, despite these techniques also being applicable to other data types.

3.2.1 Multi-Record Values

MRV [Faria and Pereira, 2023] is a novel technique that aims to mitigate the performance penalties and conflicts that arise in hot spots in distributed database systems. **MRV** does this by splitting contended values across multiple records and then using random numbers to pick the record which will be accessed, ensuring that the updates are uniformly spread.

The number of records designated to hold the value is dynamically adjusted according to the workload. Thus, it is possible for the **MRV** to never merge back into a single value if the contention on it is high enough. This does not pose a consistency problem, as **MRV** operations can be performed independently of the number of records. To read the real value of an **MRV**, the transaction performs a sum of all partial amounts.

There are two main insights presented, regarding the assignment and management of the records:

- *Clients are not statically assigned to records.* Instead, a random number is generated each time a client wants to access the **MRV**. This ensures an even spread of accesses and avoids explicit coordination between clients.

- *The number used for looking up a record is different from the number used as key of the record.* Instead, the records are stored in a ring-like structure and a constant N is selected as the upper bound for the number of records that can exist. In order to perform a lookup, the algorithm generates a random number and chooses the closest record that follows it, looping back to the beginning in case it does not find any. This reduces overhead, as the number of existing records does not need to be stored nor counted.

The paper presents three different implementation strategies: at the application level, as a middleware, and at the database engine level. This last one enables the usage of **Data Definition Language (DDL)** statements by the application and makes the underlying **MRV** implementation transparent.

In the application and middleware approaches, the transformation of a column C from a table T into **MRV** is as follows: first, the original table is renamed to T_orig; then, the values from C are extracted to a T_C table, split across several records; finally, a view is created from the join between the T_orig and T_C tables, named T, like the original table. A before and after is exemplified in Figure 3. The

primary keys of each table are in bold, with PK being the primary key of the original T table and RK the identifier of a specific record of the **MRV**.

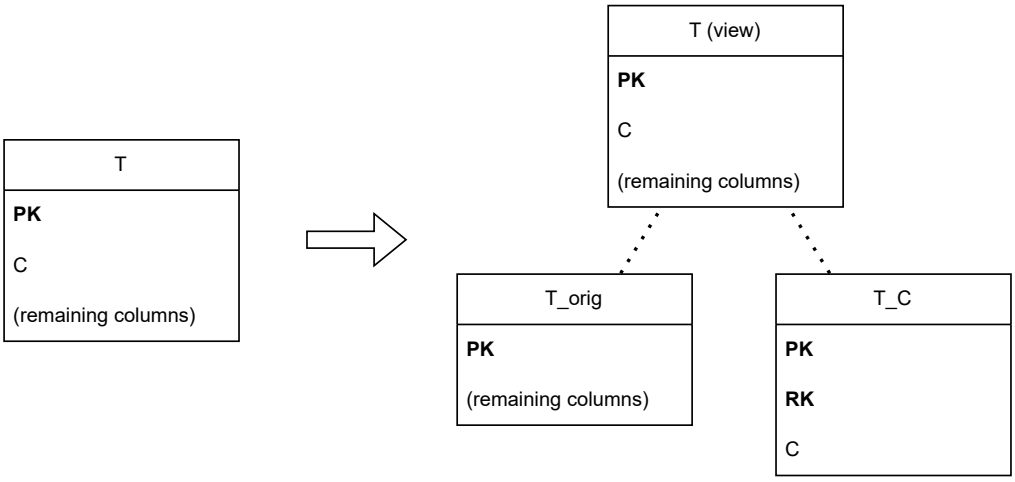


Figure 3: Transformation of a value into a MRV.

As an example, we will show how an addition is performed using the **MRV** structure shown in Figure 4. First, we pick a random integer in the $[0, N-1]$ interval. Then, we look up the first position with an index that is greater or equal to our random integer. Upon finding a record, we can perform the addition as normal. For instance, txn_1 found a record on the exact position that it had randomly picked ($r_i = i = 11$), while both txn_2 ($r_i = 3$) and txn_3 ($r_i = 7$) needed to traverse the ring to find $i = 5$ and $i = 8$, respectively.

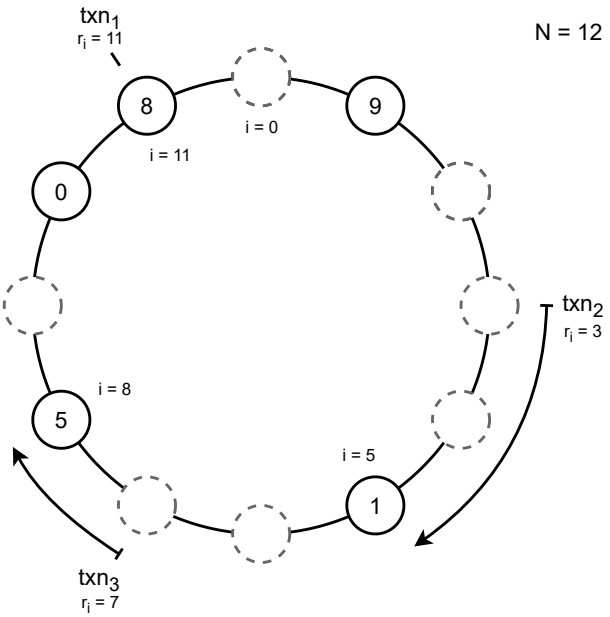


Figure 4: Example of a MRV lookup.

Subtractions start out similarly to additions. However, upon finding an initialised record, we compare the stored value to the one we subtract before performing the operation. If it is greater than or equal, we can perform the subtraction as usual. If it is not, we keep looking up other records until either the aggregated value on our looked up records is enough, resulting in a successful operation, or we loop back to our starting record, resulting in an unsuccessful operation.

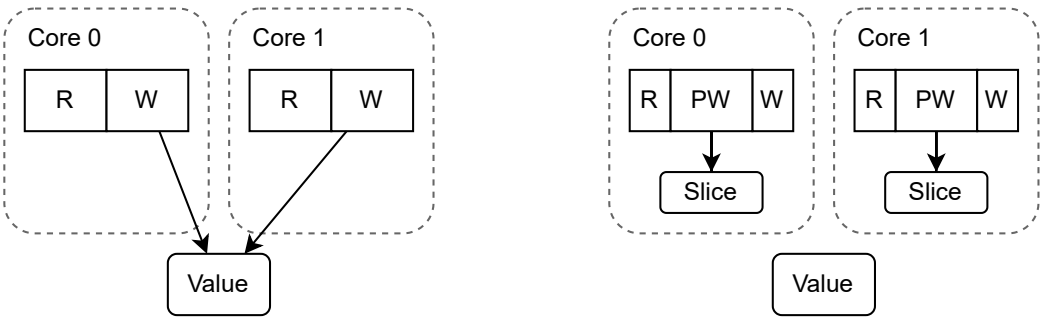
3.2.2 Phase Reconciliation

Phase Reconciliation (**PR**) [Narula et al., 2014] is a concurrency control technique for in-memory transactions, targeting workloads where small subsets of items are subject to numerous updates. In addition to numeric values, **PR** is also designed to work on more complex data structures, such as ordered tuples and top-K sets, which, as we stated, we do not consider in this work.

Doppel, the **PR** database introduced in the same paper, cycles through three distinct execution phases: the *joined* phase, the *split* phase and the *reconciliation* phase. Phase cycles are specific to single data items, i.e. one item can be in a joined phase while another distinct item is in a split phase.

The *joined* phase uses a typical **OCC** protocol and allows any kind of transaction to be executed. Once data contention reaches a level of unnecessary serial execution, the system switches to the *split* phase. Records marked as "split" are divided between cores and only a reserved type of operation—the one where contention was detected—is allowed to execute on these partial values; other types of operations block their respective transactions until the record returns to the joined phase. Finally, the *reconciliation* phase merges the values from the cores back into the global store and the cycle restarts.

In order to maintain correctness during the split phase, only a small subset of operations is available. When reconciling values, these operations must have the same result as if they were executed in a sequential order.



(a) Joined phase.

(b) Split phase.

Figure 5: Example of a PR write.

As an example, we will show how an addition is performed using the **PR** object in Figure 5, in both the joined and split phases. In the joined phase (Figure 5a), concurrent transactions on different cores access the same value in memory, and can conflict when performing an addition (overlapping write sets). If a high enough level of contention is reached, the value switches to the split phase (Figure 5b). While in the split state, different cores have their own slice of the value and can issue "private" writes without conflicting (the write sets no longer overlap). When the value undergoes the reconciliation phase, all private slices are added back together.

Subtractions work similarly, but with the possibility of failing if the slice/value (in the split and joined phases, respectively) does not contain a sufficient amount to subtract. This is in contrast to the **MRV** approach, where it does not fail if a given slice is not enough when the value as a whole is, since the subtraction can be performed over multiple slices.

3.2.3 Considerations

The design of both techniques does not pose have any significant obstacle to an adaptation to **TM**, as their transactional nature fits in with **TM**. Thus, we consider that it is worth evaluating their applicability to **TM**.

3.3 Profiling

Software profiling consists on the measurement of various metrics during a program's execution with the purpose of finding the hot path and possible optimisation targets.

3.3.1 General-purpose

There are several profiling tools available and widely used, such as the `perf` tool included in the Linux kernel [The Linux Foundation]. Some of its alternatives include `oprofile` [Levon] and `gprof` [Graham et al., 1982]. `perf`, in particular, shows stack traces of a program execution with `perf report` on the terminal, as shown on Figure 6.

Flame graphs

Flame graphs [Gregg, 2011] are a different way of visualising stack traces, showing a hierarchical view of function calls. An example can be seen on Figure 7. The x axis is in alphabetical order and does not reflect the order in which functions have been called. The width of each bar is related to how many samples were

```
Samples: 12  of event 'cycles:u', Event count (approx.): 5914570
  Children      Self  Command  Shared Object          Symbol
+   64,80%     0,00%  grep     grep                   [.] 0x00005608ddf2bf24
+   64,80%     0,00%  grep     libc.so.6              [.] __libc_start_main
+   64,80%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5a28f
+   64,80%     0,00%  grep     grep                   [.] 0x00005608ddf2a1fd
+   64,80%     0,00%  grep     libc.so.6              [.] __mbrtowc
+   64,80%     0,00%  grep     libc.so.6              [.] 0x00007fee65bf295a
+   64,80%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5be15
+   64,80%     0,00%  grep     libc.so.6              [.] 0x00007fee65bc2886
+   64,80%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5d0c4
+   39,08%    39,08%  grep     libc.so.6              [.] 0x000000000015f219
+   39,08%     0,00%  grep     libc.so.6              [.] 0x00007fee65c96219
+   31,20%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5cd8a
+   31,20%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5ca9a
+   31,20%     0,00%  grep     libc.so.6              [.] __tsearch
+   24,86%     0,00%  grep     [unknown]              [.] 0xffffffffffffffff
+   24,86%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5b444
+   20,65%    20,65%  grep     libc.so.6              [.] 0x000000000015f24f
+   20,65%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5cdc2
+   20,65%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5c9f4
+   20,65%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5c5cb
+   20,65%     0,00%  grep     libc.so.6              [.] 0x00007fee65c9624f
+   16,98%    16,98%  grep     libc.so.6              [.] 0x000000000015f22b
+   16,98%     0,00%  grep     libc.so.6              [.] 0x00007fee65c9622b
+   12,96%    12,96%  grep     libc.so.6              [.] 0x000000000015f247
+   12,96%     0,00%  grep     libc.so.6              [.] 0x00007fee65b5cd62
+   12,96%     0,00%  grep     libc.so.6              [.] 0x00007fee65c96247
+   10,19%     0,00%  grep     ld-linux-x86-64.so.2   [.] 0x00007fee65df5737
+    8,38%     8,38%  grep     ld-linux-x86-64.so.2   [.] 0x0000000000002260
+    8,38%     0,00%  grep     ld-linux-x86-64.so.2   [.] 0x00007fee65df68fb
+    8,38%     0,00%  grep     ld-linux-x86-64.so.2   [.] 0x00007fee65df4f12
+    8,38%     0,00%  grep     ld-linux-x86-64.so.2   [.] 0x00007fee65df866a
+    8,38%     0,00%  grep     ld-linux-x86-64.so.2   [.] 0x00007fee65ddc6f5
Press '?' for help on key bindings
```

Figure 6: Example of a report from `perf`.

collected in the respective function; i.e. the time spent inside the function. The y axis shows the stack depth, indicating from bottom to top which functions called which.

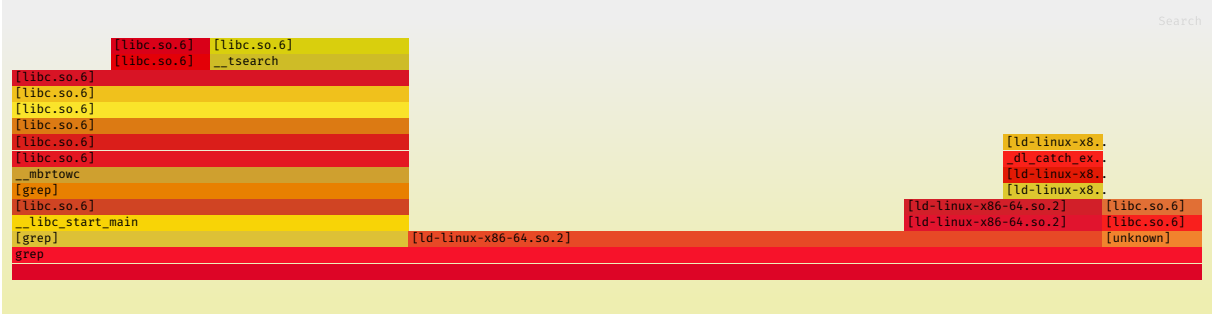


Figure 7: Example of a flame graph.

