Thread-Safe Dynamic Binary Translation using Transactional Memory

JaeWoong Chung, Michael Dalton, Hari Kannan, Christos Kozyrakis

Computer Systems Laboratory
Stanford University
{jwchung, mwdalton, hkannan, kozyraki}@stanford.edu

Abstract

Dynamic binary translation (DBT) is a runtime instrumentation technique commonly used to support profiling, optimization, secure execution, and bug detection tools for application binaries. However, DBT frameworks may incorrectly handle multithreaded programs due to races involving updates to the application data and the corresponding metadata maintained by the DBT. Existing DBT frameworks handle this issue by serializing threads, disallowing multithreaded programs, or requiring explicit use of locks.

This paper presents a practical solution for correct execution of multithreaded programs within DBT frameworks. To eliminate races involving metadata, we propose the use of transactional memory (TM). The DBT uses memory transactions to encapsulate the data and metadata accesses in a trace, within one atomic block. This approach guarantees correct execution of concurrent threads of the translated program, as TM mechanisms detect and correct races. To demonstrate this approach, we implemented a DBT-based tool for secure execution of x86 binaries using dynamic information flow tracking. This is the first such framework that correctly handles multithreaded binaries without serialization. We show that the use of software transactions in the DBT leads to a runtime overhead of 40%. We also show that software optimizations in the DBT and hardware support for transactions can reduce the runtime overhead to 6%.

1. Introduction

Dynamic binary translation (DBT) has become a versatile tool that addresses a wide range of system challenges while maintaining backwards compatibility for legacy software. DBT has been successfully deployed in commercial and research environments to support full-system virtualization [39], cross-ISA binary compatibility [7, 29], security analyses [9, 18, 25, 28, 32], debuggers [24], and performance optimization frameworks [3, 5, 38]. DBT facilitates deployment of new tools by eliminating the need to recompile or modify existing software. It also enables the use of newer hardware which need not be fully compatible with the old software architecture.

The DBT process involves dynamically translating an existing binary, and replacing or adding instructions as needed. All code execution is controlled by the DBT, and only translated code blocks are executed. A cache that stores translated basic blocks or traces is typically used to reduce the runtime overhead. For applications such as security analyses [9, 25, 28], profiling [19], and bug detection [24], DBT may also maintain *metadata* that tracks the state of memory locations. For instance, DBT can be used to implement *dynamic information flow tracking (DIFT)*, a technique that detects security vulnerabilities such as buffer overflows and other attacks based on memory corruption [8, 10, 25]. In this case, the metadata allows the DBT to taint memory locations that have been received or derived from untrusted sources (e.g., network input). If the program dereferences a tainted code pointer or executes a tainted instruction, a security vulnerability is reported.

However, the use of metadata can lead to race conditions when multithreaded binaries are dynamically translated. Since operations on metadata are handled by additional instructions, updates to data and corresponding metadata are not atomic. Multiple threads operate on the same or neighboring data and thus, this lack of atomicity can lead to inaccurate or incorrect metadata values. In the case of DIFT, this can translate to false negatives (undetected security attacks) or false positives (incorrect program termination). Existing DBT systems handle this problem by disallowing the use of multithreaded programs [4, 28], serializing thread execution [24], or requiring that tool developers explicitly implement the locking mechanisms needed to access metadata [19]. None of these solutions are satisfactory. First, multithreaded programs are becoming increasingly common due to the availability of multi-core chips. Second, thread serialization cancels the performance benefits of multithreading. Finally, providing locks for metadata accesses is a tedious process that introduces significant storage and runtime overheads and limits the scope of many DBT optimizations.

This paper improves the applicability of DBT by making it *safe* for multithreaded programs. To address the issue of metadata atomicity in a functionally-correct and low-overhead manner, we employ *transactional memory (TM)* [17]. TM has been used thus far to simplify parallel programming by supporting the atomic and isolated execution of concurrent blocks that operate on shared data. We use TM within the DBT framework to enclose the data and metadata accesses within an atomic transaction. Hence, TM mechanisms can automatically detect races on metadata and correct them through transaction rollback and re-execution. Since the TM system executes transactions optimistically in parallel assuming races are rare, threads are not serialized.

To demonstrate the synergy between DBT and TM, we use a DBT system to implement a DIFT-based secure execution framework for x86 binaries. This framework can detect memory corruption vulnerabilities such as buffer overflows and format string attacks. The DBT automatically introduces transactions in the translated code and can run on top of software-only, hybrid, or hardware-based TM systems.

The specific contributions of this work are:

- We propose the use of transactional memory to address the correctness and performance challenges for dynamic binary translation of multithreaded programs. We show that TM eliminates races on accesses to metadata without thread serialization.
- We implement a DBT-based framework for secure execution of x86 binaries through dynamic information flow tracking. The framework uses transactions at the granularity of DBT traces. This is the *first* DBT framework that handles multithreaded bi-

naries in a safe, highly performant manner.

- We study the runtime overhead of introducing TM in a DBT framework by using a software-only TM system and by approximating hardware-accelerated and hybrid TM systems. We show that even without hardware support, the overhead is approximately 40%. Software optimizations in the DBT that avoid tracking certain classes of data and hardware support for transactions can reduce the runtime overhead to 6%.

Overall, we show that TM extends metadata-based DBT tools to multithreaded applications.

The remainder of the paper is organized as follows. Section 2 summarizes DBT technology and the challenges posed by multithreaded binaries. Sections 3 and 4 present the use of TM for DBT metadata atomicity. Section 5 describes our prototype system for DBT-based secure execution of multithreaded programs. Section 6 presents the performance evaluation, Section 7 discusses related work, and Section 8 concludes the paper.

2. Dynamic Binary Translation (DBT)

2.1 DBT Overview

DBT relies on runtime code instrumentation to dynamically translate and execute application binaries. Before executing a basic block from the program, the DBT copies the block into a code cache and performs any required instrumentation. The DBT system controls all code execution and only translated code from the code cache is executed. Any control flow that cannot be resolved statically, such as indirect branches, invokes the DBT to ensure that the branch destination is in the code cache. Frequently executed basic blocks in the code cache may be merged into a longer trace. This reduces runtime overhead by allowing common hot paths to execute almost completely from the code cache without invoking the DBT system.

DBT may arbitrarily add, insert, or replace instructions when translating code. This powerful capability allows DBT to serve as a platform for dynamic analysis tools. DBT-based tools have been developed for tasks such as runtime optimization [5], bug detection [24], buffer overflow protection [18], and profiling [19]. A general-purpose DBT framework implements basic functionality such as program initialization, code cache management, and trace creation. It also provides developers with an interface (API) that they can use to implement tools on top of the DBT. Using the API, developers specify which, how, and when instruction sequences are instrumented. Finally, the DBT typically performs several optimizations on the instrumented code such as function inlining, common subexpression elimination, and scheduling.

DBT tools often maintain *metadata* that describe the state of memory locations during program execution. For example, metadata can indicate if a memory location is allocated, if it contains secure data, or the number of times it has been accessed. The granularity and size of metadata can vary greatly from one tool to another. Some tools may keep instrumentation information at a coarse granularity such as function, page, or object, while other tools may require basic block, word, or even bit-level granularity. The popular Memcheck tool for the Valgrind framework uses a combination of heap object, byte and bit-level metadata to detect uninitialized variables, memory corruption, and memory leaks [23, 34]. Metadata are allocated in a *separate* memory region to avoid interference with the data layout assumed by the original program. Whenever the program code operates on data, the DBT tool inserts code to perform the proper operations on the corresponding metadata. The additional code can be anything from a single instruction to a full function call, depending on the tool.

