International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 14, Number 2, pages 206–224, July 2024

eSilo: A Secure Transaction Processing System with SGX

Masahide Fukuyama
Faculty of Environment and Information Studies
Keio University
Kanagawa, Japan

Masahiro Tanaka
Graduate School of Media and Governance
Keio University
Kanagawa, Japan

Ryota Ogino
Faculty of Environment and Information Studies
Keio University
Kanagawa, Japan

Hideyuki Kawashima
Faculty of Environment and Information Studies
Keio University
Kanagawa, Japan

Received: February 15, 2024
Revised: May 5, 2024
Accepted: June 6, 2024
Communicated by Yasuyuki Nogami

Abstract

In the cloud computing environment, it is not easy to prove that an adversary with administrator privileges does not attack database systems. To address this issue, EnclaveDB is proposed, which applies an enclave to the database. Its logging mechanism runs sequentially and does not introduce a parallel scheme to exploit modern storage devices with parallel I/O. In this paper, we propose eSilo, which is the Silo transaction processing system with an enclave. The eSilo ensures the confidentiality of sensitive records and procedures by storing, processing, encrypting, and exporting logs inside the enclave provided by Intel SGX. Since standard C/C++ libraries are not supported by SGX, we implemented the eSilo system by replacing the alternative library included in the SGX SDK provided by Intel. We implemented the core of eSilo, extending the CCBench Silo system by adding a logging module. In the experiment with YCSB-A workload, eSilo peaked at 2.30 M tps throughput with sixty worker threads and four logger threads. Our eSilo demonstrated 9.35% performance improvement over the vanilla Silo, thanks to the superior performance of the SGX dedicated library.

Keywords: Silo, transaction processing system, database system, SGX, enclave, security

1 Introduction

1.1 Motivation

Database management systems (DBMS) in the cloud have been popular these days for their elasticity and ease of management. When confidential data is managed on a cloud DBMS, two problems should always be considered. The first problem is the existence of malicious attackers who steal important data from the database. The second problem is the administrators of the cloud providers where DBMS are hosted.

When data are stored and encrypted in the storage device, an adversary cannot read the contents of the data without obtaining a decryption key. Thus, data is secure then. However, to process such data for analysis (e.g., executing relational operators) using a conventional CPU, they need to be decrypted in the main memory. Therefore, during data processing, adversaries can steal data from the memory, which violates system security.

A promising way to cope with this situation is to use the trusted execution environment (TEE). TEE divides computer resources into a trusted and an untrusted area. Sensitive data should be located in a trusted area, and they should be processed there to avoid inappropriate situations. One of the TEEs, **Intel Software Guard Extensions or SGX** [8] provides the trusted area, or an isolated execution environment called *enclave*. A DBMS with SGX can fully protect databases from adversaries, in theory. This is the motivation of this work.

1.2 Problem

There is a seminal work called EnclaveDB [30], which is the extension of the SQL Server that processes and manages sensitive data stored in the relational database exploiting the enclave. EnclaveDB prevents unauthorized access and tampering from the OS administrator or hypervisor by holding and processing sensitive data, query engine, etc. Additionally, introducing a log tampering detection protocol makes it possible to detect if the output logs have been tampered with.

The design of EnclaveDB is based on Hekaton [9], which is a full-fledged database management system. Hekaton adopts the multi-version optimistic concurrency control protocol that exploits many-core architecture, and this concept has shown excellent performance in modern protocols of in-memory database systems [32, 34].

Modern protocols further exploit I/O parallelism and use parallel logging [26, 35, 39] while EnclaveDB adopts single logging. Parallelism can exhibit better performance in theory. However, the design and implementation details of parallel logging using SGX have not been studied in the literature.

1.3 Proposal: eSilo

We believe that designing a secure transaction processing system with an enclave using parallel logging methods is worthwhile since high performance is vital for transaction processing systems. Besides, the code should be open source for academic contribution.

To this end, we propose an extension of the Silo [34] that is the basis of modern high-performance transaction processing systems [21, 36–38] to provide confidentiality using SGX. We refer to our proposed system as **eSilo** in the rest of this paper. To design and implement eSilo, we extended the code of Silo in CCBench [32] which is a set of open-source concurrency control protocols, and the *SampleEnclave* code provided in Intel's SGX software development kit (SDK) [17].

Silo is an in-memory database system, and it stores all the records in memory. Thus, it cannot protect against attacks from adversaries who snoop on memory. To address the issue, our eSilo places all records, including sensitive data, inside an enclave. Thus, eSilo performs all the transaction processing inside the enclave and provides confidentiality for database users. Since thread creation is not allowed inside the enclave, eSilo creates worker threads *outside* the enclave, and each thread enters the enclave to execute transactions, accesses data based on optimistic concurrency control protocols, and performs logging for recovery in parallel, exploiting SSD or NVRAM. In the logging process, a log record is encrypted with a key provided by the database owner before it is moved to

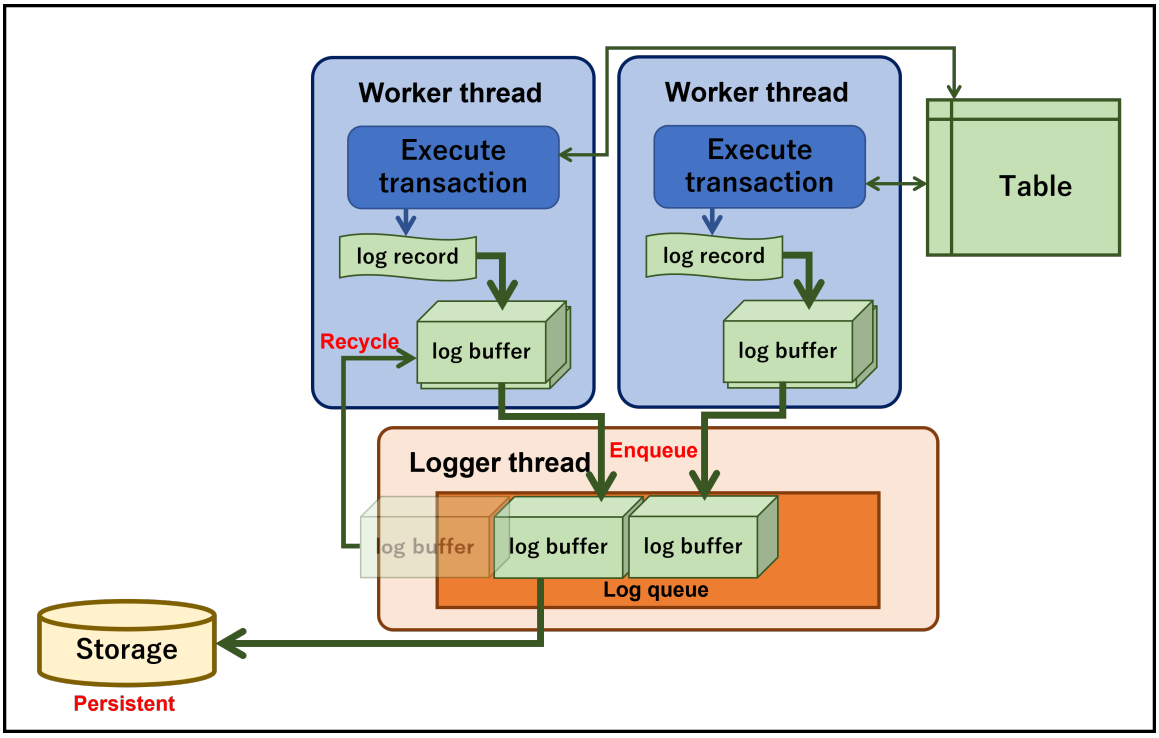

Figure 1: Overview of Silo

a storage device. Then, the encrypted log record is sent outside the enclave and transferred to a storage device to make the log record persistent. Finally, the progress of the durable epoch invokes the notification of commits to users. Our eSilo adopts Masstree [23] for indexing inside SGX to support range queries.

