Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Crashresistant Transactional Behavior in C++

Jochen Benzenhöfer

Course of Study:	Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Lukas Epple, M.Sc.
Commenced:	February 3, 2023
Completed:	August 3, 2023

Abstract

In this paper, we present a crash-resistant C++ library for executing unrestricted transactions, provided as lambdas. Our implementation bridges the gap between existing crash-resistant and unrestricted transaction implementations in C++. We first developed a non-crash-resistant version to enable the execution of lambda transactions. To achieve crash resistance, we utilized a clang plugin during compilation to modify the source file and stored critical runtime variables in a file. While our approach is approximately three times slower than transaction implementations without lambdas, the benefits of full crash resistance make it a compelling trade-off.

Kurzfassung

In diesem Artikel präsentieren wir eine absturzsichere C++-Bibliothek zur Ausführung uneingeschränkter Transaktionen, die als Lambdas bereitgestellt werden. Unsere Implementierung überbrückt die Lücke zwischen bestehenden absturzsicheren und uneingeschränkten Transaktionsimplementierungen in C++. Zunächst haben wir eine nicht absturzsichere Version entwickelt, um die Ausführung von Lambda-Transaktionen zu ermöglichen. Um die Absturzsicherheit zu erreichen, haben wir während der Kompilierung einen clang-Plugin verwendet, um die Codedateien zu modifizieren, und kritische Laufzeitvariablen in einer Datei gespeichert. Obwohl unser Ansatz etwa dreimal langsamer ist als Transaktionsimplementierungen ohne Lambdas, machen die Vorteile der vollen Absturzsicherheit dies zu einem überzeugenden Kompromiss.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AST	Abstract Syntax Tree. 19
JIT	just-in-time. 19
NVRAM	non-volatile random-access memory. 20

1 Introduction

Reliability is a critical factor in server infrastructure, and ensuring data integrity and system availability is of utmost importance. However, implementing redundant hardware systems to achieve high reliability can be prohibitively expensive. To address this challenge, the concept of using software-based solutions to prevent data loss has gained traction as an effective approach that can significantly reduce costs associated with computing time and performance. By leveraging software-based solutions, such as the implementation of transactional systems, it becomes possible to maintain data integrity and enable seamless recovery after system failures. For instance, in the event of a power outage, an active process or transaction can continue seamlessly once power is restored, eliminating the need for redundant systems or emergency generators that can incur additional costs and impact compute time. Additionally, by utilizing slower persistent memory to store critical system states, both the recovery process and the restoration of the system to its pre-failure state can be facilitated. Another significant trend in the server space is the emergence of non-volatile memory, which retains data even in the event of a server crash. Although non-volatile memory may not exhibit the same level of speed as traditional memory, advancements in its technology have progressively reduced this performance gap. This shift towards non-volatile memory has also necessitated the development of new algorithms for efficient data storage, offering opportunities to improve the speed at which data, such as logs, can be stored in persistent storage devices. Transactional systems play a vital role in ensuring both reliability and scalability in server infrastructures. Transactions enable the parallel execution of multiple concurrent operations while ensuring correctness and data consistency. In our project, we have developed a C++ library that enables the implementation of custom transactions, offering crash resistance when utilized in conjunction with persistent memory. However, the implementation of recovery mechanisms in C++ posed additional challenges, particularly concerning recreating the code that is executed within each transaction. To address this, we have adopted a recovery mechanism based on the concept of microtransactions, allowing us to recover the system state to the point following the last fully successful microtransaction as seen in [SSS20]. This approach minimizes the amount of recovery information required and reduces the computational overhead in the event of a failure. It is important to note that our implementation, although providing crash resistance, exhibits slower performance compared to a comparable basic implementation. Nonetheless, for safety-critical systems, the assurance of crash resistance may outweigh the performance trade-off, making our library a valuable solution. In summary, our project focuses on the development of a C++ library that offers users the ability to implement their custom transactions while providing crash resistance in the event of a system failure. In the subsequent chapters, we will delve into the preliminary concepts necessary to understand the project, provide detailed insights into our implementation both technically and conceptually, and conclude with a comprehensive evaluation of our library through benchmarking.

2 Preliminaries

2.1 Transaction

In the field of databases, Transactions embody the fundamental principles known as ACID: atomicity, consistency, isolation, and durability[ISO98][WSSZ07].

Atomicity

In the realm of transactions, there is no room for half-measures or indecisiveness. A transaction is either executed in its entirety, ensuring a successful outcome, or it is aborted, leaving no trace of its partial execution. It exhibits a binary nature, guaranteeing a decisive and conclusive operation[94].

Consistency

Within the transaction's domain, maintaining the integrity and coherence of the database is paramount. While it is essential to acknowledge that erroneous inputs can jeopardize semantic consistency, the transaction acts as a guardian, diligently preserving the fundamental structure and validity of the database[17].

Isolation

Picture a bustling marketplace with various vendors and customers engaged in their transactions. Similarly, in a concurrent execution environment, multiple transactions operate independently, shielded from the interference of others. This isolation prevents conflicts and ensures that each transaction can fulfill its purpose without disrupting the integrity of the system[LK08].

Durability

The durability aspect of a transaction serves as an insurance policy against unforeseen mishaps. Regardless of potential system crashes or failures, a successfully committed transaction stands as an unwavering testament to its durability. The changes it introduces persist and become permanent, safeguarded from any adverse events[ZHMS06].

Serialization

In the quest for consistency, the execution order of operations is of utmost importance. Serialization addresses this concern by establishing a consistent order of operations, regardless of whether they occur in parallel or sequential execution. It ensures that the result of a program remains the same, regardless of the chosen execution strategy[Pap79].

Linearization

Within the parallel execution landscape, maintaining the original order of operations is crucial. A transaction adheres to the principle of linearization, guaranteeing that if operation A completes before operation B begins in a parallel scenario, the same order will be preserved in sequential execution. This preserves the logical flow and integrity of the program's execution.[HW90]

By abiding by these principles, transactions ensure a reliable and robust foundation for database operations. To achieve these ACID properties [SSS20] provided an interesting concept: The transaction is split up into micro transactions. To understand why this concept can be useful we have to take a look on how to achieve the atomicity. We have three possible options(and some combinations of those): Option one: We only use instructions that can be run on hardware atomic. On current instruction sets the options for such instructions are quite big, but using only one of those is mostly not enough for transactions. So we need one of the other options. Option two: using a redo log: We store a copy of the current state of the system. In case of a failure we restore the old state with our saved copy. This provides the problem that we can not easily run multiple transactions at the same time because if one transaction fails and one succeed before the fail but was started after creating the copy we restore the copy before the successful transaction and we need to redo the successful transaction additionally to the failed one. To deal with those outdated copies there is option three: using a undo log. In the undo log we write down what sub steps we already run. In case of an failure we can then run the inverse operations of the sub steps in the reverse order to get back to the original state and try the transaction again as if nothing happened. But the sub steps need a inverse function to succeed. We rely on the undo log in our implementation, but do not rely on the user to input an inverse function to every sub step, instead we only require those sub steps to be idempotent. An idempotent function is a function that run on its output produces the same output as if run on its input mathematically: Let f be an function and $D(f)$ be the Domain of f. f is idempotent if $f(x) = f(f(x))\ \forall x, f(x) \in D(f)$. What that means in our case is: We can run the function as often as we want on the output and we always get the same result. That way we only need to know which sub step succeeded completely at the failed point and redo the following sub steps to run the transaction completely.

2.2 Programming Language

To understand why we implemented the transactions in c++ we have to take a look at the difference between java and c++:

Java

Java is a dynamically compiled language, which utilizes a just-in-time (JIT) compiler. This means that the compilation process occurs at runtime, precisely when the code is needed for execution. The JIT compiler, as the name suggests, compiles the source code on the fly, right before it is executed. Consequently, the source code must be provided just before running the program. One advantage of this approach is that it allows for runtime modifications and the ability to access the source code, as long as the specific statement being executed is not currently in progress. While this dynamic compilation process facilitates easy identification of the next statement to execute, it can also impact program efficiency. Since the code is translated during execution, it introduces a certain level of overhead, potentially slowing down the program. In summary, Java, as dynamically compiled language, employ JIT compilation, which enables on-the-spot compilation of the source code when it is required for execution. While this approach allows for flexibility and runtime modifications, it can lead to decreased program efficiency due to the additional overhead introduced by the compilation process during execution.[CFM+97]

Abstract Syntax Tree (AST)

An AST serves as a crucial intermediary step for compilers in transforming human-readable source code into machine-readable byte code. The source code undergoes parsing, where it is analyzed and broken down into tokens. Each token represents a distinct node within the AST. By arranging these tokens in the appropriate structure, meaningful statements and code can be constructed. Eventually, the AST is converted into machine code, resulting in the creation of byte code.[Cla23]

Clang

Clang, a popular compiler framework, offers various approaches to access and manipulate the AST. One method involves creating a tool that generates the AST when executed. Once the AST is obtained, it can be modified, including both the AST itself and the corresponding source code. The modified source code can then be stored for further use. However, it is important to note that the tool does not provide a direct way to compile the program within the tool itself.

C++

C++ is classified as a statically compiled language, meaning that the source code is compiled before the program's execution. Consequently, during runtime when the program is executed, we cannot directly search for and execute specific statements, as the source code must be compiled beforehand. Instead, an alternative approach involves inspecting the compiled binary code to locate and execute a desired statement. However, this method presents challenges as the compiler may optimize or alter statements, and the binary code itself is not easily readable by humans. To overcome these challenges and enable the execution of specific statements at runtime, an alternative strategy is employed: modifying the source file during compilation. This is achieved through the use of Clang compiler plugins, which are code segments executed after the compiler has constructed the AST, but before generating the final output. By potentially modifying the AST, it becomes theoretically possible to generate modified binaries based on the altered AST. However, it is important to note that modifying the AST at every statement is not universally supported by the Clang compiler. For instance, while certain modifications like deleting function arguments are permissible, adding new arguments is not feasible due to the memory layout constraints of the Clang AST. As a result, the chosen approach involves editing the source file itself and restarting the compiler after making the necessary modifications. In summary, C++ is a statically compiled language, necessitating the compilation of source code before execution. To execute specific statements at runtime, the source file can be modified during compilation using compiler plugins. However, the limitations of AST modifications within Clang require editing the source file and restarting the compilation process.

Clang Plugin

In our project, we opted to utilize a Clang plugin. The advantage of using a plugin is that it grants us access to the compiler by intercepting its execution. The Clang plugin is incorporated into the compilation process and runs after the AST has been parsed. This provides an opportunity to potentially modify the AST, and in theory, compile the modified AST. However, it should be noted that modifying the AST itself can be a complex and sometimes even impossible task. To overcome these limitations, we chose to modify the source code, as previously explained. By leveraging the plugin's ability to interact with the current compiler, we were able to replicate the existing settings and restart the compilation process from the beginning. Theoretically, this approach eliminates the need for a second compiler invocation. However, in practice, we encountered challenges in achieving the correct recompilation, possibly due to compatibility issues with the newer version of Clang being used. In summary, Clang provides different avenues for accessing and manipulating the AST, such as through the creation of tools or the utilization of plugins. While plugins offer the advantage of interacting with the compiler itself, modifying the AST directly can be complex. As a workaround, modifying the source code and restarting the compilation process was pursued as archived in [Nic22]. However, challenges were encountered in achieving proper recompilation, potentially influenced by compatibility issues with the newer Clang version employed.[Cla23]

To conclude we used C++ for this project since the implementation in java should be quite trivial.

2.2.1 non-volatile random-access memory (NVRAM)

NVRAM is capable of retaining data even when the power is turned off. There are two main types of NVRAM used for storing a computer's system state: static random access memory (SRAM) and electrically erasable programmable read-only memory (EEPROM). SRAM uses a battery for non-volatility, used for example as BIOS chips in PCs, while EEPROM relies on floating gate transistors. Both SRAM and EEPROM require higher power to operate compared to dynamic random access memory (DRAM), which is commonly used in standard PCs as ”RAM”. Another type of NVRAM is NAND flash, which offers greater storage density and lower cost. NVDIMMs are a combination of NVRAM and flash memory, designed to fit into DIMM slots on a computer's motherboard. Looking to the future, potential NVRAM types include ferroelectric RAM (FRAM), magnetoresistive RAM (MRAM), and phase-change memory (PCM). FRAM is suited for specific industrial applications due to its durable nature, while MRAM shows promise for higher storage density. PCM changes material state with electric current, providing fast read and write times.[Bro18] Although our approach is designed to run on NVRAM, we did not test it on actual NVRAM devices due to the ongoing trend of NVRAM becoming faster, aiming to achieve performance levels similar to DRAM while remaining non-volatile. So a use of today's DRAM should be a good simulation of NVRAM in the future.

3 Related Work

This chapter is about related work we used and how our work improves on those projects.