2.2 Case Study: DBT-based DIFT Tool

In this paper, we use dynamic information flow tracking (DIFT) as a specific example of a metadata-based DBT tool. However, the issues with multithreaded programs are the same for all other metadata-based DBT tools (profiling, bug detection, fault recovery, etc.). DIFT is particularly interesting because it provides security features that are important for multithreaded server applications such as web servers.

DIFT is a powerful dynamic analysis that can prevent a wide range of security issues [10, 25]. DIFT tracks the flow of untrusted information by associating a taint bit with each memory byte and each register. The OS taints information from untrusted sources such as the network. Any instruction that operates on tainted data *propagates* the taint bit to its destination operands. Malicious attacks are prevented by *checking* the taint bit on critical operations. For example, checking that code pointers and the code itself are not tainted allows for the prevention of buffer overflows and format string attacks.

There are several DBT-based DIFT systems [4, 8, 25, 28]. The metadata maintained by the DBT tool are the taint bits. Code is instrumented to propagate taint bits on arithmetic and memory instructions and check the taint bits on indirect jumps. The research focus has been primarily on reducing overheads using optimizations such as eliminating propagation and checks on known safe data and merging checks when possible [28]. For I/O-bound applications, the overhead of DBT-based DIFT over the original runtime (without DBT) is as low as 6% [28]. However, none of the DBT-based DIFT tools provide highly performant, safe support for multithreaded programs.

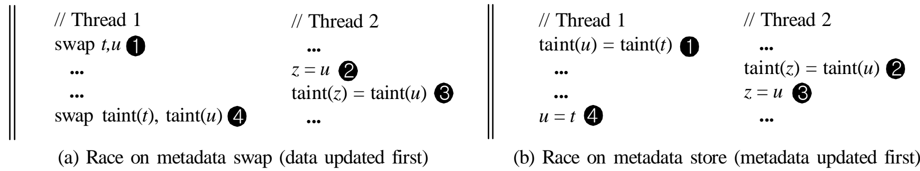
2.3 Metadata Races

DBTs cannot be readily applied to multithreaded binaries because of races on metadata access. Metadata are stored separately from the regular data. They are also operated upon by separate instructions. Since the DBT metadata can be of arbitrary size and granularity, it is impossible to provide hardware instructions to atomically update data and metadata in the general case. The original program is unaware of the DBT metadata and thus, cannot use synchronization to prevent metadata races. Hence, if the DBT does not provide additional mechanisms for synchronization, any concurrent access to a *(data, metadata)* pair may lead to a race.

Figure 1 provides two examples of races for a DBT-based DIFT tool. Consider a multithreaded program operating on variables `t` and `u`. The DBT introduces the corresponding operations on the metadata, `taint(t)` and `taint(u)`. Initially, `t` is tainted (untrusted) and `u` is untainted (trusted). In Figure 1.(a), thread 1 swaps `t` and `u` using a single atomic instruction (❶). The DBT inserts a subsequent instruction that swaps `taint(t)` and `taint(u)` as well (❹). However, the pair of swaps is not an atomic operation. As thread 2 is concurrently reading `u` (❷), it gets the new value of `u` and the old value of `taint(u)` (❸). Even though thread 2 will use the untrusted information derived from `t`, the corresponding taint bit indicates that this is safe data. If `z` is later used as code or as a code pointer, an undetected security breach will occur (false negative) that may allow an attacker to take over the system.

Figure 1.(b) shows a second example in which the DBT updates the metadata before the actual data. Thread 1 uses a single instruction to copy `t` into `u` (❹). The previous instruction does the same to the metadata (❶). However, the pair of instructions is not atomic. As thread 2 is concurrently reading `u`, it gets the new value of the metadata (❷) and the old value of the data (❸). Even though thread 2 will use the original, safe information in `u`, the corresponding taint bit indicates that it is untrusted. If `z` is later used as a code pointer or code, an erroneous security breach will be reported (false positive) that will unnecessarily terminate a

Initially t is tainted and u is untainted.



(a) Race on metadata swap (data updated first) (b) Race on metadata store (metadata updated first)

Figure 1. Two examples of races on metadata (taint bit) accesses.

legitimate program.

In general, one can construct numerous scenarios with races in updates to *(data, metadata)* pairs. Depending on the exact use of the DBT metadata, the races can lead to incorrect results, program termination, undetected malicious actions, etc. Current DBT systems do not handle this problem in a manner that is both functionally-correct and fast. Some DBTs support multithreaded programs by serializing threads, which eliminates all performance benefits of multithreading [24]. Note that in this case, thread switching must be carefully placed so that it does not occur in the middle of a non-atomic *(data, metadata)* update. Other DBTs assume that tool developers will handle this problem using locks in their instrumentation code [5, 19]. If the developer uses coarse-grain locks to enforce update atomicity, all threads will be serialized. Fine-grain locks are error-prone to use. They also lead to significant overheads if a lock acquisition and release is needed in order to protect a couple of instructions (data and metadata update). More importantly, since locks introduce memory fences, their use reduces the efficiency of DBT optimizations such as common sub-expression elimination and scheduling.

2.4 Implications

Metadata races can be an important problem for *any* multithreaded program. Even if the application is race-free to begin with, the introduction of metadata breaks the assumed atomicity of basic ISA instructions such as aligned store or compare-and-swap. Numerous multithreaded applications rely on the atomicity of such instructions to build higher-level mechanisms such as locks, barriers, read-copy-update (RCU) [20], and lock-free data-structures (LFDS) [16]. It is unreasonable to expect that the DBT or tool developers will predict all possible uses of such instructions in legacy and future binaries, and properly handle the corresponding metadata updates. For instance, it is difficult to tell if the program correctness relies on the atomicity of an aligned store.

We discuss the DBT-based DIFT tool as an illustrating example. Metadata races around locks and barriers do not pose a security threat, as lock variables are not updated with user input. On the other hand, RCU and LFDS manipulate pointers that control the correctness and security of the application and may interact with user input. Hence, metadata races can allow an attacker to bypass the DIFT security mechanisms.

Read-copy update (RCU) is used with data-structures that are read much more frequently than they are written. It is a common synchronization technique used in several large applications including the Linux kernel. RCU directly updates pointers in the protected data structure, relying on the atomicity of aligned, word-length stores. These pointers may be influenced by user input and may reference objects that include critical code pointers. For instance, Linux uses RCU to protect core components such as IPv4 route caches, directory caches, and file descriptors. NSA's Security Enhanced Linux uses RCU to protect access control information [33].

Lock-free data structures use atomic instructions such as compare-and-swap (CAS) to allow concurrent access without conventional locking [16]. While they are complex to implement, LFDS versions of queues, lists, and hash-tables are often part of libraries used in performance-critical software. Since they manipulate pointers within data structures, LFDS may access data from untrusted sources.