We implemented the eSilo system and evaluated its performance using a many-core machine with 64 cores. The results of experiments showed that eSilo demonstrated 2.30 M transactions per second (tps) throughput with sixty worker threads and four logger threads. The performance was 9.35% higher than that of the vanilla Silo.

This journal paper is the extended version of our previous conference paper [13]. The differences between this paper and the previous one are as follows: (1) The experiments with the Masstree index to accelerate data access. The results are shown in Figure 3-5, 9. (2) The evaluation with varying the number of operations in a transaction, which is shown in Figure 11-16. (3) The integrity checking protocol for parallel logging with SGX, which is shown in Section 3.3. (4) The analysis of performance for g++, tlibcxx, libc++, which is shown in Appendix.

1.4 Organization

The rest of this paper is structured as follows. Section 2 describes preliminaries: Silo, TEE, and the threat model in this study. Section 3 presents the design and implementation of our proposed system, eSilo. Section 4 describes the evaluation of eSilo. Section 5 articulates related work. Section 6 finally concludes this paper.

Algorithm 1 Worker thread

```
Input: queue is the logger's queue
 1: Receive a transaction
 2: Do read phase
 3: Locking records in the write-set {lockPhase}
 4: Read the global epoch
 5: Validating records in the read-set {validationPhase}
 6: Compute Transaction ID
 7: if logBuffer has a space then
 8:   logBuffer.push(log)
 9: else
10:   queue.push(logBuffer)
11:   Switch logBuffer
12:   Raise Logger
13: end if
14: Writing records in write-set to database{writePhase}
15: Unlocking records in write-set
```

Algorithm 2 Logger thread

```
1: while true do
2:   if Wake up call has come then
3:     queue.pop()
4:     Write log to storage
5:     Initialize logBuffer
6:   end if
7: end while
```

2 Preliminaries

2.1 Transaction Processing System: Silo

Silo [34] is a transaction processing system designed for exploiting many-core architectures and large-scale memories. For its novel design and excellent performance, it has attracted attention [7, 21, 36–38].

The concurrency control protocol of Silo is optimistic and has read and commit phases. It does not acquire read locks in the read phase for invisible read [32]. It adopts a decentralized locking mechanism in which each record maintains its locking status without using an expensive centralized locking manager. Silo also uses an epoch system to provide persistence for log records and garbage collection. By default, it is updated every 40ms. Classical transaction processing mechanisms used transaction start timestamps to assign log sequence numbers to determine transaction ordering, but these numbers were generated sequentially, creating a bottleneck [26]. Silo eliminates the bottleneck by having each worker thread issue a transaction ID using epoch.

Fig. 1 shows an overview of Silo. The worker thread that executes transactions is shown in Algorithm 1, and the logger thread that performs logging is shown in Algorithm 2. Silo consists of several worker threads and corresponding logger threads. Each worker thread executes transactions provided by users. The worker thread first completes the read phase and then enters the commit phase (Algorithm 1 - line 5). If the worker thread successfully passes the validation phase, it constructs a log record by using the write set of the transaction and pushes it into the log buffer so that a corresponding logger thread pulls it for logging (Algorithm 1 - line 8). When the log buffer is full, or a new epoch begins, the worker thread enqueues the log buffer to a log queue of the corresponding logger thread (Algorithm 1 - line 10) and wakes up the waiting logger thread (Algorithm 1 - line 12).

A worker thread uses a log buffer at a time and can use multiple log buffers to avoid its blocking.

After enqueueing a log buffer to the logger thread's log queue, the worker thread switches the target log buffer to another one and continues processing the transaction. The logger thread that wakes up writes the log buffer at the head of the log queue to the storage device to persist (Algorithm 2 - line 4). After persistence is complete, the log buffer is returned to the worker thread for recycling and waits for the next wake-up statement (Algorithm 2 - line 5).

2.2 Trusted Execution Environment and SGX

The Trusted Execution Environment (TEE) is a technology that provides a secure execution environment, often referred to as an *enclave*, that is isolated from the operating system or hypervisor through hardware support.

TEE partitions computer resources into trusted and untrusted areas. Sensitive data is then protected and securely processed within the trusted area.

Within the enclave, data is encrypted as it is stored in DRAM and then transferred to the CPU cache, where it is decrypted for processing. This encryption and decryption process is typically handled by a dedicated component, such as a Memory Encryption Engine (MEE) or Total Memory Encryption (TME), which provides confidentiality to prevent unauthorized reads by the operating system or hypervisor.

TEE has been implemented on several platforms. These include technologies available in commercial CPUs, like Intel TDX [15] and AMD-SEV [18], which encrypts the entire VM, and Intel SGX [8, 16], which encrypts part of the application.

In Intel SGX, the conventional version referred to as *SGX1* has a memory limitation, allowing the enclave to use only 128MB of memory space to ensure EPC integrity using dedicated components such as the Memory Encryption Engine (MEE) [8, 14]. However, the latest version, referred to as *SGX2* or *Scalable SGX*, implemented in the 3rd Gen Intel Xeon Processor (code-named Ice Lake) and later versions, employs Total Memory Encryption (TME) to expand the enclave memory space up to 512GB per socket. Yet, this expansion results in a loss of memory integrity protection [2, 16].

2.3 Threat Model

The goal of this study is to improve database performance by leveraging parallel logging while maintaining data confidentiality. We primarily consider the strong adversary defined by Cipherbase [1] as our threat model. A strong adversary has the ability to observe the contents of the disk, memory, CPU bus, communications, operating system, and hypervisor on the server at any given time. However, they cannot observe the state and computation within the enclave.

Confidentiality is also ensured for logs that are output to untrusted storage to ensure database persistence because they are encrypted. The adversary can only view the cipher text.

Cipherbase also defines the weak adversary that can only access disk or memory once to take a snapshot (e.g., through a cold boot attack). Since the memory is encrypted within the enclave, confidentiality is maintained in this case as well.

Both strong and weak adversaries are considered passive adversaries, which can only be observed. In contrast, an active adversary can observe and manipulate the computational logic and data. The enclave is not resistant to tampering due to its vulnerability to SGX side-channel attacks [6, 24] and the lack of integrity-checking methods. Therefore, our scope is limited to strong adversaries, and consideration of active adversaries is left for future work.