Transactions on Red-black and AVL trees in NVRAM [SSS20]

This paper by Schütt, Thorsten and Schintke, Florian and Skrzypczak, Jan is about the need for a generic transaction mechanism to update complex data structures in-place with constant memory overhead in byte-addressable NVRAM to support distributed, parallel, and cluster computing. Byte-addressable NVRAM requires transactional updates for complex data structures to ensure recoverability. Traditional database technologies like logs and journals have significant space overhead and induce non-trivial performance costs, making them less suitable for fine-grained, byte-addressable NVRAM. The presented approach offers a generic transaction mechanism with constant memory overhead for updating complex data structures like Red-Black Trees and AVL Trees in local and remote NVRAM while ensuring durable linearizability for multi-reader single-writer access. They implemented their project in c++ so we were able to use the same principles. We also adepted the idea of the micro transactions and used the red and black tree implementation as a example for transactions. In our paper we implemented a way to use lambdas as micro transactions. Thereby we allow for a broughter spectrum of transactions to be run than just tree operations.

Transactional memory in c++

The C++ Transactional Memory Technical Specification [JTC15] presents a proposal for standardizing language constructs for transactional memory in C++. The main goals of the Transactional Memory Technical Specification are to provide minimal performance overhead for non-transactional programs and to offer improved safety compared to custom transactional memory implementations. The Transactional Memory Technical Specification introduces two types of transaction blocks: ”atomic” and ”synchronized”. These transaction blocks have subtle differences in their capabilities and rollback behavior. ”Atomic” blocks contain code explicitly intended to be a transaction and must pass static checks to ensure they can safely abort and roll back at any time. In contrast, ”synchronized” blocks can perform I/O and system calls and are allowed to transition to a serial mode when attempting irrevocable operations, ensuring smoother execution. To support separate compilation and provide speculation safety, the Transactional Memory Technical Specification utilizes ”transaction-safe” function types. These function types enable static checking for irrevocable operations within the code reachable from an atomic block, offering a strong benefit compared to synchronized blocks. The Transactional Memory Technical Specification implementation in GCC duplicates transaction-safe functions and instruments the clones to ensure correct software transactional memory support without impacting the performance of non-transactional code[ZZB+19].

Despite its merits, the Transactional Memory Technical Specification remains a technical specification and has not achieved widespread adoption by compilers. Additionally, programmers are limited in their use of specific transaction-safe functions, which restricts the full potential of transactional memory support in C++ applications.

Simplifying Transactional Memory Support in C++ [ZZB+19]

This related approach aims to enable Transactional Memory support in C++ through an LLVM compiler plugin. This plugin performs transformations on the code containing lambdas that request transactional execution. By analyzing and transforming the lambdas, the plugin makes them compatible with both Hardware Transactional Memory and Software Transactional Memory execution environments. However, both of these approaches lack a crucial aspect addressed by our proposed implementation. In the existing methods, the code executed within transactions is not saved or persisted. This limitation results in a lack of fault tolerance, as the code cannot be recovered in case of system restarts or failures, leading to potential data inconsistencies. Our proposed approach builds upon the lambda-based transactional execution, similar to the first approach [Reference 1]. However, our implementation goes a step further by introducing code persistence. This means that the code executed within transactions is saved, enabling recovery in the event of system restarts or failures. With code persistence, our approach ensures that transactional operations can be resumed from the point of interruption in the event of system restarts or failures. This improvement enhances the fault tolerance of the application and minimizes the risk of data inconsistencies. The code recovery feature reduces the need for manual checkpointing, providing a more reliable and streamlined programming experience for developers. Our proposed approach, combining lambda-based transactional execution with code recovery, presents a significant enhancement to existing methods for Transactional Memory support in C++. By addressing the limitation of code persistence, we aim to improve fault tolerance, reliability, and overall programming convenience.

4 Methology and Implementation

This section is about the details of the implementation. A overview about the used classes can be seen in Figure 4.1. This graphic shows how the classes are included by each other. The only File not in the graphic is the file of the Plugin. This File does not include any of the other files. But the plugin interacts with the other files by editing the main function and the transaction manager file to allow for crash resistant behavior. In the following sections we will take a closer look at the implemented classes.

4.1 CrashResistTransactions

This section presents the implementation of a Clang plugin that adds essential functionality to the project, allowing crash recovery in case of unexpected failures. The plugin is designed to ensure data integrity and system reliability, providing an effective solution without relying on costly redundant hardware.

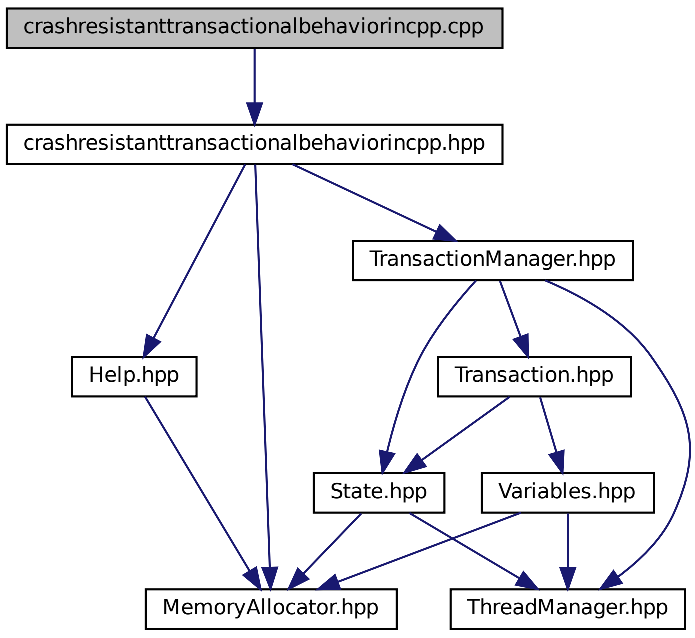


Figure 4.1: Overview of the implemented classes

Listing 4.1 The Cmake additions to use the plugin

```
add_compile_options(
  "SHELL:-Xclang -load"
  "SHELL:-Xclang [path to CrashresistTransactions.so]"
  "SHELL:-Xclang -add-plugin"
  "SHELL:-Xclang editFunctions"
)
```

4.1.1 Plugin Integration

To incorporate the plugin into a C++ project, developers need to load it into the project's build process. This can be achieved by adding the following command line argument when running the Clang compiler: `-load [path to CrashresistTransactions.so] -add-plugin editFunctions` It is important to note that the current implementation utilizes the cc1 part of the compiler and is not directly compatible with Cmake. To use the plugin in a CMakeLists.txt file, developers must add specific lines, as shown in Listing 4.1, to ensure successful integration.

4.1.2 CrashresistTransactionsAction

The Clang plugin's entry point is the CrashresistTransactionsAction class, which extends the PluginASTAction class. Upon invocation, this class triggers the functionality of the Crashresist-TransactionsConsumer class, responsible for handling the core operations of the plugin. In this class, we additionally attempted to restart the compiler on the edited files without the plugin after completing the rest of the plugin's functionality as done by [Nic22]. However, we encountered errors during this process, presumably due to our clang version, and we were unable to achieve the desired restart. The aim of restarting the compiler without the plugin was to ensure that the modifications made by the plugin were successfully integrated into the compiled application.

4.1.3 CrashresistTransactionsConsumer

The CrashresistTransactionsConsumer class effectively traverses the C++ AST using the ReadVisitor class, an implementation of RecursiveASTVisitor, gathering essential information required for crash recovery. This includes identifying callers of the TransactionManager class, to calculate the size of the State type. As well as analyzing calls to the createVector function, to extract and create lambdas necessary for correct restoration of the source code in case of crashes. Additionally, the class evaluates the number and sizes of local and global variables to approximate the overall size of crash-resistant memory needed using the Memory Layout shown in Figure 4.3. When creating the main storage file, we adopt a specific approach to indicate whether a crash has occurred previously or if the program has not run yet. We achieve this by setting all memory within the file to the value ”1” as a signal. To facilitate this, we initialize only one of the contained values, which is the ”variables offset” within the main storage file. The rest of the memory is set to ”1”, which serves as a flag to represent an uninitialized state or a lack of previous execution. However, there is a limitation to this approach. The method of setting all memory to ”1” can only be applied to the first 1024 Bytes of the file. Beyond this range, there might be issues with memory alignment and unintended overwriting of critical data. To address this limitation and ensure correct memory alignment while preserving the ”1” initialization signal, we set the maximum length of the file to 1020 Bytes (`1024 - sizeof(void*) + sizeof(int)`). This allows for proper memory alignment while still enabling the entire file to be initialized with the ”1” value. By utilizing this approach, we can maintain a consistent indication of whether the program has previously run or if any crashes have occurred. This is essential for the subsequent reading part, where we use this information to determine the appropriate state to recover to in case of a crash.

4.1.4 ReadVisitor

In the ”ReadVisitor” class, we focus on examining various parts of the code to gather important information for crash recovery. We first search for callers of the ”TransactionManager” class and analyze the first provided template argument to determine the size of the State type. Additionally, we inspect all calls to the ”createVector” function to extract lambdas from the arguments. By creating new lambdas, similar to the ”createTransaction” function in the transaction class, we enable the recovery of the code to run in case of a crash, overcoming the limitation of the zero storage policy of C++ lambdas. However, we currently encounter challenges when working with transactions input to the ”createVector” function as a transaction variable. We were not able to find the code nested within these transactions, which makes it difficult to recover them in case of a crash. Though there might be a possibility to search for the place of creation of those transaction variables, this feature has not been implemented yet. Furthermore, we search for calls to the ”setNumberOfThreads” function to evaluate the maximum number of threads. By monitoring calls to ”resizeLocalVariables” and ”resizeVariables”, we determine the maximum number of local and global variables in the program. Additionally, we analyze the first template argument of the calls to functions like ”createTransactionWithReturn”, ”setVariable”, and ”createTransaction” (for functions with two arguments and a return value) to ascertain the sizes of the variables set in the program. To handle the approximation of variable sizes, we save only the biggest size. This is often still not enought space, considering that many variables contain pointers, which are only accounted for as sizeof(void*) in our calculations.

4.1.5 WriteVisitor

In the ”WriteVisitor” class, we visit various points of interest within the source code to make actual edits and modifications. We have gathered the necessary information during the visitation process in the ”ReadVisitor” class, now we leverage this data to make the required changes to enable crash recovery. Specifically, we search for the ”TransactionManager” constructor. This is a crucial step because we need to open the crash-resistant file and add the state and variables to it once they are initialized. The crash-resistant file is responsible for storing the essential information that allows the program to recover in case of a crash. To open the crash-resistant file and map it to the local memory, we need to import specific files as shown in Listing 4.2. By importing these necessary components, we can efficiently manage the crash-resistant file and handle memory mappings appropriately. During the editing process, we carefully manage certain length variables and create the required variables or state using the Memory Allocator at the correct location, following the layout shown in

Listing 4.2 The files that need to be imported for the crash resistant file to work

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Figure 4.3. Additionally, we read out the template variables of the "initState" function to ensure that we use the correct types in the state. This step is crucial for proper recovery, as the correct data types must be used to accurately restore the program's state.

4.1.6 Recovery Process

Finally, the recovery part of the project. We do this by editing the main function. The recovery process occurs in two main parts. First, we need to recreate the variables and states that were present at the time of the crash. This involves opening and mapping the crash-resistant file, as mentioned earlier, and properly initializing the variables and state(for this we again need to imports from Listing 4.2). An important aspect of this part is the remapping of pointers in the crash-resistant file. As mentioned earlier, the crash-resistant file stores pointers to various data structures. However, after a crash, the memory mappings can change, leading to incorrect pointers. To address this, we remap all pointers in the file to the new mapping. We achieve this by subtracting the old mapping pointer from all pointers and adding the new mapping pointer to ensure proper alignment and access to the correct data. In the second part of the recovery process, we recreate the code needed to run the lambdas. This is done by utilizing the lambdas extracted and edited in the "ReadVisitor" class. We manage the state and run each lambda as required to restore the program to the state it was in before the crash. Once all active transactions have been executed and recovered, we delete the mapping, as it cannot be easily reused in the new Transaction Manager. Subsequently, we proceed to run the main file, effectively completing the crash recovery process.

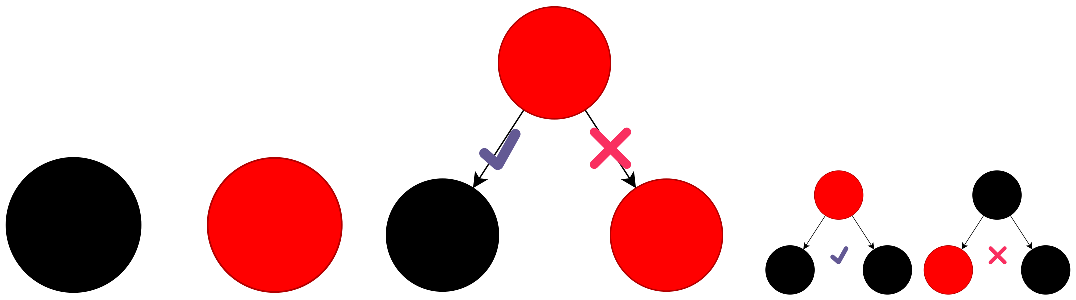
4.2 crashresistanttransactionalbehaviorincpp

This is the current main file containing the main function. We use this file to test the other classes and as an example for use cases. This is the file the user has to implement the lambda functions with the transactions into. But the name of this file does not matter, what matters is the contained main function. We imported the most important files in the crashresistanttransactionalbehaviorincpp.hpp as shown in Figure 4.1 file to allow the user to import only one file instead of all the classes.