Additionally, attackers may use memory safety vulnerabilities to deliberately introduce race conditions into race-free applications. Attacks such as buffer overflows and format string vulnerabilities give the attacker the ability to write anywhere in the program's address space. For example, if network input is used as an array index, the attacker can use one thread to overwrite thread-private or stack data in other threads. This essentially bypasses any techniques used to guarantee race-freedom in the original code. Moreover, an attacker can target metadata races in order to bypass security checks in DBT tools such as DIFT. By having one thread write malicious code or data to another thread's stack or thread-local data, an attacker can induce false negatives similar to the one in Figure 1.(a). Hence, the attacker may be able to overwrite critical pointers such as return addresses without setting the corresponding taint bit. This attack would not be possible on a single-threaded program as the DIFT propagation and checks would flag the overwritten data as untrusted.

3. DBT + TM = Thread-Safe DBT

To address metadata races in multithreaded programs, we propose the use of transactional memory (TM) within DBT systems. This section concentrates on the functional correctness of this approach, while Section 4 focuses on performance issues.

3.1 Transactional Memory Overview

TM can simplify concurrency control in shared memory parallel programming [17]. With TM, a programmer simply declares that code segments operating on shared data should execute as atomic transactions. Multiple transactions may execute in parallel and the TM system is responsible for concurrency control. A transaction starts by checkpointing registers. Transactional writes are isolated from shared memory by maintaining an undo-log or a write-buffer (data versioning). Memory accesses are tracked in order to detect read/write conflicts among transactions. If a transaction completes without conflicts, its updates are committed to shared memory atomically. If a conflict is detected between two transactions, one of them rolls back by restoring the register checkpoint and either by restoring the undo-log or by discarding the write-buffer.

There have been many proposed TM implementations. Software TM systems (STM) implement all TM bookkeeping in software by instrumenting read and write accesses within a transaction [12, 15, 30]. The overhead of software read and write barriers can be reduced to 40% with various compiler optimizations [1, 30]. Hardware TM systems (HTM) implement both data versioning and conflict detection by modifying the hardware caches and the coherence protocol. HTM systems do not require software barriers for read and write accesses within a transaction, and thus have minimal overhead [14, 21]. Hardware performs all bookkeeping transparently. More recently, there have been proposals

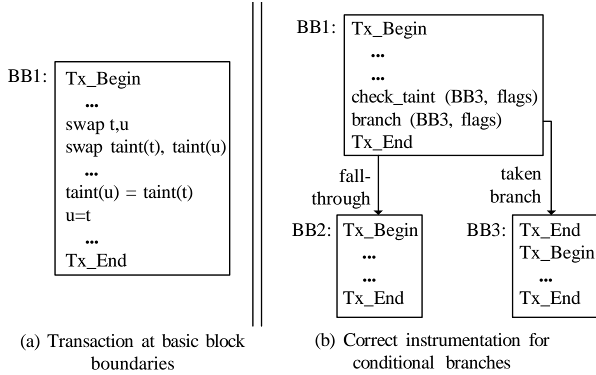


Figure 3. Transaction instrumentation by the DBT.

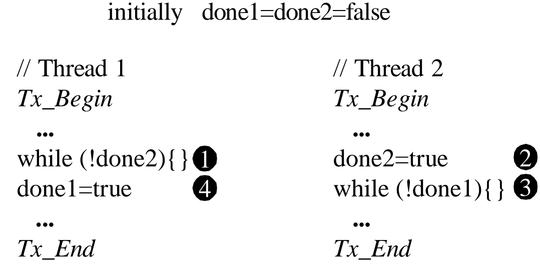


Figure 4. The case of a conditional wait construct within a DBT transaction.

for hybrid TM systems [6, 31, 36]. While they still require read and write barriers, hybrid systems use hardware signatures or additional metadata in hardware caches in order to drastically reduce the overhead of conflict detection in software transactions.

3.2 Using Transactions in the DBT

DBT systems can eliminate metadata races by wrapping any access to a *(data, metadata)* pair within a transaction. Figure 2 shows the code for the two race condition examples in Section 2.3, instrumented with transactions. The additional code defines transaction boundaries and provides the read/write barriers required by STM and hybrid systems. In Figure 2.(a), thread 1 encloses the data and metadata swaps within one transaction. Hence, even if thread 2 attempts to read `u` in between the two swaps, the TM system will detect a transaction conflict. By rolling back one of the two transactions, the TM system will ensure that thread 2 will read either the old values of both `u` and `taint(u)`, or the new values for both after the swap. Similarly, transactions eliminate the race in Figure 2.(b). Since transactions are atomic, the order of data and metadata accesses within a transaction does not matter.

For now, we assume that all data and metadata accesses are tracked for transactional conflicts in order to guarantee correctness. This is the default behavior of hardware TM systems. For software and hybrid TM systems, this requires at least one barrier for each address accessed within a transaction. We revisit this issue in Section 4 when we discuss optimizations. We focus here on the placement of transaction boundaries. To guarantee correctness, data and corresponding metadata accesses must be enclosed within a single transaction. Fine-grain transactions, however, lead to significant performance loss as they do not amortize the overhead of starting and ending transactions. Moreover, they interfere with many DBT optimizations by reducing their scope. To partially alleviate these problems, our base design introduces transactions at basic block boundaries, enclosing multiple accesses to data and metadata pairs as shown in Figure 3.(a). The Tx_Begin statement is added at the beginning of the block. In most cases, the Tx_End statement is added after the last non control-flow instruction in the block (branch or jump).

The Tx_End placement is complicated for any DBT-based tool that associates metadata operations with the control-flow instruction that ends a basic block. For instance, DIFT must check the address of any indirect branch prior to taking the branch. If the address is tainted, a security attack is reported. In this case, the DBT must introduce code that checks the metadata for the jump target in the same transaction as the control flow instruction to avoid metadata races. We handle this case by introducing the Tx_End statement at the beginning of the basic block that is the target of the indirect jump. For conditional branches, we introduce Tx_End at the beginning of both the fall-through block and the target block as shown in Figure 3.(b). If the target block includes a transaction to cover its own metadata accesses, Tx_End is immediately followed by a Tx_Begin statement.

3.3 Discussion

The DBT transactions may experience false conflicts due to the granularity of conflict detection in the underlying TM system, and the layout of data and metadata in memory. False conflicts can be a performance issue, but do not pose correctness challenges. Even if the contention management policy of the TM system does not address fairness, the DBT can intervene and use fine-grain transactions to avoid these issues.

The original binary may also include user-defined transactions. If a user-defined transaction fully encloses a DBT transaction (or vice versa), the TM system will handle it correctly using mechanisms for nested transactions (subsuming or nesting with independent rollback). Partial overlapping of two transactions is problematic. To avoid this case, the DBT can split its transactions so that partial overlapping does not occur. As long as the accesses to each *(data, metadata)* pair are contained within one transaction, splitting a basic block into multiple transactions does not cause correctness issues.

A challenge for transactional execution is handling I/O operations. For DBT transactions, I/O is not an issue as the DBT can terminate its transactions at I/O operations. Such operations typically terminate basic blocks or traces, and act as barriers for DBT optimizations. Handling I/O operations within user-defined transactions is beyond the scope of this work.