3 Proposal: eSilo

3.1 Research problem

In vanilla Silo, sensitive data and the transaction processing engine are located in the main memory. Therefore, an attacker with administrator privileges can steal sensitive data and programs. Thus, confidentiality and integrity are not guaranteed. Readers may think that just porting Silo for the enclave clearly diminishes this problem. However, it is difficult for the two reasons described below.

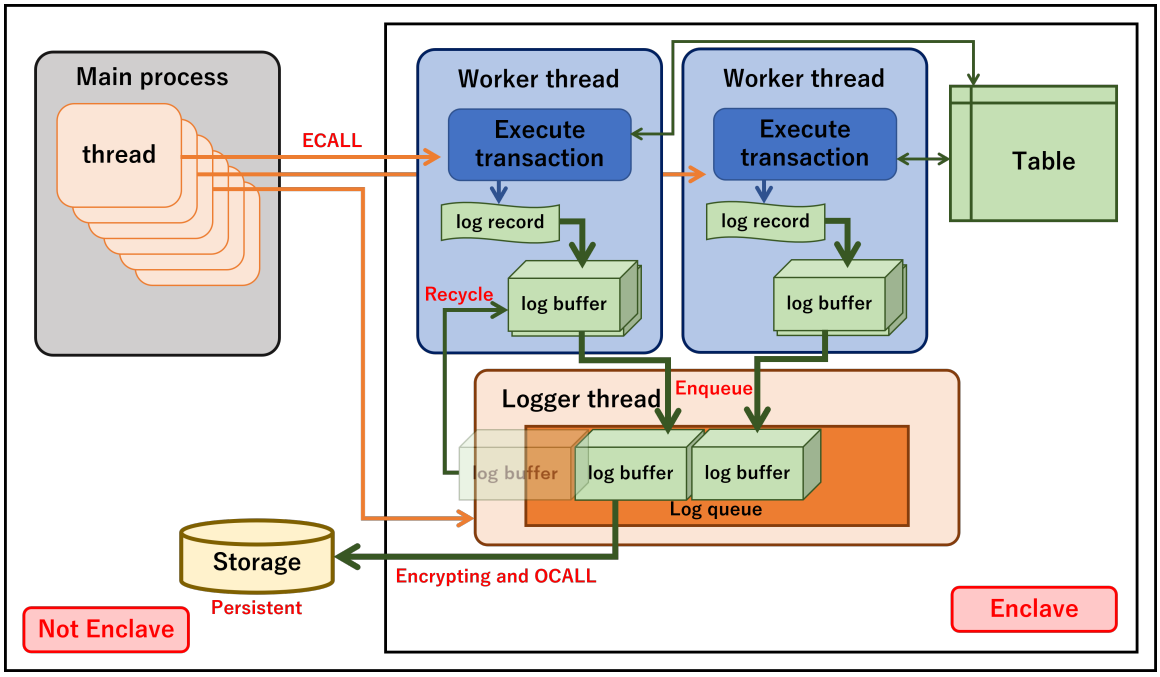

Figure 2: Overview of eSilo. It has potential performance bottlenecks compared with vanilla Silo. Due to the SGX specification, eSilo creates worker threads outside the enclave, and each thread enters the enclave to run both the worker threads and the logger threads. Besides, file access is not allowed inside the enclave, and thus, eSilo first generates log records inside the enclave and encrypts them, passes them to the outside of the enclave, and stores them on the device.

- Due to the inherent property of SGX, it requires a clear specification of the program and data to be protected or not.
- Since SGX is designed to be isolated from the operating system, it prevents the use of system calls or any functions that depend on them. This requires an architectural redesign to determine the level of dependency on OS functionality and re-implementation of functions that become unusable by the SGX specifications.

3.2 Architecture Design

Fig. 2 illustrates an overview of the eSilo design. The key components of eSilo are similar to that of Silo. However, it contains potential performance bottlenecks because of the SGX specification that threads cannot be created inside the enclave. For this reason, eSilo creates worker threads *outside* the enclave. Using *ECALL*, each thread enters the enclave to process both the worker threads and the logger threads.

Besides thread creation, file access is not allowed inside the enclave. Thus, in our design, eSilo first generates log records inside the enclave and encrypts them. Then it passes them to the outside of the enclave using *OCALL*. Finally, eSilo transfers the encrypted log records from memory to a storage device in parallel to make the logs persistent. This parallel logging protocol is based on the SiloR protocol [39], which is the logging system of the Silo.

To implement the design of eSilo described above, we separated a Silo system into two modules. The first module requires protection with an enclave, and the second does not. We also changed the logger threads that transfer log records into storage devices. We show the behavior of the logger threads inside an enclave in Algorithm 3, and that outside an enclave in Algorithm 4.

A logger thread first encrypts log buffers pushed with a key provided by the database owner after database initialization (Algorithm 3 - line 4). This key is encrypted and stored using a hardware

Algorithm 3 Logging algorithm (*inside* enclave)

Input: key is the encryption/decryption key specified by the database owner after database initialization.

1: $encData = \textbf{EncryptData}(logData, key)$
2: $OcallStore(threadID, encData, size(encData))$
3: $Free(encData)$

Algorithm 4 Logging algorithm (*outside* enclave)

Input: $threadID, encData, encSize$
Function: OcallStore

1: $f \leftarrow getFileDescriptor(threadID)$
2: $s = 0$
3: **while** $s < encSize$ **do**
4: $write(f, pointer(encData) + s, encSize - s)$
5: $s+ = r$
6: **end while**

key (e.g., a sealing key derived from the root sealing key). Then, it passes the encrypted log buffers outside the enclave (Algorithm 3 - line 5). Finally, it transfers them to the storage device (Algorithm 4 - line 4), and the log records in the log buffers are persisted [8]. Please note that a record encrypted by sealing in an enclave E can only be decrypted in E.

3.3 Log Integrity and Recovery Protocol

To enhance fault tolerance in databases, it is essential to ensure log persistence by writing to storage devices. Since logs contain sensitive data such as keys and values, they must be encrypted to ensure confidentiality. Additionally, since administrators can potentially tamper with storage data, verifying the integrity of logs is crucial. While EnclaveDB proposes an efficient protocol to ensure log integrity, it is designed for sequential logging and thus cannot be applied to systems that utilize parallel logging.

In this paper, we propose a log integrity verification protocol designed for parallel logging. Our recovery protocol employs a three-tiered hash chain to ensure the integrity of log data. The three-tiered hash chain is composed of the following elements:

Log-Level Hash Chain Each log within the same epoch carries the hash of the previous log, forming a circular hash chain with the head of the chain holding the hash of the last log.

Epoch-Level Hash Chain Logs generated by the same logger thread contain the hash of the previous epoch's log set, creating a unidirectional hash chain.

Thread-Level Hash Chain Each logger thread links a unidirectional hash chain to both credential information and the epoch file, thereby constructing a circular hash chain.