4.3 Help

This file presents a basic implementation of a Red-Black Tree, which is a type of balanced binary tree. The Red-Black Tree has specific properties that govern its structure:

1. Each node in the tree is assigned a color: either red or black as shown in Figure 4.2a.
2. All children of red nodes must be black nodes graphically: Figure 4.2b.



(a) Red-Black Trees Rule 1 **(b)** Red-Black Trees Rule 2 **(c)** Red-Black Trees Rule 3

Figure 4.2: Red-Black Trees properties

3. Every path from a given node to its leaf nodes contains an equal number of black nodes as depicted in Figure 4.2c.

If property 2 (coloring of children) or property 3 (number of black nodes) is violated, it is known as a red violation or black violation, respectively. Red-Black Trees are considered half-balanced because the shortest path from a node to its leaf nodes consists only of black nodes. On the other hand, the longest path from a node to its leaf nodes alternates between black and red nodes and is twice the length of the shortest path. To avoid red violations (property 2) between the root node (head) and its children, the head node is always colored black. Additionally, every empty node is also considered black to prevent red violations for leaf nodes [SSS20].
The implementation includes a Node class shown in Figure 4.8a, which holds several public attributes. These attributes include an enumeration representing the color of the current node (either black or red), and two references to the left and right child nodes, denoting the head of the respective subtrees. Each node also contains an integer key and an integer value. As the implementation is recursive, each node can be seen as the head of a subtree. Null pointers represent empty nodes or subtrees. There are two constructors provided in the Node class. The constructor without arguments creates an empty node, while the constructor with arguments sets the key and value for a node, with the default color being black. Additionally, there are three additional functions in the class:

- The ”print” function recursively traverses the subtree, printing the key and value of each node. It firstly prints the left subtree, then the right subtree, and finally the current node to the standard output. The function also returns the number of nodes in the subtree.
- The ”count” function recursively counts the number of nodes in the subtree by counting the nodes in the left and right subtrees and adding one for the head node.
- The ”deleteAll” function recursively deletes the entire subtree rooted at the current node. It deletes the left subtree first, followed by the right subtree, and finally deletes the node itself using the deleteOne operation from the implemented MemoryAllocator.

The included MemoryAllocator is necessary to enable the allocation and deletion of the tree within a given memory range.

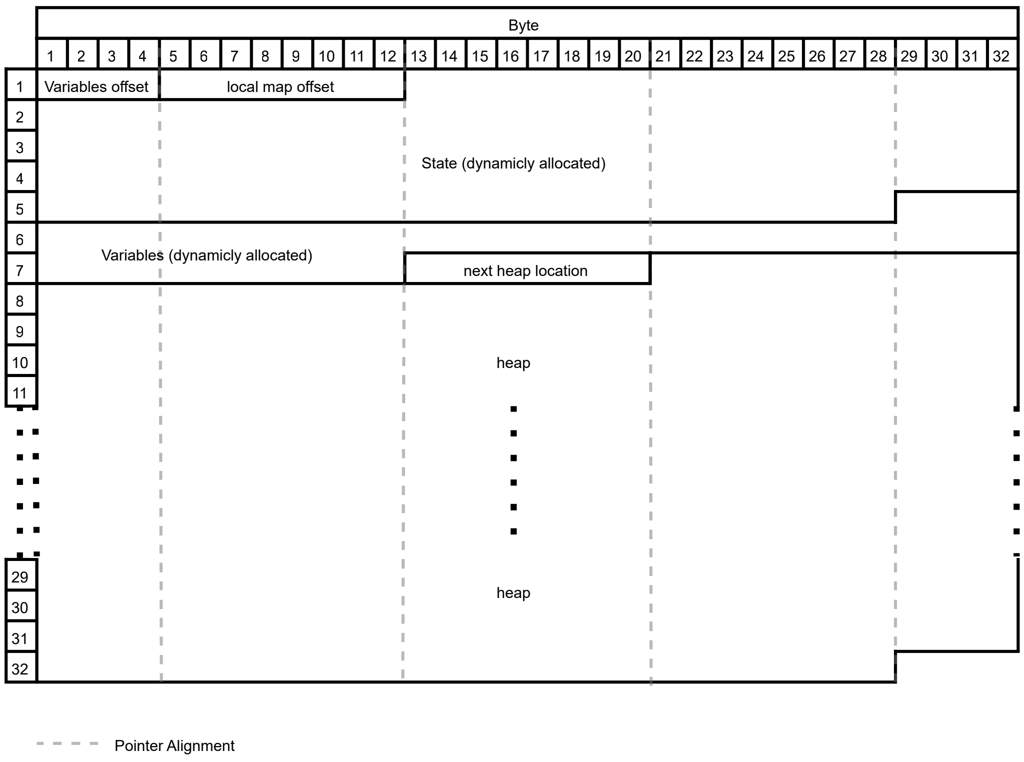


Figure 4.3: Memory Layout of the first file

4.4 MemoryAllocator

The Memory Allocator plays a crucial role in reserving and freeing memory within a specified memory range. An overview of this file is shown in Figure 4.5b. The allocator relies on two global variables: ”start” and ”min”, which determine the highest and lowest byte addresses, respectively, where variables can be stored. By default, the ”start” variable is set as a null pointer, indicating that no memory range is defined. In this case, the default new allocator is used. Similarly, the ”min” variable is initially a null pointer, signifying the absence of an end point for the memory range. The ”checkAvailableSpace” function is responsible for assessing the availability of memory within the current range. If sufficient memory exists, the ”start” address is adjusted accordingly. However, if more memory is required, the function reserves new memory for the application. This is accomplished by creating a shared memory (shm) file using the layout shown in Figure 4.4 with a size of 2MB and mapping it to the local memory using mmap. The majority of the file is then utilized as additional memory, with the exception of the first eight bytes, which represent a pointer to the next reserved memory (if necessary). The allocated memory files and mapped memory ranges are stored as pointers in the ”sharedHeapFiles” and ”mappedHeapFiles” vector variables, respectively, allowing access to the contained memory. When portions of the memory are freed, the addresses are mapped to their corresponding available space and stored to facilitate reuse and optimize memory usage. The main constructor of the Memory Allocator can be utilized with zero to two arguments. If no arguments are provided, the ”start” variable is set to the null pointer (if it was not set previously). If the variables were previously set (e.g., by constructing

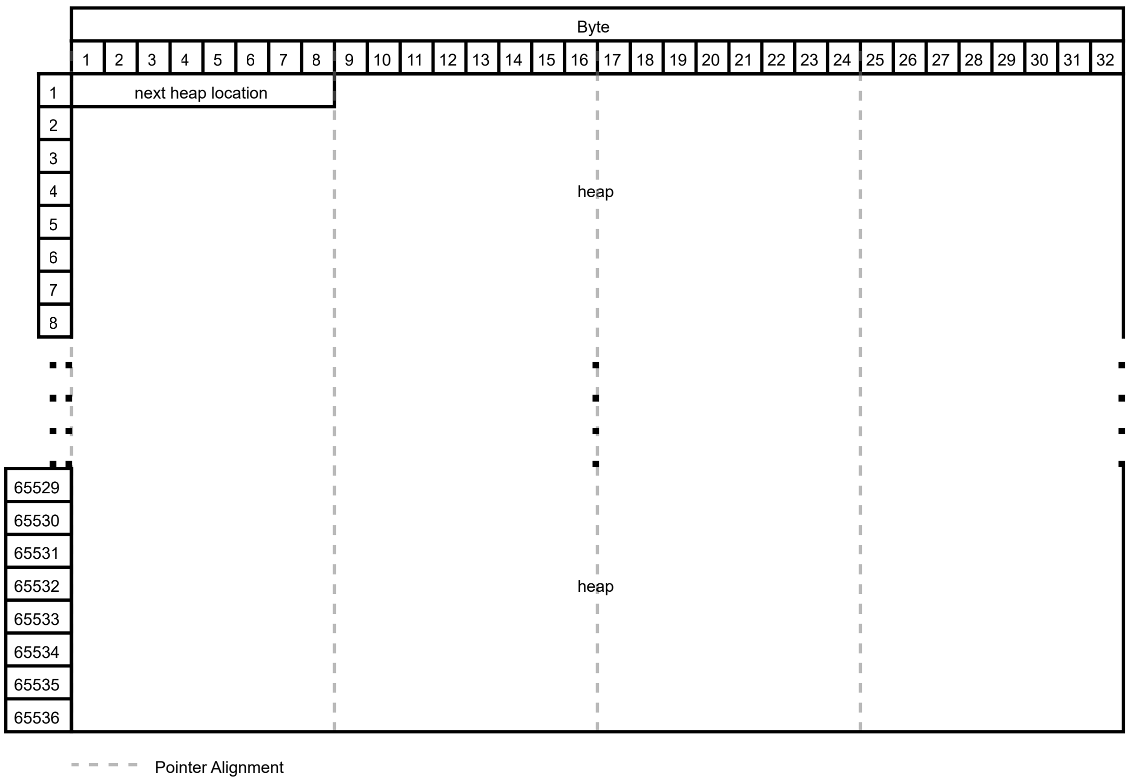


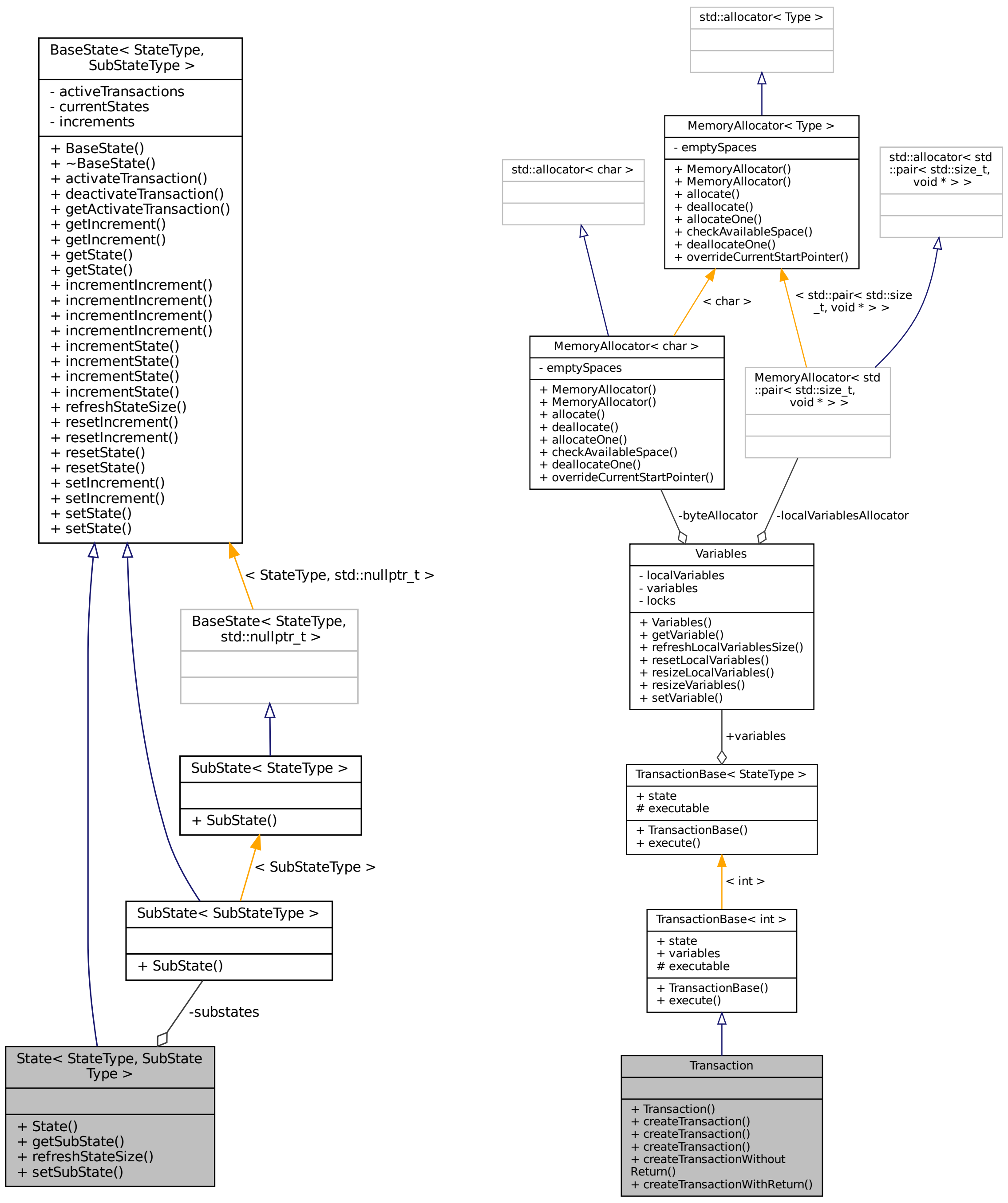
Figure 4.4: Memory Layout of additional heap files

the allocator for another template parameter), no changes are made. This approach enables the Allocator to allocate memory using the basic Allocator or new without specifying an allocation address. If the constructor is invoked with a single argument, the memory will be used starting from the given address (decreasing), but without an endpoint (unless the ”min” variable was set previously). It’s important to note that this approach can lead to unexpected behavior and is thus not recommended. Using the constructor with both variables initializes the allocator to utilize the provided memory range for allocating new objects and reserving additional memory as needed. Additionally, the allocator implements an additional constructor and the ”rebind” function to enable its use in vectors as an allocator. The ”allocate” function allocates the specified number of elements, either by utilizing the base allocator or searching for previously allocated memory of the same size. If previously allocated memory is found, the function returns a pointer to it; otherwise, it checks with the ”checkAvailableSpace” function for memory and creates a pointer using the placement new operator. The ”allocateOne” function allocates and creates a single element using the given arguments, which are forwarded to the constructor. To release the allocated memory, the ”deallocate” and ”deallocateOne” functions are provided. These functions store the byte address provided by the pointer, as well as the size and number of elements if available, to allow for subsequent reuse of the memory. If the ”start” variable is not set, the base deallocator (in the ”deallocate” function) or delete function (in the ”deallocateOne” function) is used.