Causal profiling

Curtsinger and Berger [2015] argue that typical profilers have limited applicability and are not useful for finding significant optimisation spots. This is due to their type of analysis, which usually consists on the report of time spent inside a function. A speed-up on a function that runs for a long time can be useless if it runs alongside another function that takes equally long. The authors exemplify this with a function

that draws a loading animation during a download; a speed-up on the drawing will not make the program faster, since it is still limited by the download time.

With this in mind, they propose a new type of profiling called *causal profiling*, which indicates precisely the lines of code that should be targeted for optimisations and quantifies their impact on the overall performance of the program. This is achieved by virtually speeding up code, where pauses are inserted in concurrent code. The key insight is that the slowdown has the same relative effect as running a specific line of code faster.

3.3.2 Specific for STM

Ansari et al. [2009] proposed one of the first profiling frameworks for **STM**, built on the **DSTM**2 [Herlihy et al., 2006] implementation. Some of the more important collected metrics are as follows: *speed-up* (how it scales with different counts of threads), *in transactions* (percentage of time spent executing transactions), *wasted work* (percentage of time spent executing transactions that aborted) and *aborts per commit*. In addition, the paper introduced two new **TM** metrics: *transaction execution time histograms* (spread of the duration of committed transactions) and *instantaneous commit rate* (percentage of committed transactions at a given moment).

Zyulkyarov et al. [2010] presented new profiling techniques for **TM**, these being focused on finding performance problems on the application using **TM** and not on the **TM** implementation itself. One of the techniques, called *conflict point discovery*, identifies the code statements that were involved in a given conflict, along with the context needed to know where the transaction was called from and which of the involved transactions committed/aborted. In addition, it reports on all conflicting data inside a transaction, not only on the first line where the conflict occurred.

Gottschlich et al. [2012] presented TMProf, a profiler with graphical capabilities for visualising how transactions interact with each other. It also supports playback of replays from program executions, for analysis purposes, and comparisons side-by-side for these executions, to measure the impact of applying a given optimisation.

3.3.3 Energy profiling

Energy efficiency has become an important consideration in recent times, largely due to climate change concerns. Thus, it becomes essential to not only measure software in terms of performance (throughput, latency, etc.), but also in terms of energy consumption.

To perform these measurements, we can usually resort to power meters. They are either external or

internal, but, for our purposes, we are only considering internal ones. These are power meters built into the hardware itself, like on the **Central Processing Unit (CPU)** or the **Graphics Processing Unit (GPU)**. Through software **APIs**, they can provide raw measurements for the consumption of the component at any given point in time. We found that the best supported platforms were the ones built by Intel (**Running Average Power Limit (RAPL)**) and NVIDIA (**NVIDIA Management Library (NVML)**), for **CPUs** and **GPUs** respectively.

There are several applications that make use of these hardware counters to provide a human-readable measurement of energy/power consumption, such as Intel Power Gadget [Intel], nvidia-smi [NVIDIA, 2012], PowerAPI/SmartWatts [Fieni et al., 2020], Scaphandre [Hubblo, 2023], turbostat [Brown], and others.

Chapter 4

Contribution

This chapter presents the development of our value-splitting techniques on **TM**. We begin by defining our approach and the main obstacles we faced when starting out our work. Then, we present detailed descriptions of our developed programs/libraries, for both benchmarks and the splitting techniques themselves. For the benchmarks, we present overviews of the features and targeted workloads. For the techniques, we present the built architecture and reasoning behind every major decision, backed by empirical data.

4.1 Approach

Broadly speaking, our contribution consists of two major components: the implementation of value-splitting techniques in a **TM** system and the creation of benchmarks to evaluate said techniques.

Starting with the implementation of the techniques, we first need to address the main challenges inherent to their adaptation, as databases differ significantly from **TM** systems:

- **Portability:** **Structured Query Language (SQL)** is a widespread and well-established interface for interacting with database systems, regardless of how the system itself is implemented underneath. In the case of the **MRV**'s application/middleware level implementations, it allows for a portable design of the **MRV**, capable of running on multiple systems. **TM**, in contrast, still has numerous differing programming interfaces and not one "tried-and-tested" approach. This means that our optimisations will have to be focused on a single **TM** system and will not work out-of-the-box on other systems.

- **Data access/management:** Data in a database server can be accessed and modified by multiple processes at the same time through the exposed database **API**. **TM**, on the other hand, runs on the application's own address space, restricting any kind of outside access. This means that any kind of splitting needs to occur inside the application process and be connected to the **TM** system

in order to fetch runtime statistics, like the abort rate for each contended value.

As we require a target **TM** system to evaluate the feasibility of our implementation, we outlined some requirements it needs to meet:

- **Be disjoint-access parallel (on word-based systems):** This is crucial, since each split of the value must be accessed without interfering with each other; otherwise, our optimisations will be rendered useless.
- **Is readily available and fully-featured:** Most of the systems described in §2 are proofs-of-concept, with implementations that are not either publicly available or are lacking in features needed for the value-splitting techniques. Our system of choice should ideally work out-of-the-box for our purposes.

Settling on Wyatt-STM [Hall] as our target **TM** system, we have implemented our adaptations as a C++ 20 library. We initially tested the **TM** technical specification for C++, but we found the existing documentation to be sparse and some required features were not implemented, such as explicit transaction cancellation.

The usage of a profiler would help in identifying the parts of the code where we should focus our optimisations. Unfortunately, from what we gathered in §3.3, profilers made for **TM** did not gain any adoption and research on the topic has mostly been dormant since 2012. The causal profiler Coz [Curtsinger and Berger, 2015] has also not received updates in a while, along with missing support for recent distributions of Linux.

4.2 Benchmarks

4.2.1 Microbenchmark

Our microbenchmark models the stock of a single product, subject to several singular operations by multiple clients. It is mainly built to highlight contention issues on transactional values. There is a thread per client, that runs on loop for a fixed amount of time. On each iteration, the client starts a new transaction where it randomly selects which operation it will perform on the product: a read, an addition, or a subtraction. Each operation's chance is determined by startup parameters on the microbenchmark.

As transactional workloads do not usually perform just one operation, we have extended the duration of each benchmark transaction with a loop of computational work, both before and after the operation on

the target object. Our aim is to represent the time that would be spent processing other tasks, as it occurs in real-world workloads. Without this, we would effectively be measuring the time wasted on the creation of the transaction itself, and not the time spent on useful work.

The main parameters of the benchmark are as follows:

- **Clients:** The number of concurrent threads that will run transactions.
- **Duration:** The amount of time (in seconds) that the benchmark should run for.
- **Read percentage:** The percentage of transactions that read the product.
- **Time padding:** The amount of wasted cycles in a transaction to artificially induce longer running transactions.
- **Scale:** The ratio of subtractions per one addition. For example, a scale of 1000 makes the microbenchmark do one 1000 unit addition to the product per 1000 one unit subtractions.

As output, we return three different types of metrics:

- **Throughput:** The amount of committed transactions per second. We present individual throughput counters for reads and writes.
- **Abort rate:** The ratio of transactions that have aborted to the ones that have been executed (aborted plus committed), between zero and one. Each retry counts as a distinct abort.
- **Average worker processing time:** The amount of time between each worker iteration (workers are specific to **MRV** and **PR**; more details on §4.3.1). Essentially, how much time it takes for a background worker to perform maintenance on all objects of a given value-splitting type.

Throughout this chapter, we rely on slight adaptations of this benchmark to decide on specific implementation parameters of our value-splitting techniques. When otherwise stated, we default to the usage of eight threads, five runs of 60 s each, a warm-up of 5 s, and a padding of 100000; we found these values to be optimal with the **TM** system in question (details in §5.2). We also default to the hardware/software configuration presented on §5.1.

4.2.2 STAMP Vacation

Stanford Transactional Applications for Multi-Processing (STAMP) [Minh et al., 2008] is a benchmark suite for the evaluation of **TM** systems. It is extensive, covering a wide range of domains and workloads with its eight included applications, and versatile, being compatible with numerous **TM** designs. For our tests, we have focused our attention on the Vacation application.

Vacation models a travel reservation system, implemented as a set of trees. These track customers and their respective reservations. As per the qualitative assessment made by the authors in the paper, Vacation is a low/medium contention workload with medium read/write sets and medium-size long-running transactions. Considering this, and that we consider value-splitting techniques to be a great fit for online sale systems, we opted to pick Vacation as our only application from **STAMP**, due to time constraints.

The main problem with using any of the **STAMP** applications with our **TM** system of choice is that **STAMP** is designed with word-based systems in mind. Even though it is built to be portable (it uses macros for transaction delimiting), it does not fit in with the object-based approach of Wyatt-STM. With that in mind, we had to port Vacation to Wyatt-STM ourselves, meaning we had to make some adjustments as detailed ahead.

It is important to note that we decided to base our work on the C++ **STL** port of Kilgore et al. [2015], instead of the original code in C. As that port uses standard C++ structures, it has allowed us to more quickly get started with a working version of Vacation on Wyatt-STM.

Data model

The final schema, with our adaptations, is displayed on Figure 8. At the top, there is a manager structure (`manager_t`) that is responsible for storing all the system's data, with each table implemented as a map. Customers (`customer_t`) perform reservations on the three provided resources: cars, rooms, and flights. The information about each reservation of a customer is stored in a set inside the customer itself (`reservation_info_t`), which shares its `id` with `reservation_t`. This structure stores data related with how many resources are available to be reserved and the current price for each one of them.

We have highlighted in blue the variables that have been turned into transactional objects, as is the case for the manager maps (or "tables", as they are called in Vacation) and the `reservation_t` counters. The maps themselves use a coarse-grained approach, as the whole map is protected by Wyatt-STM; we do not use transactional variables per element. We consider this to be the main limitation of our adaptation and, with purpose-built Wyatt-STM transactional maps, this bottleneck could be avoided. However, such data structures were out of the scope of this work.

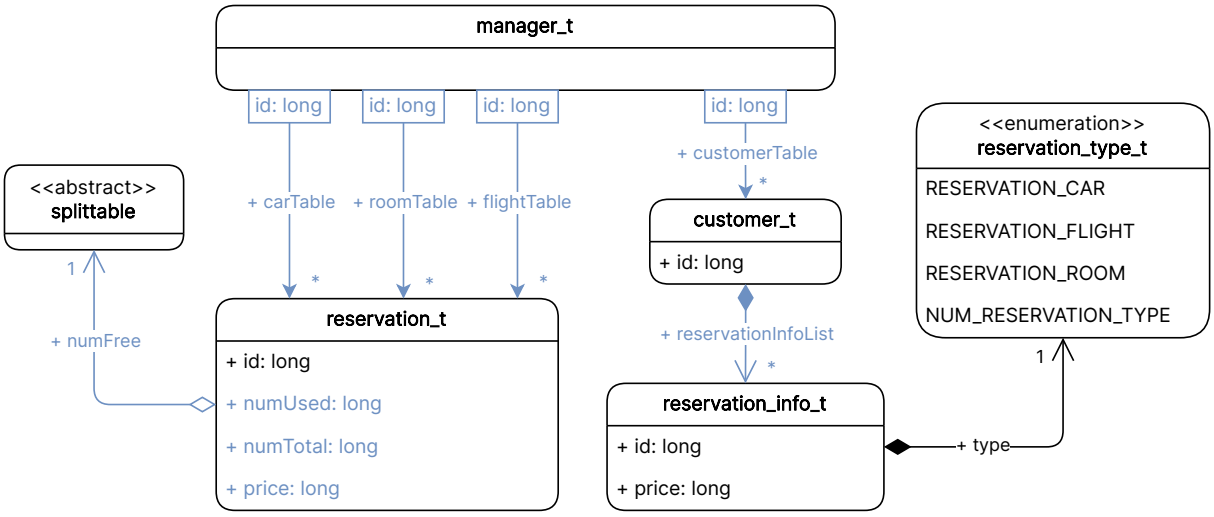


Figure 8: Vacation schema.

Our splitting target is the `numFree` variable of the `reservation_t` struct, as it was determined in the **MRV** paper to be the most common cause of abort. Other variables were not considered due to the large amount of `numFree` variables that exist in the benchmark; as we will detail ahead, our system ended up overwhelmed even with the default amount of `numFree` variables that exist.

Parameters

Vacation allows for runs with the following custom parameters:

- `-n`: Number of queries per task.
- `-q`: Percentage of relations queried.
- `-r`: Number of possible relations.
- `-u`: Percentage of user tasks.
- `-T`: Number of total tasks.
- `-t`: Number of clients (threads).

There are two suggested configurations: low contention (`-n2 -q90 -u98 -r1048576 -T4194304`) and high contention (`-n4 -q60 -u90 -r1048576 -T4194304`). We found the default relation count to be much larger than our system could handle; we made the decision to reduce the count to one-thousandth of the default (1048). This still presented issues in the value-splitting worker timing, as it is presented in later sections of this chapter, but we considered that an even lower amount would deviate too much from the original benchmark.

Transactions

Clients in Vacation run one of the following tasks at any given time:

- **Make a reservation:** The client checks the price of `-n` items and reserves some of them.
- **Delete a customer:** The client computes the total cost of a customer's reservations and then removes it from the system.
- **Add to the item tables:** The client adds `-n` new items for reservation, where an item has a unique ID number and can either be a car, a flight, or a room.
- **Remove from the item tables:** The client removes `-n` new items, where an item has a unique ID number and can either be a car, a flight, or a room.