The recovery system operates using epoch files and log files created by each Silo logger thread. The epoch file contains the durable epoch indicating the most recent and safe state of the system and the hash values corresponding to the last logs of each log file. Each log record within the log files includes the operation type, keys, values, and the hash of the previous record and is encrypted by leveraging Intel SGX's sealing capabilities.

Initially, the durable epoch is read from the epoch file. This epoch, periodically calculated and updated by Silo, indicates the most recent state of the system recoverable through logs. Then the hash values of the last logs generated by each logger thread are read.

Each log file contains encrypted log entries. The recovery system retrieves logs corresponding to the current epoch from these files, using epochs incremented up to the durable epoch.

Once the logs for the current epoch have been loaded from each log file, the integrity of each log set is checked. A log set consists of one or more log records, and each record contains a hash calculated from the information in the previous log record (the first log record in the set contains the hash of the last log record). Recalculating these hashes confirms the overall integrity of the log set.

After the log-level integrity check, the system performs epoch-level checks to ensure that each log set is part of the continuous unidirectional hash chain created by the same logger thread.

To integrate the epoch-level hash chain, each logger thread ties the beginning of its logs to credential information and the end to the epoch file. By verifying this structure, it is possible to ensure the overall integrity of the log data, which can be recovered back to the durable epoch.

3.4 Implementation

Our eSilo implementation is based on the CCBench Silo system [32]. Since it has only a concurrency control module, we additionally implemented the logging module, including logger threads and log buffers based on the SiloR protocol [39]. Our code is divided into two modules. The first one includes enclave-based codes, and it includes concurrency control protocol, Masstree index, optimistic cuckoo hashing, log record generation, and notifier. The second one includes initialization, thread activation, and log record synchronization.

Masstree is a concurrent index tree and is mandatory for range search queries with O(log N). The typical database requires range search, and Silo itself uses Masstree. So we adopted Masstree. It combines the self-balancing and high-efficiency order maintenance of B+ trees with the quick search capabilities of the trie tree through prefix matching. Masstree consists of one or more layers of B+ trees, where each layer indexes keys segmented every 8 bytes. Optimistic cuckoo hashing (OCH) is a hash table that is only beneficial for point search queries with O(1). Hash can not be used for range search but is faster than tree. So we adopted a popular hash (OCH) for our work. It utilizes optimistic concurrency control to enable concurrent operations. It uses two hash functions and a two-dimensional table, ensuring that any stored data is located at one of two predetermined positions determined by the results of hashing the key.

The LoC for eSilo and Masstree are around 2800 and 2200 in C++, respectively. Each log record contains TID, key, and data.

Transaction ID (TID) is a 64-bit integer used to identify transactions and maintain the state of records. The TID is structured as follows [34]: The upper 32 bits represent the epoch. The middle 29 bits identify transactions that were committed within the same epoch. The lower 3 bits indicate the record's status, showing whether it is locked, in its latest version, or has been deleted. Each record holds a TID, which is updated by the most recent transaction that modified the record. After the validation phase, a worker calculates the smallest TID that (a) is greater than the TID of any record read or written by the transaction, (b) exceeds the worker's most recently selected TID, and (c) falls within the current global epoch. This calculation is decentralized, allowing each worker thread to independently calculate its own TID.

Entering or leaving an enclave, we need to cross the boundary of an enclave. To describe codes for this purpose, we used the Edger8r tool, which generates interfaces between inside and outside of an enclave provided by SGX SDK [17].

We also implemented a vanilla Silo system that does not use enclaves. It is referred to as **vanilla Silo** in this paper. These implementations are publicly available [11, 12].

4 Evaluation

4.1 Environment

The machine used to evaluate the performance and the SGX SDK are shown in Table 1. The CPUs are two Intel Xeon Gold 6326 2.90 GHz with 32 physical cores, and the machine has 64 logical cores in total. DRAM size is 256 GB, and all the databases are placed on DRAM. Thus, undesirable I/O by swapping does not occur.

Table 1: Evaluation Environment

Device	Description
CPU	Intel Xeon Gold 6326 2.90 GHz Physical cores in a socket: 16 Total physical/logical cores: 32/64 L1d/L1i/L2/L3 cache: 1.5MiB/1MiB/40MiB/48MiB
DRAM	256GB (DDR4 3200 REG ECC 16GB × 16)
OS	Ubuntu 20.04.6 LTS
SGX SDK	sgx_linux_x64_sdk_2.23.100.2 for ubuntu20.04-server
SSD	Samsung SSD 980 PRO 1TB

The number of data access operations (i.e., read or write) in a transaction is 10. The workloads are YCSB-A (read/write: 50%/50%), YCSB-B (read/write: 95%/5%), and YCSB-C (read/write: 100%/0%). The number of records in the database is one million. The data access skew is set to 0, which is under a uniform distribution and provides low contentions. The measurement time is 10 s.

4.2 Basic Performance

4.2.1 Non-Skewed Case

The results of experiments with Masstree are shown in Fig. 3, 4, 5 respectively. In all the experiments, eSilo exhibits better performance than the vanilla Silo. The difference in performance increases as concurrency increases. The performance of YCSB-A is relatively low compared with other two workloads because it includes more write operations that require the generation of log records and storage access.

It should be noted that both eSilo and vanilla Silo do not encounter aborts because the skew of these experiments is zero. In this case, a transaction accesses only ten records among one million records, and the concurrency is at most 60. Thus, the access space of transactions is sparse, and transactions rarely conflict there.

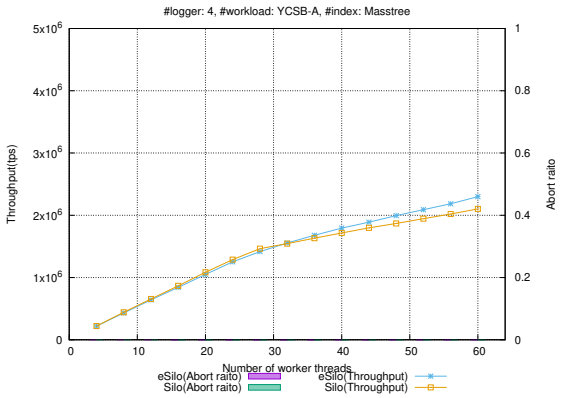

Figure 3: YCSB-A (read/write:50%/50%) with Masstree

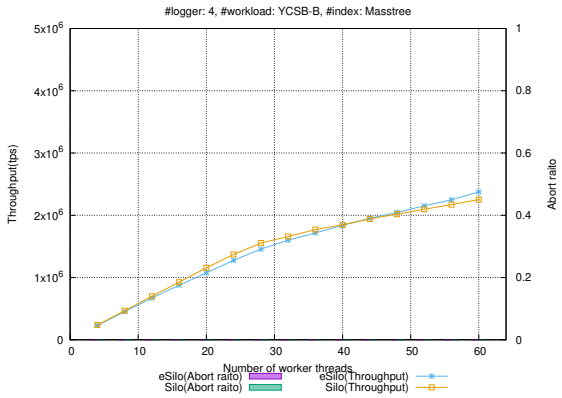

Figure 4: YCSB-B (read/write:95%/5%) with Masstree

The results with optimistic cuckoo hash [20] are shown in 6, 7, 8 respectively. Similar to the results with Masstree, eSilo outperforms vanilla Silo. The performance of these results is higher than that of Masstree. This is because the structure of the hash is more simple than that of the tree. The complexity for the hash is $O(1)$ while that of the tree is $O(log\ N)$.

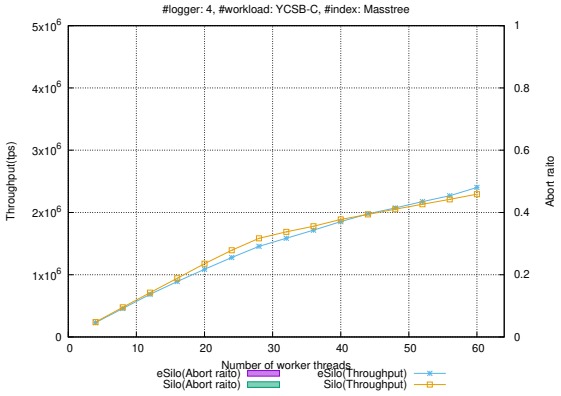

Figure 5: YCSB-C (read/write:100%/0%) with Masstree

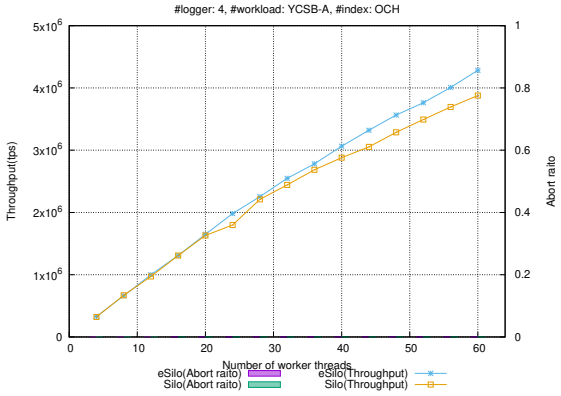

Figure 6: YCSB-A (read/write:50%/50%) with optimistic cuckoo hash

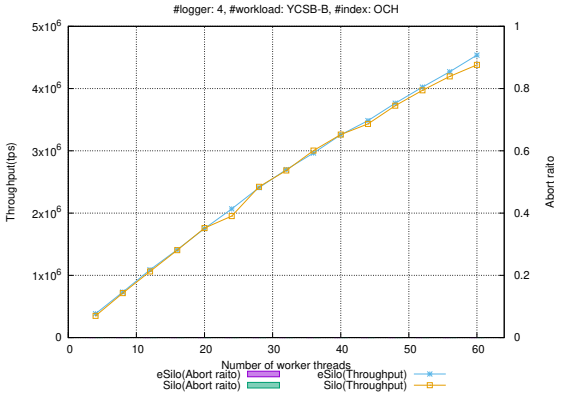

Figure 7: YCSB-B (read/write:95%/5%) with optimistic cuckoo hash

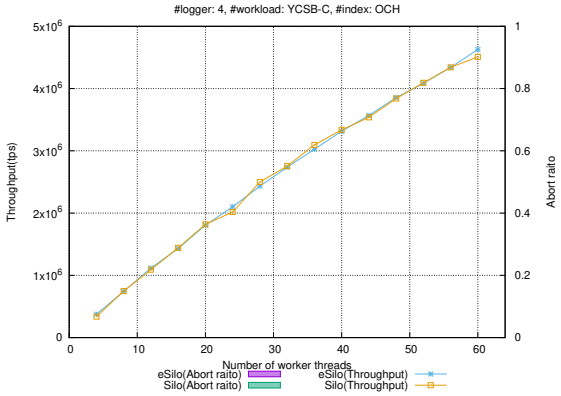

Figure 8: YCSB-C (read/write:100%/0%) with optimistic cuckoo hash

Table 2: The performance of each vector library. Measured 100 times using index and iterator to search in ascending order on a vector array with 10 million elements.

	tlibcxx (SGX)	STL
Index loop	20ms	20ms
Iterator loop	38ms	96ms

4.2.2 Varying Skew

When the skew parameter is zero, transactions rarely encounter conflicts. With a higher skew, transactions are inclining to conflict, in theory. The results of experiments with varied skews are shown in Fig. 9 and 10. As can be seen there, as skew increases, throughput deteriorates while the abort ratio increases. Still, eSilo exhibits higher throughput than that of vanilla Silo.

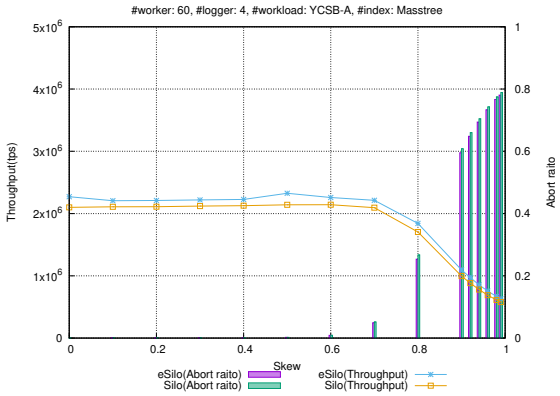

Figure 9: YCSB-A (skew:0.00∼0.99) with Masstree

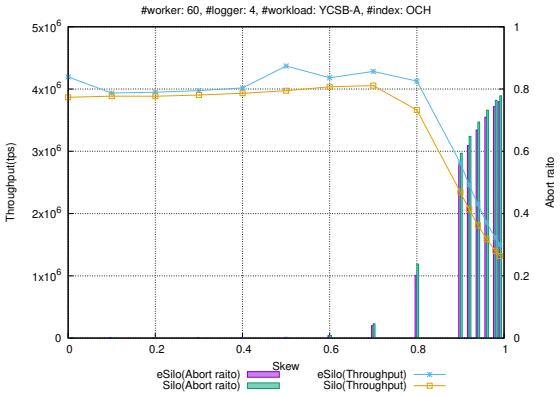

Figure 10: YCSB-A (skew:0.00∼0.99) with optimistic cuckoo hash

4.2.3 Varying Operation Size

A transaction can issue multiple operations. With more operations, a transaction requires more time to finish, which increases the possibility of conflicts with other transactions, in theory. We varied the number of operations in a transaction and measured the throughput for both Masstree and OCH, as shown in Figures 11, 12, 13, 14, 15, 16 respectively.

Due to encryption overhead, eSilo exhibits slower performance than vanilla Silo when transactions involve a small number of operations. However, as the number of operations per transaction increases, eSilo becomes faster, thanks to the library performance.

4.3 Discussion

4.3.1 Why eSilo is Faster?

All the experiments above showed that eSilo is more efficient than vanilla Silo. This reason is not trivial because eSilo requires additional functions to provide security based on SGX. To clarify it, we measured the performance of basic libraries in implementing the systems. We chose the *vector* library and measured search performance using an index or an iterator over ten million elements. We show the result in Table 2.

Surprisingly, tlibcxx (SGX) exhibits a comparable performance with STL in the index and more than twice the performance of STL in the iterator. We think this is caused by the excellent implementation of the tlibcxx.

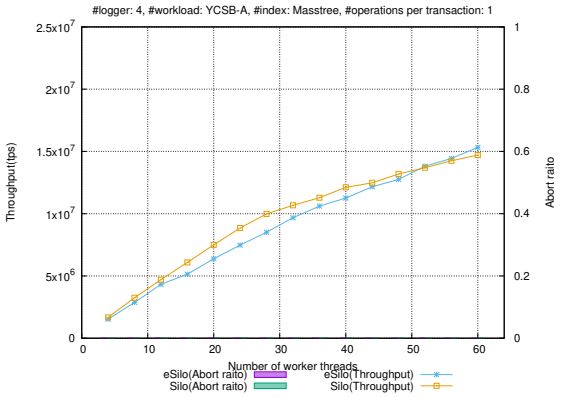

Figure 11: YCSB-A (operations per tx:1) with Masstree

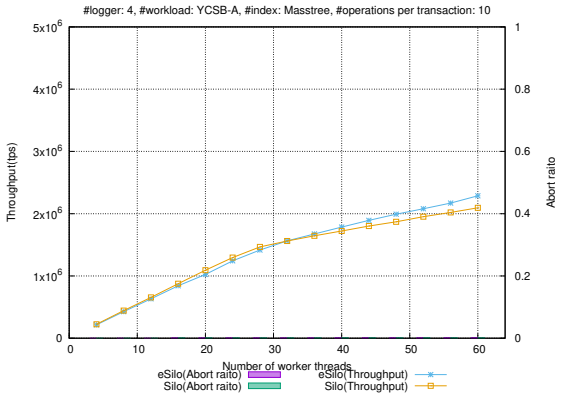

Figure 12: YCSB-A (operations per tx:10) with Masstree

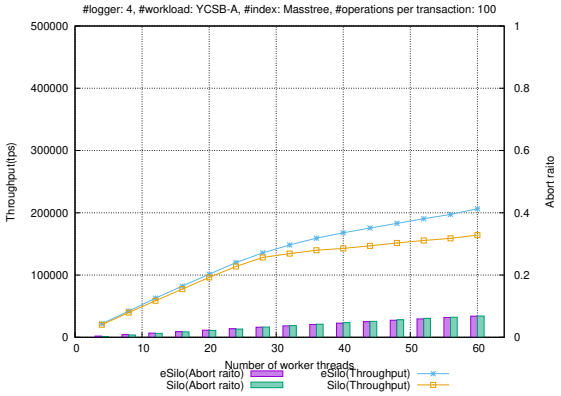

Figure 13: YCSB-A (operations per tx:100) with Masstree

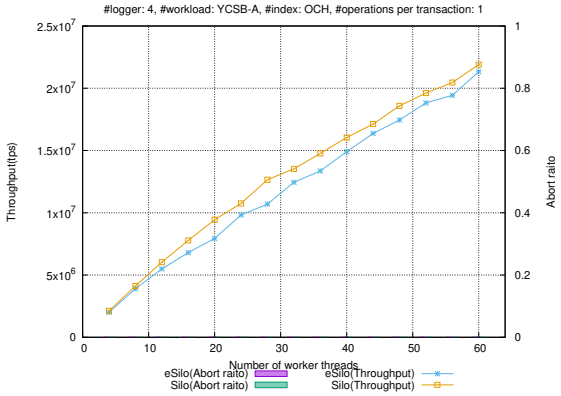

Figure 14: YCSB-A (operations per tx:1) with optimistic cuckoo hash

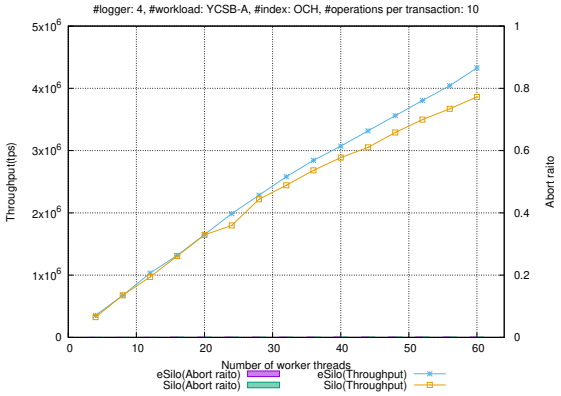

Figure 15: YCSB-A (operations per tx:10) with optimistic cuckoo hash

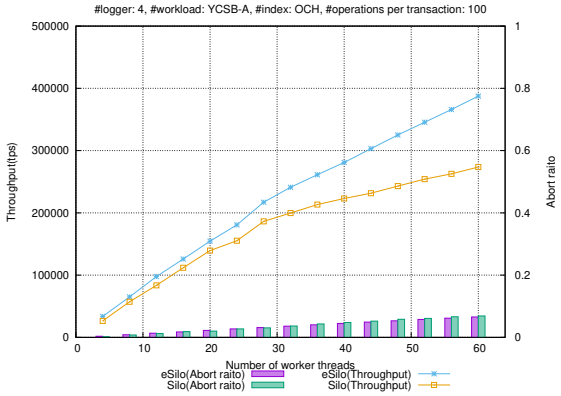

Figure 16: YCSB-A (operations per tx:100) with optimistic cuckoo hash

4.3.2 Memory Allocation and Release

Table 3: Dynamic memory allocation latency based on the number of parallel threads. Time to allocate and release memory for a record object (48 bytes).

#Thread	Normal		Enclave	
	new (ns)	delete (ns)	new (ns)	delete (ns)
1	47	19	79	22
2	47	19	402	381
4	48	19	697	1560
8	54	19	1512	4871
16	154	19	3373	12128
32	295	20	11693	29335
64	413	25	34140	38688
128	486	28	87312	75668

Since there are no insert or delete operations for YCSB-A, B, and C, memory allocation-related performance at SGX cannot be evaluated using them. For this purpose, we evaluate the performance of memory allocation. We measured the new and the delete operations for the normal case and the enclave case varying concurrency. The results of experiments are shown in Table 3. The performance of the enclave case is much worse compared with the normal case. This means that SGX does not exhibit novel performance for workloads with frequent inserts or deletes,

Therefore, even in YCSB-A, protocols based on multi-version concurrency control protocols like SSN [19], Cicada [21], and snapshot isolation [5] would not perform appropriately with this version of SGX. This is because multi-version protocols need to create a new version for each update and delete stale and obsolete versions with garbage collection. Thus, using SGX is preferred for single-versioned concurrency control protocols [7, 22, 36–38].

5 Related Work

5.1 Confidential Databases

Several techniques have been proposed to ensure the confidentiality and integrity of databases. There are two main approaches for this purpose. The first approach uses homomorphic cryptography to encrypt data before processing [29, 33]. It does not decrypt data for analysis, and thus, it prevents adversaries from stealing data. Still, it is far from being practical due to query restrictions caused by severe performance degradation provided by extremely expensive costs for homomorphic encryption computation.

The second approach is based on TEE [1, 3, 4, 30]. The use of TEE protects data in memories and ensures the confidentiality and integrity of data. None of them exploits parallel logging, and thus, the problem with the log buffer is not addressed.

A seminal work, EnclaveDB [30], guarantees the integrity of log records in storage besides tuples in the buffer pool. It is based on Hekaton [9], which does not adopt a parallel logging scheme, and it is based on a sequential logging protocol. Our work is based on Silo, which is designed to exploit parallel I/O devices such as SSD. Besides, the code is not publicly available, and thus, implementation details with SGX are not provided.

5.2 Fast Transaction Processing Systems

The acceleration of transaction processing has been addressed recently. They can be divided into deterministic protocols and non-deterministic protocols.

Aria [22] is a novel deterministic protocol that executes a transaction against a database snapshot and deterministically chooses whether to commit or not at transaction commit time. This

flexibility provides novel performance. Piece-wise visibility (PWV) [10] allows partitioned execution by constructing a dependency graph. By analyzing the dependency, it can execute non-conflicting transactions in parallel. Caracal [31] exploits the epoch concept to increase concurrency in its design. Its scheduling space is conflict serializable, but it exploits multi-version storage. It first creates a version array for each data item in the initialization phase and executes them in the subsequent phase.

Various non-deterministic serializable concurrency control protocols have been proposed. These protocols especially work efficiently for short transaction-oriented workloads like YCSB or TPC-C. Some protocols exploit a wide scheduling space [19, 27, 28] based on multi-version based-scheduling space. In contrast, conflict serializability-based protocols use a bunch of optimization methods exploiting hardware specifications (i.e., NUMA), the decentralization of timestamp generation [34] or speculative execution [7, 25, 37]. The purpose of exploitation is for performance improvement.

To our knowledge, none of the above is securely extended with SGX, except for our eSilo.

6 Conclusions and Future Work

In this study, we designed and implemented a secure transaction processing system based on the enclave and Silo.

We observed eSilo with a maximum of 2.30 million tps on sixty worker threads and four logger threads in YCSB-A workloads. Vanilla Silo, which does not use enclaves, observed a maximum of 2.10 million tps on sixty worker threads and four logger threads. The improvement, driven by the SGX dedicated library, shows eSilo boosts performance by 9.35% over vanilla Silo, proving enhanced confidentiality doesn't compromise speed, contrary to our initial concerns about a performance trade-off. Additional experiments show that parallel dynamic memory allocation within enclaves dramatically reduces performance.

The following future work should be addressed. The first work is the detection of log tampering. EnclaveDB provides this function, while it is not supported in eSilo. It is because this function can be naturally implemented with sequential logging, but it is not trivial for parallel logging that needs to use multiple separated log files. The second work is coping with the crack of the timestamp counter inside the CPU (TSC). SGX currently does not support any time systems, such as TSC, that depend on the CPU clock. Thus, if an adversary controls the CPU clock, serious problems can occur. If the clock is accelerated, the epoch will be frequently updated. Then it would exhaust log buffers in a very short period. If it is slowed down, then the epoch will not be updated, and it makes the latency of the system very long.

Acknowledgment

This paper is based on results obtained from the project, "Research and Development Project of the Enhanced infrastructures for Post-5G Information and Communication Systems" (JPNP20017) and the project (JPNP16007) commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and JSPS KAKENHI Grant Number 22H03596.

References

[1] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD Conf.*, page 1033–1036, 2013.

[2] Pierre-Louis Aublin, Mohammad Mahhouk, and Rüdiger Kapitza. Towards TEEs with large secure memory and integrity protection against HW attacks. `https://systex22.github.io/papers/systex22-final15.pdf`. Accessed: 2023-7-12.

[3] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *TKDE*, 26(3):752–765, 2014.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 33(3):1–26, 2015.

[5] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[6] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In *Security*, 2022.

[7] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Plor: General transactions with predictable, low tail latency. In *SIGMOD Conference*, pages 19–33. ACM, 2022.

[8] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD Conf.*, pages 1243–1254, 2013.

[10] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, jan 2017.

[11] Masahide Fukuyama. Code of esilo. `https://github.com/Noxy3301/enclaveSilo`, 2023.

[12] Masahide Fukuyama. Code of vanilla silo. `https://github.com/Noxy3301/silo_minimum`, 2023.

[13] Masahide Fukuyama, Masahiro Tanaka, Ryota Ogino, and Hideyuki Kawashima. esilo: Making silo secure with sgx. In *2023 Eleventh International Symposium on Computing and Networking (CANDAR)*, pages 107–112, 2023.

[14] Shay Gueron. A memory encryption engine suitable for general purpose processors. *Cryptology ePrint Archive*, 2016.

[15] Intel. Intel trust domain extensions. `https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf`. Accessed: 2023-7-15.

[16] Intel. Supporting intel sgx on muli-socket platforms., 2021. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf`.

[17] Intel. Sgx sdk, 2023. `https://download.01.org/intel-sgx/latest/linux-latest/distro/ubuntu20.04-server/sgx_linux_x64_sdk_2.23.100.2.bin`.

[18] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. `https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf`. Accessed: 2023-7-15.

[19] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687. ACM, 2016.

[20] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[21] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35. ACM, 2017.

[22] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[23] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, 2012.

[24] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *IEEE S and P*, pages 1466–1482, 2020.

[25] Tatsuhiro Nakamori, Jun Nemoto, Takashi Hoshino, and Hideyuki Kawashima. Decentralization of two phase locking based protocols. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 281–282, 2022.

[26] Yasuhiro Nakamura, Hideyuki Kawashima, and Osamu Tatebe. Integration of tictoc concurrency control protocol with parallel write ahead logging protocol. *IJNC*, 9(2):339–353, 2019.

[27] Sho Nakazono, Hiroyuki Uchiyama, Yasuhiro Fujiwara, Yasuhiro Nakamura, and Hideyuki Kawashima. Nwr: Rethinking thomas write rule for omittable write operations. *arXiv preprint arXiv:1904.08119*, 2019.

[28] Jun Nemoto, Takashi Kambayashi, Takashi Hoshino, and Hideyuki Kawashima. Oze: Decentralized graph-based concurrency control for real-world long transactions on bom benchmark. *arXiv preprint arXiv:2210.04179*, 2022.

[29] Raluca Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[30] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *IEEE S&P*, pages 264–278, 2018.

[31] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 180–194, 2021.

[32] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *PVLDB*, 13(13):3531–3544, 2020.

[33] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[34] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[35] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[36] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, 2016.

[37] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. Polaris: Enabling transaction priority in optimistic concurrency control. *Proc. ACM Manag. Data*, 1(1):44:1–44:24, 2023.

[38] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. SIGMOD '16, page 1629–1642, 2016.

[39] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI'14*, pages 465–477, October 2014.

Appendix

Since SGX is designed to be isolated from the operating system, it restricts the use of system calls and any functions that depend on them. This limitation requires the use of specialized libraries for enclave operations, which are designed to function independently of the OS-specific features. For this reason, SGXSDK provides alternative libraries, such as tlibc and tlibcxx, which are customized for enclave use. Our analysis indicated that tlibcxx is based on LLVM 12's libc++, customized to disable functions and libraries dependent on system calls. The similarity between the tlibcxx library and various versions of libc++ is shown in Figure 17.

The SGXSDK v2.23.100.2 used in the performance evaluation of eSilo explicitly disabled certain libraries such as cfenv, chrono, clocale, csignal, fstream, future, iostream, regex, and thread, along with some functions.

By excluding the standard library from g++ and linking LLVM's libc++ and libc++abi instead, the number of call instructions in the assembly was reduced, leading to enhanced performance. The tlibcxx library used within the enclave, being based on LLVM 12's libc++, benefits from these performance improvements.

To illustrate this, consider Algorithm 5, which demonstrates the use of a loop with an iterator. The code iterates through the test vector using an iterator (itr) and compares each element with a position counter (pos) to ensure they match.

Table 4: Function Calls and Assembly Instructions Comparison

Operation	g++ with libstdc++	g++ with libc++ (tlibcxx)
test.begin()	**call** std::vector<int, std::allocator <int>>::begin()	**leaq** test(%rip), %rdx
test.end()	**call** std::vector<int, std::allocator <int>>::end()	**movq** 8+test(%rip), %rax
itr++	**leaq** -104(%rbp), %rax **movq** %rax, %rdi **call** __gnu_cxx::__normal_iterator <int*, std::vector<int, std::allocator<int>>>::operator ++(int)	**leaq** -1008(%rbp), %rax **movq** %rax, -488(%rbp) **movq** -488(%rbp), %rax **movq** (%rax), %rax **leaq** 4(%rax), %rdx
assert((*itr) == pos)	**leaq** -104(%rbp), %rax **movq** %rax, %rdi **call** __gnu_cxx::__normal_iterator <int*, std::vector<int, std:: allocator<int>>>::operator*() const **movl** (%rax), %eax **cmpl** %eax, -124(%rbp)	**movq** -1008(%rbp), %rax **movl** (%rax), %eax **cmpl** %eax, -1024(%rbp)

The assembly generated for invoking member functions and operator overloads of vector and iterator libraries in cases using g++ with the standard library (libstdc++) versus g++ with libc++ (tlibcxx) is shown in Table 4.

This analysis presents the assembly generation without specific optimizations. It demonstrates that when g++ is combined with libc++, the call instructions are eliminated from the assembly. Call instructions expand into multiple opcodes when compiled into an executable. Their frequent

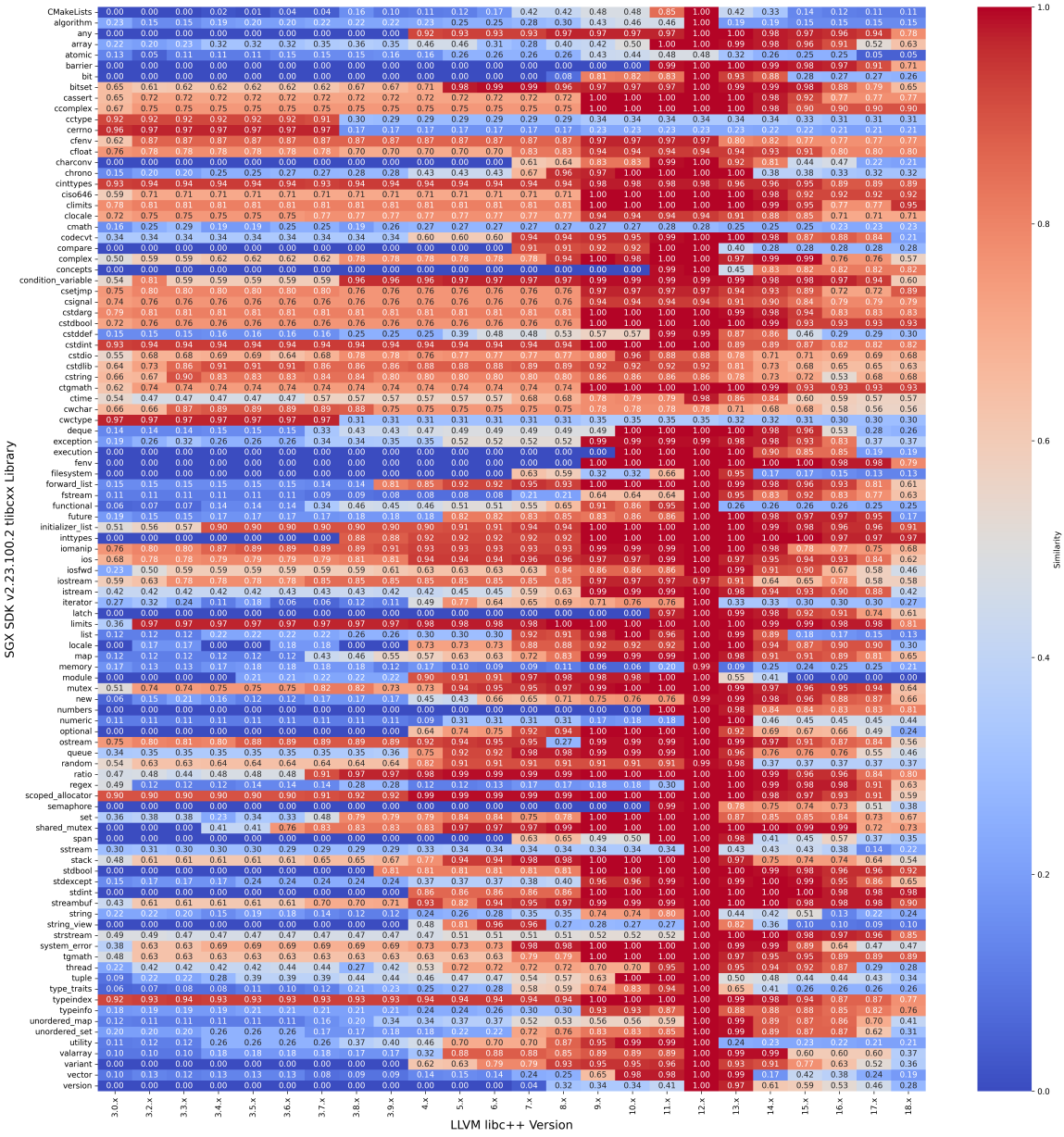

Figure 17: Library similarity between SGXSDK tlibcxx and LLVM libc++

Algorithm 5 Vector iteration code written in C++

```
1: for (auto itr = test.begin(); itr != test.end(); itr++) {
2:     assert((*itr) == pos);
3:     pos = pos + 1;
4: }
```

use in each iteration introduces significant overhead. However, eliminating these call instructions by acquiring the iterator's pointer before the loop starts and directly incrementing the address during iterations can greatly reduce this overhead. It was confirmed that this adjustment effectively removes the performance differential.

Therefore, by removing the standard library from g++ and using LLVM's libc++ and libc++abi instead, the number of call instructions decreases, which improves performance. The tlibcxx library, a customized version of LLVM's libc++, benefits greatly from this improvement, resulting in specific processes running faster inside the enclave.