4.5 State

This file is designed to store information about the last completely run transaction. It consists of several classes that provide functionality for managing states. An overview is provided in Figure 4.5a. The primary class is BaseState, which implements basic functionality for the state. It serves as the foundation for the main State class, which takes two template parameters and is responsible for providing the state to other classes like the ”TransactionManager”. Additionally, there is a second State class that provides a single template parameter. This class serves as a sub-state and can be used independently within a single transaction without interfering with the main state across multiple consecutive (micro-) transactions. All state classes can be constructed with the same arguments as the ”MemoryAllocator”. This is essential for crash-resistant operations since the state needs to be stored in a manner that allows for its reconstruction in the event of a crash, enabling the execution of the remaining (micro-) transactions. To achieve this, the state must be saved in non-volatile memory to survive a system reboot, as well as within a shared address space to allow a fresh process to access the old state. These tasks are facilitated by the Memory Allocator when used with the appropriate arguments. Each concurrently running thread requires its own state to ensure accurate recreation. To simplify this process, the state class is implemented with the necessary mechanisms to manage all states of concurrent threads within a single state class. This involves deleting or adding internal states once a concurrent thread is added or removed. In addition to the state information (indicating the current micro-transaction), it is necessary to store the active transaction information for each thread. The Memory Allocator is utilized for this purpose as well. Furthermore, there is a need to change the increment of the state. This is necessary because if the state is modified while a micro-transaction is running, it may not be possible to correctly restore the current state if the application crashes after manipulating the state but before completing the micro-transaction. To avoid using the increment after a crash, this information is stored in the local memory space without relying on the Memory Allocator. The ”refreshStateSize” function handles this task by querying the number of concurrent threads from the included ”ThreadManager” and adjusting the internal states accordingly. The file provides multiple functions for manipulating the state:

- ”resetState”: Resets the state of a given thread by constructing a new element using the constructor with no input parameters of the template parameter ”StateType”. This allows for alternative reset methods rather than simply setting the internal state to zero.
- ”resetIncrement”: Resets the increment variable (intended to be added to the state of a given thread after completing the current (micro-) transaction). It sets the increment to one if the template parameter ”StateType” is int. Otherwise, it constructs a new element using the constructor with no input parameters of the template parameter ”StateType” and adds one afterward.
- ”incrementState”: Increments the state of the given thread by the increment set for that thread or the given increment value. If no increment was set, it increments by one. This function is intended to be used between two (micro-) transactions.
- ”incrementIncrement”: Increments the increment value added to the state when using the ”incrementState” function for the given thread. Internally, the ++ operator is used, so the ”StateType” must implement this function appropriately, if no other increment is passed as a parameter.



(a) UML Style Diagram for the state classes **(b)** UML Style Diagram for the Transaction classes

Figure 4.5: Some UML Style Diagrams

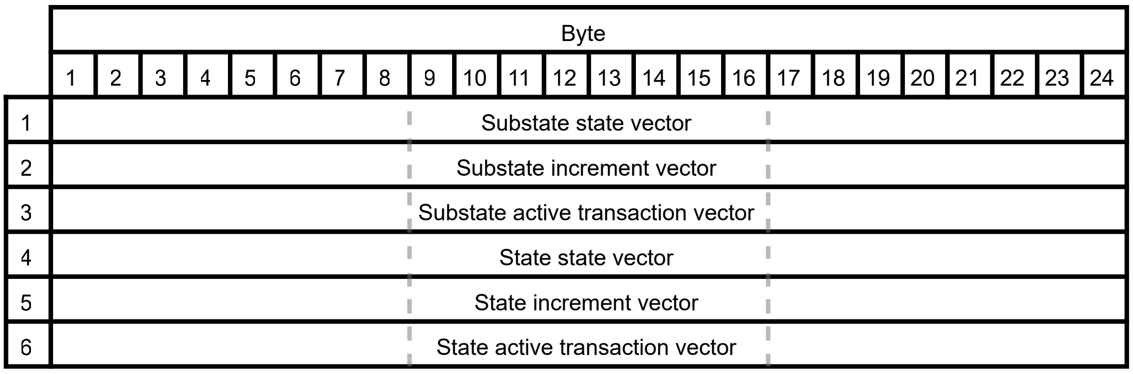


Figure 4.6: Memory Layout of the State class

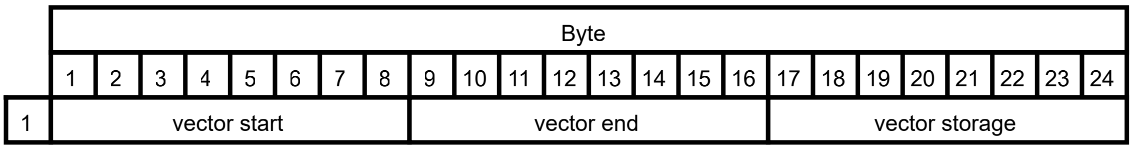
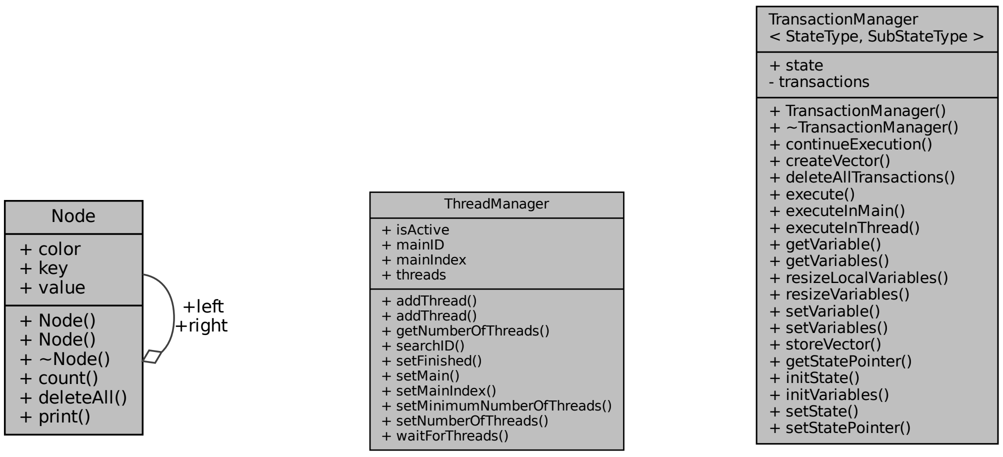


Figure 4.7: Memory Layout of a vector

- Other getter and setter functions are provided for increment, state, substate (if available), and the active transaction. The active transaction is represented by an integer from zero upwards, depending on the registration order, with ”-1” indicating no active transaction in the given thread.

Many of the functions in the file contain duplicated versions that require an additional ”index” integer as an argument. These duplicated functions are designed to minimize the number of times the current thread index needs to be requested, especially in cases where the index is already known. By providing the ”index” argument directly, these duplicated functions avoid the need to retrieve the current thread index repeatedly, which can improve performance. Instead, the known thread index is directly used in the function, eliminating the need for additional computations or calls to obtain the index. This approach is particularly useful in scenarios where the thread index is readily available or has already been determined in the calling context. By passing the index as an argument, the duplicated function can directly utilize this information, improving efficiency. In summary, the duplicated functions with an additional ”index” integer argument are designed to optimize performance by minimizing the number of times the current thread index needs to be retrieved or determined within the function itself. The basic size of a State object is 144 Bytes (on our test systems), depicted in Figure 4.6, with additional heap memory depending on the template arguments ”StateType” and ”SubStateType”, as well as the dynamically used number of threads. The Basic size is determined by the layout of the vector implementation. In our case vectors consist of three pointers as shown in Figure 4.7 [Shw21]



(a) UML Style Diagram for the Node class **(b)** UML Style Diagram for the ThreadManager class **(c)** UML Style Diagram for the Transaction Manager class

Figure 4.8: Some more UML Style Diagrams

4.6 ThreadManager

In modern server infrastructure, the demand for concurrent transactions while ensuring high reliability and data integrity is of paramount importance. However, achieving concurrent execution of transactions in a controlled manner can be challenging. The Thread Manager class, implemented in C++ using jthreads, addresses this challenge by managing multiple threads for running transactions concurrently. It provides an efficient and flexible approach to harness the power of multithreading.

The Thread Manager class facilitates the concurrent execution of transactions by managing multiple threads. Our design embraces microtransactions but does not treat each of them as mutually excluded sections. This approach allows transactions to execute concurrently without excluding others, thereby optimizing resource utilization. However, to maintain data consistency, the responsibility of implementing mutual exclusion lies with the user. The Thread Manager class offers a collection of static functions shown in Figure 4.8b, enabling straightforward access without requiring an instance of the class. A critical feature is its ability to map a unique thread ID to an index, a functionality crucial for accessing the correct content within other classes, such as the State class. The ”searchID” function retrieves the corresponding index associated with a given thread ID. Notably, thread IDs must be registered with the manager before accessing the index, ensuring a seamless mapping process. To register thread IDs, the Thread Manager class provides two methods. The first approach involves designating a given ID as the main ID using the ”setMain” function, and mapping it to the main thread index (defaulted to ”0”). The main index's active status can be altered using the ”setMainIndex” function. The second method, ”addThread”, allows for registering new threads by either providing an already running thread as an argument or passing thread arguments directly. The latter option ensures efficient thread creation and registration while avoiding delays during state changes. Flexibility in thread management is essential for scalability. The ”setNumberOfThreads” function allows for dynamically adjusting the number of indices (or threads) to accommodate concurrent transactions. The main index is distinct and not counted in the total number of indices. Alternatively, the ”setMinimumNumberOfThreads” function increases the indices count while preserving existing ones. The ”getNumberOfThreads” function provides a means to retrieve the total number of indices, including the main index. To ensure controlled execution and prevent conflicts among concurrent transactions, the ”setFinished” function allows marking an index (or thread) as finished, signifying its availability for reuse. By employing the ”waitForThreads” function, all threads can be rejoined back to the main thread upon completion, facilitating proper termination. Overall, the Thread Manager class serves as a crucial component of our C++ library, enabling efficient concurrent transaction execution with dynamic thread management. By delegating mutual exclusion to users, we strike a balance between maximizing resource utilization and maintaining data consistency.