The tasks are picked based on the frequencies displayed in Table 1. The `-u` parameter defines the percentage of transactions that do not operate on the tables themselves, just make a reservation.

Table 1: Frequency of Vacation transactions.

Transaction	Frequency
Make a reservation	U %
Delete a customer	$(100 - U)/2$ %
Add item to the item tables	$(100 - U)/4$ %
Remove item from the item table	$(100 - U)/4$ %

Implementation challenges

We faced several challenges in the adaptation of Vacation to Wyatt-STM, and, as such, some adjustments were necessary.

One of the main differences lies on the data structures used to store all the travel system's data. The original C version used custom-built trees, which were replaced with `std::map` on the C++ port. There was no functional loss there, as `std::map` uses a red-black tree underneath. However, our implementation uses `immer::map`, which is instead an unordered mapping of values, since *immer* does not provide any type of ordered maps. We analysed the code and determined that this does not pose a problem, as the order of the values is not relevant to the overall operation of the travel system.

The **STL** port uses a **TM** system based on the original GNU (not to be confused with the C++ **TM** draft proposal, which was later also implemented by **GCC**). Listing 4.1 presents the conversion of the customer deletion transaction, to highlight some differences between the original system and Wyatt-STM.

Listing 4.1: Conversion of a Vacation transaction to Wyatt-STM.

```
1  bool done = true;
2  while (1) {
3    __transaction_atomic {
4      long bill =
   ↪  managerPtr->queryCustomerBill(customerId);
5      if (bill >= 0) done = done &&
   ↪  managerPtr->deleteCustomer(customerId);
6      if (done) break;
7      else {
8        assert(0);
9        __transaction_cancel;
10     }
11   }
12 }
```

```
1  WSTM::Atomically([&](WSTM::WAtomic& at) {
2    bool done = true;
3    long bill =
   ↪  managerPtr->queryCustomerBill(at,
   ↪  customerId);
4    if (bill >= 0) done = done &&
   ↪  managerPtr->deleteCustomer(at,
   ↪  customerId);
5    if (!done) WSTM::Retry(at);
6  });
```

The first major change is reflected on transactional accesses. Wyatt-STM requires the usage of the `WAtomic` object (created by the transaction) to access and modify transactional variables; it is not possible to read or write to a transactional value outside of a transaction. On the other hand, the **STL** port does not provide any annotation mechanism to indicate that some variables should only to be used inside of transactions. Like with locks, care must be taken to avoid data races on these objects.

The other major change is in terms of control flow. Wyatt-STM's transactions are enclosed in function objects, which makes the `break` on line six of the original implementation useless. By using the `Retry()` method of Wyatt-STM, we can directly force a transaction to restart whenever it is unable to complete, without needing an explicit loop.

4.3 Implementation

4.3.1 Common implementation details

Both **MRV** and **PR** share key characteristics, as they are both value-splitting techniques. Therefore, in our implementation, we have opted to build a shared core between the two. Our value-splitting interface specifies the following operations, with the respective code in Listing 4.2:

- `read`**:** Fetches the entire value.
- `add`**:** Takes a value and adds it.

- sub**:** Takes a value and tries to subtract it. It can fail if the stored value (minuend) is smaller than the taken one (subtrahend).

Listing 4.2: Interface for value-splitting objects.

```
class splittable {
public:
  auto virtual read(WSTM::WAtomic& at) -> uint = 0;
  auto virtual add(WSTM::WAtomic& at, uint value) -> void = 0;
  auto virtual sub(WSTM::WAtomic& at, uint value) -> void = 0;
};
```

All operations take an `WAtomic` object by reference, which enables the use of the value-splitting objects on Wyatt-STM transactions and promotes composability. Initially, we planned on having the sub operation return a boolean value, to indicate whether the subtraction was successful or not. However, Wyatt-STM uses exceptions for control flow, namely for transaction cancellation. As such, if the subtraction fails, it will throw an exception.

We use various functional immutable data structures from the *immer* library [Puente, 2017]. Updates can be done efficiently and safely by creating a new instance that shares the physical representation of previous content. Therefore, they lend themselves quite well to concurrent scenarios and, in the particular case of transactional memory, to rolling back transactions on complex data structures.

Since our main goal is to study the feasibility of value-splitting techniques in **TM** systems, we have not evaluated every implementation strategy presented in the **MRV** and **PR** papers. Instead, we chose "good enough" defaults, and, where applicable, it will be stated in this dissertation what those defaults are, along with the explicit deviations that were made from the original research.

It should be noted that our implementations do not account for overflows. That is, in C++, unsigned integer arithmetic has a defined behaviour of wrapping around, e.g. adding one to the maximum value (`UINT_MAX`) results in zero. Our library does not check for situations where this might occur, which means that a value split into multiple chunks can silently wrap around when chunks are merged together.

Chunk alignment

CPUs make use of their integrated cache to minimise latency in memory accesses. When the **CPU** needs to fetch something from memory, it first checks if it is already in cache. If it is not, it fetches an entire cache line worth of contiguous data—usually 64 bytes—containing the data that was requested by the program (and more). In multithreaded code, when a thread writes to any position on the cache line, it invalidates

the whole cache line for the other threads on different cores. Thus, two threads can modify two distinct elements on the line and end up forcing cache updates to maintain cache coherency, hindering overall performance. This issue is known as *false sharing*.

A common solution for false sharing involves the usage of an explicit memory alignment, such as forcing one shared object per cache line. C++ offers `std::hardware_destructive_interference_size` since C++ 17, a built-in constant that specifies the cache line size in bytes. Combined with the `alignas` specifier, the alignment of an object can be determined at compile-time, based on the hardware being used to compile the code. Listing 4.3 demonstrates how two contiguous atomic variables in a struct avoid false sharing with the usage of the aforementioned technique.

Listing 4.3: Specifying an explicit alignment for variables in C++.

```
1 struct counters_t {
2   alignas(std::hardware_destructive_interference_size) std::atomic<unsigned int> first;
3   alignas(std::hardware_destructive_interference_size) std::atomic<unsigned int> second;
4 };
```

Our implementations use vectors to store the chunks of value-splitting objects, each chunk being represented by a transactional variable. This could lead one to think that we are susceptible to false sharing. However, the Wyatt-STM's underlying implementation of transactional variables uses shared pointers, which makes it impossible to make any assumption about how the values—that the pointers point to—are organised on the heap, if they are contiguous or not. Therefore, we most likely do not need to account for the possibility of false sharing in our implementations.

To confirm our hypothesis, we ran some tests to measure the impact of explicit alignment on transactional throughput. The first results (Table 2) use `std::hardware_destructive_interference_size` as the alignment value. We also present results with a hard-coded alignment of 64 bytes (Table 3), since C++ erroneously assumed a size of 256 on our testing machine. We verified on the **Operating System (OS)** that the cache line size was, in fact, 64 bytes, using the "`getconf LEVEL1_DCACHE_LINESIZE`" command.

The tables indicate a throughput measurement in writes per second, with an additional column indicating the difference in performance obtained from using explicit alignment.

The test consists in the successive increment of transactional variables, stored contiguously in a vector. Each thread acts exclusively on a variable assigned to it, as we aim to simulate a false sharing scenario. We measure performance using two distinct vector implementations: `flex_vector`, from the *immer*

Table 2: Transactional throughput (writes/s) with and without explicit alignment, for 256 bytes.

	`immer::flex_vector`			`std::vector`		
Threads	Non-aligned	Aligned	Δ%	Non-aligned	Aligned	Δ%
1	12 796.0	12 796.4	0.00	12 796.0	12 797.0	0.01
2	25 447.4	25 467.4	0.08	25 461.2	25 473.0	0.05
4	50 688.2	50 815.0	0.25	50 804.0	50 845.0	0.08
8	86 221.6	90 493.8	4.95	88 372.8	84 064.2	−4.88
16	42 129.4	41 703.6	−1.01	40 971.4	37 935.2	−7.41
32	25 341.2	25 301.6	−0.16	25 520.8	24 534.6	−3.86
64	9679.4	10 329.4	6.72	9785.8	9068.4	−7.33
128	5438.8	3919.8	−27.93	4291.6	6010.8	40.06

Table 3: Transactional throughput (writes/s) with and without explicit alignment, for 64 bytes.

	`immer::flex_vector`			`std::vector`		
Threads	Non-aligned	Aligned	Δ%	Non-aligned	Aligned	Δ%
1	12 796.0	12 795.6	0.00	12 796.0	12 794.6	−0.01
2	25 447.4	25 476.2	0.11	25 461.2	25 471.4	0.04
4	50 688.2	50 493.4	−0.38	50 804.0	50 753.8	−0.10
8	86 221.6	89 556.2	3.87	88 372.8	88 122.6	−0.28
16	42 129.4	44 465.0	5.54	40 971.4	44 822.4	9.40
32	25 341.2	25 792.8	1.78	25 520.8	24 598.8	−3.61
64	9679.4	10 292.0	6.33	9785.8	10 240.4	4.65
128	5438.8	5232.4	−3.79	4291.6	4688.0	9.24

library, and `vector`, from the C++ **STL**. These were picked due to their usage on our value-splitting implementations (MRV and PR, respectively).

Since `flex_vector` is an immutable data structure, we were not able to store the transactional variables directly in it, as we would not be able to modify them. Instead, we stored them inside shared pointers, which could be then stored inside the vector as they would not change, only the values that they point to. `std::vector` has no such limitations and, therefore, we did not use pointers in it, allowing us to avoid the additional level of indirection.

Both tables let us conclude that there is not a substantial (positive) difference between using and not using cache line alignment, indicating the absence of false sharing for both vectors. In the lower thread counts (up to four), the difference is under 1%, which we could consider inside the margin of error. Above that, and notably in the higher thread counts, we verify a larger discrepancy in throughput, both positive and negative, but nonetheless not results we can expect from solving a false sharing situation. False sharing would become noticeable even with only two threads. Thus, we have opted not to specify a custom alignment for our transactional variables.

Managers

Our **MRV** and **PR** implementations are independent of each other, but both follow a similar architecture (Figure 9). At the top level, there is an object manager, which is a singleton that stores pointers to all the active objects of its respective type. These are used by the manager's workers, the ones responsible for periodically adjusting the values to the running workload. The pointers are all contained in an immutable *immer* map, so that worker threads are able to traverse through all the objects without causing conflicts with the concurrent additions/removals that result in a new instance.

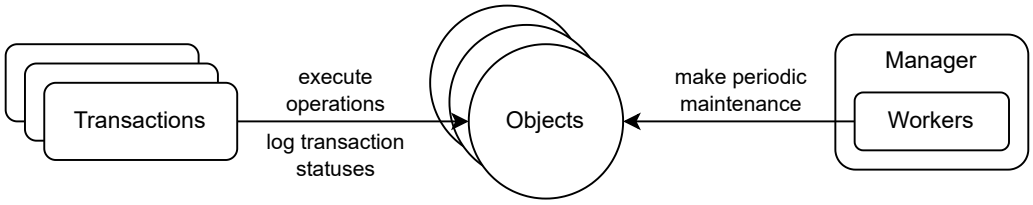


Figure 9: Generic architecture for both techniques.

Our workers run in loop from the moment they are instantiated: they sleep for a fixed amount of time, iterate through all the registered objects, perform their maintenance on them, and start again. Despite being single-threaded, they make use of task distribution across multiple threads. The `std::for_each` function, built into the C++ **STL**, lets developers specify the execution policy of the loop (e.g. sequential or

parallel). For our purposes, and since our loop iterations are independent (each iteration does not depend on the results from previous iterations), we have picked `std::execution::par_unseq` as our execution policy. This way, we do not need to worry about creating the right amount of threads for the amount of objects we have to manage, as the function automatically distributes the workload in a balanced manner. Moreover, the implicit barrier on the loop guarantees us that no two threads are able to perform the same type of maintenance on a given object at the same time, as the worker only exits the loop body when all the threads are finished. Concurrency control is achieved with transactions on each maintenance task, which are started whenever the need to modify a transactional variable arises.

Abort/commit counters

To track contention on our objects, each operation logs its status upon abort or commit. For our purposes, an abort is registered whenever a transaction fails to commit, e.g. due to read/write conflicts on the same **MRV** partition or running out of stock on a **PR** partition. This includes retries, which would make a transaction that retried N times before committing count as N aborts.

The aborts/commits are stored as a single 32-bit atomic unsigned integer, as shown in Figure 10, where its 16 most significant bits are designated for the aborts and the 16 least significant bits for the commits.

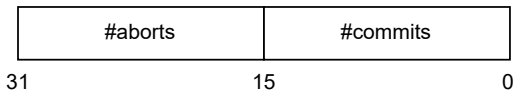


Figure 10: Atomic counter for transactional statuses on a MRV object.

For **PR**, since there is the need for additional data, we use a 64-bit atomic variable instead. As shown in Figure 11, there are two new counters: *aborts for no stock*, which tracks how many of the aborts have been caused by no stock, and *waiting*, which tracks how many transactions are waiting for a phase transition (e.g. trying to read the value while in a split phase).

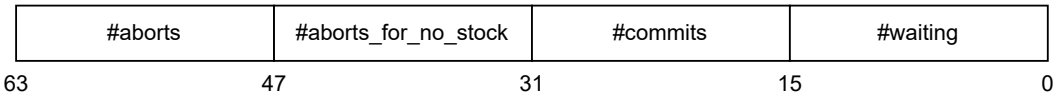


Figure 11: Atomic counter for transactional statuses on a PR object.

This approach allows us to reset both counters atomically, without the need for additional synchronisation mechanisms like locking or **TM**, which would incur unnecessary overhead. We guarantee that the manager's workers always read consistent values, e.g. it is impossible for the **PR** worker to observe

non-zero aborts for no stock while also reading zero aborts. These counters are updated with the code shown on Listing 4.4.

