



國立臺灣大學重點科技研究學院積體電路設計與自動化碩士學位學程

碩士論文

Program for Integrated Circuit Design and Automation

Graduate School of Advanced Technology

National Taiwan University

Master's Thesis

以協程架構實現切換性軟體事務性記憶體的性能提升

Enhancing Performance of Switchable Software Transactional Memory
through Coroutine Architecture Implementation

施墨然

Mo-Ran Shih

指導教授：郭斯彥 博士

Advisor: Sy-Yen Kuo, Ph.D.

中華民國 113 年 7 月

July, 2024

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

以協程架構實現切換性軟體事務性記憶體的性能提升

Enhancing Performance of Switchable Software Transactional Memory through Coroutine Architecture Implementation

本論文係 施墨然 R11K41017 在國立臺灣大學重點科技學院 積體電路設計與自動化碩士學位學程 完成之碩士學位論文，於民國 113 年 7 月 16 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Program for Integrated Circuit Design and Automation on 16/07/2024 have examined a Master's thesis entitled above presented by Shih, Mo-Ran R11K41017 candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

鄧斯齊

(指導教授 Advisor)

雷欽隆

呂昌明

林崇男

袁世一

主任 Director:

黃俊郎





Acknowledgements

在臺灣大學研究所度過了將近兩個學年，這段時間充滿了挑戰與成長，我深刻感受到了學術探索的無限魅力。在這段旅程中，我得到了許多人的幫助與支持，使我能夠順利完成我的研究工作。

首先，我要衷心感謝郭斯彥教授，他給予我進入實驗室的機會，還提供了優良的研究環境和豐富的資源。在我的研究過程中，他給予了我無私的指導和寶貴的建議，使我能夠不斷向前。同樣地，我也要感謝袁世一副教授，他的引導與提點使我的研究更加完整，並幫助我解決了許多困難。

接著，我要特別感謝謝宙穎學長。是他提供了寶貴的研究題目，在探索研究背景時告訴我該如何快速切入重點，並在整個過程中給予我耐心的引導與指正，讓我不斷進步，更在我遇到研究困境時給予了我啟發式的想法，幫助我突破了困境。倘若沒有他無私的分享與引導，或許至今我仍在為研究方向而煩惱。謝宙穎學長無疑是我不可或缺的良好良師益友，對此我深感感激。

此外，我還要感謝實驗室的所有同學。在此我收穫了無數的鼓勵與幫助。特別要提到的是在台積電實習時認識的蔡家洋同學，在這段時間裡所給予的陪伴與協助對我來說意義重大。

最後，我要感謝我的家人和朋友們。在我追求學術目標的道路上給予了我無盡的支持和理解，使我能夠克服各種困難繼續前行。

謹此，向旅途中所有美好的事物致敬。





摘要

本研究探討以協程 (coroutine) 實現軟體事務性記憶體 (Software Transactional Memory) 的運算排程，透過在協程間切換來提升其運算效能及可擴展性，軟體事務性記憶體用於處理協調衝突的方法，能夠大致區分為兩種架構：專注於衝突解決的衝突管理器 (Contention Manager) 或者是專注於衝突避免的排程器 (scheduler)，衝突管理器根據衝突解決策略決定衝突的事務 (transaction) 該中止或繼續執行，而排程器則透過排程來防止衝突再次發生，然而兩者都對可擴展性造成了極大的限制，許多文獻探討的機制將導致執行緒進行無謂的閒置，或者進行無效工作最終被中止，從而低效地利用計算資源。

本文提出了一種新穎的計算框架，稱為切換性軟體事務性記憶體 (switch-STM)，將任務封裝為協程進行計算。當發生衝突時，切換協程以繼續計算，防止執行緒不必要地閒置。該框架不受任務排程限制，並具有高度的可擴展性。此外，它與衝突管理器兼容，可同時使用進一步提升計算效率。本文提出了三種協程切換策略，通過探索這些切換策略，推進軟體事務性記憶體計算框架的可擴展性和性能。

關鍵字：事務性記憶體、運算架構、排程、平行運算、平行編程





Abstract

This study explores the computational scheduling of Software Transactional Memory (STM) using coroutines, aiming to enhance its computational efficiency and scalability by switching between coroutines. Methods employed by STM to handle coordination conflicts can be broadly classified into two architectures: the Contention Manager, which focuses on conflict resolution, and the Scheduler, which focuses on conflict avoidance. While Contention Manager determines whether conflicting transactions should proceed or halt based on conflict resolution policies, Scheduler prevents conflicts from reoccurring by scheduling. However, both architectures impose significant limitations on scalability. Many existing mechanisms result in thread idleness or futile work that will ultimately be terminated, thereby inefficiently utilizing computational resources.

This paper proposes a novel computational framework called Switchable Software Transactional Memory (SwitchSTM), which encapsulates tasks into coroutines for computation. When conflicts arise, coroutines are switched to continue computation, preventing

unnecessary thread idling. This framework is not constrained by task scheduling limitations and exhibits high scalability. Additionally, it is compatible with Contention Manager, further enhancing computational efficiency. Three coroutine-switching strategies are proposed in the paper, advancing the scalability and performance of STM computational frameworks through the exploration of these switching strategies.

Keywords: transnational memory, computing architecture, scheduling, Parallel Computing, Parallel Programming



Contents

	Page
Acknowledgements	iii
摘要	v
Abstract	vii
Contents	ix
List of Figures	xi
List of Tables	xiii
Chapter 1 Introduction	1
Chapter 2 Background	5
2.1 Parallel computing	5
2.2 Software Transactional Memory	7
2.3 TinySTM	10
Chapter 3 Performance Improvement	11
3.1 Conflict	11
3.2 Contention manager	14
3.3 Scheduler	17
3.4 Scalability Issue	19



Chapter 4	Coroutine	21
4.1	Concepts of coroutine	21
4.2	Symmetric Coroutine vs. Asymmetric Coroutine	23
4.3	Libaco	24
Chapter 5	SwitchSTM	27
5.1	Task Iterative STM	27
5.1.1	Task Iterative STM system architecture	27
5.1.2	Task Iterative STM Interaction	30
5.2	Core design of SwitchSTM	31
5.2.1	SwitchSTM system architecture	31
5.2.2	Transaction switching execution	33
5.2.3	Switcher implementation	35
5.2.4	Correctness of switching	38
Chapter 6	Evaluation	39
6.1	Configuration	39
6.2	Performance comparison with basic STM	40
6.3	Performance comparison with prior work	42
6.4	Abort ratio comparison with prior work	43
Chapter 7	Conclusion	45
Chapter 8	Future Work	47
References		49



List of Figures

2.1	Transaction Execution	9
3.1	Contention Manager actions	16
3.2	Scheduler actions	17
3.3	Comparison of methods for performance improvement	19
4.1	function vs. coroutine	22
4.2	Libaco API	25
5.1	Task Iterative STM	28
5.2	Task Iterative STM	30
5.3	SwitchSTM system architecture	32
5.4	Transaction switching execution	33
5.5	Switcher architecture	35
5.6	Correctness issue: transaction switch example	38
6.1	Performance improvement comparison of SwitchSTM	41
6.2	Performance improvement comparison with prior work	42
6.3	Abort ratio comparison with prior work	44





List of Tables

6.1	The configuration of STAMP benchmark.	40
-----	---	----



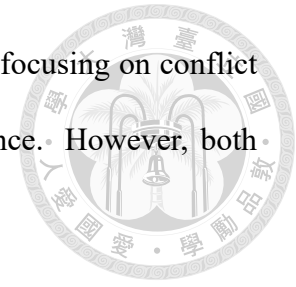


Chapter 1 Introduction

Software Transactional Memory (STM)[13] [16] is a technique used in parallel computing to manage shared resources. In multi-threaded programming, simultaneous access to shared resources can lead to issues like race conditions and deadlocks. STM offers a high-level abstraction to address these problems, allowing programmers to implement parallel programs in a simpler manner without needing to consider the low-level logic of implementation. In STM, programmers mark program blocks requiring atomic operations as transactions, and the STM system is responsible for managing the execution of these transactions. When multiple transactions are executed concurrently, STM tracks their interactions and performs undo operations, reverting modifications to system data when necessary to ensure resource consistency, or maintains a redo log to apply transactional operations to the system at the appropriate time. Compared to traditional locking mechanisms, STM provides higher code readability and scalability while reducing the complexity of errors and debugging. Software Transactional Memory offers a more flexible and convenient approach to parallel programming, enabling developers to more effectively leverage the potential of multi-core processors and distributed systems.

However, STM also faces challenges such as performance overhead and memory usage, making careful evaluation and optimization necessary in practical applications. There are various methods to address the performance loss caused by aborts, which can be cat-

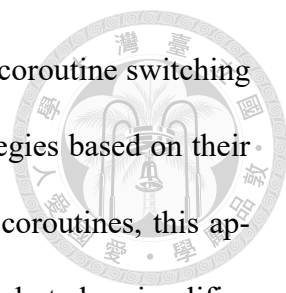
egorized into two main frameworks: contention manager, primarily focusing on conflict resolution, and scheduler, mainly concentrating on conflict avoidance. However, both frameworks have certain scalability limitations.



Under the contention manager framework[2], the emphasis lies in resolving conflicts between transactions to ensure minimal overhead in advancing computation progress while avoiding recurring conflicts. Several common conflict resolution strategies include suicide, aggressive, polite, delay, karma, polka[12], among others. While these strategies can reduce conflicts and transaction aborts to some extent, they may also lead to increased system overhead and complexity, resulting in reduced scalability.

On the other hand, under the scheduler framework, the primary focus is on avoiding conflicts by dynamically adjusting the execution order of transactions or resource allocation. The goal of this approach is to minimize the occurrence of conflicts during runtime, thereby reducing the likelihood of transaction aborts. However, implementing this method often requires complex scheduling or limiting the number of threads to reduce conflicts, which may inadvertently decrease concurrency[10], making system resources inefficiently utilized and thus limiting scalability.

While both contention manager and scheduler methods have shown some effectiveness in reducing STM performance loss, they exhibit certain limitations in scalability[5]. To address this issue, this study proposes a novel computational framework aimed at ensuring program correctness while efficiently utilizing system resources. The key to this new approach lies in wrapping STM tasks as coroutines[11] to execute. When a transaction aborts, the system automatically switches to other coroutines for computation. This design leverages system resources that contention managers or schedulers cannot fully uti-



lize, further enhancing computational efficiency. Additionally, three coroutine switching strategies are proposed, allowing developers to choose existing strategies based on their requirements or develop their own. By combining STM tasks with coroutines, this approach not only enables more efficient utilization of system resources but also simplifies the design and testing process of parallel programs, thus improving system performance and stability.

The remaining chapters of this paper mainly cover background to existing research and detailed analyses of the proposed new computational framework. Chapter 2 will introduce the existing research background, including the use of tinySTM for developing computational frameworks. Chapter 3 discusses the mechanisms currently used to improve STM performance, including the Contention Manager focused on conflict resolution and the Scheduler focused on conflict avoidance, while also questioning their scalability limitations. Chapter 4 explains the tool, coroutine, used to implement transaction switching functionality. Chapter 5 will present the new computational framework proposed in this paper, SwitchSTM, along with its implementation details. Chapter 6 will explore the performance of SwitchSTM and compare it with past research cases to validate its performance advantages. Finally, Chapter 7 will summarize the research findings of the entire paper and draw conclusions, proposing possible future research directions.