4.7 Transaction

The Transaction file plays a crucial role in managing microtransactions within our C++ library. An overview of the file is shown in Figure 4.5b We implemented a Base Transaction class using the template parameter ”StateType” to allow for various implementations. This template parameter specifies the type used for the (sub-)state within the transactions, offering flexibility and variability. Furthermore, the implementation allows for the incorporation of different kinds of lambdas, such as coroutines, to be utilized as transactions. In the Base Transaction class, two static references are stored: one for Sub States and the other for variables, making access more convenient. Additionally, the executable to be run as a transaction is stored as a function without any input or return parameters. However, this variable only points to an empty class without the actual executable code. As a result, we cannot utilize this variable to restore the transaction in the event of a failure, and thus, it is not saved with the Memory Allocator. To create a basic transaction that runs the identity function, the constructor of the Base Transaction class is used. In this way, a function is always stored, although the executable must be overwritten to execute a meaningful operation. To execute the currently stored executable, the ”execute” function can be employed. Within the Transaction class, a specialization for lambdas/functions with integer as the sub-state type, we implemented multiple static function instances of ”createTransaction”. These functions can be used with different arguments to create transactions that serve specific purposes. For instance, if a transaction is passed as an argument, it is simply returned as is. In the case of a lambda without input or output parameters, an empty transaction is created, the given executable is set, and the newly created transaction is returned. To handle lambda functions with return parameters, we have two additional functions: ”createFunctionWithReturn” and ”createFunctionWithoutReturn”. The ”createFunctionWithoutReturn” function takes a lambda function (with input parameters and no return parameter) as an argument, followed by $n \in \mathbb{N}_0^+$ input argument names (as integers) and their respective types as template parameters. The transaction is created by first generating an empty transaction and then a new lambda function. In this function, the parameters are retrieved from the variables, forwarded to the given lambda function, and executed. The empty lambda in the transaction is then replaced with the newly created one, and the transaction is returned. Similarly, the ”createFunctionWithReturn” function handles lambda functions with both input and output parameters. The transaction is created similarly to the previous case, where an empty transaction is generated, and a new lambda (without input or output parameters) is created. In this lambda function, the input parameters are retrieved from the variables, forwarded to the input lambda function, and executed. The returned value is then stored in the variables, and the empty lambda in the transaction is replaced with the newly created one. In conclusion, the Transaction file provides the ability to convert various types of lambda functions into ones without input and output parameters, effectively using them as transactions. This design allows for versatile and efficient microtransactions, ensuring proper control and management of resources within our C++ library.

4.8 TransactionManager

The TransactionManager file plays a central role in managing various aspects of transactions within our C++ library. In this section, we describe the implemented functions and their functionalities. An overview can be found in Figure 4.8c. It's important to note that this file is one of the subjects of the plugin and is modified in the compile step (see Section 4.1 for more information). The template arguments ”StateType” and ”SubStateType” are used to specify the corresponding template arguments in the state, substate, and transaction, respectively. This is necessary to manage the specialized version within this transaction manager. In the constructor of the TransactionManager, we create and set the state in this file. Additionally, we copy a reference to the substate of the created state to the transaction class to allow for microtransactions to access the substate created here. We also construct and set the variables object used in the transaction class. The creation of the state and variables object is done in functions(initState and initVariables respectively) to facilitate easier modification by the plugin. Since this class is the main file the user interacts with, we provide some functions for the state and variables class. We offer a function to retrieve a pointer to the state using ”getStatePointer” as well as the ability to set the pointer via the ”setStatePointer” function. Additionally, the transaction manager can be used to set the state via the ”setState” function, which will forward the state given to the corresponding ”setState” function of the state class (object). To manage the variables, we also provide several functions: ”resizeVariables” and ”resizeLocalVariables”. These functions forward the given value to the variables object and allow for changing the number of variables to store locally or globally, respectively. Furthermore, we provide a function ”setVariable” to set the value given to the given variable name. The template parameter describes the type of value to store, which is crucial for reserving the correct amount of memory for the value. In some situations, it is necessary to store multiple values at once. Therefore, we also provide the function ”storeVariables”. Here, the types of values to store are specified as template parameters, and the values along with their corresponding names are input as pairs. The function then forwards each pair to the ”setVariable” function of the Variables class. Similarly, we also provide functions to retrieve variables: the ”getVariable” function retrieves a value given a variable name from the variables. The type of the variable to retrieve must be given by the template parameter to allow for the correct type of the returned value. Additionally, we provide a ”getVariables” function to retrieve multiple variables at once. The names of the variables to retrieve are the parameters, and the template parameters describe the type of retrieved values. The function returns the found values as a vector of pairs, where each pair consists of the name of the value followed by the value itself. In addition to forwarded functions, we also provide some functions that utilize the transaction manager itself. For example, the ”createVector” function takes multiple transactions or lambdas and creates a vector of those transactions (lambdas will be converted to transactions beforehand). The created vector is stored locally, and an integer marking the index of this transaction vector is returned. An already existing vector can also be passed to the TransactionManager via the ”storeVector” function.

This function stores the vector internally and returns an integer index of the stored function. The provided index can then be used to run the vector of transactions. There are multiple ways to run the transactions:

- The ”executeInMain” function runs the transaction vector provided on this thread set as the main thread.
- The ”execute” function runs the code without checking if the thread running the vector is set correctly. However, this function will reset the local variables and reset the state before running the provided transaction.
- The ”continueExecution” function does not take any arguments. It searches if there is an active transaction mapped to the thread ID of the calling thread and continues the transaction without resetting the state or local variables.
- Finally, there is the function ”executeInThread”, which creates a new thread using the Thread-Manager that runs the code of the provided transaction index and returns a reference to the created thread.

Lastly, the ”deleteAllTransactions” function iterates over all stored vectors of transactions and deletes them. In summary, the TransactionManager file serves as a comprehensive tool to manage transactions effectively, providing flexibility and control over microtransactions within the C++ library. The functionalities described here play a crucial role in ensuring seamless execution and proper management of transactions.

4.9 Variables

The Variables file plays a crucial role in managing variables that are passed between multiple microtransactions or even multiple transactions within our C++ library. An overview of this file is shown in Figure 4.5b The management of variables revolves around two main concepts: global variables and local variables. Global variables are named with positive integers or zero($n \in \mathbb{N}_0^+$) and are accessible to all threads. They persist even when starting a run of a vector and are thus used for inter-transaction communication. In contrast, local variables are named with negative integer values (not zero)($n \in \mathbb{N}^- \backslash \{0\}$. These variables exist separately for each thread, meaning that each thread has its own set of local variables. It is essential that each thread only uses the local variables provided to it. Local variables are cleared when starting a run of a vector, making them useful for inter-microtransaction communication. These variables play a critical role in ensuring correct recovery of the system after a crash, necessitating their preservation with the Memory Allocator. To create the Memory Allocator, the constructor of the Variables class requires the corresponding parameters for the heap start and end locations. In the constructor, we initialize the Memory Allocator as well as the storage for the variables. Initially, we provide only one storage place for global variables and one for local variables. However, this may not be sufficient for most transactions. To accommodate this, we offer functions for extending and shrinking the number of variables that can be stored. The ”resizeLocalVariables” function is used to adjust the number of local variables to be stored to the given number, while the ”resizeVariables” function provides the same functionality for global variables. To ensure that each thread has a separate storage for its local variables, we provide a function called ”refreshLocalVariablesSize”. This function resizes the number of threads to match the number announced by the user and also resizes the number of

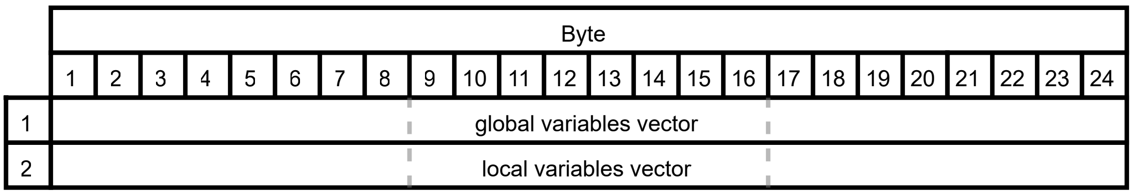


Figure 4.9: Memory Layout of the Variables class

local variables stored for each thread to match the number of variables that can be stored by the first (main) thread. As local variables should not persist across multiple vectors of microtransactions, we provide a way to reset all local variables of a given thread using the ”resetLocalVariables” function with the corresponding thread index. This function sets all local variables to the null pointer. The main functions of this class are the ”setVariable” and ”getVariable” functions. Both functions take an integer, interpreted as a name, as input. This name serves as the descriptor for the corresponding variable, and the respective value is set or retrieved accordingly. Internally, we store the values as pairs, where the first value is the size of the object (size_t), and the second argument is a void pointer. In the ”setVariable” function, we use template parameters to determine the size (using the sizeof function) of the value to be stored. If the size is smaller or equal to the size of a void pointer, we reinterpret the value as a void pointer and store it in the second part of the corresponding pair, while the first part holds the size of the type. However, if the size exceeds the capacity of a void pointer, we allocate storage using the Memory Allocator, specifying Bytes (chars) as a template argument and the size of the type as ”numberOfElements”. Subsequently, we save the newly allocated pointer in the pair, the size of the type in the first part, and the value to store at the allocated storage. To enable access to global variables by multiple threads simultaneously, we added locks to allow for mutual exclusion. These locks are not stored in the crash-resistant file, as they are declared as static variables. This decision is intentional to prevent storing the locks crash-resistantly. In the event of a crash while setting a global variable, accessing the variable upon restart would be impossible. The basic size of the Variables class is 48 Bytes (24 Bytes for the variables vector followed by 24 Bytes for the local variables vector the vector layout is depicted in Figure 4.7) shown in 4.9. However, additional storage is needed and allocated for the variables themselves. In conclusion, the Variables file provides efficient management of global and local variables, ensuring proper communication between transactions and microtransactions while enabling reliable recovery after system crashes. The careful handling of variables and the incorporation of the Memory Allocator contribute to the stability and robustness of our C++ library.

5 Evaluation

In this chapter we describe how we benchmark our implementation to evaluate the performance.

5.1 Setup

To evaluate the performance of our implementation and compare it to a basic function-based implementation, we conducted benchmark tests using red-black trees as the basis. The benchmarks were designed to measure the efficiency of insert, search, and remove operations in both the basic function and transaction implementations, utilizing our library. For benchmarking, we employed the Google benchmark implementation, creating separate benchmarks for each tree operation. We chose red-black trees as they provide a suitable example for transactions that can be implemented through idempotent functions as did [SSS20]. To account for potential variations due to memory implementation, we ran each test twice: once with the basic memory implementation (using ”new” to create objects/pointers) and once with the memory allocator. Specifically, we used the memory allocator in both scenarios, but in one run, we executed the plugin beforehand, and in the other, we did not. These benchmark tests covered various scenarios encountered when running the library. However, the tests did not include measuring the impact of the recovery process in case of a system crash. Since the plugin cannot directly access the main function of the benchmark code, we wanted to use perf (hotspot) to measure the recovery process in a program compiled using our plugin. This did not succeed due to no impact showing up. So we copied the recovery process into a Google Benchmark to obtain a measurement. We focused on pointer manipulation as one of the significant time-consuming steps in mapping the file correctly. Consequently, we ran the benchmark twice: once on an empty file, as if the plugin had just created it, and once on a file that had been used to run some transactions previously. This allowed us to determine the difference between recovering from a crash and starting the file without a previous crash. Each operation benchmark was conducted on trees containing varying numbers of elements before the operation, ranging from 1 to 2^{18} in increments of 2^{12}. By doing so, we obtained an approximation of the runtime for each operation for different tree sizes. To ensure statistical significance, Google Benchmark runs each benchmark multiple times. However, only one representative time (calculated from all runs) is reported in the output. To account for this, we ran all benchmarks multiple times (ten times), saving the results in a usable format (CSV) using a shell script as shown in Listing 5.1. Afterward, we analyzed the benchmark files using Python to derive meaningful insights and comparisons. All tests were conducted on a Windows 11 PC using WSL2.