A final issue that can occur if DBT transactions span multiple basic blocks (see Section 4), is that of conditional synchronization in user code. Figure 4 presents a case where transactions enclose code that implements flag-based synchronization. Without transactions, this code executes correctly. With transactions, however, execution is incorrect as the two transactions are not serializable. For correctness, statement ❶ must complete after statement ❷ and statement ❸ after statement ❹. Once transactions are introduced, statements ❶ and ❹ must complete atomically. Similarly, statements ❷ and ❸ must complete atomically. Regardless of the type of the TM system and the contention management policy used, the code in Figure 4 will lead to a livelock or deadlock. We handle this case dynamically. If the DBT runtime system notices that there is no forward progress (timeout on a certain trace or repeated rollbacks of transactions), it re-instruments and re-optimizes that code to use one transaction per basic block.

Initially `t` is tainted and `u` is untainted.

```
// Thread 1                  // Thread 2
Tx_Begin                     ...
 wrbarrier(t)                Tx_Begin
 wrbarrier(u)                 rdbarrier(u)
 swap t,u                     wrbarrier(z)
 ...                          z = u
 ...                          rdbarrier(taint(u))
 wrbarrier(taint(t))          wrbarrier(taint(z))
 wrbarrier(taint(u))          taint(z) = taint(u)
 swap taint(t), taint(u)     Tx_End
Tx_End                       ...
```

(a) Elimination of the metadata swap race

```
// Thread 1                  // Thread 2
Tx_Begin                     ...
 rdbarrier(taint(t))         Tx_Begin
 wrbarrier(taint(u))          rdbarrier(taint(u))
 taint(u) = taint(t)          wrbarrier(taint(z))
 ...                          taint(z) = taint(u)
 ...                          rdbarrier(u)
 rdbarrier(t)                 wrbarrier(z)
 wrbarrier(u)                 z = u
 u = t                       Tx_End
Tx_End                       ...
```

(b) Elimination of the metadata store race

Figure 2. Addressing metadata races using transactions. The read and write barriers are necessary for STM and hybrid TM systems, but not for HTM systems.

4. Optimizations for DBT Transactions

The use of transactions in DBT eliminates metadata races for multithreaded programs. However, transactions introduce three sources of overhead: the overhead of starting and ending transactions; that of the read and write barriers for transactional bookkeeping; and the cost of rollbacks and re-execution. This section focuses on the first two sources of overhead, which are particularly important for STM and hybrid systems. In Section 6, we show that rollbacks and are not common for DBT transactions.

4.1 Overhead of Starting/Ending Transactions

Longer transactions improve performance in two ways. First, they amortize the cost of starting and ending a transaction. Starting a transaction includes checkpointing register state. Ending a transaction requires clean up of any structures used for versioning and conflict detection. Second, long transactions increase the scope of DBT optimizations, such as common subexpression elimination, instruction scheduling, and removal of redundant barriers [1].

There are two ways to increase the length of DBT transactions:

- **Transactions at trace granularity:** Modern DBT frameworks merge multiple basic blocks into hot traces in order to increase the scope of optimizations and reduce the number of callbacks to the DBT engine [4, 19, 24]. We exploit this feature by introducing transactions at the boundaries of DBT traces. As the DBT creates or extends traces, it also inserts the appropriate Tx_Begin and Tx_End statements.
- **Dynamic transaction merging:** We can also attempt to dynamically merge transactions as DBT traces execute. In this case, the DBT introduces instrumentation to count the amount of work per transaction (e.g., the number of instructions). When the transaction reaches the Tx_End statement, the STM checks if the work thus far is sufficient to amortize the cost of Tx_Begin and Tx_End. If not, the current transaction is merged with the succeeding one. Dynamic transaction merging is especially helpful if DBT transactions are relatively common, as is the case with a DBT-based DIFT tool.

While both methods produce longer transactions, they differ in their behavior. Trace-level transactions incur additional overhead only when traces are created. However, the transaction length is limited by that of the DBT trace. Dynamic transaction merging incurs some additional overhead as traces execute, but can create transactions that span multiple traces. Both approaches must take into account that increasing the transaction length beyond a certain point leads to diminishing returns. Moreover, very long transactions are likely to create more conflicts. The ideal length of transactions depends on the underlying TM system as well. Hardware

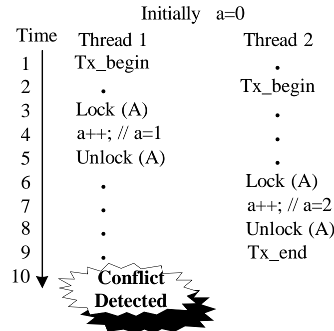


Figure 5. An example showing the need for TM barriers on data accesses even in race-free programs.

and hybrid TM systems typically use hardware mechanisms for register checkpointing and allow shorter transactions to amortize the fixed overheads.

4.2 Overhead of Read/Write Barriers

For STM and hybrid TM systems, the DBT must introduce read and write barriers in order to implement conflict detection and data versioning in transactions. Despite optimizations such as assembly-level tuning of barrier code, inlining, and elimination of repeated barriers, the performance overhead of software bookkeeping is significant [1]. Hence, we discuss techniques that reduce the number read and write barriers for DBT transactions.

4.2.1 Basic Considerations for Barrier Use

In Section 3, we stated that all data and metadata accesses in a DBT transaction should be protected with a TM barrier. This is definitely the case for DBT-based DIFT tools. During a buffer overflow attack, a thread may access any location in the application's address space (see Section 2.4). Unless we use barriers for all addresses, we may miss metadata races that lead to false negatives or false positives in the DIFT analysis. For other DBT-based tools that require metadata, it may be possible to eliminate several barriers on data or metadata accesses based on knowledge about the sharing behavior of threads. One has to be careful about barrier optimizations, as aggressive optimization can lead to incorrect execution. One must also keep in mind that TM barriers implement two functions: they facilitate conflict detection and enable rolling back the updates of aborted transactions.

For instance, assuming that the original program is race-free, one may be tempted to insert barriers on metadata accesses but eliminate all barriers on the accesses to the original data. Figure 5 shows a counter example. The two threads use a lock to increment variable `a` in a race-free manner. The DBT introduces transactions that enclose the lock-protected accesses. The transactions include TM barriers for metadata but not for `a`. At runtime, thread 1 sets `a` to 1 at timestep 4 and thread 2 updates `a` to 2 at timestep 7. At timestep 10, the TM system detects a conflict between the transaction in thread 1 and another thread due to a different metadata access. The system decides to rollback the transaction of thread 1. Since there was no barrier on the access to `a`, its value cannot be rolled back as there is no undo-log or write-buffer entry.

Now, assume that we execute the same code with a TM barrier on `a` that does data versioning (e.g., creates an undo-log entry) but does not perform conflict detection on `a`. At timestep 4, thread 1 will log 0 as the old value of `a`. When the system rolls back thread 1 at timestep 10, it will restore `a` to 0. This is incorrect as it also eliminates the update done by thread 2. The correct handling of this case is to insert TM barriers for `a` that perform both conflict detection and data versioning, even though the original code had locks to guarantee race-free accesses to `a`.

4.2.2 Optimizations Using Access Categorization

For DBT-based tools unlike DIFT, it is possible to eliminate or simplify certain TM barriers by carefully considering data access types. Figure 6 presents the five access types we consider and the least expensive type of TM barrier necessary to guarantee correct atomic execution of the code produced by the DBT:

- **Stack and Idempotent_stack accesses**: For accesses to thread-local variables on the stack [13], we need barriers for data versioning but not for conflict detection. Moreover, if the access does not escape the scope of the current transaction (Idempotent_stack) there is no need for TM barriers at all.
- **Private accesses**: Similar to stack variables, certain heap-allocated variables are thread-local. Their accesses do not require barriers for conflict detection. If the transaction updates the variable, a TM barrier is needed for data versioning.
- **Benign_race accesses**: There are access to shared variables that are not protected through synchronization in the original program. Assuming a well-synchronized application, any races between accesses to these variables are benign. Hence, there is no need for TM barriers for conflict detection, but there may be barriers for data versioning on write accesses.
- **Shared accesses**: Any remaining access must be assumed to operate on truly shared data. Hence, the DBT framework must insert full read and write barriers depending on the type of access. The only optimization possible is to eliminate any repeated barriers to the same variable within a long transaction [1].

The DBT framework classifies accesses during trace generation using well-known techniques such as stack escape and dynamic escape analysis [13, 35]. It also adds instrumentation that collects the information necessary for runtime classification [35]. The success of the classification partially depends on the nature of the DBT. For example, the information available in object-based binaries such as Java bytecode increases the analysis accuracy and provides additional optimization opportunities.

To identify Benign_race accesses, the DBT must know the synchronization primitives used by the application. It can be the case that the lack of synchronization is actually a bug. The translated code may change if and when the bug manifests in the execution. This problem is not specific to DBTs. Any small system perturbation such as the use of a faster processor, the use of more threads, or a change in memory allocation may be sufficient to mask or expose a race.

Instruction Type	Example	Taint Propagation
ALU operation	r3 = r1 + r2	Taint[r3] = Taint[r1] OR Taint[r2]
Load	r3 = M[r1+r2]	Taint[r3] = Taint[M[r1+r2]]
Store	M[r1+r2] = r3	Taint[M[r1+r2]] = Taint[r3]
Register clear	r1 = r1 xor r1	Taint[r1] = 0
Bounds check	cmp r1, 256	Taint[r1] = 0

Table 1. Taint bit propagation rules for DIFT.

5. Prototype System

To evaluate the use of transactions in DBT, we used the Pin dynamic binary translator for x86 binaries [19]. We implemented DIFT as a Pintool and ran it on top of a software TM system.

5.1 DIFT Implementation

The analysis in our DIFT tool is similar to the one in [25, 28]. We maintain a taint bit for every byte in registers and main memory to mark untrusted data. Any instruction that updates the data must also update the corresponding taint bit. Table 1 summarizes the propagation rules. For instructions with multiple source operands, we use logical OR to derive the taint bit for the destination operand. Hence, if any of the sources are tainted, the destination will be tainted as well.

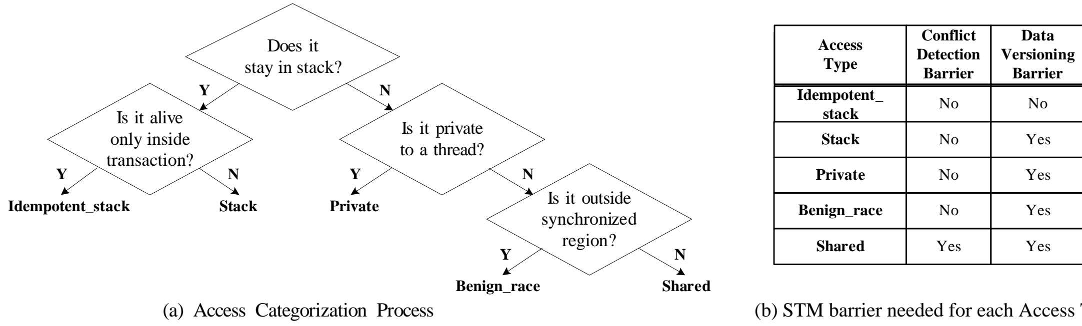
To execute real-world binaries without false positives, register clearing and bounds check instructions must be recognized and handled specially. For the x86 architecture, instructions such as `sub %eax, %eax` and `xor %eax, %eax` are often used to clear registers as they write a constant zero value to their destination, regardless of the source values. The DIFT tool recognizes such instructions and clears the taint bit for the destination register. Programs sometimes validate the safety of untrusted input by performing a bounds check. After validation, an input can be used as a jump address without resulting in a security breach. We recognize bounds checks by untainting a register if it is compared with a constant value [25, 28].

To prevent memory corruption attacks, taint bits are checked on two occasions. First, when new code is inserted into the code cache, we check the taint bits for the corresponding instructions. This check prevents code injection attacks. Second, we check the taint bit for the operands of indirect control-flow operations (e.g., register-indirect jump or procedure call/return). This ensures that an attacker cannot manipulate the application's control-flow.

5.2 Software TM System

We implemented an STM system similar to the one proposed in [1]. For read accesses, the STM uses optimistic concurrency control with version numbers. For writes, it uses two-phase locking and eager versioning with a per-thread undo-log. The conflict detection occurs at word granularity using a system-wide lock table with 2^{10} entries. Each lock word contains a transaction ID when locked and a version number when unlocked. The least significant bit identifies if the word is locked or not. Given a word address, the corresponding lock word is identified using a hash function. This approach may lead to some false conflicts but can support transactional execution of C and C++ programs.

Figure 7 shows the pseudocode for the STM. Tx_Begin() clears the per-transaction data structures for data versioning and conflict detection and takes a register checkpoint using the Pin API. Three barrier functions are provided to annotate transactional accesses. RD_barrier() adds the address to the read-set for conflict detection. WR_barrier() adds the address to the write-set for conflict detection and creates an undo-log entry with the current value of



Access Type	Conflict Detection Barrier	Data Versioning Barrier
Idempotent_stack	No	No
Stack	No	Yes
Private	No	Yes
Benign_race	No	Yes
Shared	Yes	Yes

(a) Access Categorization Process (b) STM barrier needed for each Access Type

Figure 6. The five data access types and their required TM barriers.

```
Tx_Begin(){
 PIN::Checkpoint(buf);
 RdSet.clear();
 WrSet.clear();
 UndoQ.clear();}

Tx_Commit(){
 foreach addr in RdSet{
  lock=lockTable.get(addr);
  if(lock!=myID && lock!=RdSet.get(addr)){
   Tx_Abort();} }
 foreach addr in WrSet{
  nextVer=WrSet.get(addr)+2;
  lockTable.set(addr, nextVer);}}

Tx_Abort(){
 foreach addr in UndoQ{
  *addr=UndoQ.get(addr);}
 foreach addr in WrSet{
  nextVer=WrSet.get(addr)+2;
  lockTable.set(addr, nextVer);}
 PIN::Restore(but);}
```

```
WR_barrier(addr){
 lock=lockTable.get(addr);
 if(lock==myID){
  UndoQ.insert(addr,*addr);
  return;
 } elif(lock%2==0 && CAS(myID,lock,addr)){
  WrSet.insert(addr,lock);
  UndoQ.insert(addr,*addr);
  return;}
 Tx_Abort();}

RD_barrier(addr){
 lock=lockTable.get(addr);
 if(lock==myID){
  return;
 } elif(lock%2==0){
  RdSet.insert(addr,lock);
  return;}
 Tx_Abort();}
```

Figure 7. The pseudocode for a subset of the STM system.

the address. WRlocal_barrier() does not create an undo-log entry but adds the address into the write-set. This barrier is used for the Stack, Private, and Benign_race access types.