Chapter 2 Background

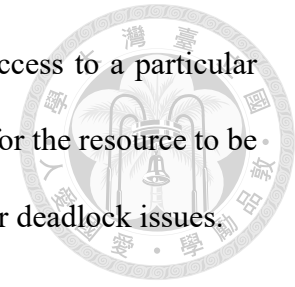
This section introduces the research background of parallel computing, provides contextualized explanations of the evolution of parallel computing, and the emergence of STM. It briefly describes the computational architecture of STM and introduces TinySTM, which we use as the development framework.

2.1 Parallel computing

Parallel computing is a computational approach that divides computational tasks into multiple independent parts and executes these parts simultaneously on multiple processing units. This approach greatly enhances computational efficiency and performance, especially in fields such as big data processing, scientific computing, and machine learning. However, parallel computing also presents challenges, one of which is ensuring synchronization among multiple processing units and correct access to shared resources.

Failure to properly allocate resource access among multiple processing units can lead to race conditions or resource contention. A race condition[6] occurs when two or more threads or processes attempt to access or modify a shared resource simultaneously, with the final outcome depending on the execution order or timing. Due to the uncertainty of execution order, the final outcome may not be the expected value. Resource contention

arises when multiple threads or processes simultaneously request access to a particular resource, potentially causing some threads to be blocked or waiting for the resource to be released, resulting in performance degradation, system congestion, or deadlock issues.



In parallel computing, when multiple processing units simultaneously access shared resources, race conditions and resource contention problems can arise. This is where the concept of the "critical section" comes into play. A critical section[15] is a segment of code where only one processing unit can execute at a time. By ensuring that only one processing unit can enter this section of code at any given time, race conditions and resource contention problems can be avoided.

To implement a critical section, various synchronization mechanisms are commonly used, including locks, semaphores, and condition variables. Locks are the most basic synchronization mechanism, allowing a processing unit to lock it before entering the critical section, thereby preventing other processing units from entering. However, locks are prone to deadlocks[1] and lock contention issues, and they cannot support advanced synchronization requirements. Semaphores are a more complex synchronization mechanism that allows limiting the number of processing units entering the critical section simultaneously, but they can suffer from inefficiency and high complexity issues. Finally, condition variables provide a mechanism for entering the critical section based on specific conditions, but they may also introduce deadlocks and complexity issues.

These synchronization mechanisms for implementing the critical section often involve dealing with many low-level details and complexities. While they ensure safe access to shared resources, they lack sufficient abstraction, requiring programmers to consider many low-level implementation details when building programs. This often increases the

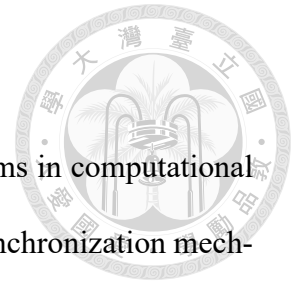
complexity and difficulty of development.

To address these challenges, various synchronization mechanisms in computational frameworks have been researched and developed. One significant synchronization mechanism is Software Transactional Memory (STM)[13]. STM offers a higher level of abstraction, allowing programmers to express concurrent control logic more naturally without worrying about low-level synchronization details.

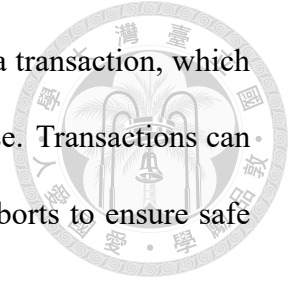
STM is based on the concept of transactions, allowing programmers to bundle a series of operations into a transaction. This transaction corresponds to the critical section in the original computational framework, and it executes atomically, meaning that all the code within the transaction must either succeed or fail entirely—it does not allow for partial success or failure. If the transaction successfully completes, the changes to shared resources are permanently committed. However, in case of conflicts or errors, the transaction is rolled back to its initial state, allowing for retries or other handling. The STM system orchestrates the execution of transactions to ensure the progress of the entire program. Its emergence provides programmers with a higher level of abstraction, making it easier and more intuitive to implement concurrent control in multi-threaded or multi-process environments. It helps reduce dependence on low-level synchronization mechanisms, improves code readability and maintainability, and enables programmers to build parallel computing programs more efficiently.

2.2 Software Transactional Memory

Software Transactional Memory (STM) is a parallel programming technique that achieves correct execution of concurrent programs by managing the execution of transac-



tions. In STM, a segment of code in a program is encapsulated into a transaction, which is executed atomically, similar to executing a transaction in a database. Transactions can include a series of operations such as reads, writes, commits, and aborts to ensure safe access and modification of shared resources.



Transactions in STM include four basic operations:

- **Read.** The Read operation is used to read the value of shared resources. When a transaction wants to read a shared resource, it reads the value of that resource into the transaction's workspace for subsequent operations to use.
- **Write.** The Write operation is used to modify the value of shared resources. When a transaction wants to modify a shared resource, it writes the new value into the transaction's workspace. However, this modification is only visible within the scope of the transaction and has not been committed.
- **Commit.** The Commit operation[8] is used to commit the read and write operations of a transaction, making its changes to system resources permanent. When a transaction successfully completes its operations without conflicts with other transactions, it can be committed, causing the modifications made by the transaction to become visible to other transactions.
- **Abort.** The Abort operation is used to cancel the operations of a transaction, rolling it back to its initial state. When a transaction encounters conflicts with other transactions or encounters other error conditions, it aborts, causing its read and write operations to have no effect on other transactions. The abort operation does not affect the execution of other transactions and allows system resources to revert to a consistent state.

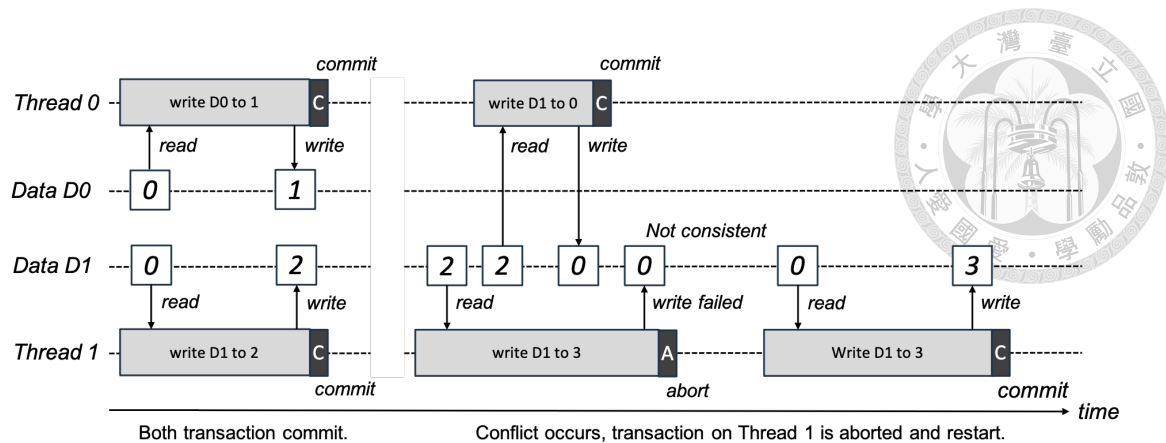


Figure 2.1: Transaction Execution

Through these four basic operations, STM can achieve the safe execution of concurrent programs. Each transaction can freely read and modify shared resources, while STM is responsible for managing the commit and rollback of transactions to ensure resource consistency and reliability. In this way, STM provides a high level of abstraction, making the development of concurrent programs easier and more intuitive.

Figure 2.1 illustrates the scenarios of several transactions. On the left side are examples where both transactions commit successfully. Two transactions executed on different threads perform read and write operations on data D0 and D1 respectively. As there are no conflicts, both transactions are allowed to commit by the system.

On the right side is an example where a conflict occurs, leading to an abort. Both transactions attempt to read and write data D1. Since the transaction on Thread 0 completes its write and commits first, the transaction on Thread 1 encounters inconsistency when attempting to commit. Consequently, it aborts and restarts, performing read and write operations on D1 again. This time, without conflicts with other transactions, the system allows the transaction to commit.



2.3 TinySTM

TinySTM[7] is an open-source STM implementation. It provides a lightweight and easily extensible STM framework, enabling developers to effortlessly utilize TinySTM for researching and testing new STM architectures.

Key features of TinySTM include:

- **Lightweight.** TinySTM is designed with a minimalist approach, consuming minimal system resources and exhibiting lower runtime overhead, making it suitable for resource-constrained environments or high-performance applications.
- **Ease of Use.** TinySTM offers a simple and intuitive interface, allowing programmers to seamlessly integrate transactional memory into their applications and quickly get started.
- **Ease of Development and Extension.** With its modular architecture, TinySTM is easy to extend and customize. Programmers can modify and extend TinySTM according to their specific requirements, adapting it to various application scenarios and needs.

Due to these advantages, many research projects are based on TinySTM for development. For instance, SwitchSTM, proposed in this study, is built upon the foundation of TinySTM. TinySTM provides a stable and reliable base, allowing researchers to focus on developing new parallel computing models or optimization techniques without worrying about the underlying STM implementation details. This has made TinySTM one of the widely-used STM implementations in the academic community.



Chapter 3 Performance Improvement

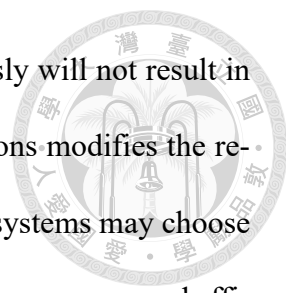
This section begins by discussing the impact of conflicts on the performance and throughput of STM systems, and introduces two mechanism frameworks provided by existing research to reduce the overhead caused by conflicts. Subsequently, it explores the scalability issues that these existing mechanisms may encounter. Finally, it presents the main approach of this study, attempting to propose a new method that can overcome the limitations of existing mechanisms.

3.1 Conflict

In software transactional memory (STM) systems, system performance faces numerous challenges, with one of the most significant being the resolution or avoidance of conflicts. A conflict refers to the competition or clash that arises when two or more transactions simultaneously operate on shared resources. This can lead to performance degradation, data consistency issues, and even system crashes.

Common types of conflicts in STM systems include:

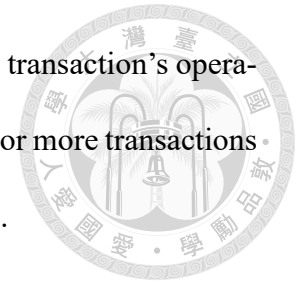
- **Read-Read Conflict.** Two transactions simultaneously attempt to read data from a shared resource. Read operations typically do not modify the resource, so mul-



multiple transactions reading from the same resource simultaneously will not result in inconsistent or erroneous outcomes unless one of the transactions modifies the resource, leading to a Read-Write or Write-Read conflict. Some systems may choose to allow multiple transactions to read concurrently to improve concurrency and efficiency, while others may choose to limit the concurrency of read operations to avoid subsequent Read-Write or Write-Read conflicts and ensure resource consistency.

- **Read-Write Conflict.** One transaction is trying to read data from a shared resource while another transaction is performing a write operation. In this scenario, if the transaction performing the read operation completes before the transaction performing the write operation modifies the value of the resource and commits first, it will result in the read operation obtaining outdated data that is not allowed to be committed by the system. This conflict typically leads to the rollback or retry of the transaction performing the read operation to ensure it does not read inconsistent or outdated data.
- **Write-Read Conflict.** Similar to the Read-Write conflict, one transaction is attempting to perform a write operation on a shared resource while another transaction is executing a read operation. If the read operation does not complete before the write operation is committed, it may result in the read operation obtaining outdated or inconsistent values, leading to the termination of the read operation and causing the transaction to rollback and retry.
- **Write-Write Conflict.** Write-Write conflicts typically occur when multiple transactions are concurrently modifying the same resource. When two or more transactions attempt to write to the same resource simultaneously, a race condition may occur where the write operation of one transaction may override the write result of

another transaction, resulting in the discard or overwrite of one transaction's operation. This conflict typically leads to the rollback or retry of one or more transactions to ensure the consistency and atomicity of the write operations.