Listing 5.1 shell script to run the benchmarks multiple times and save the output

```
for VAR in `seq 0 9`
do
  ./benchmarks --benchmark_out_format=csv --benchmark_out=benchmarkResult$VAR
  echo $VAR
done
```

operation	data points	minimum	maximum	median	mean
insert	1290	0.995	5.164	2.775	2.731
search	1290	1.391	4.576	2.933	2.898
remove	1290	1.37	4.211	2.915	2.905
initialize empty file	20	2542.36	3749.18	2581.485	2642.156
initialize used file	20	2808.57	4283.99	3514.855	3527.175

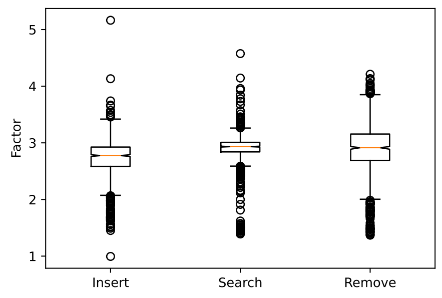
Table 5.1: Results all data Table – Data used in the notch plots using all data

operation	data points	minimum	maximum	median	mean
insert	645	1.621	5.164	2.807	2.801
search	645	1.391	4.576	2.968	2.96
remove	645	1.39	3.679	2.883	2.882
initialize empty file	10	2542.36	3749.18	2565.14	2682.525
initialize used file	10	2808.57	4167.03	2845.19	3347.459

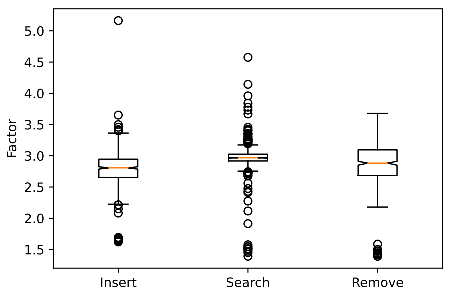
Table 5.2: Results with Memory Table – Data used in the notch plots with our memory implementation

operation	data points	minimum	maximum	median	mean
insert	645	0.995	4.132	2.744	2.662
search	645	1.392	3.932	2.868	2.84
remove	645	1.37	4.211	2.949	2.929
initialize empty file	10	2549.47	2632.23	2608.31	2601.787
initialize used file	10	2887.07	4283.99	4182.555	3706.892

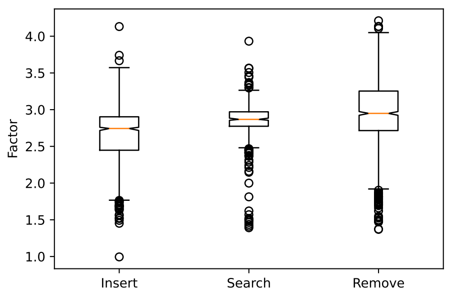
Table 5.3: Results no memory data Table – Data used in the notch plots using the implementation without memory modifications



(a) using all data collected



(b) using only the data collected using the memory modification



(c) using only the data collected without using the memory modification

Figure 5.1: Notch plots for the factor between baseline and our implementation for all operations with fliers

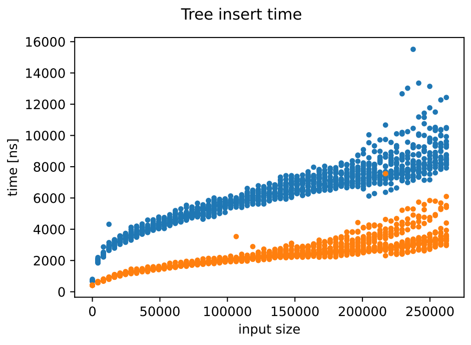
5.2 Results

In this section we present the resulting data from the benchmarks. In total we had 10038798 iterations across all benchmarks. From those iterations we got 3920 Data points in the output file we analyzed.

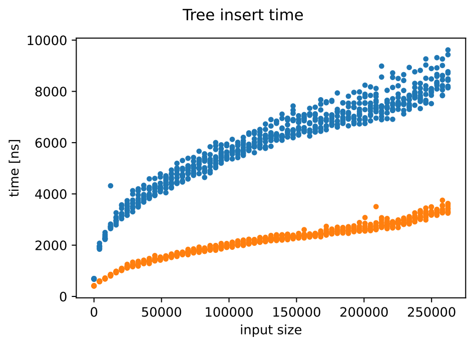
5.2.1 Tree operation results

insert operation

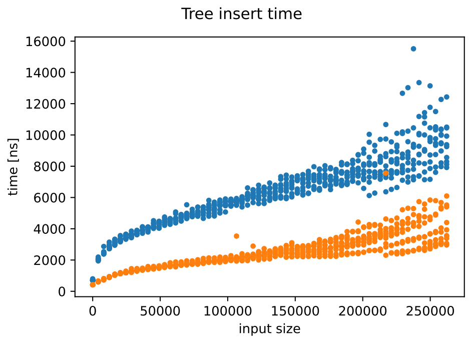
In this section, we analyze the results obtained from benchmarking the insert tree operation. The data points are plotted in Figure Figure 5.2, showcasing the relationship between the size of the tree and the time it takes to insert one element. We noticed an outlier in the data, specifically at input size 12,288, where the insert operation took 150,850 ns. This outlier was removed from the graphics to ensure better visibility of the overall trends. As expected, the basic implementation consistently outperformed our implementation, as managing variables and states incurs additional overhead in



(a) using all data collected



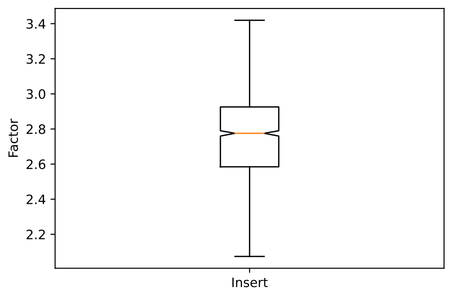
(b) using only the data collected using the memory modification



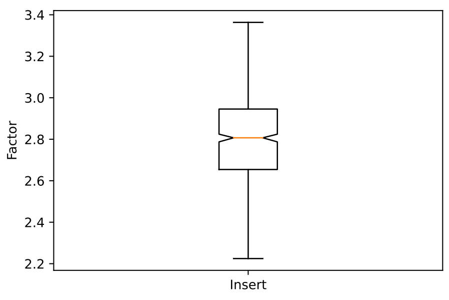
(c) using only the data collected without using the memory modification

Figure 5.2: time in ns it took to insert an element to a tree of given size. orange:Baseline, blue:our implementation

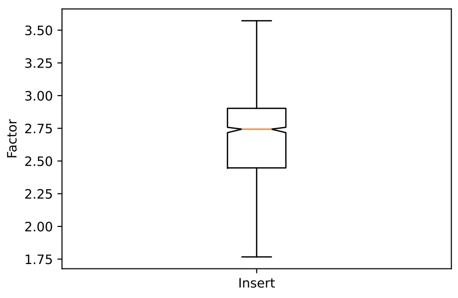
our approach. The difference between the two implementations was not as significant as anticipated, as shown in Figure 5.2a. This performance gap reinforces the importance of carefully considering the trade-offs between functionality and performance when choosing between the basic and our implementation. When comparing the data of the memory implementation (Figure Figure 5.2b) to the basic memory management (Figure Figure 5.2c), we observed that the time data did not scatter as much for higher tree sizes when using the memory manager. This could be due to the memory manager reusing previously freed memory, resulting in more efficient memory usage compared to the basic new operator, which requires allocating new heap memory for each new node inserted. Additionally, the use of files instead of random memory addresses in the heap might improve caching behavior. To quantify the performance difference between the two implementations, we calculated a factor by dividing each time of our implementation by the corresponding time of the basic implementation. The notch plots with fliers (outliers) and without fliers (Figure 5.1 and Figure 5.3, respectively) provide a visual representation of these factors. Additionally, we created separate notch plots for the data using the memory modification (Figure 5.3b) and without the memory allocator (Figure 5.3c). By comparing these plots with the data presented in Tables Table 5.1, Table 5.2, and Table 5.3, we can conclude that the factor of the implementation using the basic memory implementation is slightly lower than the factor using the modified memory.



(a) using all data collected



(b) using only the data collected using the memory modification



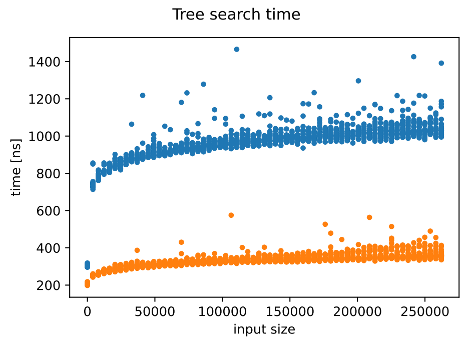
(c) using only the data collected without using the memory modification

Figure 5.3: Notch plots for the factor between baseline and our implementation for the insert operation without fliers

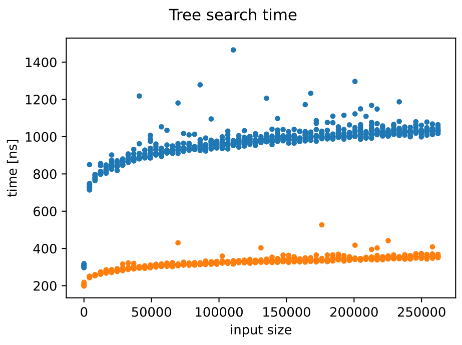
However, the memory implementation significantly reduces the variance in the factor, indicating more consistent performance. So we can determine a factor of about 2.8 as the performance deterioration factor.

search operation

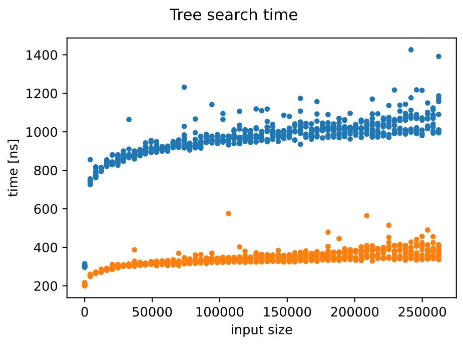
In this section, we delve into a comprehensive analysis of the results obtained from our benchmarking of the search tree operation. By plotting the data points in Figure 5.4, we gain valuable insights into the relationship between the tree size and the time taken to search for a single element. As expected, the basic implementation consistently outperformed our approach, mainly due to the additional overhead introduced by managing variables and states. Although the performance gap between the two implementations (approximated fromFigure 5.4a) was not as substantial as initially anticipated, the significance of carefully weighing functionality against performance becomes evident. Comparing the data of the memory implementation (Figure 5.4b) to the basic memory management (Figure 5.4c), intriguing patterns emerge. Notably, the time data showed reduced scattering, particularly for larger tree sizes, when employing the memory manager. This observation



(a) using all data collected



(b) using only the data collected using the memory modification

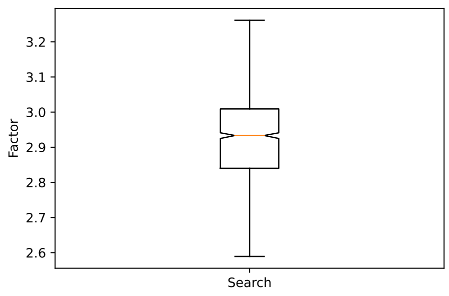


(c) using only the data collected without using the memory modification

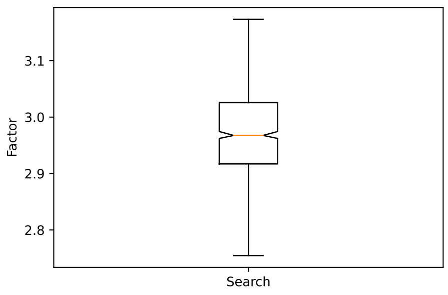
Figure 5.4: time in ns it took to search an element in a tree of given size. orange:Baseline, blue:our implementation

hints at the memory manager's efficiency in reusing previously freed memory, as opposed to the basic new operator, which requires allocating new heap memory for each new node inserted. Moreover, leveraging file-based memory addressing might contribute to improved caching behavior.

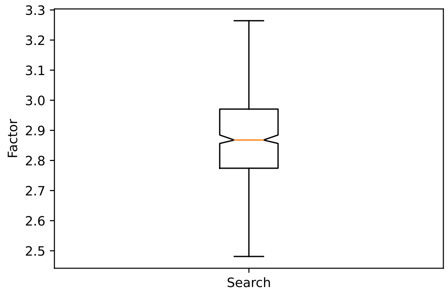
To gauge the quantifiable difference in performance between the two implementations, we calculated a factor by dividing the time of our approach by the corresponding time of the basic implementation. The notch plots with and without fliers (outliers) depicted in Figure 5.1 and Figure 5.5, respectively, provide insightful visual representations of these factors. We further created separate notch plots for the data using the memory modification (Figure 5.5b) and without the memory allocator (Figure 5.5c). From these plots and the data presented in Tables Table 5.1, Table 5.2, and Table 5.3, we established a performance deterioration factor of approximately 2.9. An difference between the data sets can again be determined in that the memory modification increases the factor slightly while also decreasing the variance.