Listing 4.4: Updating the transaction status counters.

```
1 at.OnFail([this]() { this->add_aborts(1u); });
2 at.After([this]() { this->add_commits(1u); });
```

Both `OnFail()` and `After()` methods are provided by the `WAtomic` object of Wyatt-STM. They allow for the registration of a function to be executed upon commit failure or success, respectively.

4.3.2 Multi-Record Values

The underlying data structure of our **MRV** implementation is an immutable *immer* vector of transactional integers—the splits. Making the vector itself immutable is the key to letting it be shared between threads. Since our transactions can, at times, traverse the entire vector (in `read` operations, for example), we needed a way to let a thread be able to iterate through all the splits while another thread added/removed some to/from the vector. The transactional integers escape this immutability, since they are not reassigned, instead they are only internally mutated. To add/remove splits, an updated copy of the vector is made, with the new changes applied. The *immer* library allows for the use of this functional approach without taking a substantial performance hit.

The second key insight of **MRVs**, which was to decouple the lookup numbers from the record keys, was not considered in our implementation. Due to the internal structure of our adaptation, the total number of splits is known—the vector needs it—and can be accessed at no extra performance cost. Every element of the vector is also filled with an available transactional variable, making each lookup exact and not approximate like the original implementation.

MRVs require two background workers, named *adjust* and *balance*. The adjust worker manages the dynamic growth and shrinkage of the **MRV**, which means that, depending on the workload, elements are either added or removed. Care is taken to perform updates on elements being removed, in order to force conflicts with concurrent application threads. The balance worker evens the values stored in the splits in order to reduce contention, namely, to avoid subtract operations from acting on the same splits. Balancing naturally causes conflicts with concurrent threads accessing the same elements, thus ensuring consistency. Our workers follow the same periods as the original adjust and balance workers, which are 1000 ms and 100 ms, respectively.

The original paper evaluated the impact of different window sizes, that is, the interval of time that workers considered for measuring contention (e.g. a window size of 1 s would make the workers take in consideration only the aborts/commits that occurred in the last second). We opted to discard this concept of windows and instead consider all the information that followed the last fetch. That is, we have simple counters for the total number of aborts and commits, and they are reset every time the workers fetch the data. In our case, only the adjust worker needs this data, so it is reset every 1000 ms.

Random number generation

MRVs rely on random numbers to decide which of the splits is going to be picked for a value. Thus, it is important that the used **Random Number Generator (RNG)** adheres to the following properties:

- **Follow a uniform distribution:** All values in the specified range of the **MRV** must have an equal (or near equal) chance of being picked, in order to avoid contention on specific splits.
- **Be thread independent:** Since random number generation is a common operation, the **RNG** must be fast and should not rely on thread synchronisation mechanisms. Our target with the **MRVs** is to promote independent work among threads as much as possible.
- **Use different seeds per thread:** In case of a deterministic **RNG**, the initial seeds that are picked on each thread must be different. Otherwise, all threads will generate the same sequence of numbers and end up competing for the same splits.

It should be noted that, for our purposes, achieving true randomness is not necessary. The optimal **RNG** is one where threads collide as little as possible, while not leaving any of the splits unused. Each one of the three presented points can be achieved with the built-in C++ library: uniform distribution is available through `std::uniform_int_distribution`; thread independence is obtained by using a **RNG** per thread, with the `thread_local` keyword; different seeds per thread can be acquired with `std::random_device`.

Listing 4.5 shows the implementation of our random index after applying the aforementioned insights. For the moment, let us assume the usage of a templated **RNG**. The generator is stored in a static thread-local variable, so that it is only initialised once and each thread has its own. We chose `std::random_device` as our seed, despite having an implementation-defined determinism. Preliminary tests proved this to be effective on our machine of choice, so it remained as our pick. Finally, we feed the generator to a uniform distribution, resulting in a random integer.

Listing 4.5: Implementation of the RNG.

```
template <typename T>
auto random_index(size_t min, size_t max) -> size_t {
    static thread_local T generator(std::random_device{}());
    std::uniform_int_distribution<size_t> distribution(min, max);
    return distribution(generator);
}
```

The missing piece, then, is the **RNG** itself. The C++ **STL** provides three different engines: *linear congruential* (minstd), *Mersenne Twister* (MT), and *subtract with carry* (RANLUX). In addition, we also consider in this work the Xoroshiro/Xoshiro [Blackman and Vigna, 2021] family of engines, due to their popularity. Of all these engines, we want to aim for the highest throughput one.

To measure this, we adapted our microbenchmark to repeatedly invoke the `random_index()` function on multiple threads at the same time, with no padding whatsoever. The results are as follows in Figure 12. As we can observe, all engines scale quite well up to 32 threads, with Xoshiro getting a small but noticeable lead. On the highest thread counts (64 and 128), the results become more inconsistent, but Xoshiro gets the overall best performance. Thus, it will be used for all remaining tests.

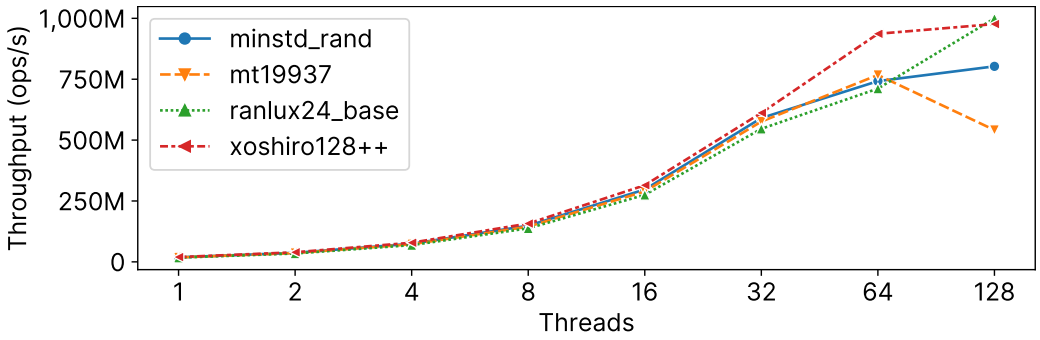


Figure 12: Throughput comparison of different RNG engines.

Besides throughput, quality is another important aspect in the decision of a **RNG** engine. However, since it is a complex topic and out of the scope of this work, we will be setting it aside. We analysed the output of all the engines using density plots and deemed them uniform enough for our purposes.

Internal chunk storage

As we have mentioned, our implementation of **MRV** uses the `flex_vector` structure from the *immer* library to store the chunks of a value. Since this vector incurs additional costs to enforce efficient immutability, we decided to test its lookup performance and observe how it compares to the C++ built-in

vector. Our results are presented in Figure 13.

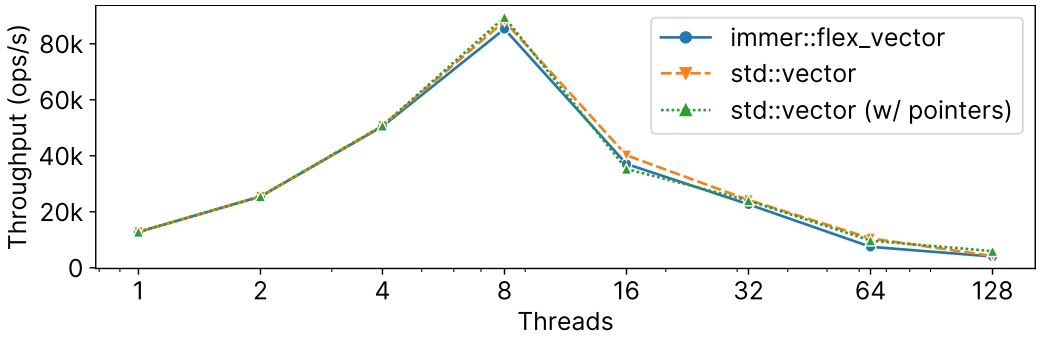


Figure 13: Write throughput on different vectors.

The test consists in the successive increment of transactional variables, stored contiguously in a vector. Each thread acts exclusively on a variable assigned to it, in order to avoid transactional conflicts. We performed tests for three distinct cases:

- `immer::flex_vector`: What we use on our implementation, due to its immutability. Its elements need to be copyable and can only be accessed through `const` methods, therefore forcing us to add a level of indirection (with shared pointers) to maintain mutability within the transactional variable itself.
- `std::vector`: The baseline case, a standard vector from the C++ **STL**.
- `std::vector` *(w/ pointers)*: The same as `std::vector`, but with the additional level of indirection applied to `immer::flex_vector`; works as an intermediate case between the two.

We can observe that performance is identical in all cases. From this, we conclude that `flex_vector` itself is not a bottleneck, not at least in terms of lookup operations. We could also perform comparisons with vector insertions and deletions, but `std::vector` cannot be used in the Wyatt-STM's transactions as it is. Transactions can sometimes restart and, with that, repeat operations several times. Non-idempotent methods, as is the case of pushing to a vector, would yield incorrect results. We could use the built-in vectors as immutable vectors, but that would severely hinder performance due to all the required copies on every vector update, something that *immer* is able to avoid.

Adjust worker

The number of records a given **MRV** object should have is determined by the abort rate of its operations. Like in the original work, the adjust worker is guided by a minimum and goal abort rates. Our target

is to keep the abort rate in the interval of the two. To achieve that, we add records when the rate is above the goal and remove when it is below the minimum. Too few records and we would have unneeded contention; too many records and we would have an unnecessarily large memory footprint, along with balancing issues.

To determine how many nodes we need to add, we use the following formula, based on the original implementation:

$$min(round(1 + \text{size} \times \text{abort\_rate}), \text{max\_nodes} - \text{size})$$

The left-hand side of the min allows for a quick/slow growth of the number of records, depending on the observed abort rate, while the right-hand side lets us limit the maximum number of records on every object, to avoid infinite growth. The removal of nodes is done one at a time, to minimise the impact on other running transactions.

We have mentioned the usage of *immer* due to the thread-safety guarantees it offers. However, we have glossed over the only non-thread-safe operation it exposes: variable assignment. Assignment is how we are able to update the splits' vector and store it in the same original variable, allowing the propagation of changes to other threads. It is clear that we need to protect access to the vector to let the worker perform the adjustments correctly, but we also must take care to not disrupt other running transactions.

The logical first step is to employ the transactional system we are already using. However, placing the vector inside a transactional variable would severely impact performance, as it would conflict with every single running transaction. This is particularly undesirable for record additions, as they do not logically interfere with any existing record. Record removals interact with (at most) two existing records: the one to be removed (the last on the vector, as it is the easiest to remove) and a random one from the remaining, to absorb the contents of the removed one (in case it contains a non-zero value).

As a consequence, in order to achieve high levels of performance, we have disregarded the usage of transactions and resorted, instead, to atomic variables. By using a `std::atomic<std::shared_ptr>`, we enable other running transactions to progress while vector adjustments are being performed. When an operation starts on a **MRV**, it fetches the current version of the vector. Since we have the guarantee that the vector is immutable, we do not need to worry about concurrent modifications causing errors during the program execution.

With atomics, we are able to add new records to the vector without any kind of transaction. Our worker design, as detailed previously, guarantees us that, at any given time for a given object, only one thread is performing an adjustment task. Therefore, and also considering that the adjustment is the only operation that mutates the vector itself, we can (i) safely load the vector, (ii) add however many records we deem

necessary, and (iii) store it again in the atomic pointer. No other thread has modified the vector in this interval, and therefore we are able to avoid the usage of mutual exclusion for this whole section. Once the atomic pointer is synchronised throughout all the threads, other transactions will be able to use the new records.

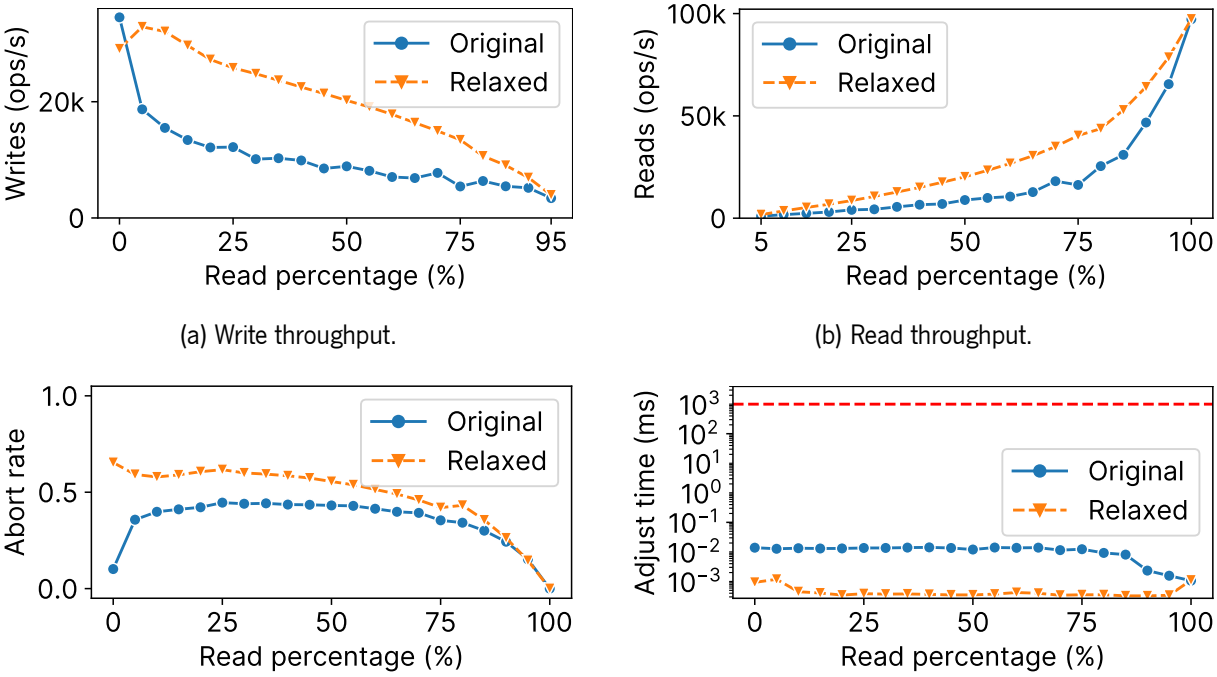
On the other hand, the removal of a record requires a more intrusive approach. When removing a non-zero record, we need to transfer its value to another record. As we need to operate on transactional variables to perform the transfer—and force conflicts on transactions that are operating on our removal target—we have no other option but to start a transaction. We can load the vector and create the new updated version before the transaction starts, ensuring that, even if the transaction restarts, we do not repeat the removal from the vector. One benefit of loading the pointer to the split *a priori*, if the transaction restarts after storing the atomic pointer to the vector and other transactions read from the vector, is that they will not be able to fetch the split that we removed. A simplified version of the final function is demonstrated in Listing 4.6.

Listing 4.6: Simplified function for a MRV record removal.

```
auto mrv_flex_vector::remove_node() -> void {
  auto splits = *std::atomic_load(&this->splits).get();
  auto size = splits.size();

  // Fetching these values here saves time on the transaction (and avoids consistency issues).
  auto last_split = splits[size - 1];
  auto absorber = splits[utils::random_index(0, size - 2)];
  auto new_splits = std::make_shared<splits_t>(splits.take(size - 1));

  WSTM::Atomically([&](WSTM::WAtomic& at) {
    auto last_split_value = last_split->Get(at);

    // This ensures that other threads reading/writing to the last split will conflict.
    last_split->Set(0, at);

    // Perform a transfer to the absorber, if the last split contains a positive value.
    if (last_split_value > 0) absorber->Set(absorber->Get(at) + last_split_value, at);

    // It does not matter if this store is repeated, in case the transaction restarts.
    // In effect, it is idempotent, as the `new_splits` variable was created outside
    // of the transaction.
    std::atomic_store(&this->splits, new_splits);
  });
}
```

The original work uses a minimum abort rate of 0.01 and a goal of 0.05. Our preliminary tests showed this to be too restrictive, as it led to the creation of unnecessarily large amounts of records and,

consequently, lower overall throughput. As an alternative, we relaxed the ideal abort rate interval to 10 times the original, with the minimum and goal now being 0.1 and 0.5, respectively. Results for the microbenchmark and Vacation are displayed in Figures 14 and 15, respectively. The balance worker was disabled for these tests to isolate the adjust performance.



(a) Write throughput.

(b) Read throughput.

(c) Abort rate.

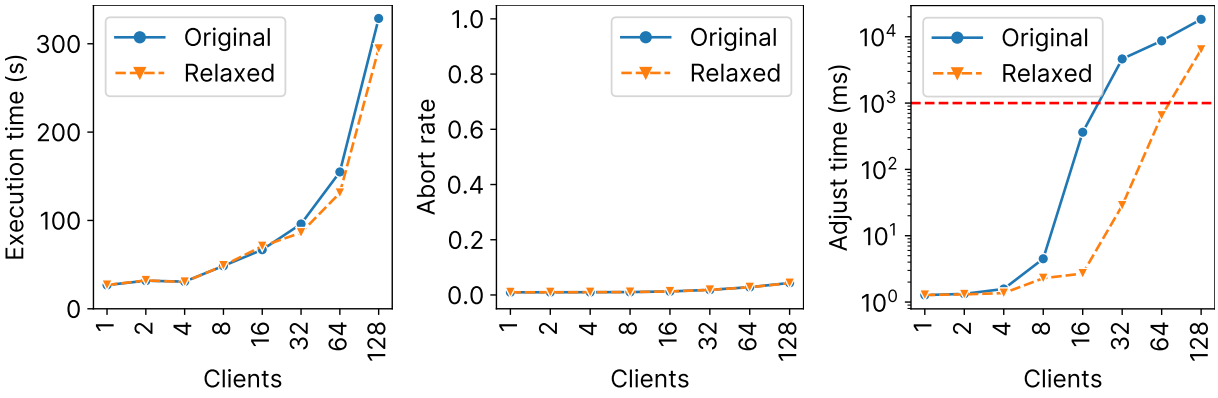
(d) Adjust time.

Figure 14: Microbenchmark results for different abort rate targets.

The microbenchmark results demonstrate the stark difference between the two abort rate settings. At the expense of an (expected) overall higher abort rate, the relaxed settings provide much higher throughputs, both in reads and writes. The adjust time graph confirms our initial findings, that there were too many records being created. With the relaxed settings, the adjust worker takes less time performing adjustments as it does not need to perform as many. In any case, since this test only contains one **MRV** object, the adjust time is well below the defined interval time, which is denoted by the dashed red line.

Results from the Vacation benchmark, using the high contention configuration, do not exhibit such distinct behaviour. There is, nonetheless, a clear advantage in terms of adjustment time. The original abort rate parameters make the worker exceed the adjust time interval at 32 clients, while the relaxed settings only make it trespass at 128 clients. Surpassing the time interval means that the worker will work non-stop, as the next batch of adjustments will already be queued by the time the worker finishes a round of adjustments.

Considering the results obtained from both benchmarks, we concluded that the best approach for the



(a) Execution time. (b) Abort rate. (c) Adjust time.

Figure 15: Vacation (high contention) results for different abort rate targets.

adjust worker was to apply the relaxed settings. Thus, moving forward, we assume this strategy for all **MRV** tests.

Balance worker

Due to the random nature of **MRV** writes, splits can become unbalanced over time—that is, some splits can end up holding a majority of the total value, while others have near-zero counts. The process of balancing consists of (i) picking a certain amount of splits, (ii) combining their values, and (iii) dividing the sum equally among them.

The original paper proposes four different strategies to balance records:

- **None:** The worker does not do anything; no records are affected. This will serve as our baseline.
- **Random:** The worker picks two random records from the entire set to balance.
- **Min-max:** The worker picks the largest and smallest values of the set to balance the two of them.
- **All:** The worker balances all values of the set.

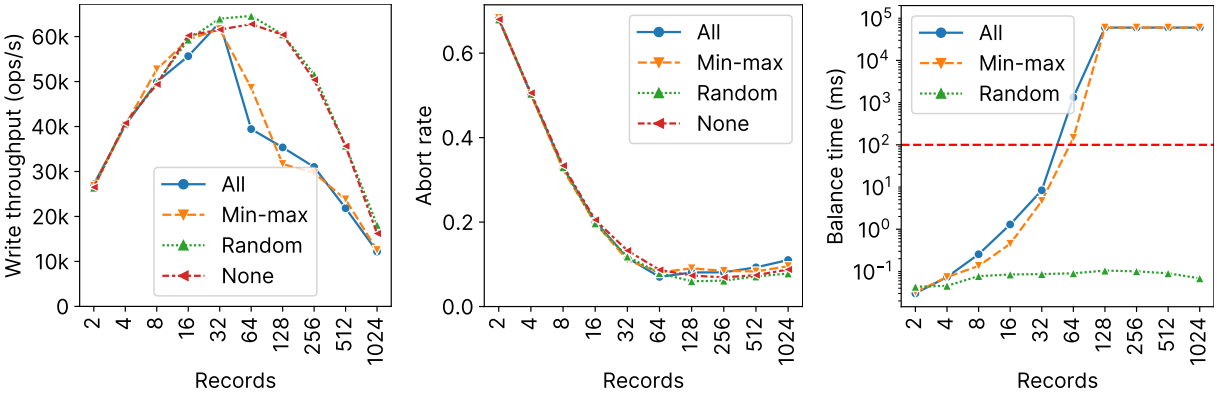
Like in the original implementation, we have applied a small optimisation to *random* and *min-max*: if the values of the two chosen records are close enough (up to a difference of five units), we do not perform the balance and no changes are applied. This way, we avoid making unnecessary balances that would hinder overall performance.

To determine the optimal balance strategy, we used both our microbenchmark and **STAMP** Vacation. These present two significantly different workloads: the microbenchmark only contains a single value where all threads act upon, creating a substantial level of contention, and Vacation has a large set of **MRV**

objects where client threads are dispersed throughout, creating contention at the worker level since it has to manage a considerable amount of values.

We decided to disable the adjust worker for these experiments and test with fixed amounts of records per **MRV**, to isolate balance performance for different **MRV** sizes. The results are presented in Figures 16, 17 and 18.

In regard to the microbenchmark, Figure 16 displays results for a write-only workload (0 % reads), while Figure 17 presents a mixed workload (50 % reads).

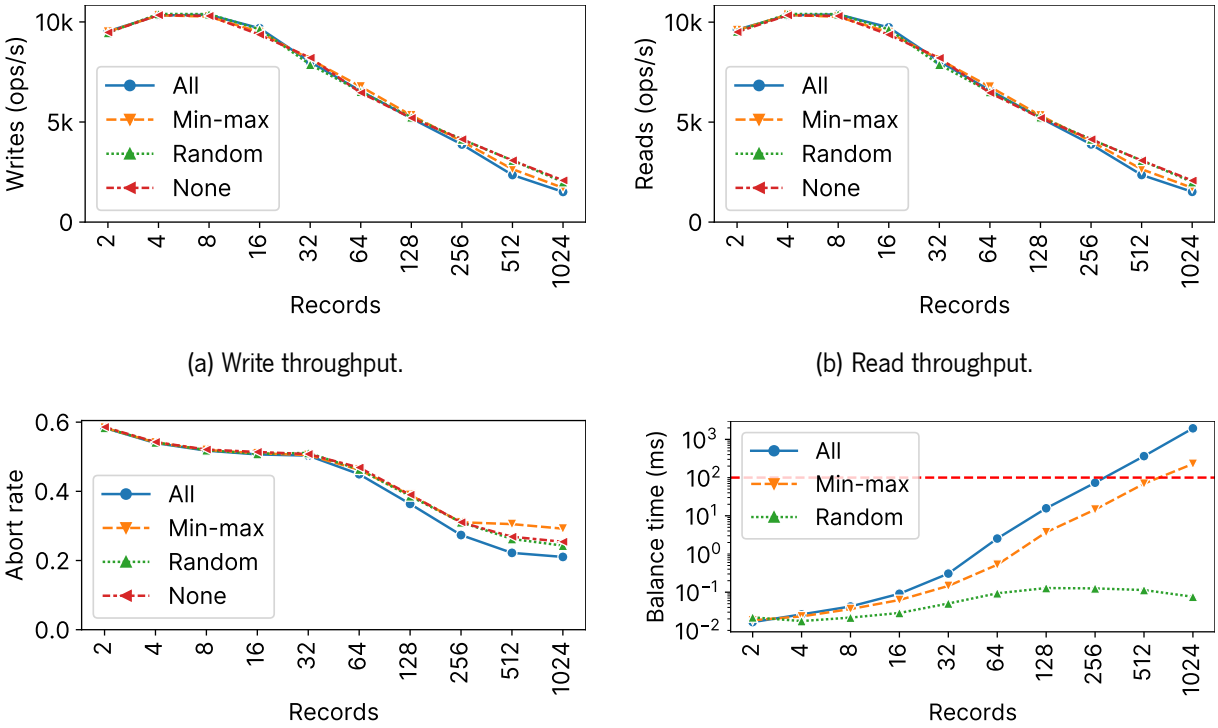


(a) Write throughput. (b) Abort rate. (c) Balance time.

Figure 16: Microbenchmark (0% reads) results for different balance strategies.

Overall, we were surprised with the outcome. Both microbenchmark scenarios demonstrate that our balancing algorithms do not help in terms of performance when compared to not performing balances, even causing a negative impact in some cases. *Min-max* and *all* prove to be too expensive to perform, with balancing times nearing or surpassing the expected balance interval for higher record counts (Figures 16c and Figure 17d). Considering that the microbenchmark only has one single value, the processing times are clearly not suited for more realistic scenarios, with multiple **MRV** objects in memory; this is the kind of scenario Vacation tests. This balancing delay is reflected on the write throughput (Figures 16a), where we see a significant drop at 32 records on the write-only workload. *Random*, being the only algorithm that does not scan the entire **MRV**, is able to sustain a very low balance time and performance on par with using no balancing. It is worth noting that the balance time for higher record counts in Figure 16c maxes out at 60 s since the test only runs for 60 s.

In regard to the Vacation benchmark, Figure 18 presents results for the high contention parameters. As with the microbenchmark results, we again observe that no balancing algorithm has better performance than performing no balancing. Figure 18c shows the balance worker lagging behind the expected balance time interval, due to the higher number of objects it has to manage. No balancing algorithm is able to stay



(a) Write throughput.

(b) Read throughput.

(c) Abort rate.

(d) Balance time.

Figure 17: Microbenchmark (50% reads) results for different balance strategies.

in the specified balance interval at any record count. Except for *all*, which takes a longer time to finish the benchmark, all balance strategies perform identically.

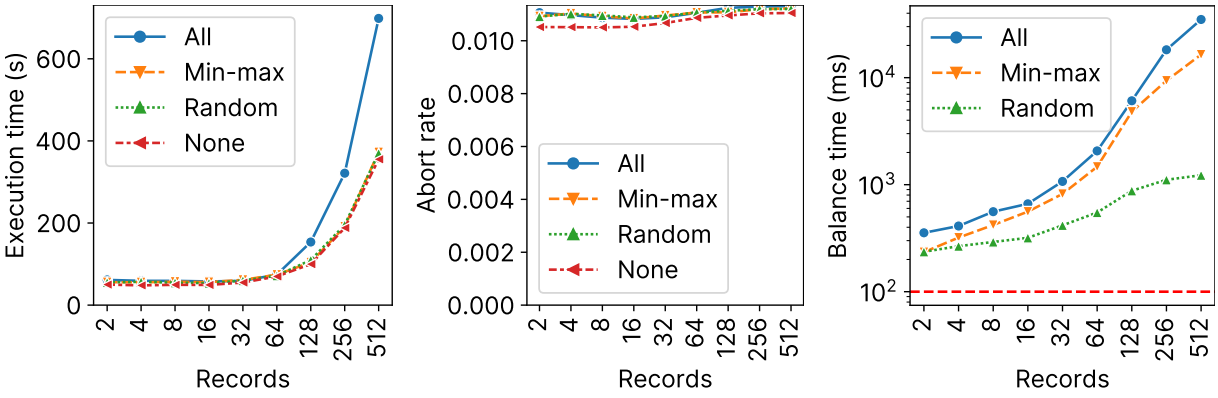
Considering the results obtained from both benchmarks, we concluded that the best approach for the balance worker was not to use it. Thus, moving forward, we assume no balancing for all **MRV** tests.

4.3.3 Phase Reconciliation

PR, due to its phase shifting nature, has a different internal structure depending on the phase. When the object is in the joined phase, it consists of a single transactional integer. When it is in the split phase, it consists of a vector of transactional integers, each exclusive to its assigned thread. We use a transactional boolean to indicate the object state, if it is in a split or joined phase. Since all transactions that operate on the object need to fetch this boolean, a phase change will conflict with all running transactions and force a restart.

PR has only one background worker, responsible for triggering phase transitions on the objects. Every 20 ms, as in the original paper, the worker fetches the current metadata and decides whether to transition or not.

One downside of **PR** is the static thread allocation. In particular, in our implementation, threads must



(a) Execution time. (b) Abort rate. (c) Balance time.

Figure 18: Vacation results for different balance strategies.

register themselves with the **PR** system before any operation is executed and cannot be later unregistered. By comparison, **MRVs** are not bound to the number of threads that are interacting with them, allowing for more flexible workloads.

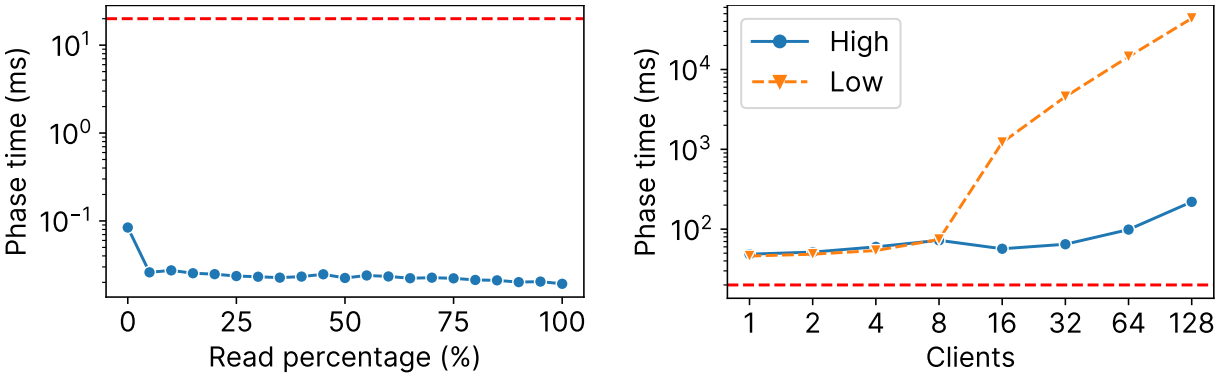
Phase worker

Phase transitions occur as detailed in §3.2.2. Note that the reconciliation phase is not treated as an explicit state in our implementation, since the split phase transitions directly into the joined phase within a single transaction, avoiding consistency issues. Reconciliation is triggered similarly to the adaptation used in the **MRV** paper [Faria and Pereira, 2023]: it can happen if there is any client waiting (e.g. a client tried to perform a read and is now blocked until the joined phase) or if there was any abort due to no stock (zeroed counter).

The short transition interval (20 ms), combined with the aforementioned reconciliation triggers, means that our worker must be responsive and able to transition objects that have a pending phase switch. This is a crucial factor for **PR**, even more so than with **MRVs**, since client threads can be blocked on certain operations while the phase is not switched. No **MRV** operations require real-time worker intervention, which enables client threads to progress despite slow workers and not ideal **MRV** parameters.

We have applied our benchmarks to determine how the phase worker performs, with results demonstrated in Figure 19. The red dashed line represents the phase worker interval (20 ms); the worker should not surpass it in order to keep a responsive operation.

On the microbenchmark (Figure 19a), regardless of the read percentage, the worker is able to keep up with the phase transitions in time. Considering the benchmark only acts on a single value, it is essential that the obtained time is well below the limit, to enable high object counts.



(a) Microbenchmark.

(b) Vacation (high and low).

Figure 19: Phase transition intervals.

On the Vacation benchmark (Figure 19b), we present results for both high and low configurations. In both workloads, it is clear that the phase worker lags behind the target. In practice, this means that client threads waiting on a phase transition (e.g. the object is in split phase and a client want to read the value) will freeze for longer, while the worker processes other objects. The low contention configuration has a higher phase transition time due to its necessity of having more transitions; the higher contention workload can make the objects function for longer on the split phase.

There is no clear solution for this phase transition delay. The worker must check for phase switches at a regular interval, as it is the core of how **PR** works. The conflict-inducing nature is also inherent to **PR**, as a phase transition requires the modification of the value as a whole.

Chapter 5

Evaluation

This chapter presents an evaluation of the value-splitting techniques with the developed benchmarks. First, we detail the test environment used for this evaluation, along with a study conducted to determine the ideal test parameters. Then, we present the main performance evaluation, using our benchmarks. We evaluate both throughput and abort rate, to determine the viability of using value-splitting in **TM** systems. Finally, we also present an energy consumption analysis of several selected benchmark scenarios.

5.1 Test environments

We defined two distinct test environments for our evaluation, each one with a different purpose. The performance tests were all executed on the following ARM machine, due to its high core count:

- **CPU:** 2x HiSilicon Kunpeng 920 (ARM)
- **RAM:** 8x 32 GB DDR4 2666 MT/s
- **OS:** Rocky Linux 8.7
- **Kernel:** Linux 4.18.0
- **Compiler:** gcc 12.2.1, with flags `-O3` and `-march=native`

Each one of our Kunpeng **CPUs** houses two **Non-Uniform Memory Access (NUMA)** nodes of 32 cores each, with no **Simultaneous Multithreading (SMT)** capabilities. Therefore, in total, the system contains 128 logical processors. As such, we limited the maximum thread count for our tests to 128.

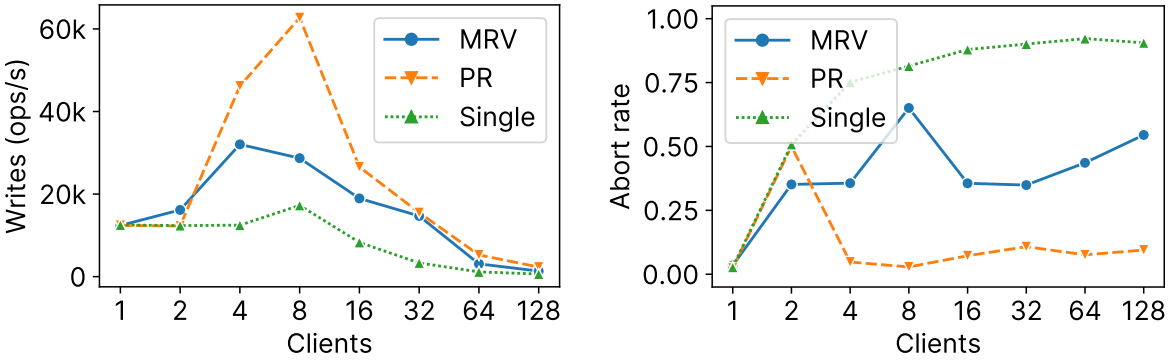
Ideally, the energy consumption tests would also be performed on this machine. However, as the profiling tools we have researched do not target ARM processors, we were forced to employ different hardware. Thus, we used a laptop with the following configuration:

- **CPU:** Intel Core i5-5300U
- **RAM:** 8 GB DDR3 1600 MT/s
- **OS:** Arch Linux (rolling release), last updated on 14/08/2023
- **Kernel:** Linux 6.4.8
- **Compiler:** gcc 13.2.1, with flags `-O3` and `-march=native`

This **CPU** contains only four logical processors, a significantly inferior count when compared to the other test environment. Despite this, we still considered worthwhile the analysis of the obtained results.

5.2 Optimal parameters

We created our first experiment with the goal of determining the optimal number of clients (threads that run transactions) for our tests, using our microbenchmark. The results are presented in Figures 20 and 21, for write-only and mixed workloads, respectively.



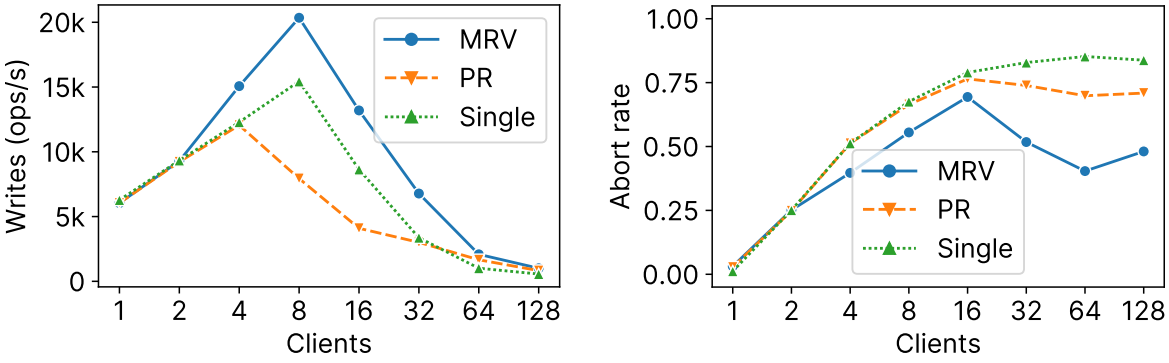
(a) Write throughput.

(b) Abort rate.

Figure 20: Pure write workload with variable number of clients.

In a pure write scenario (Figure 20), **PR** has the best overall performance of the three, due to static thread assignment. Since the test only performs additions and no reads, the object can stay in the split phase, and each client can effectively work independently of the others. Overall, **MRVs** do not fall too far behind—save for the four and eight client tests—while also delivering better performance compared to the baseline case. The single-value approach does not scale as transactions are not able to concurrently update the object, as otherwise we would have data inconsistencies due to read/write conflicts. This is made clear in Figure 20b, where the *single* abort rate increases as the number of clients increases. Both

value-splitting techniques significantly reduce the abort rate, with **PR** nearing a 0 % rate due to its split phase. It should be noted that for two clients, we observe a similar behaviour between *single* and **PR**, since the **PR** object stays in the joined phase.



(a) Write throughput.

(b) Abort rate.

Figure 21: Mixed (50 %) workload with variable number of clients.

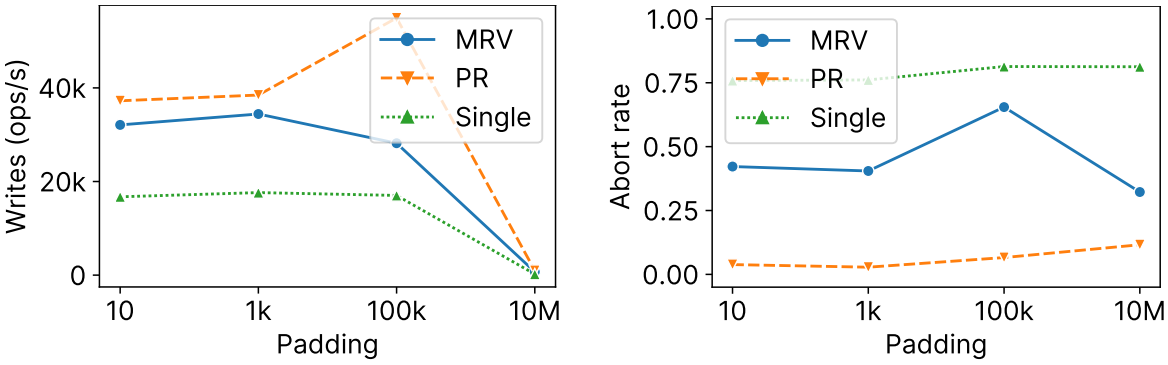
The mixed workload[1] in Figure 21 shows the inherent limitations of **PR**. Since half of all transactions perform a read on the value, the system must be on the joined phase to be able to execute them. Taking in consideration that any phase transition conflicts with every single running transaction, it becomes clear that the constant switching between phases proves detrimental to overall system performance. Comparatively, **MRVs** are able to maintain a small but substantial lead, both in throughput and abort rate.

In summary, the most interesting (and overall best) results are obtained with only eight clients. Therefore, this will be our target for the remaining tests.

The padding we mentioned before is a simple loop of successive additions to a local variable not used elsewhere. We have used a value of 100k iterations on the previous test, which we found to be a large enough amount without too big of an impact in performance, as shown in Figures 22 and 23.

Value-splitting is useful independently of the transaction length, but even more so on long-running ones. Beyond the 100k mark, the **TM** system shows its limitations and the throughput of all the implementations drops significantly.

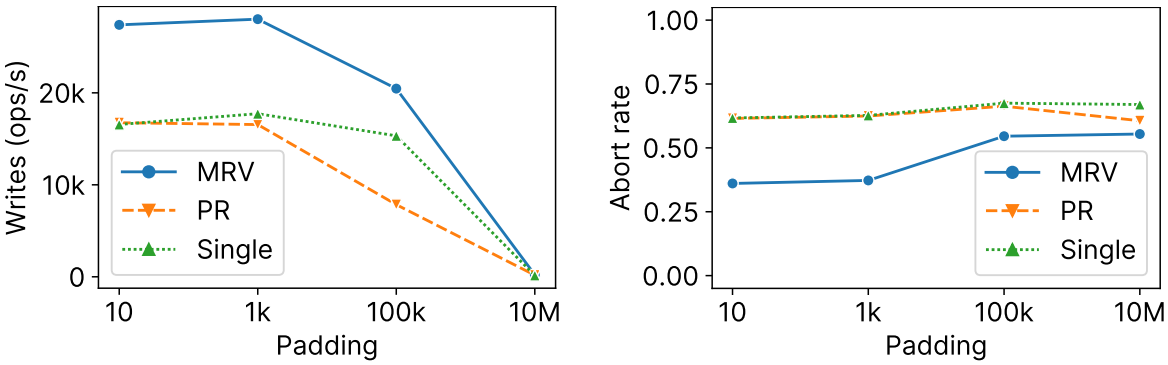
[1] Despite being a mixed workload, we have opted to show only the write graph, since the read one is identical.



(a) Write throughput.

(b) Abort rate.

Figure 22: Pure write workload for eight clients with a variable amount of padding.



(a) Write throughput.

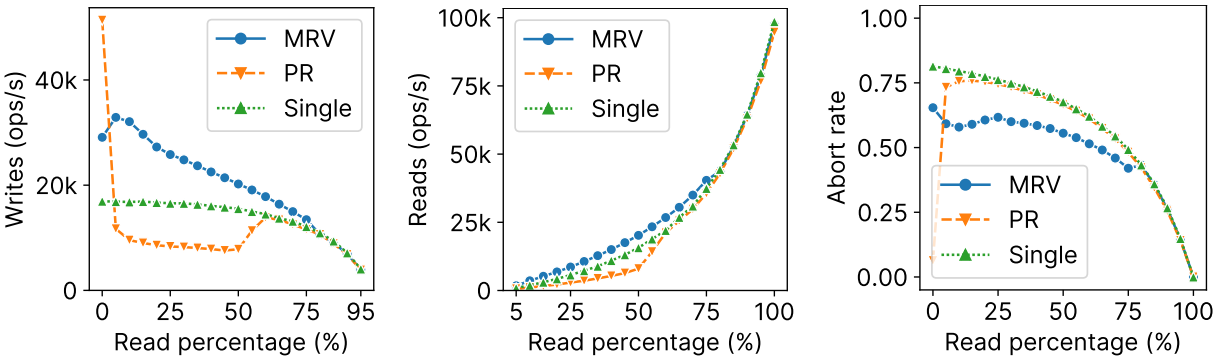
(b) Abort rate.

Figure 23: Mixed (50 %) workload for eight clients with a variable amount of padding.

5.3 Performance evaluation

5.3.1 Microbenchmark

We will start by analysing the microbenchmark results, using a variable read percentage to understand how the different value-splitting techniques adapt to different workloads. As stated in §4.2.1, we ran the test with eight threads, five runs of 60 s each, a warm-up period of 5 s, and a padding of 100k. Results are presented on Figure 24.



(a) Write throughput. (b) Read throughput. (c) Abort rate.

Figure 24: Microbenchmark results.

As expected from the previous section, in pure write workloads, both value-splitting techniques have a major performance improvement over the baseline code in terms of throughput (Figure 24a). However, as reads are introduced into the workload, the **PR** write performance drops by a significant amount, even falling below the baseline, as frequently switches back and forth between joined and split phases. With a share of at least60 % read operations, it stays in the joined phase and effectively defaults to a single value. **MRV**, on the other hand, is able to keep well above *Single* performance up until the 80 % mark, when it also effectively defaults to being a single value.

In terms of read throughput (Figure 24b), we can observe a similar behaviour as for write throughput. **MRVs** perform slightly better than *Single* while **PR** stays below; both end up following the *Single* curve on the same points of Figure 24a. It is interesting seeing **MRV** surpassing *Single* on reads; the significantly higher write performance mitigates the penalty created by having multiple chunks to read. PR suffers, again, from a performance penalty in low read workloads, due to the need to wait for the joined phase to proceed with the read.

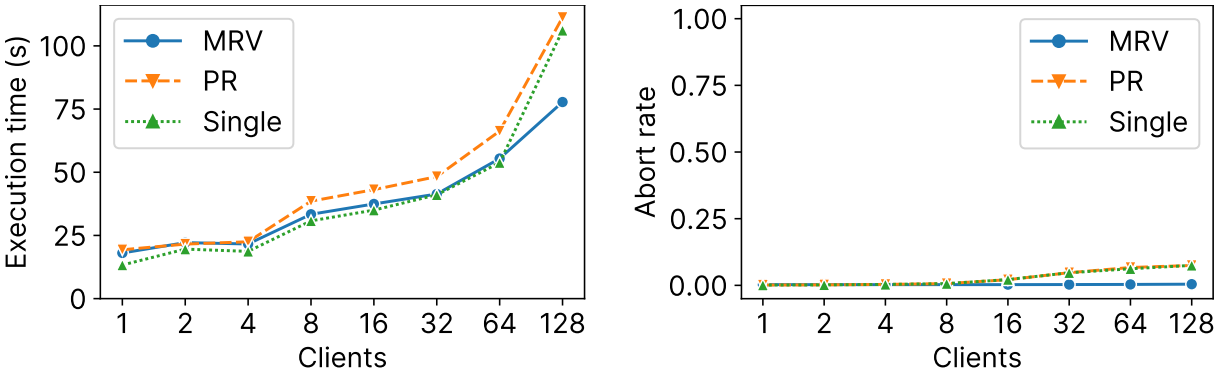
Looking at the abort rate line plot (Fig. 24c), the advantage that **PR** had on our previous write-exclusive scenarios has disappeared. It is able to achieve a close-to-zero abort rate with a 0 % read percentage, as

expected, but it quickly ends up following the abort rate curve for the *Single* value on percentages higher than 10 %. **MRV** is able to keep a lower abort rate in write-dominated scenarios, eventually following the other two implementations in high read workloads, as they all default to a single value.

In the end, **MRVs** are the all-round better option of the three implementations we presented, notably on write-dominated scenarios, which were our initial evaluation target. On these workloads, **MRVs** offer the best performance by a noticeable margin, while also offering performance identical to *Single* on read-heavy workloads, which is already ideal.

5.3.2 STAMP Vacation

We will start our Vacation analysis with the low contention workload (Figure 25), with varying numbers of clients.



(a) Execution time.

(b) Abort rate.

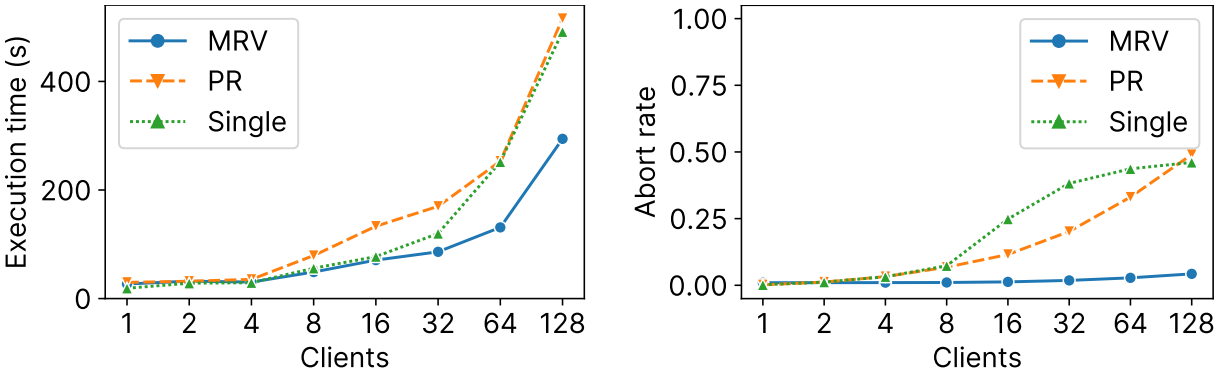
Figure 25: Vacation (low contention) results.

Considering the execution time results on Figure 26a, *Single* provides the best overall performance by a slight margin. Since we are dealing with a low contention workload, it is normal for the value-splitting techniques to not offer an improvement when compared to a plain value. However, we can observe the higher inherent contention of having 128 clients having an effect on the results, where *Single* has a near-double increase in execution time and **MRV** is able to take the lead. **PR** stays behind for all client counts, even if only so slightly.

Regarding the abort rates (Figure 26b), they are, in general, quite low. **PR** and *Single* have a slight increase starting at 16 clients, while **MRV** is constantly near zero throughout all tests.

For the high contention workload (Figure 26), we can see a similar behaviour as of the last test, only with the contention being noticeable starting at lower client counts.

If on the previous test *Single* only fell back (performance wise) with 128 clients, this time that happens



(a) Execution time.

(b) Abort rate.

Figure 26: Vacation (high contention) results.

starting at 32 clients. **MRV** is able to maintain notably better performance on these higher client counts. The abort rate graph (Figure 26b) provides part of the explanation as to why. We can observe that **MRVs** keep a quite low near-constant rate, while the other two techniques rise starting at eight clients.

In sum, while the **MRV** performance benefit is not as clear as on the microbenchmark, there is still a clear advantage in using it on higher contention scenarios. **PR**, largely due to its sluggish phase worker, is not able to keep up with the phase transitions in time and offer any kind of performance improvement over the single value, in this particular benchmark.

5.4 Energy consumption

We measure efficiency with the following formula:

$$\text{efficiency} = \text{operations}/\text{energy}$$

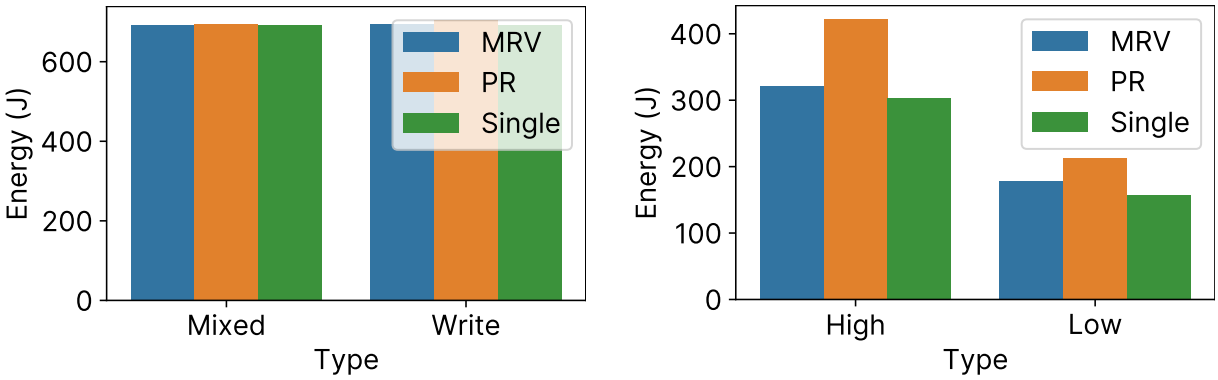
In this context, *operations* is the count of successfully committed transactions and *energy* is a measurement in joule of the total energy consumption.

The energy consumption statistics were collected using *turbostat*, a Linux utility for checking processor statistics. *Turbostat* measures the energy consumption for the whole program, thus it also includes the energy wasted on benchmark setup and teardown. Since all the tests perform this work and we verified that they do not take a significant amount of time to perform it, we consider it as a constant and it does not affect the overall results.

We considered four specific scenarios, using the same parameters as the previous tests. For the microbenchmark, we tested with a write-only workload (*Write*) and a 50 % read scenario (*Mixed*). For the Vacation benchmark, we tested with the two default configurations (*High* and *Low*). Results are presented

on Figures 27, 28 and 29, for energy consumption, performance and efficiency, respectively. Performance results are again included here for reference purposes, as we are using different hardware.

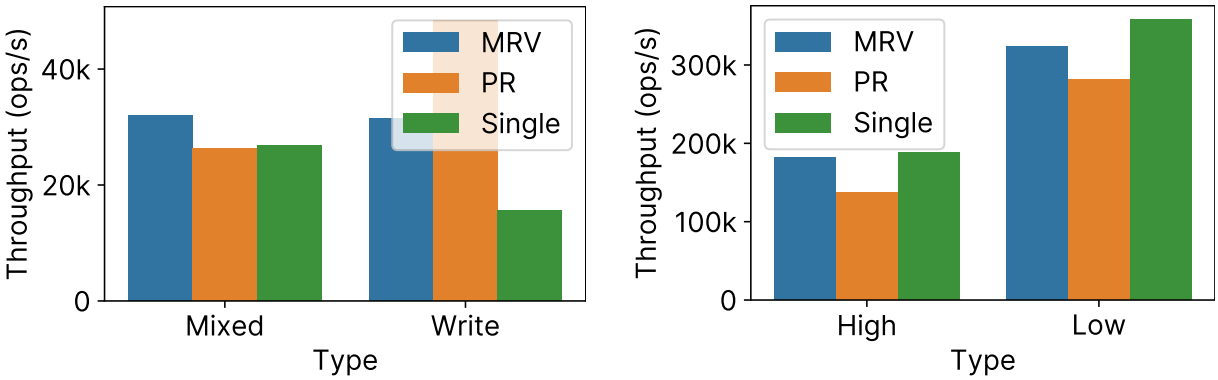
Up until now, every Vacation test presented elapsed time as the main performance metric, as it is how the benchmark was originally designed. However, to make it in line with the microbenchmark results, we have opted to show throughput instead (Figure 28b).



(a) Microbenchmark.

(b) Vacation.

Figure 27: Energy consumption results.



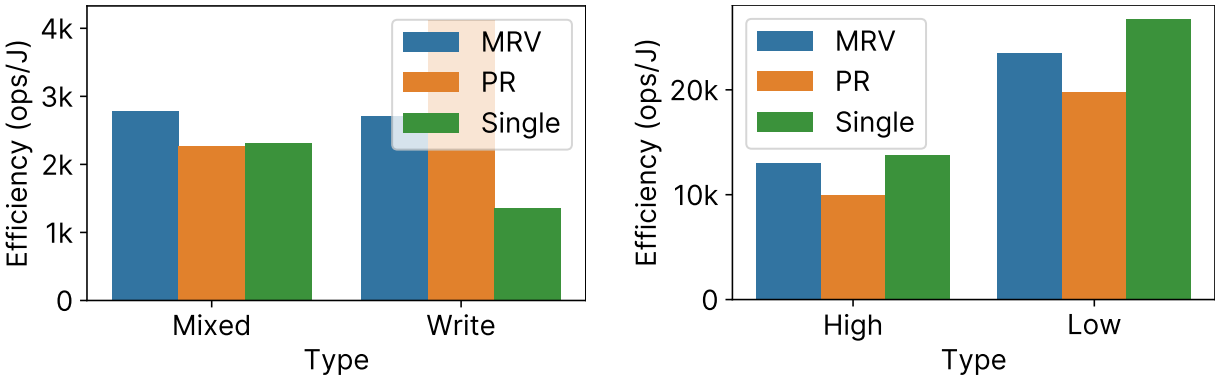
(a) Microbenchmark.

(b) Vacation.

Figure 28: Performance results.

In the microbenchmark (Figure 29a), **PR** offers the best efficiency in the write-exclusive scenario. Despite this, and as we have noted in the previous section, **PR**'s performance drops as reads are introduced, which is reflected in its mixed workload efficiency. On the other hand, **MRVs** maintain a consistent level of efficiency in the two scenarios, surpassing the single value in both. As we can observe from Figure 27a, all implementations consume a similar amount of energy.

In Vacation (Figure 29b), we can observe similar behaviour on the two configurations. The single value offers the best efficiency, with **MRVs** and **PR** taking the 2nd and 3rd places, respectively. In terms of



(a) Microbenchmark.

(b) Vacation.

Figure 29: Energy efficiency results.

energy (Figure 27b), **PR** is the implementation with the highest level of consumption, with **MRV** offering a similar amount of spent energy to the single value. It is clear, for any of the implementations, that the higher contention setting leads to a higher amount of energy consumption.

Overall, we can conclude that **PR** is the worst implementation in terms of energy efficiency. **MRV** is able to perform better than the single value in both microbenchmark runs, only falling behind on Vacation.

Chapter 6

Conclusions and Future Work

In this work, we presented the implementation of two value-splitting techniques in **TM**, based on previous research for database systems. The final result is a C++ library integrated in the Wyatt-STM system, with fully-working value-splitting objects and optimised background workers. We tested several approaches to implementation in **TM** and determined which strategies were feasible and which were not, as the original targets for value-splitting worked in a fundamentally different manner. Moreover, we built two different benchmarks to evaluate our system: a microbenchmark, purposefully made to highlight the problem our optimisation aims to solve, and an adapted Vacation, based on the original reference application from the **STAMP** benchmark suite.

6.1 Conclusions

Experimental results show that **PR** is hindered in most contexts due to static thread allocation and high maintenance requirements. The former prohibits any kind of dynamic workload, where the number of threads is either unknown at the startup of the application or it changes during the execution. The latter deters the usage of large amounts of value-splitting objects, since the delay on phase transitions harms normal operation. While we presented cases where the results weighted heavily in favour of **PR**, such as the write-exclusive scenarios of the microbenchmark, we also consider them as not truly indicative of real-world workloads, rendering their applicability limited.

MRVs, on the other hand, show that their biggest strength lies in their adaptability, with great performance on write-dominated workloads. Despite suffering similar setbacks in regard to worker timings, they are to a lesser degree and they do not have such a significant impact on performance, since **MRV** operations are able to advance no matter the state of the **MRV**. This is in contrast with **PR**, where the phase an object is in can limit the operations that are performed on it.

In regard to energy consumption, it is not so clear that value-splitting is as beneficial for efficiency as

it is for performance. Despite the performance improvements in high contention scenarios, as shown in the presented results, the usage of value-splitting techniques incurs in additional processing costs, due to the need for background workers.

Overall, our results show that value-splitting is worth exploring in the context of **TM** systems. The analysis of these types of write-heavy workloads is of relevance, since they can be easily found in the online sale of highly contended items, e.g., limited-edition releases or concert ticket sales for popular artists.

6.2 Prospect for future work

In future work, we believe that further adjustment of the value-splitting parameters and the usage of different underlying structures could have a significant impact on the performance achieved. We used *immer* in this work as its immutability aspect streamlined the overall implementation, but we believe that data structures made with **TM** in mind could be beneficial. Workloads with dynamic amounts of contention over time could also prove useful in emphasising the strengths of **MRVs** over **PR**.

In this work, we have only focused on simple integer values for our splitting techniques, but we could also apply some of our insights to more complex data structures. The original **PR** proposal is already applicable to ordered tuples and top-K sets, which could be an interesting addition to value-splitting in **TM**.

Besides Wyatt-STM, value-splitting techniques could be applied to other **TM** systems, specifically ones with different implementation types. Word-based systems could have great performance benefits, due to their inherent fine-grained method of operation.

The rise of **Persistent Memory (PM)** devices sparked up new interest on **TM**, with the development of **Persistent Transactional Memory (PTM)** (also known as **Durable Transactional Memory (DTM)**) systems. While **PTM** takes in consideration many of the ideas behind **TM**, the durability aspect of **PM** creates other factors of concern, making the usage of standard **TM** implementations unfeasible. Despite this, we do not think that there are any major issues with the value-splitting techniques that could cause problems if implemented on **PTM** systems; testing their viability on **PM** could be an interesting endeavour for future research.

Bibliography

AMD. Advanced Synchronization Facility: Proposed Architectural Specification. Technical report, AMD, 2009.

C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005. doi: 10.1109/HPCA.2005.41. ISSN: 2378-203X.

Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. Profiling Transactional Memory Applications. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 11–20, February 2009. doi: 10.1109/PDP.2009.35. ISSN: 2377-5750.

Arm. A-Profile Architecture. URL https://developer.arm.com/Architectures/A-Profile%20Architecture.

David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software*, 47(4):1–32, December 2021. ISSN 0098-3500, 1557-7295. doi: 10.1145/3460772. URL https://dl.acm.org/doi/10.1145/3460772.

T. Editors of Encyclopaedia Britannica. Moore's law. URL https://www.britannica.com/technology/Moores-law.

Len Brown. turbostat - Debian Manpages. URL https://manpages.debian.org/testing/linux-cpupower/turbostat.8.en.html.

Clojure. Refs and Transactions. URL https://clojure.org/reference/refs.

Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815409. URL https://doi.org/10.1145/2815400.2815409.

Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 336–346, New York, NY, USA, October 2006. Association for Computing Machinery. ISBN 978-1-59593-451-2. doi: 10.1145/1168857.1168900. URL https://doi.org/10.1145/1168857.1168900.

João P. L. de Carvalho, Guido Araujo, and Alexandro Baldassin. The Case for Phase-Based Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):459–472, February 2019. ISSN 1558-2183. doi: 10.1109/TPDS.2018.2861712. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

Nuno Diegues and Paolo Romano. Bumper: Sheltering distributed transactions from conflicts. *Future Generation Computer Systems*, 51:20–35, October 2015. ISSN 0167-739X. doi: 10.1016/j.future.2015.04.002. URL https://www.sciencedirect.com/science/article/pii/S0167739X15000941.

Nuno Faria and José Pereira. MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting. *Proceedings of the ACM on Management of Data*, 1(1):1–27, May 2023. ISSN 2836-6573. doi: 10.1145/3588723. URL https://dl.acm.org/doi/10.1145/3588723.

Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488, May 2020. doi: 10.1109/CCGrid49817.2020.00-45.

FSF. GNU libitm. URL https://gcc.gnu.org/onlinedocs/libitm/index.html.

Justin E. Gottschlich, Maurice P. Herlihy, Gilles A. Pokam, and Jeremy G. Siek. Visualizing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 159–170, New York, NY, USA, September 2012. Association for Computing Machinery. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370842. URL https://doi.org/10.1145/2370816.2370842.

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, June 1982. ISSN 0362-1340. doi: 10.1145/872726.806987. URL https://doi.org/10.1145/872726.806987.

Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of Seventh International*

Conference on Very Large Databases, volume 81, pages 144–154. Tandem Computers Incorporated, 1981.

Brendan Gregg. Flame Graphs, 2011. URL https://www.brendangregg.com/flamegraphs.html.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL https://doi.org/10.1145/289.291.

Brett Hall. Wyatt-STM. URL https://github.com/bretthall/Wyatt-STM. original-date: 2015-12-18T17:34:53Z.

Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Springer International Publishing, Cham, 2010. ISBN 978-3-031-00600-5 978-3-031-01728-5. doi: 10.1007/978-3-031-01728-5. URL https://link.springer.com/10.1007/978-3-031-01728-5.

HaskellWiki. Software transactional memory. URL https://wiki.haskell.org/Software_transactional_memory.

Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, February 2008. Association for Computing Machinery. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345237. URL https://doi.org/10.1145/1345206.1345237.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, May 1993. ISSN 0163-5964. doi: 10.1145/173682.165164. URL https://doi.org/10.1145/173682.165164.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, July 2003. Association for Computing Machinery. ISBN 978-1-58113-708-8. doi: 10.1145/872035.872048. URL https://doi.org/10.1145/872035.872048.

Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Notices*, 41(10):253–262, October 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167495. URL https://doi.org/10.1145/1167515.1167495.

Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1–16, New York, NY, USA, April 2016. Association for Computing Machinery. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901348. URL https://doi.org/10.1145/2901318.2901348.

Hubblo. Scaphandre, August 2023. URL https://github.com/hubblo-org/scaphandre. original-date: 2020-10-16T14:10:05Z.

Intel. Intel® Power Gadget. URL https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html.

Intel. Desktop 4th Generation Intel® Core™ Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family - Specification Update. Technical report, Intel, 2015.

Intel. Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue. White paper, Intel, 2021.

ISO. Technical Specification for C++ Extensions for Transactional Memory. Technical specification, ISO, 2015.

Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 151–160, New York, NY, USA, August 1994. Association for Computing Machinery. ISBN 978-0-89791-654-7. doi: 10.1145/197917.198079. URL https://doi.org/10.1145/197917.198079.

Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, and Michael Spear. Transactional Tools for the Third Decade. In *Proceedings of the 10th ACM SIGPLAN Workshop on Transactional Computing*, page 8, 2015.

Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice*

of parallel programming, PPoPP '06, pages 209–220, New York, NY, USA, March 2006. Association for Computing Machinery. ISBN 978-1-59593-189-4. doi: 10.1145/1122971.1123003. URL https://doi.org/10.1145/1122971.1123003.

Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. In *Workshop on Transactional Computing (Transact)*, page 11, 2007.

John Levon. OProfile - A System Profiler for Linux. URL https://oprofile.sourceforge.io/.

Berenice Mann. Arm releases SVE2 and TME for A-profile architecture, April 2019. URL https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/new-technologies-for-the-arm-a-profile-architecture.

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008. doi: 10.1109/IISWC.2008.4636089.

Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, OSDI'14, pages 511–524, USA, October 2014. USENIX Association. ISBN 978-1-931971-16-4.

NVIDIA. NVIDIA System Management Interface, June 2012. URL https://developer.nvidia.com/nvidia-system-management-interface.

Juan Pedro Bolívar Puente. Persistence for the masses: RRB-vectors in a systems language. *Proceedings of the ACM on Programming Languages*, 1(ICFP):16:1–16:28, August 2017. doi: 10.1145/3110260. URL https://dl.acm.org/doi/10.1145/3110260.

James Reinders. Transactional Synchronization in Haswell, 2012. URL https://web.archive.org/web/20120413203120/http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, August 1995. Association for Computing Machinery. ISBN 978-0-89791-710-0. doi: 10.1145/224964.224987. URL https://doi.org/10.1145/224964.224987.

Michael Spear, Hans Boehm, Victor Luchangco, Jens Maurer, Michael L. Scott, and Michael Wong. Transactional Memory Lite Support in C++. Technical specification, ISO, 2021. URL https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1875r2.pdf.

The Linux Foundation. Perf Wiki. URL https://perf.wiki.kernel.org/index.php/Main_Page.

Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael Spear. Simplifying Transactional Memory Support in C++. *ACM Transactions on Architecture and Code Optimization*, 16(3): 25:1–25:24, July 2019. ISSN 1544-3566. doi: 10.1145/3328796. URL https://doi.org/10.1145/3328796.

Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. Discovering and understanding performance bottlenecks in transactional applications. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 285–294, September 2010.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.