The figure 2.1 in the previous section illustrates an example of a Write-Write conflict, where two transactions simultaneously attempt to write to the same data. Without proper synchronization mechanisms, these two transactions may interfere with each other, leading to data inconsistency or incorrect results. This competitive scenario is an example of a conflict.

Conflicts often result in the termination of transactions, rendering the work done by the aborted transactions wasteful and meaningless for the system. Therefore, conflicts are a major factor contributing to poor system performance. To address the impact of conflicts on STM system performance, some research has proposed effective conflict resolution mechanisms[9][12][14]. These solutions typically involve optimizing conflict detection and resolution strategies in STM systems to minimize the overhead incurred when conflicts occur.

On the other hand, some research suggests that instead of waiting until conflicts occur to resolve them, efforts should be made to prevent these conflicts from happening in the first place[4]. These studies often focus on transaction scheduling strategies in STM systems to reduce the occurrence of conflicts and improve system performance.

These two approaches can be classified as contention managers and schedulers. Contention managers focus on handling conflicts after they occur, while schedulers aim to take measures to avoid conflicts before they happen. Both resolving and avoiding conflicts are crucial for achieving efficient STM. They can reduce system overhead and latency, thereby

enhancing system throughput and efficiency, enabling STM systems to better cope with high concurrency and large-scale parallel computing demands.



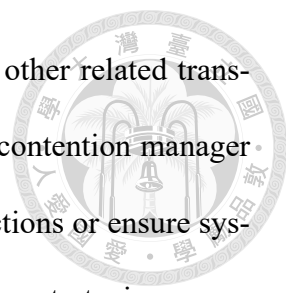
3.2 Contention manager

A contention manager is focused on conflict resolution. It is primarily responsible for monitoring, detecting, and resolving conflicts when multiple transactions access shared resources simultaneously and conflicts occur, to ensure the correct execution of transactions. The main tasks of a contention manager include conflict detection, conflict resolution, and transaction rollback.

Firstly, the contention manager monitors the conflicts among transactions. When multiple transactions operate on the same resource, it detects the existence of these conflicts. Subsequently, based on predefined policies such as transaction priority or other relevant criteria, the contention manager resolves conflicts. This may involve determining which transaction should continue execution and which transaction needs to be rolled back and retried. Finally, if necessary, the contention manager coordinates the rollback operations of transactions to ensure system consistency and correctness.

When handling conflicts, a contention manager typically operates on the behavior of transactions, which can be classified into the following actions:

- **Kill self.** The contention manager marks the transaction it belongs to as needing to be aborted. This means that when a conflict occurs, the transaction is forced to abandon its current operation and undergo rollback or termination. This is usually done to ensure system consistency and integrity, avoiding inconsistent states.

- 
- **Kill other.** This indicates that the contention manager marks other related transactions as needing to be aborted. When a conflict occurs, the contention manager chooses to abort other transactions to prioritize specific transactions or ensure system stability. This is often based on transaction priorities or other strategies.
 - **Halt.** In some cases, the contention manager chooses to suspend the execution of all related transactions to wait for conflict resolution or for the system to return to a safe state. This prevents conflicts from further escalating or leading to more severe consequences.
 - **Restart.** The contention manager rolls back the transactions involved in the conflict to their previous states and then restarts these transactions to re-execute their operations. This action is typically used to restore transaction consistency and correctness after resolving conflicts, allowing the system to continue operation.

These actions are the basic behaviors that a contention manager can take when handling conflicts, and their selection and implementation depend on the system's design and requirements to ensure performance, consistency, and reliability.

Figure 3.1 illustrations showcase several examples of different conflict managers. In the Suicide conflict manager, when a transaction conflicts with others, the conflict manager commands that transaction to execute the "kill self" operation. The Aggressive conflict manager, on the other hand, chooses to restart other transactions when conflicts occur. The Backoff conflict manager combines "kill self" with "halt," similar to Suicide, but it requires the transaction to pause for a random period before restarting to avoid conflicting again with the same transactions, leading to another termination and restart.

Besides deciding whether to abort or continue transactions based on conflict situa-

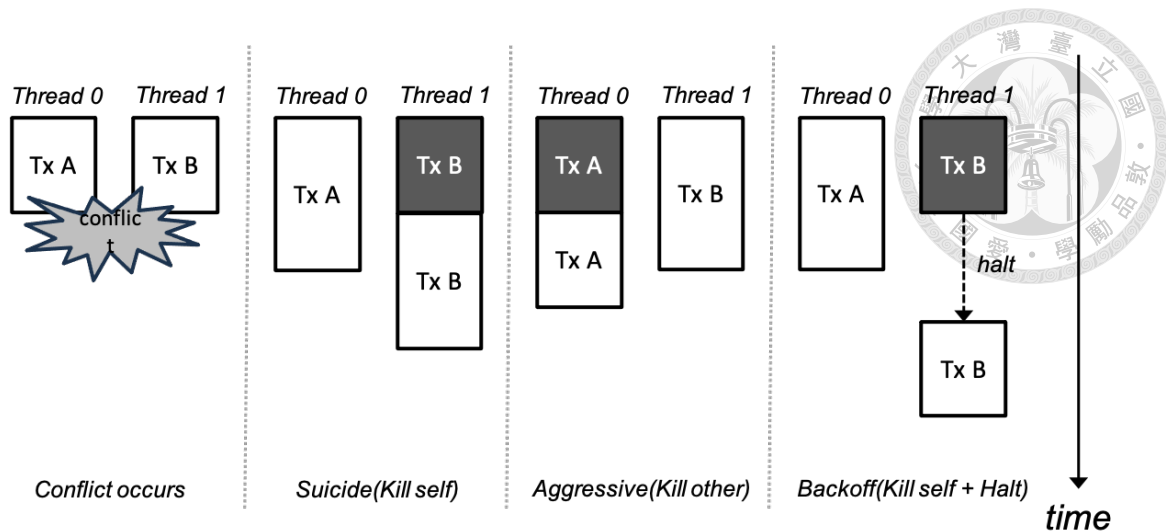


Figure 3.1: Contention Manager actions

tions, conflict managers can also make decisions based on specific decision-making strategies. Karma is a decision-making strategy used by a conflict manager, where transactions are terminated based on the amount of work completed. This means transactions that have completed less work might be terminated to increase the chances of executing transactions that have completed more work, thereby minimizing work wastage caused by conflicts.

Polka combines the advantages of Karma and backoff. Similar to Karma, Polka also decides which transaction to terminate based on the amount of work completed. However, the terminated transaction needs to pause for a period, which is exponentially proportional to the work done. This operation aims to enhance system efficiency, making the terminated transaction more likely to succeed in submission after restarting.

Timestamp is another decision-making strategy that tends to abort younger transactions to ensure the oldest transactions have the highest priority for execution. The purpose is to ensure the system can quickly process the oldest transactions, thereby reducing overall latency and maximizing system performance.

The primary goal of a conflict manager is to minimize the overhead caused by con-

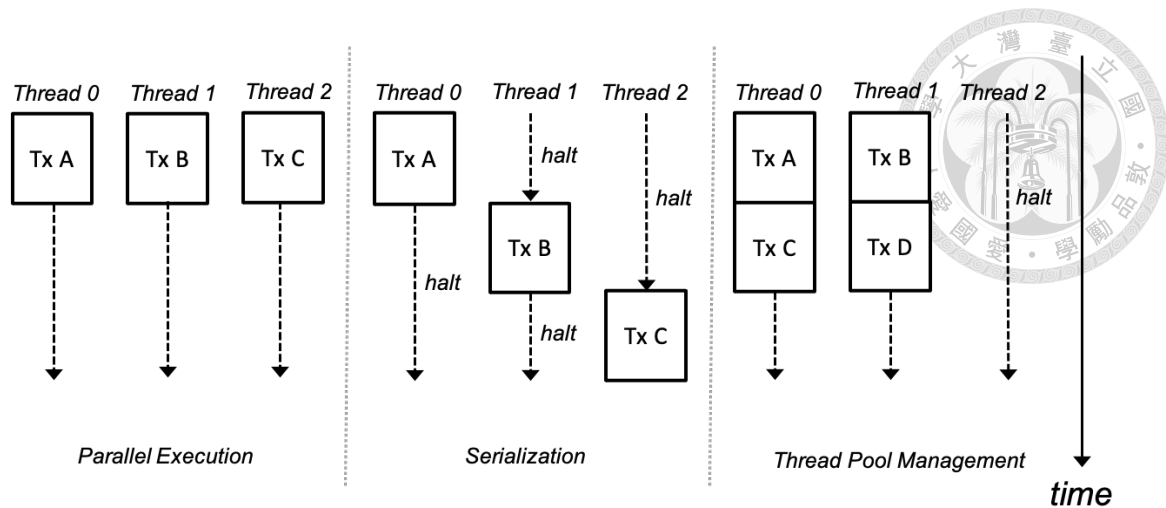


Figure 3.2: Scheduler actions

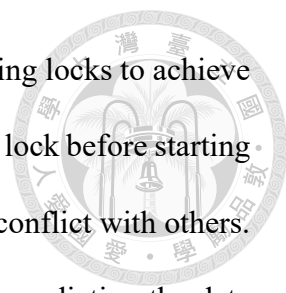
fluctuates to maximize system performance and throughput. To achieve this goal, it can utilize various conflict resolution strategies. These strategies help the conflict manager make the most appropriate decisions when facing conflicts, effectively enhancing system performance and reliability.

3.3 Scheduler

The scheduler[4] focuses on taking measures to prevent conflicts before they occur. It is responsible for scheduling and executing transactions to minimize the likelihood of conflicts. The scheduler typically devises appropriate scheduling strategies based on transaction attributes and system state. These strategies may include adjusting the order of transactions and optimizing resource allocation.

Figure 3.2 examples illustrate various behaviors of schedulers, where the system takes different measures when faced with high conflict rates or specific decision logics.

In the Serialization scenario, the system transforms transactions that could initially run simultaneously into sequential execution. This means that at any given time, the sys-



tem allows only one transaction to execute, typically implemented using locks to achieve serialization. When a transaction needs to execute, it must acquire the lock before starting execution, ensuring that the currently executing transaction does not conflict with others. Shrink is an example of a scheduler adopting Serialization behavior, predicting the data that the next transaction may access based on the transaction's past read or write sets. If this data is being accessed by other transactions, the scheduler may require the transaction to execute serially.

Thread pool management, on the other hand, is an example of resource allocation. In this scenario, the system decides to suspend the use of certain resources (such as pausing some threads), reducing the number of concurrent transactions to decrease the probability of conflicts. F2C2-STM exemplifies Thread pool management behavior, dynamically observing the relationship between available thread count and throughput during program execution and adjusting thread count in the system to maximize throughput. By adjusting resource utilization, the system can more effectively manage conflicts, thereby enhancing overall performance and reliability.

The goal of a scheduler is to identify potential conflict points in advance and take preventive measures to reduce system overhead and improve performance. The design and implementation of schedulers often need to consider system complexity and scalability to ensure their applicability in different use cases.

The selection and implementation of these schedulers depend on the system's requirements, characteristics, and goals. Whether adopting Serialization or Thread pool management strategies, the aim is to optimize system performance and reliability. Through proper design and tuning of schedulers, the system can more effectively manage conflicts, im-

prove overall performance, and ensure smooth execution of various tasks.



3.4 Scalability Issue

Scalability refers to the ability of a system to maintain or improve overall performance and throughput when increasing available resources, such as the number of threads. However, contention managers and schedulers often encounter scalability issues when facing high conflict rates.

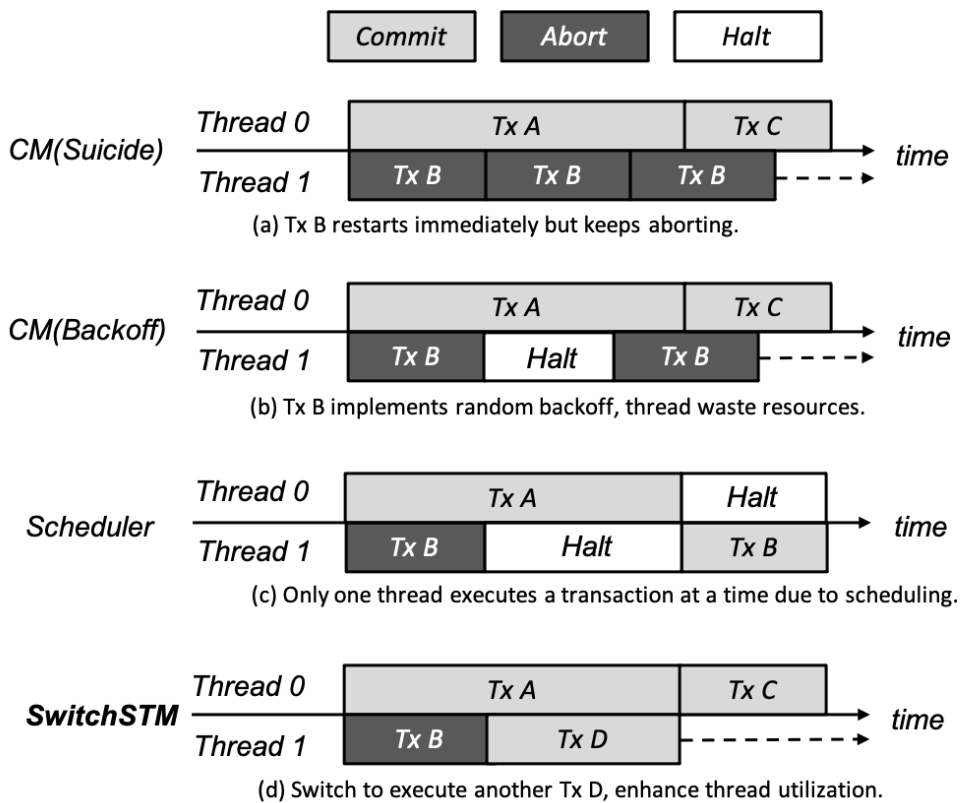


Figure 3.3: Comparison of methods for performance improvement

In some cases within the contention manager, actions such as halting or arbitrarily restarting transactions may result in ineffective work execution or resource wastage. As system resources increase, these issues can exacerbate, as more conflicts and competition lead to a higher number of transactions being aborted or restarted, consequently diminishing system performance and throughput.

Serialization is a common behavior in schedulers where only one thread can execute a transaction at a time, leaving other threads waiting and leading to resource wastage. With an increase in the number of threads, the wasted resources correspondingly increase, thereby affecting system scalability and performance.

Typically, wasted system resources are proportional to the overall system resources. However, within the frameworks of contention managers and schedulers, the utilization of resources gradually decreases as system resources increase, raising concerns about scalability. Effectively utilizing these eventually wasted system resources can contribute to improving system performance.

Based on this premise, this study proposes a novel architecture, SwitchSTM. When conflicts arise, this architecture switches to another transaction. If this new transaction does not conflict with the currently executing transaction, it can effectively perform work, advancing the overall progress of the program. Such an architecture enhances system throughput and resource utilization.



Chapter 4 Coroutine

To achieve the functionality of switching transactions, specific mechanisms need to be designed. This chapter introduces coroutines used to implement transaction switching functionality, along with the coroutine library Libaco that we have adopted.

4.1 Concepts of coroutine

In programming, a coroutine[11] is a special function construct that can pause its execution and resume at a later point in time. Unlike regular functions, coroutines have the ability to suspend and resume execution multiple times within the function without starting from the beginning, which sets them apart from regular functions. This feature makes coroutines particularly useful for tasks that require interleaved execution, such as asynchronous operations, event handling, and multitasking.

The advantage of coroutines lies in their ability to improve program efficiency and performance. Compared to traditional threads, coroutines allow for fast switching between different tasks. Traditionally, a thread can only execute one task within a specific time frame before moving on to the next. However, with coroutines, multiple coroutines can be executed simultaneously within a single thread, and quick switching between these coroutines enables the interleaved execution of multiple tasks.

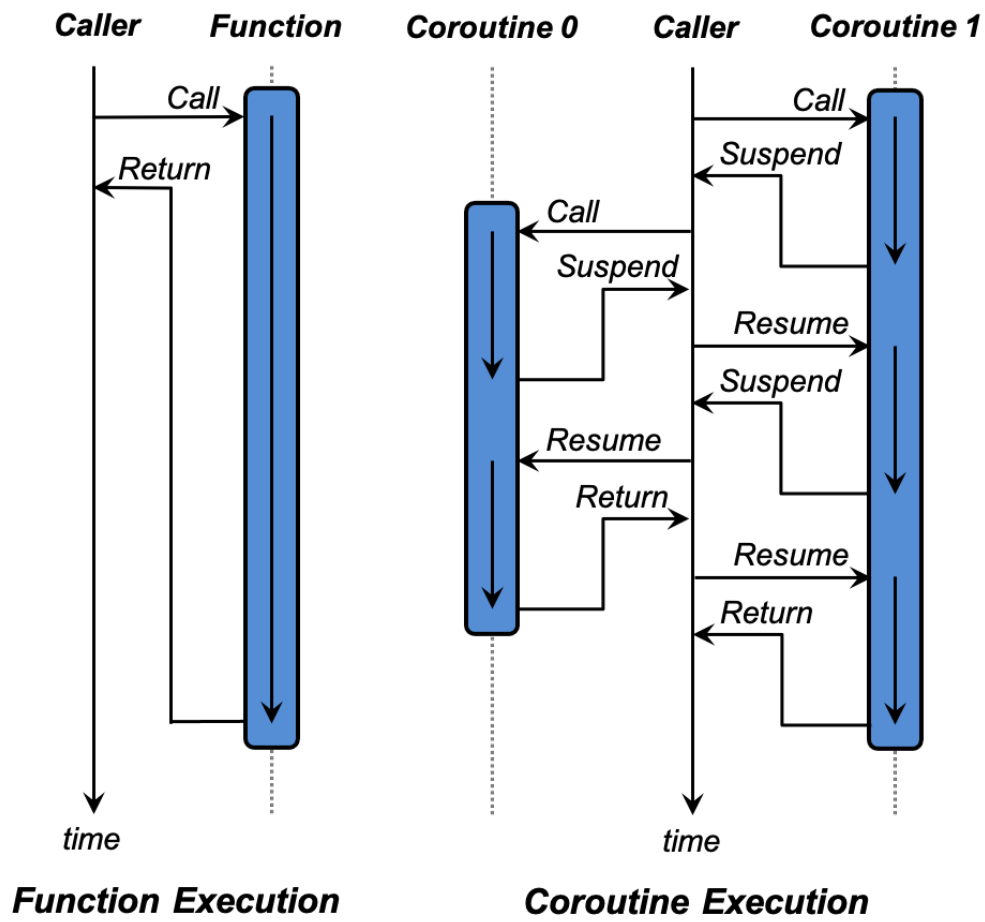
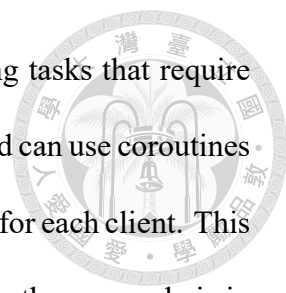


Figure 4.1: function vs. coroutine



This capability makes coroutines particularly useful for handling tasks that require interleaved execution. For example, in a network server, a single thread can use coroutines to handle multiple client requests without needing to start a new thread for each client. This helps save system resources and improves the server's efficiency. Another example is in game engines, where coroutines can be utilized. Games often need to handle various tasks such as rendering graphics, processing input, and computing game logic. With coroutines, a game engine can quickly switch between these tasks within the same thread, enabling smooth game execution and efficient resource utilization.

In practical applications, coroutines can be implemented through libraries or frameworks. For instance, the `asyncio` module in Python provides built-in coroutine support, making it easier to write asynchronous programs. Similarly, many other programming languages and platforms offer similar coroutine support, such as JavaScript's `Promise` and `async/await`, as well as C's `c-coroutine` and `Libaco`. In the STM architecture proposed in this study, the `Libaco` library is used to construct the transaction-switching functionality.

4.2 Symmetric Coroutine vs. Asymmetric Coroutine

Coroutines can be classified into symmetric coroutines and asymmetric coroutines[3], which are two different types of coroutines with notable differences in design and usage:

- **Symmetric Coroutine:**

In symmetric coroutines, all coroutines have equal rights and capabilities. This means that any coroutine can voluntarily yield control to other coroutines and regain control when needed. Under this framework, the switching between coroutines is symmetric, and each coroutine has the potential to control execution. Symmet-

ric coroutines are typically used in cooperative multitasking, where each task can independently decide when to yield control to other tasks.



- **Asymmetric Coroutine:**

In asymmetric coroutines, there is a clear relationship between a main coroutine and non-main coroutines. Typically, the main coroutine holds the control and can actively transfer execution control to non-main coroutines, while non-main coroutines passively wait for control transfer. In other words, in asymmetric coroutines, control of execution is unidirectional. Asymmetric coroutines are often used to represent two different roles, such as producers and consumers in the producer-consumer model, where producers are responsible for generating data and consumers are responsible for consuming data.

In summary, symmetric coroutines allow for equal switching and control among all participants, while asymmetric coroutines involve a clear master-slave relationship, where one coroutine controls the execution of others.

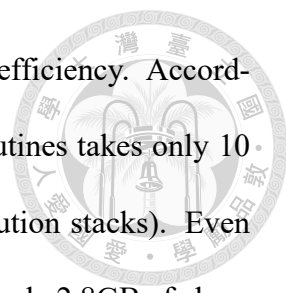
4.3 Libaco

Libaco is an efficient and lightweight C library for asymmetric coroutines, with its name derived from "asymmetric coroutine". Despite its core implementation being less than 700 lines of code, Libaco encompasses all necessary functionalities of a coroutine library. In addition to providing a production-grade implementation of C coroutines, Libaco also offers comprehensive documentation covering the implementation of fast and correct coroutine libraries, including rigorous mathematical proofs.



```
1: /* coroutine_initialization */
2: aco_thread_init()
3:
4: /* create main_co */
5: main_co = aco_create(NULL, NULL)
6:
7: /* create non-main co */
8: co = aco_create(Task, Task_arg)
9:
10: /* run decided co */
11: aco_resume(co)
12:
13: -----coroutine start-----
14: .....//some code
15: /* return to main_co */
16: aco_yield()
17: .....//some code
18: if(Task_structure.isEmpty()):
19:     aco_exit()
20: -----coroutine end-----
```

Figure 4.2: Libaco API



Libaco boasts extremely high performance and memory usage efficiency. According to performance test results, context switching between two coroutines takes only 10 nanoseconds on an AWS c5d.large instance (with independent execution stacks). Even with ten million coroutines concurrently executing, Libaco consumes only 2.8GB of physical memory (using tcmalloc, with each coroutine utilizing a 120B copied stack). Users have the option to create new coroutines with either independent execution stacks or to share one execution stack with any number of other coroutines.

In our research, we chose to use Libaco to build our STM system due to its lightweight and efficient capabilities, as well as its simple and understandable API as shown in the figure. The efficiency of Libaco allows us to implement fast coroutine switching in the system, which is crucial for achieving rapid switching between transactions. Additionally, the straightforward and user-friendly API provided by Libaco makes it easy for us to integrate and use, thereby accelerating the development and deployment process of the system. Overall, Libaco provides us with an efficient, reliable, and easy-to-use method for coroutine switching.



Chapter 5 SwitchSTM

This chapter presents the implementation of SwitchSTM. Firstly, we explain the architecture of Task Iterative STM, followed by an explanation of how coroutine integration facilitates the implementation of SwitchSTM. Furthermore, we delve into the details and correctness of the approach.

5.1 Task Iterative STM

The computational framework of STM encompasses various types. The SwitchSTM proposed in this study primarily targets the optimization of task iterative computational frameworks. This section will elucidate the fundamental structure of task iterative STM and illustrate how this framework interacts with the program through pseudo code.

5.1.1 Task Iterative STM system architecture

The Task Iterative STM architecture is a parallel program execution model based on tasks, where tasks serve as the fundamental units of execution within the system. Each task consists of multiple transactions, representing different sequences of operations within the task. In addition to transactions, tasks may also include non-transactional parts, typically representing tasks that do not require transactional processing.

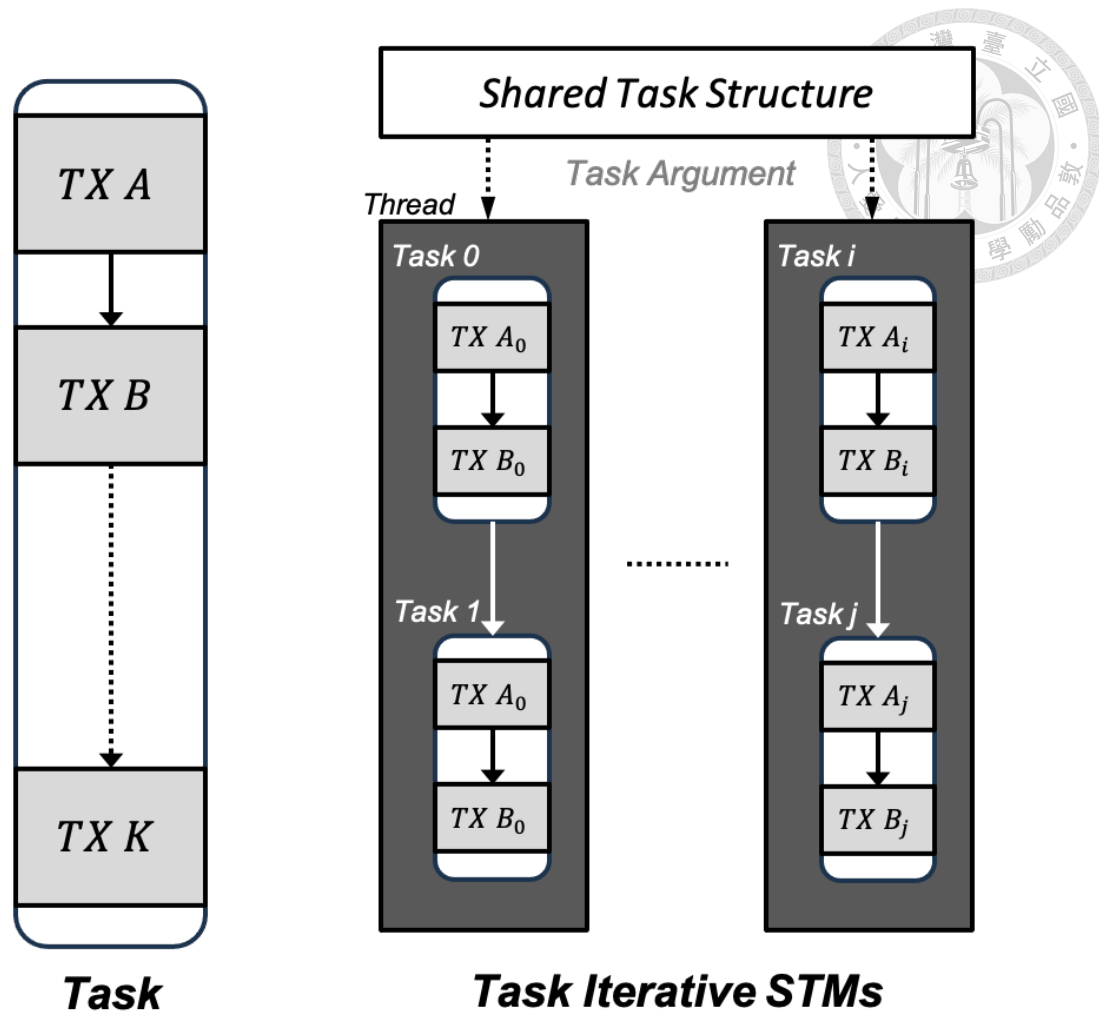


Figure 5.1: Task Iterative STM

The arguments for these tasks, which need to be iterated upon, are stored in a shared task structure, which may exist explicitly or implicitly within the system. STM systems typically allocate a task argument for each thread and enable these threads to interact with the shared task structure for iterative task execution, as illustrated on the right side of Figure 5.1.

The Task Iterative STM architecture comprises the following elements:

- **Task.** As depicted on the left side of Figure 5.1, in Task Iterative STM, a task encompasses a series of operations, which may include multiple transactions as well as other non-transactional operations. Each thread can execute only one task at a

time and iterates through these tasks as needed.

- **Task Argument.** Task Argument refers to the parameters or data used to specify the execution context of a task. When executing a task, each thread determines its specific execution content based on the Task Argument it receives. Task Arguments typically contain the initial data, parameters, and other relevant information required by the task.
- **Shared Task Structure.** The Shared Task Structure is a system-wide structure for storing tasks, which may exist explicitly or implicitly within the system. This structure stores all pending task arguments, and each thread needs to retrieve task arguments from this structure to execute tasks.
- **Thread.** Threads are the smallest units of execution in an operating system, responsible for executing tasks or operations. In the Task Iterative STM architecture, each thread is responsible for executing tasks within the system. Threads retrieve tasks from the shared task structure based on the Task Argument they receive and proceed with execution. Through the concurrent execution of multiple threads, the system achieves more efficient task processing and computation.

In Task Iterative STM, each thread interacts with the Shared Task Structure to retrieve task arguments for iteration. Threads sequentially execute transactions and non-transactional parts of each task until either all tasks are completed or convergence goals are achieved. When convergence goals are met or all tasks are executed, the program's execution is completed.





5.1.2 Task Iterative STM Interaction

The development framework adopted in this research is TinySTM, which employs the Task Iterative STM architecture. Using pseudo code, we illustrate the process as depicted in Figure 5.2.

During the initialization phase, the system initializes STM system and the shared task structure through `STM_init()` and `Task_structure_init()`, respectively. Subsequently, the program enters the parallel region, where `TM_THREAD_ENTER()` indicate the beginning and `TM_THREAD_EXIT()` indicate the end of the parallel code region. Within the parallel code region, each thread continuously retrieves task parameters from the shared task structure until there are no remaining task parameters available for execution. Finally, the system is closed via `STM_exit()`.

```
1: /* STM_initialization */
2: STM_init()
3: Task_structure.init()
4:
5: ----- parallel region start -----
6: TM_THREAD_ENTER()
7: while (Task_structure.isNotEmpty()):
8:     /* fetch a Task argument */
9:     Task_arg = Task_structure.pop()
10:    Thread.runTask(Task_arg)
11: TM_THREAD_EXIT()
12: ----- parallel region end -----
13:
14: STM_exit()
```

Figure 5.2: Task Iterative STM



5.2 Core design of SwitchSTM

This section will provide a detailed explanation of the implementation details of SwitchSTM. We will start by introducing the overall system implementation of SwitchSTM, followed by a thorough exploration of the execution method of transaction switching and the implementation approach of the Switcher. Finally, we will explain the correctness of the underlying logic for switching transaction execution.

5.2.1 SwitchSTM system architecture

The system architecture of SwitchSTM is illustrated in Figure 5.3. Its core concept lies in utilizing coroutines to encapsulate tasks, enabling dynamic scheduling of task execution. This allows the system to efficiently switch between multiple transactions, thereby better addressing the challenges posed by concurrent environments and enhancing overall parallel execution performance.

In SwitchSTM, each thread executes tasks by running coroutines, where these tasks may comprise multiple transactions. The presence of coroutines ensures that the thread's state is maintained during task switches, thus avoiding the overhead associated with conventional thread context switching. Whenever a task completes execution within a coroutine, the coroutine requests a new task argument from the Shared Task Structure to continue executing the next task.

When conflicts arise between transactions in SwitchSTM, the system provides a mechanism to allow the currently executing task to switch to another task for continued execution. This switching behavior is facilitated by the Switcher, which is one of the

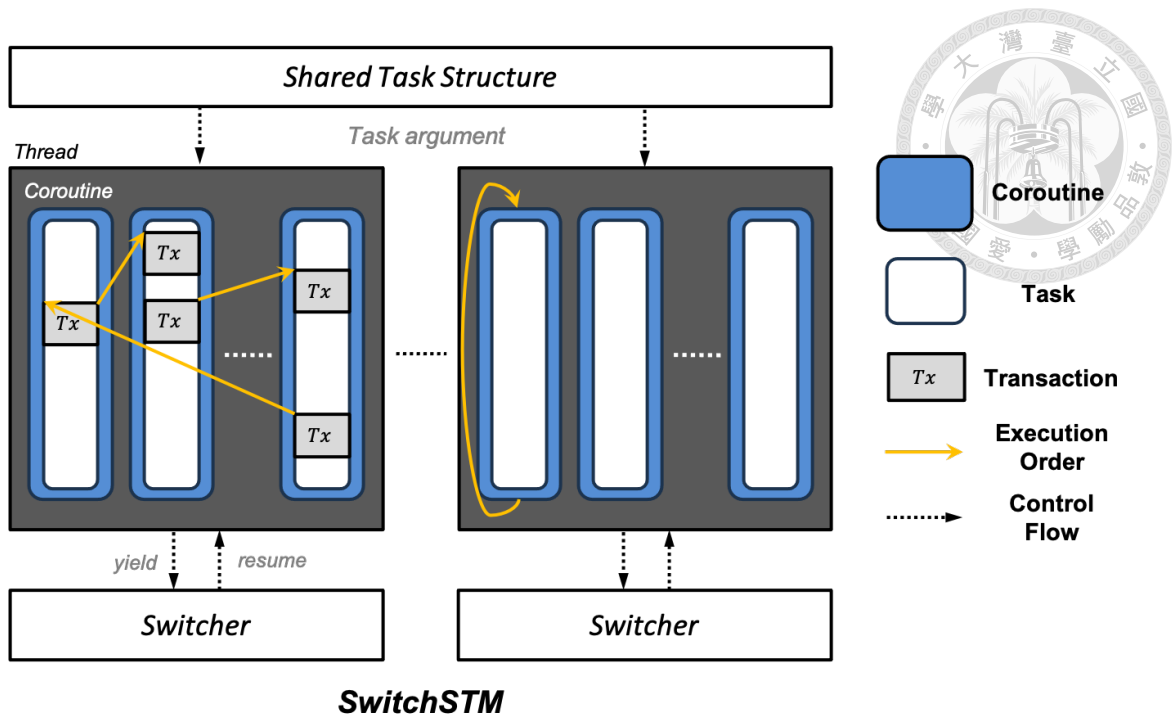


Figure 5.3: SwitchSTM system architecture

core modules of SwitchSTM. When conflicts occur, the transaction relinquishes control to the Switcher, which then determines whether to initiate a switch based on the system's state and predefined policies, or to choose other appropriate actions. This flexibility enables SwitchSTM to dynamically switch to other tasks when handling conflicts, thereby enhancing system efficiency and performance.

SwitchSTM is designed to maximize parallel execution performance. Through the combination of coroutines and the Switcher, the system can swiftly and flexibly schedule task execution, effectively addressing the challenges posed by concurrent environments. The implementation details of SwitchSTM involve in-depth exploration of the execution mechanism of transaction switch and the implementation of the Switcher, which will be elaborated upon in the following sections.



5.2.2 Transaction switching execution

In our system architecture, the execution behavior of transactions is closely related to and highly integrated with coroutines. We have made some adjustments to the execution behavior of transactions to enable switching functionality.

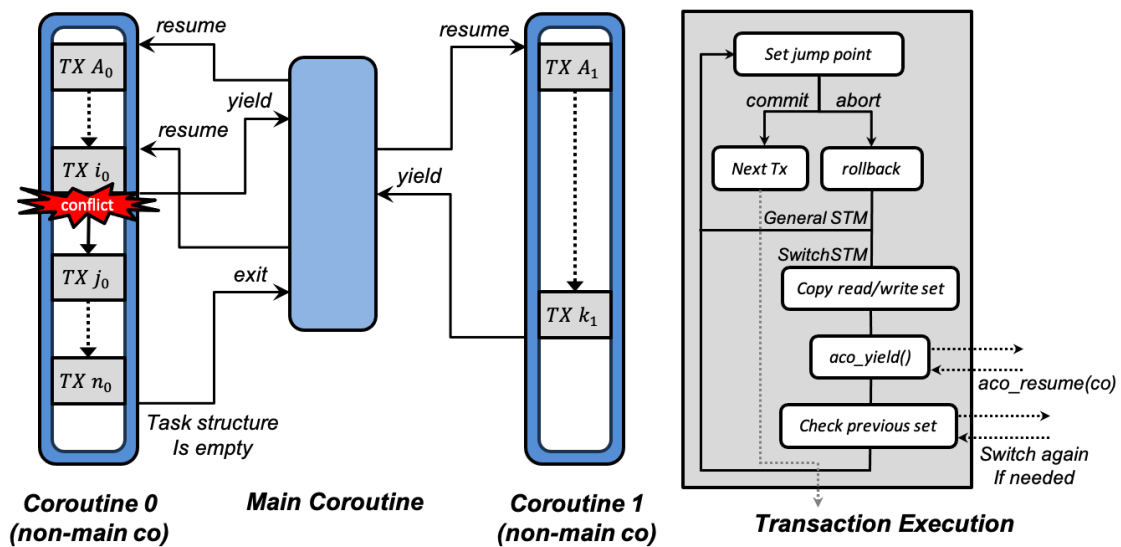
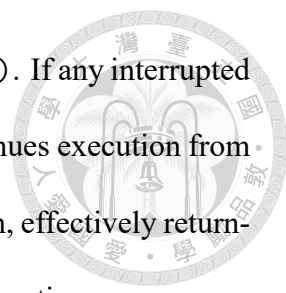


Figure 5.4: Transaction switching execution

As shown in the diagram on the right in Figure 5.4, each transaction sets a checkpoint at the beginning of its execution to record the system environment at that point. This ensures accurate tracking of the execution state during runtime. If a transaction successfully executes and commits, it proceeds to execute the subsequent code in the task or the next transaction to fulfill its corresponding tasks. However, if a transaction is aborted for any reason, a rollback operation is performed. The purpose of rollback is to completely eliminate any traces of the transaction's execution and initialize its state for subsequent operations.

Typically, after a transaction is rolled back, it attempts to restart by returning to the initial checkpoint to re-execute the transaction. However, in our system architecture, inter-



rupted transactions must yield control to the switcher via `aco_yield()`. If any interrupted transaction is selected by the switcher for execution, the system continues execution from the line of code immediately after the `aco_yield()` in that transaction, effectively returning to the initial checkpoint. This design ensures that interrupted transactions can resume execution from the appropriate execution point after switching, thereby guaranteeing the correctness and reliability of the system.

During execution, a transaction uses read sets and write sets to record the addresses it reads from and writes to. In SwitchSTM, each time a transaction aborts, its read set and write set are recorded in a previous set. When a transaction is re-executed, it might access these addresses again, which has a high likelihood of causing a conflict and another abort. Therefore, before switching to a transaction, we first check if the addresses in its previous set are currently being used by other transactions. If other transactions are accessing these addresses, the transaction is marked as unswitchable, and control is returned to the switcher to attempt another switch. This design effectively avoids frequent conflicts, thereby enhancing the overall efficiency and performance of the system.

We utilized the Libaco library introduced in section 4.3 to implement the transaction switching functionality. Libaco is an asynchronous coroutine library with a Main coroutine responsible for scheduling the execution of non-main coroutines. When a conflict occurs, indicating the need for transaction switching, the coroutine encapsulating that transaction (i.e., non-main coroutine) yields control to the Main coroutine, known as the Switcher. The Switcher then determines the next course of action based on the system's state and predefined policies. If a task completes execution and there are no more task parameters available in the Shared Task Structure, the corresponding coroutine also yields control back to the Switcher.

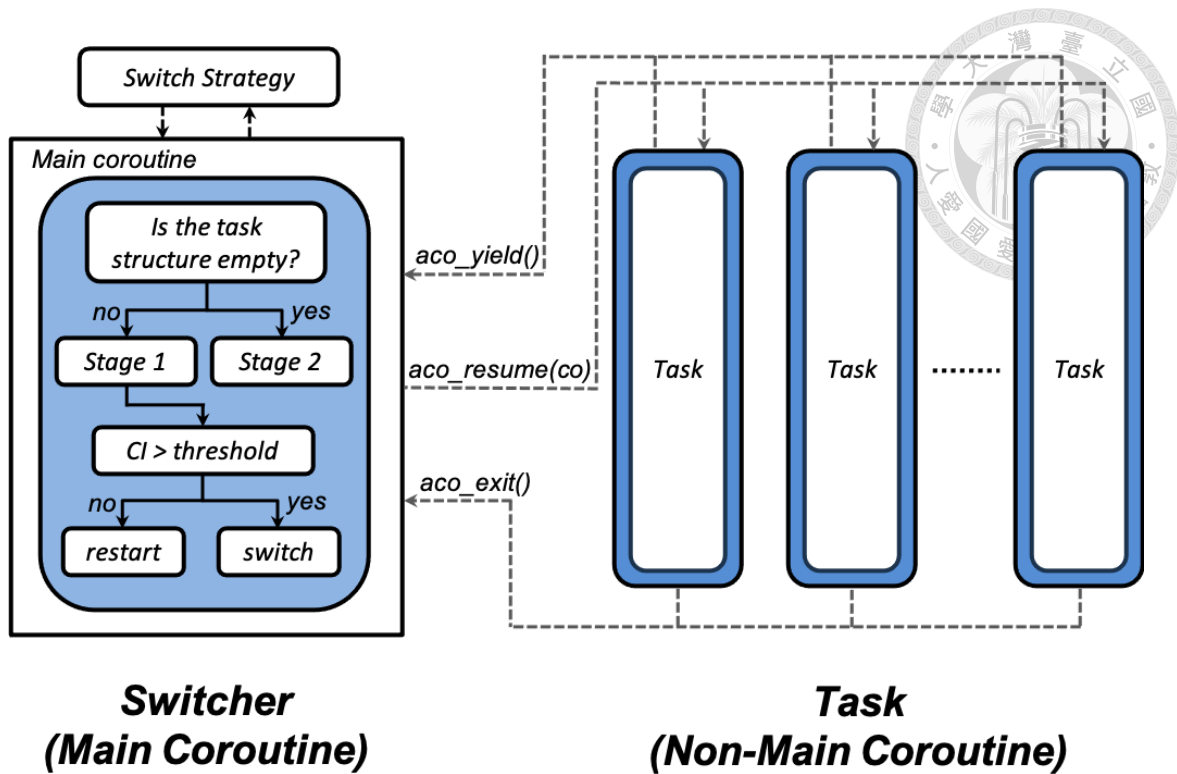


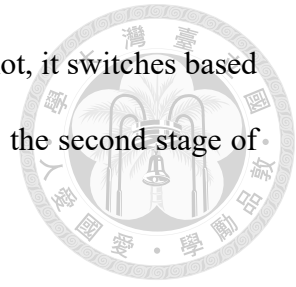
Figure 5.5: Switcher architecture

5.2.3 Switcher implementation

The implementation of the Switcher essentially functions as a main coroutine responsible for managing and scheduling the execution of other coroutines. When a transaction needs to switch, the corresponding coroutine yields control to the Switcher. This enables the Switcher to determine the next course of action based on the system's state and predefined policies.

The Switcher operates using a two-stage switching mechanism. In the first stage, it can switch to any coroutine. In the second stage, it avoids switching to coroutines that have exited execution. When a coroutine completes the execution of a task and the shared task structure becomes empty, indicating that there are no more task parameters available for execution, that coroutine is marked as exited. Whenever control is yielded back to the

Switcher, it first checks if the shared task structure is empty. If it is not, it switches based on the defined switching policy. However, if it is empty, it executes the second stage of the switching mechanism.



In our current implementation, we assume that there is no priority among all transactions, making the choices for switching equivalent. The Switcher can select coroutine to resume without considering priority levels. We adopted three switching strategies:

- **Random Switching Strategy.** Under the random switching strategy, the Switcher randomly selects an available coroutine for execution without being constrained by other factors. This strategy is the simplest and most basic, suitable for scenarios where execution order or specific conditions are not considered. The random switching strategy helps achieve more uniform task distribution because it does not favor any specific coroutine but rather selects the next one to execute randomly.
- **Sequential Switching Strategy.** The sequential switching strategy selects the next coroutine for execution based on its numbering. In other words, each switch chooses the coroutine that has been least recently executed, potentially selecting the one that has been idle for the longest time. This strategy aims to minimize the likelihood of conflicts between the coroutine being switched to and the one currently executing, reducing the likelihood of conflicts with the current transaction.
- **Least Aborts Priority Strategy** The least aborts priority strategy considers the number of times each coroutine's transactions have been aborted. Under this strategy, the Switcher prioritizes coroutines with fewer aborts for execution to minimize the overhead caused by aborts. This strategy is suitable for scenarios where minimizing the total number of aborts in the system is desired, effectively improving

system performance.



These switching strategies can be chosen based on the system's needs and characteristics to achieve optimal performance and efficiency. In addition to considering different switching strategies, it is also important to pay attention to the current level of contention in the system. Our experience shows that performing switches when the system's contention rate is high can more effectively enhance overall system performance. This is mainly due to conflict affinity.

Therefore, we introduced contention intensity (CI) to dynamically measure the current level of contention in the system. During the operation of the STM system, contention intensity (CI) is an indicator that reflects the degree of contention in the system. The CI value is directly proportional to the current level of contention in the system; in other words, the more conflicts there are, the higher the CI value. We will only perform switch operations when the CI value exceeds a certain threshold. This design ensures that in severe contention situations, the switching strategy can better optimize system performance and avoid the extra overhead of unnecessary switch operations. This dynamic adjustment mechanism allows us to respond flexibly to different operating conditions, achieving higher efficiency and better performance.

The main functionalities of Switcher include receiving and handling control from other coroutines, determining coroutine switching behavior based on switching strategies, and deciding whether to execute a switch or perform restart. Since Switcher itself is a coroutine, it can utilize the functionalities provided by the Libaco library for highly efficient coroutine scheduling and switching. The implementation of Switcher enables the system to flexibly manage switches between transactions, thereby enhancing system per-

formance and efficiency.



5.2.4 Correctness of switching

When switching transactions between coroutines, ensuring correctness is crucial. Each coroutine represents an independent task and needs its own local storage to correctly save and restore its state during switches. This local storage includes variables and state information required for each transaction's execution. Shared local storage can lead to state confusion or data corruption when transactions switch between coroutines.

Referencing the example illustrated in Figure 5.6, when executing transaction A on coroutine 0, it may be forced to abort due to certain conflicts or conditions, and then switch to coroutine 1 to execute transaction B. Assuming transaction B is successfully committed and modifies the storage, the system then switches back to coroutine 0 to continue executing transaction A. At this point, if coroutine 0 and coroutine 1 share the same storage space, transaction A may be affected by the storage modified by transaction B. This could result in incorrect or undefined behavior for transaction A because the environment in which transaction A initially executed has been altered by transaction B, making it unable to anticipate the original state, thus leading to erroneous execution results. This highlights the importance of ensuring that each coroutine has its own local storage to maintain transaction consistency and correctness during transaction switches.

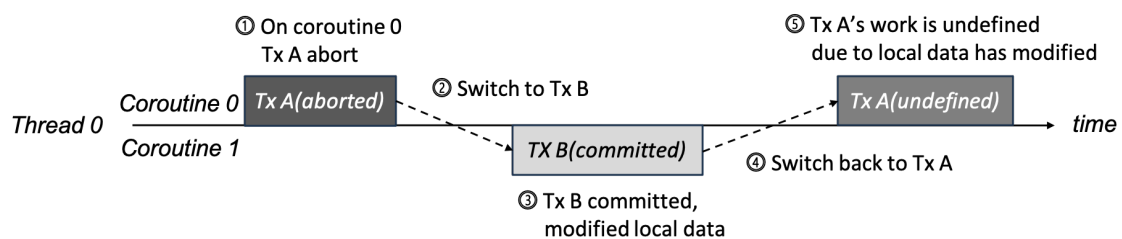


Figure 5.6: Correctness issue: transaction switch example



Chapter 6 Evaluation

In this chapter, we will evaluate the performance of SwitchSTM. First, we will introduce the system configuration and the STAMP benchmarks used in our tests. Finally, we will compare the performance of SwitchSTM with several methods designed to enhance STM efficiency.

6.1 Configuration

The experiments in this study were conducted on a server equipped with dual Xeon Silver 4208 CPUs running at 2.1GHz, with 32GB of DDR4 RAM. The server ran on the Linux kernel version 5.11.0 and utilized GCC 9.4.0 for compilation. The STM library used was TinySTM version 1.0.5, and the benchmark suite employed was STAMP version 0.9.10. Based on experience, we set up five coroutines for each thread and calculate the Contention Intensity (CI) using the formula $CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC$. The Contention Intensity is calculated at each commit or abort. When committing, CC is 0, and when aborting, CC is 1, α is an adjustable parameter, and in our design, we chose 0.5.

Throughout this study, we adhere to the standard STAMP configuration for each benchmark. Detailed parameters can be found in Table 6.1.

Table 6.1: The configuration of STAMP benchmark.

Name	Configurations
bayes	bayes -v32 -r4096 -n10 -p40 -i2 -e8 -s1
genome	genome -g16384 -s64 -n16777216
intruder	intruder -a10 -l128 -n262144 -s1
kmeans-low	kmeans -m40 -n40 -t0.00001 -i random-n65536-d32-c16.txt
kmeans-high	kmeans -m15 -n15 -t0.00001 -i random-n65536-d32-c16.txt
labyrinth	labyrinth -i random-x512-y512-z7-n512.txt
ssca2	ssca2 -s20 -i1.0 -u1.0 -l3 -p3
vacation-low	vacation -n2 -q90 -u98 -r1048576 -t4194304
vacation-high	vacation -n4 -q60 -u90 -r1048576 -t4194304
yada	yada -a15 -i ttimeu1000000.2

6.2 Performance comparison with basic STM

First, we compared the three switching strategies of SwitchSTM with the most basic STM, which uses the suicide contention manager. Under the condition of 32 threads, SwitchSTM achieved an average speedup of 6.41 times. As shown in the figure, as the number of threads increases, the performance of SwitchSTM gradually improves. In contrast, when the number of threads increases, the suicide strategy experiences a rapid decline in throughput due to high conflict rates. However, SwitchSTM can switch to execute other transactions, thereby avoiding performance losses caused by conflicts. The three switching strategies of SwitchSTM show significant performance improvements in most test cases.

Figure 6.1 illustrates the performance improvements of the three switching strategies of SwitchSTM across different test cases, normalized relative to the basic STM. In the test cases, genome and SSca2 are not task iterative types, so SwitchSTM does not perform switching in these two cases, and its performance is roughly the same as the suicide strategy. For the Labyrinth algorithm, the conflict rate between transactions is very low, resulting in a slight performance decrease for SwitchSTM at lower thread counts due to

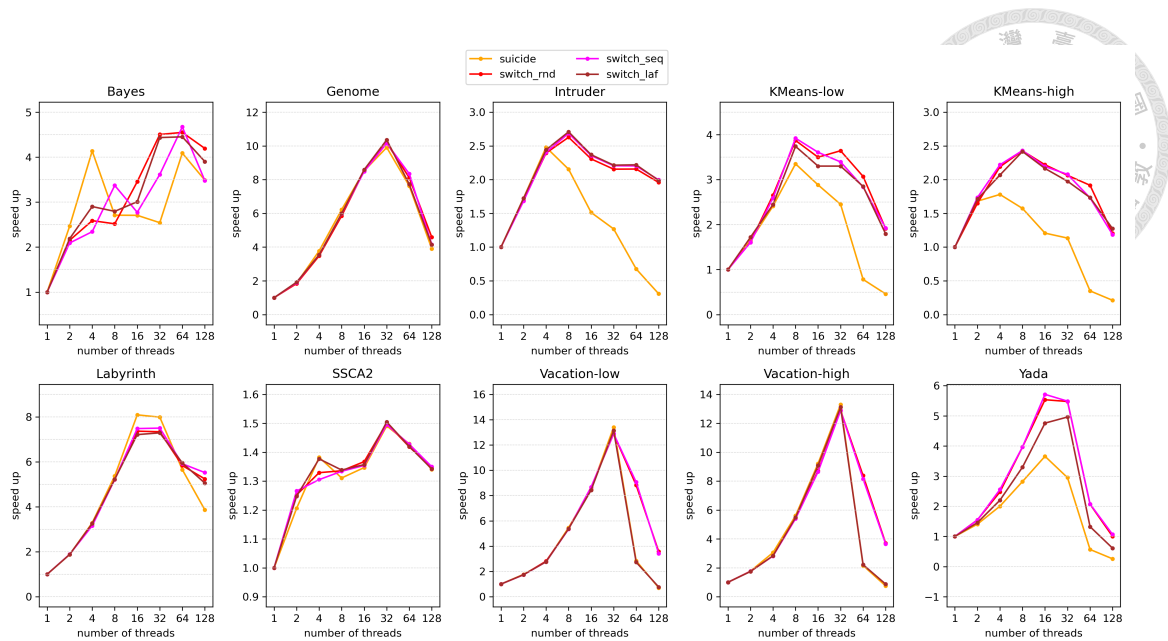


Figure 6.1: Performance improvement comparison of SwitchSTM

system overhead. It is only when the number of threads increases to 128 that a modest performance improvement is observed.

In the bayes test case, it can be seen that as the number of threads increases, the overall performance of SwitchSTM is superior to that of the basic STM. In the yada, Intruder, and kmeans test cases, the advantage of SwitchSTM is more pronounced, with significant performance improvements compared to the basic STM. In the vacation algorithm, noticeable performance improvements are observed only when the number of threads increases to 64 and 128, due to the nature of the algorithm itself.

In summary, SwitchSTM demonstrates excellent performance in most test cases, especially under high concurrency conditions. All three switching strategies effectively enhance system performance. These results indicate that SwitchSTM can efficiently manage and reduce conflicts in high-conflict environments, thereby improving overall system performance.

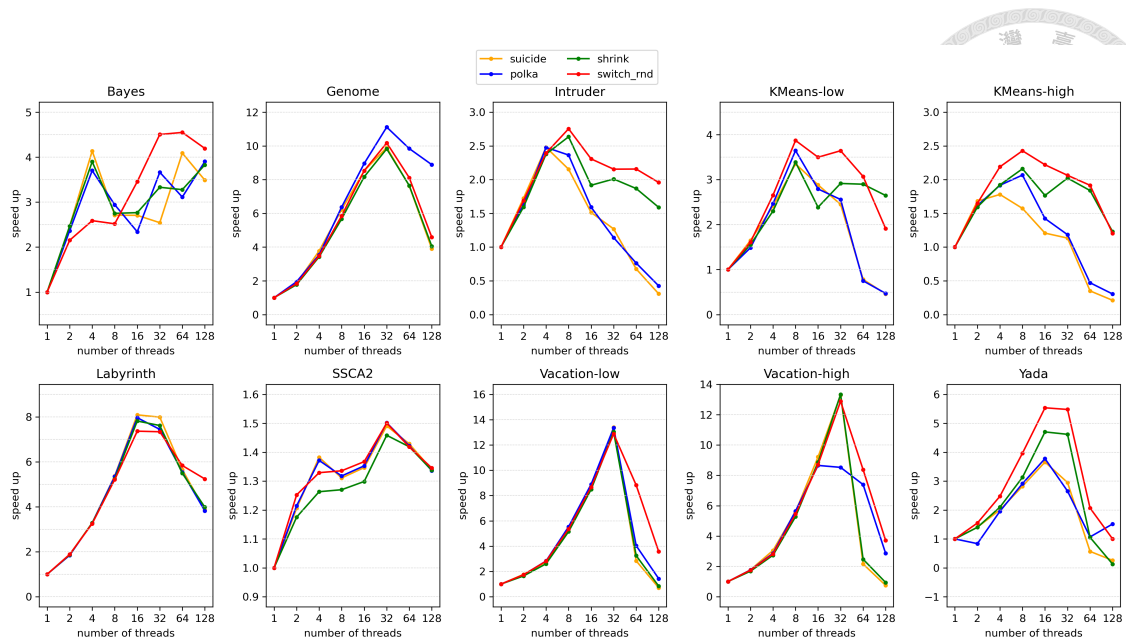
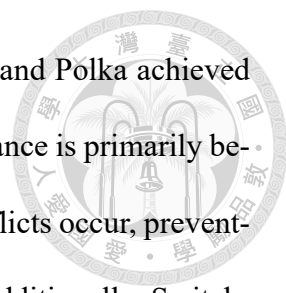


Figure 6.2: Performance improvement comparison with prior work

6.3 Performance comparison with prior work

When comparing the performance of SwitchSTM with other STM systems, we chose the SwitchSTM with a random switching strategy as the representative and conducted a detailed comparison with the Polka contention manager and the Shrink scheduler. The test results showed that although Polka and Shrink both provided some performance improvements over the basic STM, SwitchSTM still performed better overall. In many test cases, SwitchSTM delivered the best performance optimization. Figure 6.2 illustrates the performance improvement comparison.

Specifically, in most test scenarios, SwitchSTM not only managed conflicts between transactions more effectively but also significantly increased the overall system throughput. For example, in the execution of the Intruder algorithm with 8 threads, SwitchSTM achieved a $2.75\times$ improvement, whereas Shrink and Polka only reached $2.63\times$ and $2.41\times$ improvements, respectively. Similarly, when running the yada algorithm with



16 threads, SwitchSTM attained a $5.5\times$ improvement, while Shrink and Polka achieved $4.78\times$ and $3.89\times$ improvements, respectively. This superior performance is primarily because SwitchSTM can switch to execute other transactions when conflicts occur, preventing system stagnation due to conflicts within a single transaction. Additionally, SwitchSTM avoids repeatedly switching to transactions that are highly likely to abort again, further reducing the likelihood of recurring conflicts and thus enhancing system performance.

In high-concurrency environments, this flexible switching strategy of SwitchSTM effectively reduces the impact of conflicts on system performance, demonstrating significant performance advantages. Regardless of whether the conflict rate is high or low, SwitchSTM consistently maintains stable and excellent performance. This makes it a more ideal choice for managing concurrent transactions in most scenarios.

6.4 Abort ratio comparison with prior work

When comparing the abort ratio of these STM systems, we selected the random switching strategy as the representative for SwitchSTM and compared it with the Polka contention manager and the Shrink scheduler. The abort ratio is a crucial performance metric for STM systems, calculated by dividing the number of aborted transactions by the number of committed transactions. This metric reflects the system's efficiency in handling conflicts: the lower the abort ratio, the less computational resources are wasted, thus enhancing overall performance.

Figure 6.3 illustrates the abort ratio comparison. The results show that in most task-iterative test cases, SwitchSTM exhibits the smallest increase in abort ratio as the number

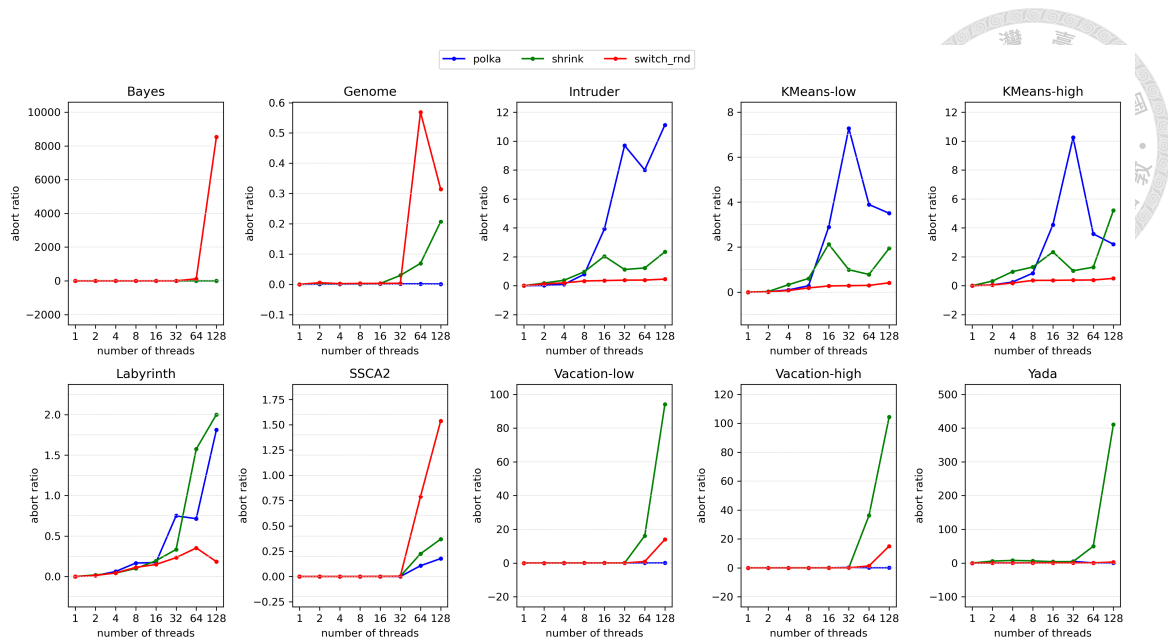


Figure 6.3: Abort ratio comparison with prior work

of threads increases (note that Genome and SSCA2 do not conform to the task-iterative form, and SwitchSTM does not effectively operate in these test cases). This indicates that in high-concurrency environments, SwitchSTM can more effectively reduce the number of transaction aborts. Specifically, as the concurrency level within the system rises, the abort ratio for Polka and Shrink increases significantly, leading to resource wastage and performance degradation. In contrast, SwitchSTM, through its effective switching strategy, avoids switching to transactions that are highly likely to abort again, thereby maintaining a lower abort ratio.

Additionally, SwitchSTM's design enables it to efficiently manage transactions in high-conflict environments, reducing the number of aborts. This not only improves the overall system throughput but also minimizes the computational resource waste resulting from transaction aborts. These results indicate that SwitchSTM outperforms other STM systems across various test scenarios, especially in handling high concurrency and high-conflict situations, where its performance advantages are even more pronounced.



Chapter 7 Conclusion

This paper presents a SwitchSTM system based on the coroutine architecture, achieving flexible task execution switching. In the event of a conflict, SwitchSTM records the resources causing the conflict and immediately switches to the transaction of another task for execution. Whenever a transaction switch is required, the system checks the previously recorded conflicting resources to avoid switching to transactions that are likely to abort again. This design significantly enhances the overall performance and stability of the system, reducing performance losses caused by repeated conflicts.

SwitchSTM is designed to address the problem of transaction conflicts in highly concurrent environments. Specifically, SwitchSTM packages tasks as coroutines and utilizes the flexible switching capabilities of coroutines. This allows each transaction to pause when encountering a conflict and then switch to another coroutine for execution. This approach enables the system to use resources more efficiently, reducing performance degradation caused by conflicts. Additionally, through the flexible coroutine switching mechanism, SwitchSTM can quickly resume normal operations, minimizing the computational resources wasted due to transaction aborts.

This study comprehensively compares SwitchSTM with STM systems based on conflict manager (CM) architectures and scheduler-based STM architectures. The test results

show that, in most cases, SwitchSTM performs exceptionally well across various test scenarios in the STAMP benchmark. Particularly in high concurrency situations, SwitchSTM can significantly reduce the transaction abort rate, thereby increasing system throughput and efficiency. This indicates that SwitchSTM's ability to handle transaction conflicts in high concurrency environments is superior to other STM systems, and its performance improvements have been validated in most test cases.

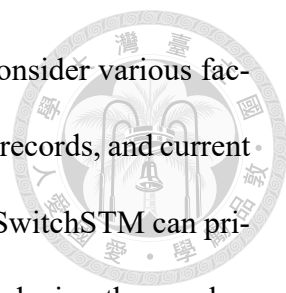
In summary, SwitchSTM demonstrates outstanding competitiveness in managing transaction conflicts and improving system performance. Compared to other STM systems, SwitchSTM performs better in high concurrency environments, proving its potential and advantages in practical applications. The flexible switching mechanism and efficient conflict handling capability of SwitchSTM make it an ideal choice for managing transactions in future high concurrency systems. By effectively reducing transaction abort rates and improving system resource utilization, SwitchSTM not only enhances overall performance but also provides a more reliable and efficient solution for concurrent computing.



Chapter 8 Future Work

This study proposes a computation framework based on switching transaction execution. Building on this foundation, we believe there are several areas of work that can further enhance the performance of SwitchSTM. In the following sections, we will provide detailed explanations of these future works.

- **Dynamic switching strategy optimization.** Future work can focus on developing more advanced dynamic switching strategies to further enhance the performance of SwitchSTM. Currently, SwitchSTM provides three basic switching strategies, but dynamically adjusting strategies based on runtime environment and load can optimize switching behavior in real time. For example, machine learning algorithms can be introduced to analyze the current system state, predict future conflicts and performance bottlenecks, and select the most appropriate switching strategy accordingly. This will make SwitchSTM more flexible and efficient, capable of adapting to different application scenarios and varying workloads.
- **Improved transaction identification.** Enhancing the performance of SwitchSTM can also be achieved by more intelligently identifying and handling transactions that have a high likelihood of aborting. This work can include developing advanced prediction models that use historical data and runtime information to predict which



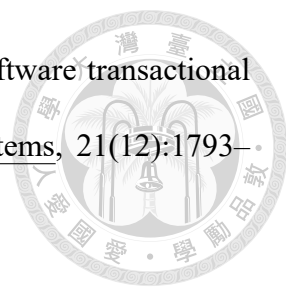
transactions are at higher risk of aborting. These models can consider various factors, such as the read and write sets of transactions, past conflict records, and current system load. By identifying high-risk transactions in advance, SwitchSTM can prioritize other, more stable transactions for execution, thereby reducing the number of aborts and improving overall system performance.

- **Expanding to more STM systems.** Future research can also focus on extending the SwitchSTM technology to a wider variety of STM systems. Currently, SwitchSTM is primarily optimized for specific STM architectures (task iterative), but different STM systems may have different designs and requirements. By adapting SwitchSTM technology to these diverse STM systems, its applicability and effectiveness can be further validated, enabling its implementation in a broader range of applications.
- **Integration with other methods.** Combining SwitchSTM with other performance optimization strategies is also an important direction for future work. STM performance optimization can come from multiple areas, and exploring how to organically integrate SwitchSTM with these strategies can further enhance system performance. For example, we can investigate combining SwitchSTM's switching mechanism with contention management, allowing SwitchSTM to decide which transaction to abort in case of a conflict, rather than simply resorting to suicide. Additionally, integrating efficient garbage collection algorithms could maximize system throughput and response speed while ensuring data consistency. These comprehensive optimization strategies will make SwitchSTM more robust and practical.

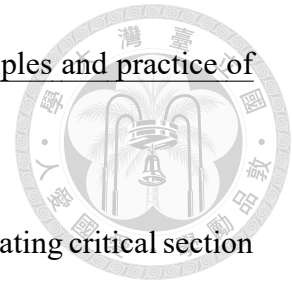


References

- [1] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging, pages 51–60, 2006.
- [2] S. Bethu, P. Srikanth, and M. A. Ahmed. Contention management policy in software transactional memory in parallel and distributed systems. In 2014 Fourth International Conference on Communication Systems and Network Technologies, pages 357–361. IEEE, 2014.
- [3] T. Crolard. A verified abstract machine for functional coroutines. arXiv preprint arXiv:1606.06376, 2016.
- [4] P. Di Sanzo. Analysis, classification and comparison of scheduling techniques for software transactional memories. IEEE Transactions on Parallel and Distributed Systems, 28(12):3356–3373, 2017.
- [5] A. Dragojevic and R. Guerraoui. Predicting the scalability of an stm: A pragmatic approach. In 5th ACM SIGPLAN Workshop on Transactional Computing, 2010.
- [6] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. ACM SIGOPS operating systems review, 37(5):237–252, 2003.

- 
- [7] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. IEEE Transactions on Parallel and Distributed Systems, 21(12):1793–1807, 2010.
- [8] J. Gray and L. Lamport. Consensus on transaction commit. ACM Transactions on Database Systems (TODS), 31(1):133–160, 2006.
- [9] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOOL'05), 2005.
- [10] T. Heber, D. Hendler, and A. Suissa. On the impact of serializing contention management on stm performance. Journal of Parallel and Distributed Computing, 72(6):739–750, 2012.
- [11] A. L. D. Moura and R. Ierusalimsky. Revisiting coroutines. ACM Transactions on Programming Languages and Systems (TOPLAS), 31(2):1–31, 2009.
- [12] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pages 240–248, 2005.
- [13] N. Shavit and D. Touitou. Software transactional memory. In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pages 204–213, 1995.
- [14] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In

Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 141–150, 2009.



[15] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. ACM SIGARCH Computer Architecture News, 37(1):253–264, 2009.

[16] Tabassum and Meenu. Transactional memory: A review. In 2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS), pages 370–375, 2020.