(a) using all data collected



(b) using only the data collected using the memory modification

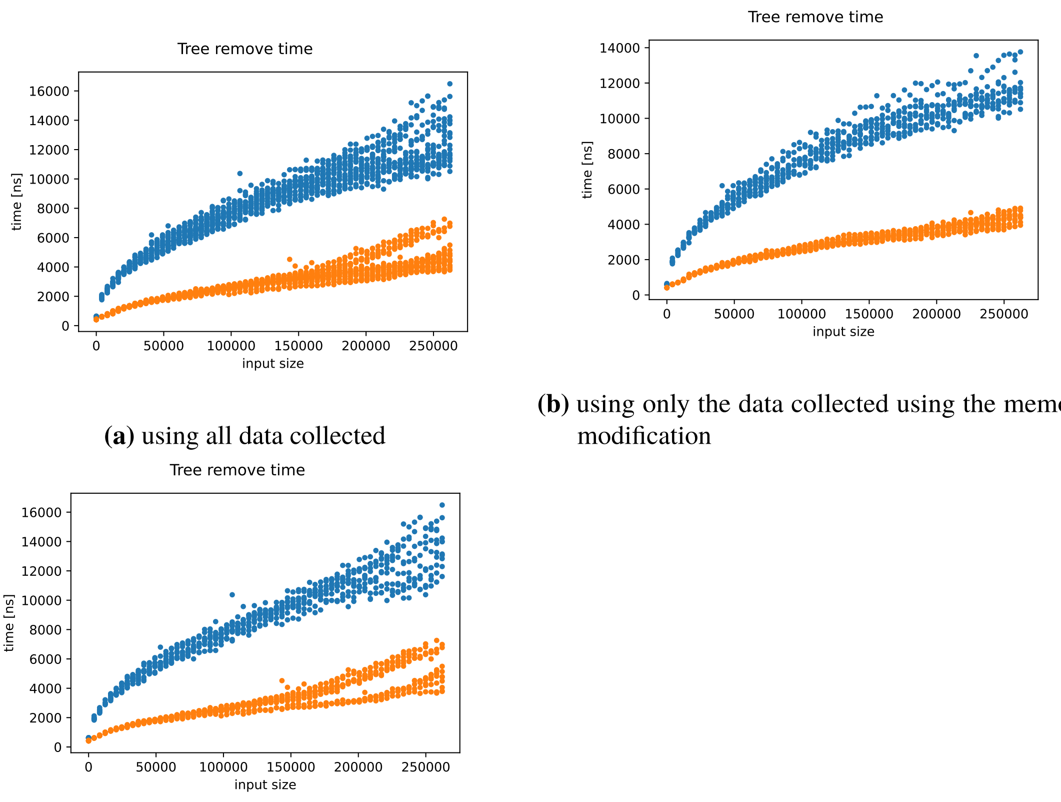


(c) using only the data collected without using the memory modification

Figure 5.5: Notch plots for the factor between baseline and our implementation for the search operation without fliers

remove operation

In this section, we embark on a comprehensive analysis of the results obtained from our benchmarking of the remove tree operation. By graphing the data points in Figure 5.6, we gain valuable insights into the relationship between the size of the tree and the time it takes to remove a single element. As expected, the basic implementation consistently outperformed our approach, primarily due to the additional overhead incurred in managing variables and states. Although the performance gap between the two implementations (approximated from Figure 5.6a) was not as substantial as initially anticipated, the significance of carefully weighing functionality against performance becomes evident. Comparing the data of the memory implementation (Figure 5.6b) to the basic memory management (Figure 5.6c), intriguing patterns emerge. Notably, the time data exhibited slightly reduced scattering, especially in the basic implementation, particularly for larger tree sizes, when leveraging the memory manager. This observation hints at the memory manager's efficiency in reusing previously freed memory, in contrast to the basic new operator, which necessitates allocating new heap memory for each new node inserted. Furthermore, the utilization of file-based memory addressing might contribute to improved caching behavior.



(a) using all data collected

(b) using only the data collected using the memory modification

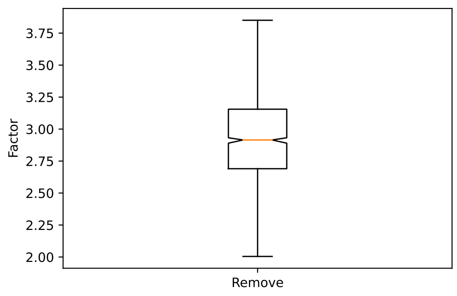
(c) using only the data collected without using the memory modification

Figure 5.6: time in ns it took to remove an element from a tree of given size. orange:Baseline, blue:our implementation

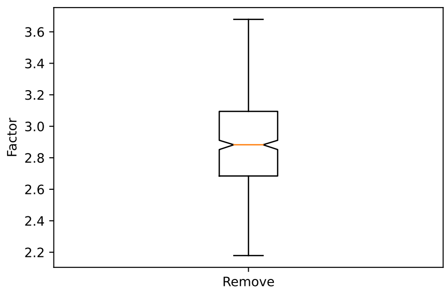
To gauge the quantifiable difference in performance between the two implementations, we calculated a factor by dividing the time of our approach by the corresponding time of the basic implementation. The notch plots with and without fliers (outliers) depicted in Figure 5.1 and Figure 5.7, respectively, provide insightful visual representations of these factors. We further created separate notch plots for the data using the memory modification (Figure 5.7b) and without the memory allocator (Figure 5.7c). From these plots and the data presented in Tables Table 5.1, Table 5.2, and Table 5.3, we established a performance deterioration factor of approximately 2.9. An difference between the data sets can again be determined in that the memory modification decreases the variance without increasing the factor as significantly as in the previous operations.

5.2.2 Recovery results

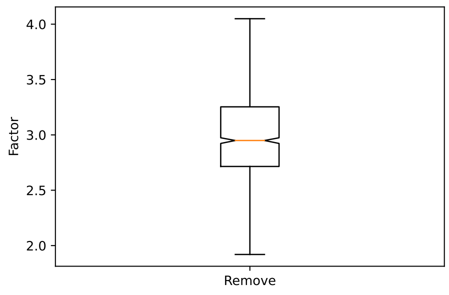
In this section, we examine the results of the recovery benchmark, which are presented in Figure 5.8. Unlike the other benchmarks, we do not have as many data points for this benchmark, as we did not vary the tree size (this has no effect). Instead, we focused on evaluating the recovery process with different scenarios. In the recovery benchmark, we measured the time it took to recover a file that had undergone previous transactions compared to a fresh file. Unsurprisingly, the results showed



(a) using all data collected



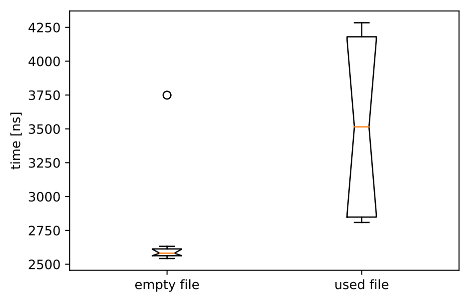
(b) using only the data collected using the memory modification



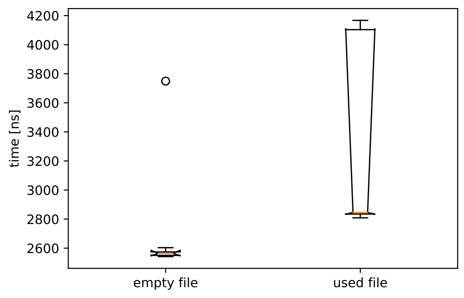
(c) using only the data collected without using the memory modification

Figure 5.7: Notch plots for the factor between baseline and our implementation for the remove operation without fliers

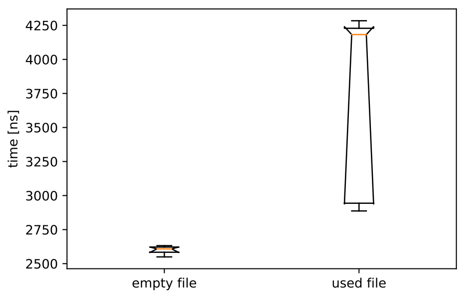
that the file with previous transactions took longer to open and recover than the fresh file. In the scenario without the memory modification (Figure 5.8c), the file used for previous transactions had only one heap file, whereas the file used in the memory modification scenario (Figure 5.8b) had 25 heap files. Despite this significant difference in the number of additional heap files(each has a size of 2MB), the time to recover the file with previous transactions was either higher in the scenario without the memory modification or at most at the same level. This observation suggests that the influence of the number of additional heap files may not be as crucial as expected in the recovery process. However, we had no further ideas while attempting to derive important parameters for an accurate estimation. Consequently, we could only compare the results of recovering an empty file with a filled file. As already stated the used file takes longer to recover, depending on the situation about 1.25 to 1.75 times as long depicted in Table 5.1, Table 5.2, and Table 5.3. Notably, even the process of opening an empty file takes about 2600 ns, which is equivalent to approximately two search operations in the tree. Therefore, optimizing the file opening speed could be a valuable area for further research to reduce startup time and improve overall efficiency. In conclusion, the recovery benchmark shed light on the intricacies of the recovery process and highlighted potential



(a) using all data collected



(b) using only the data collected using the memory modification



(c) using only the data collected without using the memory modification

Figure 5.8: Notch plots for the time it takes to recover from a crash

areas for optimization in future iterations of the implementation. By considering various scenarios and conducting more in-depth analyses, we can make informed decisions regarding the improvement and utilization of our approach in practical use cases.

5.3 Discussion

The findings of our benchmark analysis emphasize the importance of carefully considering the trade-offs between performance and functionality when evaluating the adoption of our implementation. While our approach introduces some overhead due to managing variables and states, the use of the memory manager showcases its potential to enhance stability and optimize variance. To make well-informed decisions regarding the adoption of our approach, further investigation and analysis based on specific use cases are crucial, especially in scenarios where performance plays a critical role. It is essential to weigh the benefits of crash resistance and improved stability against potential performance losses. Based on our rough estimate, our implementation is expected to exhibit a performance loss of about a factor of three, with all operations taking approximately three times longer to execute. However, in some applications, the trade-off in performance may not be as significant when compared to the vital aspect of crash resistance. The additional static increase of approximately 2600 ns for the recovery process to work should also be taken into account when evaluating overall performance. The increased compile time due to the compiler plugin should also be considered as an important factor when evaluating the adoption of the implementation. While the impact of compile time on runtime performance might be negligible, the time taken during the compilation process can significantly affect the development cycle and deployment of updates. It is important to emphasize that each application and use case is unique, and the decision to adopt our implementation should be based on a careful evaluation of specific requirements and priorities. While the potential performance trade-offs exist, the benefits of enhanced stability and crash resistance may outweigh the performance impact, making our approach a viable choice for applications that prioritize reliability and data integrity.

6 Overview of software tools used

This section is about what software we used to create this project.

6.1 ChatGPT

GPT-3 is a language model developed by OpenAI, and it stands for ”Conditional Generative Pre-trained Transformer 3.” It is the third iteration of the GPT series and is known for its impressive language generation capabilities. GPT-3 was trained on a massive amount of text data from the internet and is composed of 175 billion parameters, making it one of the largest language models ever created. With its large size, GPT-3 can generate coherent and contextually relevant responses given a prompt or a question. It has been used for various tasks, such as language translation, content generation, question answering, and more. GPT-3 works by using a transformer architecture, which allows it to understand and generate human-like text based on patterns and information it has learned during training[BKA+23]. The used version of ChatGPT-3.5(an improved version of GPT-3 released November 2022) was accessed through the OpenAI website. We used this tool to improve the language of this paper and improve the grammar.

6.2 Grammarly

Grammarly is an partly AI-enabled cloud based English communication assistance technology that works seamlessly across platforms and devices. Founded in 2009, Grammarly’s primary objective is to improve effective communication. It offers real-time interfaces through various platforms and applications, ensuring convenience and accessibility for users. By continuously advancing their technology, Grammarly strives to provide innovative solutions that facilitate better communication and deliver tangible results[Gra23]. The app was mainly used via Windows on the most recent Version 1.5.81. We used this technology for spell checking and grammar improvements.

6.3 TexMaker

Texmaker, developed since 2003 by Pascal Brachet, is a versatile and user-friendly LaTeX editor that caters to the needs of Linux, macOS, and Windows users. As a free and modern application, Texmaker brings together a comprehensive set of tools required for developing documents with LaTeX, streamlining the entire process into a single, efficient platform. With Texmaker, one can harness the power of LaTeX while enjoying a seamless editing experience. The editor offers robust support for unicode, enabling one to work with diverse characters and languages effortlessly. Additionally, Texmaker features basic spell checking functionality. To enhance productivity, Texmaker offers auto-completion, making it easier and faster to write LaTeX code. The built-in code folding feature allows one to conveniently collapse and expand sections of the document, enabling one to navigate and organize work with ease. Texmaker also boasts a built-in PDF viewer with synctex support, enabling one to synchronize source code and output PDF. This functionality allows for convenient navigation between document and the corresponding sections in the PDF, facilitating efficient editing and reviewing. By combining these powerful features into a single application, Texmaker empowers LaTeX users to create professional and visually appealing documents with ease[Bra03]. The app was used via Windows on the most recent Version 5.1.3. We used this technology to write and compile the LaTeX code for this paper.

6.4 JabRef

JabRef is a remarkable citation and reference management tool that offers users a range of valuable features. As a free and open-source software, it is accessible to all and can be used across various operating systems. One of JabRef's primary functions is assisting users in collecting and organizing sources efficiently. Whether you're conducting academic research or working on a project that requires referencing, JabRef provides a convenient platform to store and manage your references effectively. By employing JabRef's intuitive interface and customizable metadata fields, you can ensure that your reference library remains well-organized and easily accessible. Additionally, JabRef simplifies the process of locating specific papers and publications by enabling advanced search capabilities within your reference database. With its comprehensive search functionality, you can quickly retrieve the information you need, saving valuable time and effort. Furthermore, JabRef serves as a valuable tool for staying up-to-date with the latest research in your field. By offering features such as DOI and arXiv lookup, as well as integration with various academic databases, JabRef enables you to discover and access the most recent scholarly publications effortlessly. JabRef's development and maintenance are carried out by a dedicated team comprising PhD students, postdocs, and researchers from diverse backgrounds. Although they contribute to JabRef in their spare time, their commitment ensures that the software remains continually updated and responsive to user needs. Since its establishment in 2003, JabRef has grown to become a trusted resource for researchers, students, and professionals seeking an efficient and user-friendly citation and reference management solution[KKG+23]. The app was used via Windows on the most recent Version 5.9. We used this technology to create and edit the biblography for the LaTeX code for this paper.