WR_barrier() first checks the lock word for the address using compare-and-swap. If it is locked by another transaction, a conflict is signaled. If not, the transaction remembers the current version number and sets its ID in the lock word. It then creates the undo-log entry. Note that there can be multiple entries for the same address because we do not check for duplicates. Since the transactions generated the DBT are typically small (50 to 200 instructions), duplicates are not common. Hence, we prefer to waste some storage on the occasional duplicate rather than take the time to search through the undo-log on each WR_barrier() call. RD_barrier starts by checking the corresponding lock word. If it is locked, a conflict is signaled. If not, the version number is recorded in the read-set.

Tx_End() first validates the transaction. The lock words for all addresses in the read-set are checked. If any entry has a different version number than the one in the read-set, or is locked by another transaction, a conflict is signaled. Once the read-set is validated, the transaction commits by releasing the locks for all addresses in the write-set. The version numbers in the corresponding lock words are incremented by 2 to indicate that the data has been updated. On a conflict, the transaction rolls back by applying the undo-log and then releasing the locks for addresses in its write-set. It then restores the register checkpoint. We re-execute aborted transactions after a randomized backoff period.

	Register Checkpointing	Conflict Detection	Data Versioning
STM	SW (multi-cycle)	SW read-set /write-set	SW undo-log
STM+	HW (single-cycle)	SW read-set /write-set	SW undo-log
HybridTM	HW (single-cycle)	HW signatures	SW undo-log
HTM	HW (single-cycle)	HW read-set /write-set	HW undo-log

Table 2. The characteristics of the four TM systems evaluated in this paper.

5.3 Emulation of Hardware Support for TM

We also evaluated the overhead of DBT transactions by emulating three systems with hardware support for transactional execution. Table 2 summarizes their characteristics. We used emulation instead of simulation because Pin is available only in binary form. Hence, it is very difficult to run Pin on a hardware simulator with ISA extensions that control the TM hardware support.

The first hardware system, *STM+*, is the same as our initial STM but uses a fast, hardware-based mechanism for register checkpointing. Such mechanisms are generally useful for speculative execution and are likely to be common in out-of-order processors [2]. We emulate the runtime overhead of hardware-based

CPU	4 dual-core Intel Xeon CPUs, 2.66 GHz
L2 Cache	1 MByte per dual-core CPU
Memory	20 GBytes shared memory
Operating System	Linux 2.6.9 (SMP)
DBT framework	Pin for IA32 (x86) Linux

Table 3. The evaluation environment.

checkpointing by substituting the call to the Pin checkpointing function with a single load instruction. The second hardware system, *HybridTM*, follows the proposals for hardware acceleration of software transaction conflict detection [6, 31, 36]. Specifically, we target the SigTM system that uses hardware signatures for fast conflict detection [6]. We emulate HybridTM by substituting the read and write barrier code in Figure 7 with the proper number of arithmetic and load/store instructions needed to control the SigTM signatures. HybridTM uses a hardware checkpoint as well but maintains the undo-log in software just like STM. At commit time, it does not need to traverse the read-set or write-set for conflict detection. The final hardware system, *HTM*, represents a full hardware TM system [14, 21]. We emulate it by eliminating the read and write barriers as HTM systems perform transactional bookkeeping transparently in hardware. Register checkpointing takes a single cycle and a successful transaction commit takes 2 cycles (one for validation and one to clear the cache metadata).

When executing a program using STM+, HybridTM, or HTM, we cannot roll back a transaction, as we simply emulate the runtime overhead of transactional bookkeeping without implementing the corresponding functionality. This did not cause any correctness problems during our experiments as we did not launch a security attack against our DIFT tool at the same time. In terms of accuracy, our results for the hardware TM systems do not include the overhead of aborted transactions. As shown in Section 6, the abort ratio for DBT transactions is extremely low (less than 0.03% on average). We also do not account for the overhead of false conflicts in hardware signatures or overflows of TM metadata from hardware caches. Since DBT transactions are fairly small, such events are also rare. Hence, we believe that the performance results obtained through HW emulation are indicative of the results that would be obtained if detailed simulation were an option.

6. Evaluation

Table 3 describes our evaluation environment, which is based on an 8-way SMP x86 server. Our DIFT tool is implemented using the Pin DBT framework [19]. Pin runs on top of Linux 2.6.9 and GCC 3.4.6. We used nine multithreaded applications: *barnes*, *fmm*, *radix*, *radiosity*, *water*, and *water-spatial* from SPLASH-2 [40]; *equake*, *swim*, and *tomcatv* from SPECcomp [37]. These applications are compute-bound and achieve large speedups on SMP systems. They are well-suited for our performance experiments, as the overhead of introducing transactions cannot be hidden behind I/O operations. All applications make use of the Pthreads API.

Table 4 presents the basic characteristics of the transactions introduced by our DBT tool when using one transaction per DBT trace. Transactions are relatively short, with the average length varying between 50 and 250 instructions, including those needed for DIFT. The SPEComp benchmarks have longer transactions because Pin extracts longer traces from loop-based computations. When using an STM system without the optimizations in Section 4.2, our tool introduces transactions with 10 read barriers and 5 write barriers on average. Transactions rarely abort (less than 0.03% on average). This is expected as these are highly parallel applications with little sharing between parallel threads.

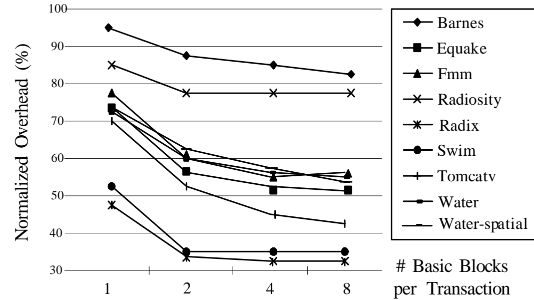


Figure 9. The overhead of STM transactions as a function of the maximum number of basic blocks per transaction.

6.1 Baseline Overhead of Software Transactions

Figure 8 presents the baseline overhead for our DBT tool when using software (STM) transactions without any hardware support for TM. These results are indicative of the performance achievable on existing multiprocessor systems. We measure the overhead by comparing the execution time of the DIFT tool with STM transactions (thread-safe) to the execution time of the DIFT tool without transactions (not thread-safe). Lower overheads are better. The DBT uses one transaction per trace in this case.

The average runtime overhead for software transactions is 41%. The cost of transactions is relatively low, considering that they make the DBT thread-safe and allow speedups of 4x to 8x on the measured system systems. The STM overhead varies between 26% (radix) and 56% (fmm). This variance is significantly lower than that observed in previous STM systems. This is because the tool introduces transactions at the level of DBT traces (a few basic blocks) where the influence of the algorithmic difference between applications is diluted to some extent. The exact value of the overhead does not always follow the transaction length and number of read/write barriers presented in Table 4 for two reasons. First, STM transactions have a higher impact on cache-bound applications, particularly when the STM barriers have low locality. Second, the averages in Table 4 hide significant differences in the exact distribution of the statistics.

6.2 Effect of Transaction Length

The results in Figure 8 assume one transaction per DBT trace, the largest transactions our DBT environment can support without significant modifications. Figure 9 illustrates the importance of transaction length. It shows the overhead of STM transactions in the DBT-based DIFT tool as a function of the maximum number of basic blocks per transaction (1 to 8). If the DBT trace includes fewer basic blocks than the maximum allowed per transaction, the transaction covers the whole trace. Otherwise, the trace includes multiple transactions.

As expected, the overhead of using one basic block per transaction is excessive, more than 70%, for the majority of applications. The computation in one basic block is not long enough to amortize the overhead of STM transactions. As the maximum number of basic blocks per transaction grows, the overhead decreases, as the cost of starting and ending transactions is better amortized. Moreover, larger transactions increase the scope of DBT optimizations and benefit more from locality. At up to 8 basic blocks per transaction, several applications reach the low overheads reported in Figure 8 because most of their traces include less than 8 basic blocks, or 8 blocks include sufficient computation to amortize

Application	# Instr. per Tx	# LD per Tx	# ST per Tx	# RD Barriers per Tx	# WR Barriers per Tx	Abort Ratio (%)
Barnes	81.21	9.07	5.49	5.61	3.60	0.01
Equake	118.68	15.42	4.90	9.93	3.35	0.02
Fmm	111.42	14.60	8.28	8.77	5.68	0.03
Radiosity	61.85	7.27	5.44	4.02	3.18	0.00
Radix	118.70	18.89	12.30	11.00	17.29	0.02
Swim	249.76	34.20	6.79	20.81	4.92	0.03
Tomcatv	118.77	13.19	6.13	8.78	4.13	0.00
Water	55.25	7.31	3.09	4.33	2.07	0.00
Water-spatial	60.93	8.03	3.79	4.81	2.53	0.00

Table 4. The characteristics of software (STM) transactions introduced by our DBT tool.

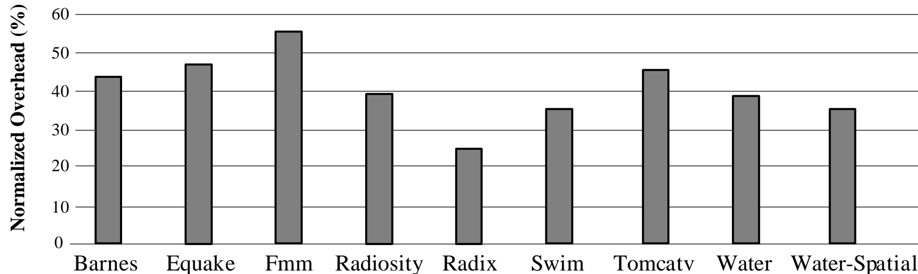


Figure 8. Normalized execution time overhead due to the introduction of software (STM) transactions.

transactional overheads. This is not the case for Barnes, Radiosity, Water, and Water-spatial which have longer traces and benefit from longer transactions.

As the results for STM+ suggest in Section 6.4, most applications would also benefit from transactions that span across trace boundaries. To support multi-trace transactions, we would need dynamic support for transaction merging as traces execute (see Section 4.1).

6.3 Effect of Access Categorization

The baseline STM transactions in Figure 8 use read and write barriers to protect all accesses to data and metadata. In Section 4.2, we discussed how, in certain cases, we can use additional analysis of access types in order to reduce the overhead of STM instrumentation. To measure the effectiveness of these optimizations, we implemented two simple analysis modules in our DBT system that identify Stack and Benign_race accesses respectively. We classify all accesses that are relative to the stack pointer, as Stack. To identify Benign_race accesses, we use the DBT to add a per-thread counter that is incremented on pthread_mutex_lock() and decremented on pthread_mutex_unlock(), the only synchronization primitives used in our applications. A memory access is classified as a Benign_race if it occurs when the counter is zero. This analysis requires a priori knowledge of the synchronization functions used by the application. Note that in both analyses, we optimize the STM overhead for data accesses. All metadata accesses are fully instrumented to ensure correctness.

Figure 10 shows the impact of the two optimizations on the runtime overhead of STM transactions. The left-most bar (unoptimized) represents the results with full STM instrumentation from Section 6.1. Figure 10 shows that optimizing STM barriers for Stack accesses reduces the overhead of transactions by as much as 15% (radix) and 7% on the average. Optimizing the STM barriers for Benign_race accesses reduces the overhead by 5% on the average. Overall, the results indicate that software optimizations based on access-type classification can play a role in reducing the overhead of transaction use in DBT-based tools.

6.4 Effect of Hardware Support for Transactions

Finally, Figure 11 shows the reduction in the overhead as the amount of hardware support for hardware execution increases. As explained in Section 5.3, we emulate three hardware schemes: STM+, which provides hardware support for register checkpointing in STM transactions; HybridTM, which uses hardware signatures to accelerate conflict detection for STM transactions; and HTM, a fully-featured hardware TM scheme that supports transactional execution without the need for read or write barriers. For reference, we also include the original results with software-only transactions.

STM+ reduces the average overhead from 41% to 28%. Hardware checkpointing is particularly useful for small traces for which a software checkpoint of registers dominate execution time. HybridTM reduces the average overhead to 12% as it reduces the overhead of conflict detection in the read and write barriers (read-set and write-set tracking). The full HTM support reduces the overhead of using transactions in the DBT-based DIFT tool down to 6% by eliminating read and write barriers within the traces. Overall, Figure 5.3 shows that hardware support is important in reducing the overhead of transactions in DBT tools. Nevertheless, it is not clear if a fully-featured HTM is the most cost-effective approach. The biggest performance benefits come from hardware register checkpointing and hardware support for conflict detection in software transactions.

7. Related Work

There have been significant efforts to develop general-purpose DBT systems, such as Dynamo [3], DynamoRIO [5], Valgrind [24], Pin [19], StarDBT [4], and HDtrans [38]. Apart from DIFT-based security tools, these frameworks have been used for performance optimizations [3, 38], profiling [19], memory corruption protection [18], and software bug detection [23, 34]. To the best of our knowledge, no existing DBT framework supports a functionally-correct, low-overhead method for handling metadata consistency issues in multithreaded programs. They require explicit locking by

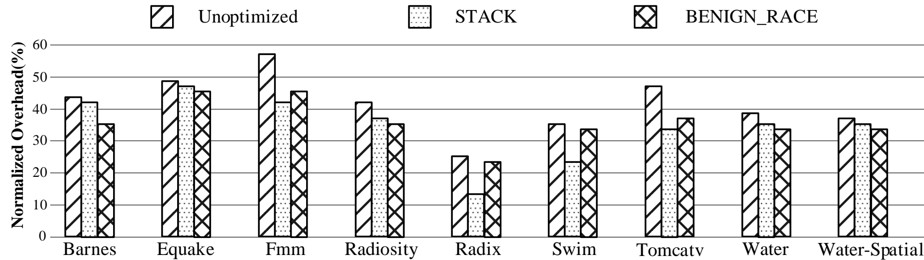


Figure 10. The overhead of STM transactions when optimizing TM barriers for Stack and Benign_race accesses.

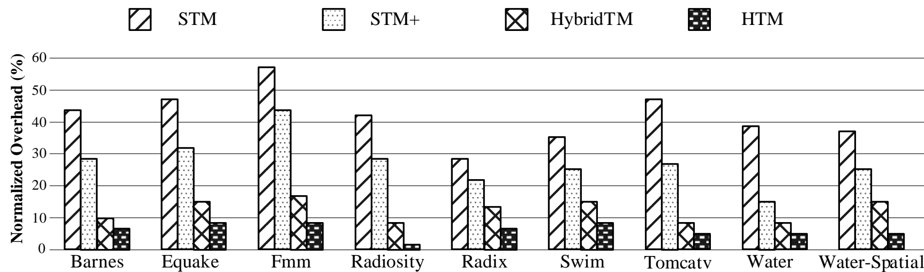


Figure 11. Normalized execution time overhead with various schemes for hardware support for transactions.

the tool developer, thread serialization, or disallow multithreaded programs altogether. This work provides the first in-depth analysis of the issue and proposes a practical solution.

Significant effort has also been made to develop DIFT as a general purpose technique for protecting against security vulnerabilities. DIFT has been implemented using static compilers [41], dynamic interpreters [26, 27], DBTs [4, 28, 8], and hardware [10, 11]. The advantage of the DBT-based approach is that it renders DIFT applicable to legacy binaries without requiring hardware modifications. The LIFT system proposed a series of optimizations that drastically reduce the runtime overhead of DBT-based DIFT [28]. Our work complements LIFT by proposing a practical method to extend DBT-based DIFT to multithreaded programs.

The popularity of multi-core chips has motivated several TM research efforts. HTM systems use hardware caches and the coherence protocol to support conflict detection and data versioning during transactional execution [14, 21]. HTMs have low bookkeeping overheads and require minimal changes to user software. STM systems implement all bookkeeping in software by instrumenting read and write accesses within transactions [12, 15, 30]. STMs run on existing hardware and provide full flexibility in terms of features and semantics. To address the overhead of STM instrumentation, researchers have proposed compiler optimizations [1] and hybrid TM systems that provide some hardware support for conflict detection in STM code [6, 31, 36]. More recently, there have also been efforts to use TM mechanisms beyond concurrency control. In [22], TM supports complex compiler optimizations by simplifying compensation code.

8. Conclusions

This paper presented a practical solution for correct execution of multithreaded programs within dynamic binary translation frameworks. To eliminate races on metadata accesses, we proposed the use of transactional memory techniques. The DBT uses transactions to encapsulate all data and metadata accesses within a trace into one atomic block. This approach guarantees correct execution as TM mechanisms detect and correct races on data and metadata updates. It also maintains the high performance of multithreaded execution as DBT transactions can execute concurrently.

To evaluate this approach, we implemented a DBT-based tool for secure execution of x86 binaries using dynamic information flow tracking. This is the first such tool that correctly handles multithreaded binaries without serialization. We showed that the use of software transactions in the DBT leads to runtime overhead of 40%. We also demonstrated that software optimizations in the DBT and hardware support for transactions can reduce the runtime overhead to 6%. Overall, we showed that TM allows metadata-based DBT tools to practically support multithreaded applications.

9. References

[1] A.-R. Adl-Tabatabai, B. Lewis, et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[2] H. Akkary et al. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *the Proc. of the 36th Intl. Symp. on Microarchitecture*, San Diego, CA, Dec. 2003.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[4] E. Borin, C. Wang, et al. Software-based Transparent and Comprehensive Control-flow Error Detection. In *the Proc. of the 4th Intl. Symp. Code Generation and Optimization (CGO)*, New York, NY, March 2006.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, Austin, TX, December 2001.

[6] C. Cao Minh, M. Trautmann, et al. An Effective Hybrid Transactional Memory System with Strong Isolation

Guarantees. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[7] A. Chernoff, M. Herdeg, et al. FX!32 A Profile-Directed Binary Translator. *IEEE Micro*, 18(2), Mar. 1998.

[8] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *the Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, London, UK, July 2007.

[9] M. Costa, J. Crowcroft, et al. Vigilante: End-to-End Containment of Internet Worms. In *the Proc. of the 20th ACM Symp. on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[10] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *the Proc. of the 37th Intl. Symp. on Microarchitecture (MICRO)*, Portland, OR, December 2004.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *the Proc. of the 20th Intl. Symp. on Distributed Computing*, Septempber 2006.

[13] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *9th Intl. Conf. on Compiler Construction (CC)*, Berlin, Germany, Mar. 2000.

[14] L. Hammond, V. Wong, et al. Transactional Memory Coherence and Consistency. In *the Proc. of the 31st Intl. Symp. on Computer Architecture (ISCA)*, Munich, Germany, June 2004.

[15] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *the Proc. of the 18th Conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003.

[16] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture (ISCA)*, May 1993.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *the Proc. of the 11th USENIX Security Symp.*, San Francisco, CA, August 2002.

[19] C. Luk, R. Cohn, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[20] P. McKenney and J. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998.

[21] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture (HPCA)*, Austin, TX, February 2006.

[22] N. Neelakantam, R. Rajwar, et al. Hardware Atomicity for Reliable Software Speculation. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Jose, CA, June 2007.

[23] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program . In *the Proc. of the 2007 ACM Intl. Conf. on Virtual Execution Environments (VEE)*, San Diego, CA, June 2007.

[24] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[25] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *the Proc. of the Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, February 2005.

[26] A. Nguyen-Tuong, S. Guarnieri, et al. Automatically Hardening Web Applications using Precise Tainting. In *Proc. of the 20th IFIP Intl. Information Security Conf.*, Chiba, Japan, May 2005.

[27] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *the Proc. of the Recent Advances in Intrusion Detection Symp.*, Seattle, WA, Sept. 2005.

[28] F. Qin, C. Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *the Proc. of the 39th the Intl. Symp. on Microarchitecture (MICRO)*, Orlando, FL, December 2006.

[29] Rosetta. `http://www.apple.com/rosetta/`.

[30] B. Saha, A.-R. Adl-Tabatabai, et al. A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *the Proc. of the 11th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[31] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *the Proc. of the 39th Intl. Symp. on Microarchitecture (MICRO)*, Orlando, FL, December 2006.

[32] K. Scott and J. Davidson. Safe Virtual Execution Using Software Dynamic Translation. In *the Proc. of the 18th Annual Computer Security Applications Conf. (ACSAC)*, Las Vegas, NV, December 2002.

[33] Security-Enhanced Linux. `http://www.nsa.gov/selinux/`.

[34] J. Seward and N. Nethercote. Using Valgrind to detect Undefined Value Errors with Bit-Precision. In *the Proc. of the USENIX 2005 Annual Technical Conf.*, April 2005.

[35] T. Shpeisman, V. Menon, et al. Enforcing Isolation and Ordering in STM. In *the Proc. of the ACM SIPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[36] A. Shriraman, M. Spear, et al. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[37] Standard Performance Evaluation Corporation, *SPEC OpenMP Benchmark Suite*. `http://www.spec.org/omp`.

[38] S. Sridhar, J. Shapiro, et al. HDTrans: an Open Source, Low-Level Dynamic Instrumentation System. In *the Proc. of the ACM/USENIX Intl. Conf. on Virtual Execution Environments (VEE)*, Ottawa, ON, June 2006.

[39] VMware. `http://www.vmware.com/`.

[40] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture (ISCA)*, Santa Margherita, Italy, June 1995.

[41] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *the Proc. of the 15th USENIX Security Conf.*, Vancouver, Canada, Aug. 2006.