6.5 Microsoft Visual Studio Community 2022

Visual Studio developed by Microsoft is a powerful and comprehensive integrated development environment (IDE) that empowers developers to complete the entire development cycle seamlessly. With features like code writing, editing, debugging, building, and app deployment, Visual Studio provides a robust platform for efficient software development. It goes beyond basic code editing and debugging, offering a wide array of tools such as compilers, code completion, source control, and extensions to enhance every stage of the development process. Supporting multiple programming languages, Visual Studio allows developers to start from simple "Hello World" programs and progress to building and deploying complex applications. Whether it's creating and testing .NET and C++ apps, designing ASP.NET pages in a web designer view, developing cross-platform mobile and desktop apps using .NET, or crafting responsive web user interfaces in C#, Visual Studio offers a versatile environment for developers[AMR+23]. The app was used via Windows on the most recent Version 17.6.4. We used this technology to write the C++ code.

6.6 Windows Subsystem for Linux

Developers can harness the capabilities of both Windows and Linux simultaneously on a Windows machine through the Windows Subsystem for Linux (WSL). By installing a Linux distribution like Ubuntu, OpenSUSE, Kali, Debian, Arch Linux, and more, developers can seamlessly utilize Linux applications, utilities, and Bash command-line tools directly on Windows, without the need for installing virtual machines or dual-boot configurations. This enables efficient cross-platform development and eliminates the overhead typically associated with traditional setups[WWT+22]. The app was used via Windows on the most recent Version 1.2.5.0 of WSL2. We used this technology to compile and run the C++ code. In the Subsystem we operated on the Kernel version 5.15.90.1 on a Ubuntu 22.04.2 LTS installation.

6.7 Doxygen

Doxygen by Dimitri van Heesch is a widely recognized tool for generating documentation from annotated source code, primarily for C++, but it also supports various other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, and to some extent D. It also provides support for the hardware description language VHDL. It enables the generation of online documentation in HTML format and/or offline reference manuals in LaTeX format from documented source files. Additionally, it supports output formats like RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the source code, ensuring consistency between the documentation and the codebase. Doxygen can be configured to extract the code structure from undocumented source files, facilitating navigation in large codebases. It also automatically generates visual representations, including include dependency graphs, inheritance diagrams, and collaboration diagrams, to illustrate the relationships between different elements. In addition to generating technical documentation, Doxygen can be used for creating regular documentation, as demonstrated in the doxygen user manual and website. Doxygen is developed primarily for Mac OS X and Linux but is designed to be highly portable, making it compatible with most Unix flavors. Executables for Windows are also available[Hee23]. The technology was used via Ubuntu on the most recent Version 1.9.1 of WSL2. We used this technology to create the html documentation and some graphics of the C++ code.

6.8 draw.io

draw.io is an open-source technology stack designed for building diagramming applications. It is widely recognized as the most extensively used browser-based end-user diagramming software in the world. draw.io is a registered trademark owned by JGraph Ltd and draw.io AG. JGraph Ltd is a company registered in England, while draw.io AG is a company registered in Switzerland. These two companies jointly develop and own the draw.io software. They operate the websites diagrams.net and draw.io, and also possess the rights to the diagrams.net and draw.io brands. One of the distinguishing aspects of draw.io is its disruptive business model. While aiming to generate reasonable profits to sustain a dedicated and professional software team, the company avoids employing artificial scarcity tactics that can lead to bloated sales-oriented organizations with excessive revenues. The mission statement of draw.io is to provide free, high-quality diagramming software accessible to everyone. This commitment to delivering valuable diagramming tools without cost barriers aligns with their vision of democratizing the creation of visual graphics[23]. For our project, the draw.io technology was utilized through the Windows app, specifically on the latest version 21.6.1. This technology was chosen to create graphics that couldn't be automatically generated using Doxygen, a popular documentation generator tool. By leveraging draw.io, we were able to enhance your project with visually appealing and informative graphics that would have otherwise been challenging to create with other tools or technologies.

7 Conclusion and Outlook

Conclusion

In conclusion, we have developed a C++ library that enables crash-resistant execution of unrestricted transactions provided as lambdas. Our implementation bridges the gap between existing crash-resistant and unrestricted transaction implementations in C++. Initially, we designed a non-crash-resistant version that allowed for the execution of lambda transactions. To achieve crash resistance, we utilized a clang plugin during compilation to modify the source file and stored critical runtime variables in a file as well as the lambda functions in the source code. While our implementation is approximately three times slower than an implementation using transactions without lambdas, the potential for full crash resistance outweighs the loss in speed. This library provides a practical solution for critical applications that require both crash resistance and unrestricted transactions in C++. By combining the flexibility of lambdas with the robustness of crash resistance, our library empowers developers to create more reliable and resilient systems. Further research and optimization could potentially narrow the performance gap, making our implementation even more attractive for a wide range of applications. In the future, we plan to explore additional optimizations and further refine our library to strike a better balance between performance and functionality. Additionally, we aim to extend the applicability of our approach to distributed systems, where the synchronization of transactions across multiple systems could benefit from the crash-resistant capabilities we have implemented. Overall, our work contributes to the growing body of research in the field of transaction management in C++, providing a valuable tool for developers seeking both flexibility and reliability in their applications.

Outlook

In this thesis, we primarily focused on the usage of our technology in local systems, particularly in the context of NVRAM. However, the potential applications of this idea extend beyond local systems, and there could be interesting opportunities in distributed systems as well. One intriguing possibility lies in leveraging our implementation to serialize the execution process of transactions, effectively serializing the lambdas used in these transactions. This concept could be applied in distributed systems to synchronize the execution of lambdas across multiple nodes. Instead of rewriting the main file on each system, we could generate the lambdas and add them to the file, allowing us to send the file to other systems for execution. By doing so, we could achieve synchronization among distributed systems running the same lambdas. This approach could lead to enhanced coordination and consistency in distributed environments. As each system executes the same set of lambdas in a serialized manner, it becomes possible to ensure that operations are carried out in the same order on all nodes, promoting data consistency and reducing the likelihood of conflicts. Additionally, by sending the files with the serialized lambdas, we can efficiently propagate updates and changes to the lambdas, maby not inc c++ but in other languages, across the distributed system, facilitating dynamic adaptation and fault tolerance. While our thesis primarily explores the application of our technology in local NVRAM-based systems, the potential for distributed system synchronization opens up exciting avenues for future research and implementation. By further investigating and testing this concept, we can explore the scalability and robustness of our approach in distributed environments and potentially uncover new possibilities for its application in various scenarios, such as distributed databases, cloud computing, and IoT networks. Overall, this extension of our technology to distributed systems presents an intriguing direction for future work, and its successful implementation could have significant implications for the advancement of distributed computing paradigms.

Bibliography

[17] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712 (cit. on p. 17).

[23] *About draw.io*. 2023. URL: https://www.drawio.com/about (visited on 07/10/2023) (cit. on p. 56).

[94] "IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory (DMA)". In: *IEEE Std 1212.1-1993* (1994), pp. i–. DOI: 10.1109/IEEESTD.1994.120263 (cit. on p. 17).

[AMR+23] M. Anand, P. Miller, D. Richards, G. Hogenson, T. G. Lee, A. Buck, J. Kirsch, T. Petersen, j-martens, J. Parente, M. Jacobs, G. Warren, C. Robertson, eds. *What is Visual Studio?* Microsoft. May 5, 2023. URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (visited on 07/04/2023) (cit. on p. 55).

[BKA+23] A. Bahrini, M. Khamoshifar, H. Abbasimehr, R. J. Riggs, M. Esmaeili, R. M. Majdabadkohne, M. Pasehvar. "ChatGPT: Applications, Opportunities, and Threats". In: *2023 Systems and Information Engineering Design Symposium (SIEDS)*. Apr. 2023, pp. 274–279. DOI: 10.1109/SIEDS58326.2023.10137850 (cit. on p. 53).

[Bra03] P. Brachet. *TEXMAKER. Free cross-platform LaTeX editor since 2003*. 2003. URL: https://www.xm1math.net/texmaker/ (visited on 07/04/2023) (cit. on p. 54).

[Bro18] R. Brown. *DEFINITION NVRAM (non-volatile random-access memory)*. June 30, 2018. URL: https://www.techtarget.com/searchstorage/definition/NVRAM-non-volatile-random-access-memory (visited on 07/20/2023) (cit. on p. 20).

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko. "Compiling Java just in time". In: *IEEE Micro* 17.3 (May 1997), pp. 36–43. ISSN: 1937-4143. DOI: 10.1109/40.591653 (cit. on p. 19).

[Cla23] L. Clang. *Introduction to the Clang AST*. 2023. URL: https://clang.llvm.org/docs (visited on 06/22/2023) (cit. on pp. 19, 20).

[Gra23] Grammarly. *About Us | Grammarly*. 2023. URL: https://www.grammarly.com/about (visited on 07/04/2023) (cit. on p. 53).

[Hee23] D. van Heesch. *Doxygen. Generate documentation from source code*. June 28, 2023. URL: https://www.doxygen.nl/ (visited on 07/04/2023) (cit. on p. 55).

[HW90] M. P. Herlihy, J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972 (cit. on p. 18).

[ISO98] ISO. *IEC 10746-1 Information technology–Open Distributed Processing–Reference Model: Overview*. 1998 (cit. on p. 17).

[JTC15] I. JTC. “Technical Specification for C++ Extensions forTransactional Memory”. In: *ISO/IEC TS* 19841 (2015) (cit. on p. 23).

[KKG+23] O. Kopp, S. Kolb, M. Geiger, T. Diez, C. Schwentker, C. C. Snethlage, D. Voigt, J. Askertop, B. Tutzer, T. Ertel, eds. *JabRef - Free Reference Manager. Stay on top of your Literature*. The efficient way to collect, organize and discover. July 5, 2023. URL: https://www.jabref.org (visited on 07/05/2023) (cit. on p. 54).

[LK08] J. Larus, C. Kozyrakis. “Transactional memory”. In: *Communications of the ACM* 51.7 (2008), pp. 80–88 (cit. on p. 17).

[Nic22] T. Nicolai. *fire-llvm*. Feb. 3, 2022. URL: https://github.com/Time0o/fire-llvm (visited on 06/23/2023) (cit. on pp. 20, 26).

[Pap79] C. H. Papadimitriou. “The Serializability of Concurrent Database Updates”. In: *J. ACM* 26.4 (Oct. 1979), pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158. URL: https://doi.org/10.1145/322154.322158 (cit. on p. 18).

[Shw21] D. Shweta. *HOW ARE VECTORS IMPLEMENTED INTERNALLY IN C++*. Jan. 7, 2021. URL: https://duggal-shweta2010.medium.com/how-are-vectors-implemented-internally-in-c-5787abf1c38f (visited on 08/01/2023) (cit. on p. 34).

[SSS20] T. Schütt, F. Schintke, J. Skrzypczak. “Transactions on Red-black and AVL trees in NVRAM”. In: *arXiv preprint arXiv:2006.16284* (2020) (cit. on pp. 15, 18, 23, 29, 41).

[WSSZ07] C. P. Wright, R. Spillane, G. Sivathanu, E. Zadok. “Extending ACID Semantics to the File System”. In: *ACM Trans. Storage* 3.2 (June 2007), 4–e. ISSN: 1553-3077. DOI: 10.1145/1242520.1242521. URL: https://doi.org/10.1145/1242520.1242521 (cit. on p. 17).

[WWT+22] M. Wojciakowski, M. Wojciakowski, Tassoman, renzo, R. DuBois, M. Patel, H. Arya, A. Jenks, T. Raj, S. Cooley, B. Nath, M. Satran, jdorsay-xenu, H. maurya, B. Carranza, D. Coulter, J. Martinez, v-surgos, eds. *What is the Windows Subsystem for Linux?* Microsoft. Dec. 8, 2022. URL: https://learn.microsoft.com/en-us/windows/wsl/about (visited on 07/04/2023) (cit. on p. 55).

[ZHMS06] X. Zhang, M. A. Hiltunen, K. Marzullo, R. D. Schlichting. “Customizable Service State Durability for Service Oriented Architectures”. In: *2006 Sixth European Dependable Computing Conference*. Oct. 2006, pp. 119–128. DOI: 10.1109/EDCC.2006.8 (cit. on p. 17).

[ZZB+19] P. Zardoshti, T. Zhou, P. Balaji, M. L. Scott, M. Spear. “Simplifying Transactional Memory Support in C++”. In: *ACM Trans. Archit. Code Optim.* 16.3 (July 2019). ISSN: 1544-3566. DOI: 10.1145/3328796. URL: https://doi.org/10.1145/3328796 (cit. on pp. 23, 24).

All links were last followed on August 1, 2023.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Druck-Exemplaren überein.

Datum und Unterschrift:

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

Date and Signature: