

tm_db: A Generic Debugging Library for Transactional Programs

Maurice Herlihy

Brown University Computer Science

herlihy@cs.brown.edu

Yossi Lev

Brown University Computer Science and Sun Microsystems

levyossi@cs.brown.edu (contact author)

Abstract

Transactional Memory (TM) has received a lot of attention as a programming API for concurrent programs on emerging multicore architectures. If the transactional programming model is to realize its promise of simplifying the problem of writing correct and scalable concurrent programs, debuggers will have to change.

In this paper, we introduce `tm_db`, an open-source library to provide debuggers with a general debugging support for transactional programs. The library helps debuggers provide programmers with generic transactional debugging features, independent of the particular TM's runtime internals. In addition, it provides TM designers with a well defined interface for transactional debugging support.

We discuss the basic debugging features we believe are essential to debug transactional programs, how they are provided by the library, and how they integrate into a general debugging infrastructure.

I. Introduction

Transactional memory (TM) promises to make multi-threaded programming easier than programming with conventional locks and condition variables, mostly by reducing non-determinism, thus simplifying the programmer's model of computation. For the same reasons, debugging transactional programs should also be easier. Many properties that we take for granted in sequential programming, such as the ability to describe data structures by invariants, and methods by pre and post-conditions, fail when using

locks and condition variables, but are mostly restored with transactional programming.

Debugging single-threaded (sequential) programs, while hardly easy, does follow certain principles. Consider the problem of debugging a complex data structure, such as a red-black tree. In a quiescent state, the tree satisfies a (possibly complicated) invariant governing how it is balanced. To debug a sequential tree implementation, one should check that the tree satisfies its invariant when each method is called, and again when the method returns. Naturally, a method in progress may temporarily violate the tree's invariant.

When debugging multi-threaded lock-based programs, these principles evaporate. A method call rotating a subtree may cause keys temporarily to disappear, or to be duplicated, or the tree to become temporarily unbalanced. In programs based on fine-grained locking, some critical section may always be in progress, and the tree may *never* be entirely consistent. Halting a thread in a critical section could reveal such anomalies, making it difficult to understand whether the tree implementation is correct. There are simply too many interleavings to distinguish good states from bad.

When debugging multi-threaded transactional programs, we can reclaim the sequential invariants. Recall that transactions execute in *isolation*, implying that no transaction can observe another's partial effects. The user (that is, the person using the debugger) should never see the partial effects of any transaction other than the one being debugged. Each transaction "sees" a quiescent tree when it starts, and the user can check that observed departures from the tree invariant are due to the partial effects of the transaction being debugged.

Transactional debugging has other, perhaps less-obvious advantages. Today, when a user stops a thread at a break-

point and steps over a function call, the Sun Studio dbx debugger [15] lets all other threads run while the debugged thread is executing this step. Clearly, this policy makes reasoning about program behavior more difficult, but it was found necessary to avoid deadlocks that could arise if the function were to wait on a condition variable signaled by a concurrent thread. By contrast, when debugging transactional code, there is no need for such a policy. The debugged transaction can always be aborted if it is blocked by another thread. Alternatively, the debugger can take advantage of the underlying TM system and find out which transaction is blocking the debugged thread, and allow the user to either abort it or step it to completion. These choices are not available using locking. If the underlying TM is obstruction-free (for example, DSTM [1]), then a single-stepped transaction will always succeed when run in isolation. Either way, stepping can be done in isolation, and is thus easier to understand.

Another advantage of transactional models is that TM runtimes naturally keep track of lots of information useful for debugging, such as transaction read and write sets, tentative versus committed values, data conflicts, and who is waiting for whom. This kind of information is difficult (or impossible) to assemble using lock-based synchronization.

To exploit these aspects of transactional programs, however, debuggers must change [2]. The problem is that a TM runtime provides only the *illusion* of isolation and atomicity. The debugger must cooperate with the runtime to preserve this illusion for the user debugging the program, by showing the state as seen by the debugged program, and not exposing unrelated artifacts of the TM runtime.

Today, the research literature encompasses many alternative software transactional memory (STM) systems¹ (for example, [3], [4], [5], [1], [6], [7], [8], [9], [10], [11], [12]) that differ in many ways. Some update objects in place, and log undo operations in case the transaction must be rolled back. Others buffer updates, and apply those updates only when the transaction commits. Some detect conflicts as they occur, while others detect conflicts only when transactions attempt to commit. There are many, many ways to implement software transactions, with many trade-offs, and there is no reason to believe these differences will be resolved soon. We think it is unreasonable, as well as undesirable, to expect each STM implementation to provide its own debugging variant, with its own debugging interface, for transactional programs that are using it.² Nevertheless, although STMs may differ

¹In this paper, we focus on debugging software transactional memory systems, although some of the ideas described here could be applied to hardware.

²See Harmanci et al. [13] for a discussion how to debug and test a TM runtime.

on internal organization, they all provide essentially the same high-level functionality. Data objects may be opened for reading or writing. Each object has a committed value, as well as a tentative value proposed by active transactions. A transaction may be committed, aborted, or in a transient cleanup state after committing or aborting.

In this paper we present `tm_db`, an open-source library that exploits these high-level similarities to define a standard debugging interface, common to all these systems, so that any debugger can be used to debug transactional programs that use almost any STM, without knowing (much) about that STM's internal structure. Our library not only provides debuggers with an interface that is independent of the particular STM in use, but also provides STM designers with a well defined interface for debugging support.

Here are this paper's contributions. We describe:

- Basic features needed to debug transactional programs, and how they can be made into an TM-independent debugging interface.
- The design of a generic debugging infrastructure, that can be used by different debuggers, and be extended to support many different TM runtimes.
- How an existing STM (SkySTM [14]) was modified to support debugging using `tm_db`.

We hope the community will extend this open-source library to support additional TM systems, and additional debugging features. There is a parallel effort (not further discussed) to extend the Sun Studio dbx debugger to support transactional debugging using `tm_db`.

The rest of this paper is organized as follows. In Section II, we describe essential features for debugging transactional programs. In Section III, we describe the overall design of our solution. Finally, in Section IV we describe our experience implementing this solution in a specific TM runtime.

II. Debugging Transactional Programs

This section describes basic debugging features we believe are essential for transactional debugging, and how they are provided by `tm_db`.

A. Logical Values

As noted, STM runtimes often deploy complex schemes to preserve the valuable illusion of simplicity provided by the transactional model. For example, consider a simple phase-based program, where the phase is governed by a shared counter. The user halts the program at a breakpoint, and observes that the counter value is 3. Can the user assume that the effects of all Phase 2 transactions are

reflected in memory? In STM runtimes that use deferred updates, some Phase 2 transactions may have committed, but their effects may not have been written to memory. Similarly, the atomicity property may be violated with STM runtimes that use undo logs (e.g., [6]), where some Phase 2 transactions may have aborted, but their effects may not yet have been undone. A naïve debugger that simply examines memory might well break the illusion, confusing the user with run-time artifacts unrelated to the program being debugged.

We therefore believe that the debugger should expose the effects of a transaction atomically (all at once), and exactly at the transaction’s commit point.³ We define the *logical value* of a memory location *L* to be the latest value written to *L* by either a committed transaction (whether or not that value actually appears in *L*), or a non-transactional write.⁴ When examining logical values, a transaction’s effect is exposed atomically when it commits. Because different STM runtimes manage memory in different ways, the `tm_db` library helps the debugger figure out the logical value of a memory location, by providing a method that returns the logical value of a memory word. Note that the logical value does not reflect tentative writes that were done by transactions that have not yet committed. We later describe how the debugger can provide information about such tentative writes when presenting data to the user.

B. Transaction Identity

We distinguish between three distinct notions of transaction. An *atomic block* is a *lexical* scope, corresponding to the lines of code executed by a transaction. A *logical transaction* occurs at run time when a thread executes an atomic block. The same atomic block can produce multiple logical transactions, one for each of its successful executions.⁵ Finally, a logical transaction can produce a sequence of *physical transactions*, because an execution of a logical transaction may be rolled back and retried, possibly several times.

To capture the distinction between logical and physical transactions, the library represents a transaction as a triple *T.L.P*, where *T* is the thread executing the transaction, *L* is the total number of logical transactions completed by the thread, and *P* is the number of physical transactions that failed executing the current logical transaction. For example, Transaction 3.4.1 indicates the 2nd try of the 5th logical transaction of thread 3. Thus, when stopping at

³By “commit point” we refer to the point in which the transaction logically takes effect (its linearization point).

⁴When the underlying TM does not provide strong atomicity, this definition assumes that there are no concurrent transactional and non-transactional writes to the same location.

⁵A successful execution does not necessarily mean that the transaction has successfully committed – it might have been self-aborted.

two different points, the user can tell what progress each thread has made, for example whether it is still executing the same physical transaction, the same logical transaction, and so on.

An atomic block is identified by the address of its first instruction, easily translated to a specific function name and offset, or to a file and line number.

The library provides the transaction and atomic block id for the *latest transaction* begun by each thread (even if the transaction has already completed). Figure 1 shows a usage example,⁶ in which a debugger uses the library to present the transaction and atomic block ids of the latest transactions for all running threads, as well as their statuses, which we describe next.

Thread	Latest TxId	Status	Atomic Block
t@1	<None>		<None>
t@2	2.1907.0	Committed	PushHead+0x0008
t@3	3.2217.1	Active	PushTail+0x0008
> t@4	4.2082.0	Aborting	PushHead+0x0008
t@5	5.2107.0	Committing	PushHead+0x0008
t@6	6.2210.0	Invalid	PushTail+0x0008

Fig. 1. Transactions Ids, Statuses, and Atomic blocks.

C. Transaction Status

The library represents a transaction’s *status* using three fields:

- The `IsRunning` flag indicates whether the transaction is still running, or has completed.
- The transaction’s `State` can be one of the following values: `Active`, `Committed`, `Aborted`, and `Invalid`.
 - An `Active` transaction is still capable of committing successfully (meaningful only for running transactions).
 - An `Invalid` Transaction can no longer commit successfully (for example, it may have read something modified by another transaction), but is not yet aware of it, and not yet in the process of aborting. Such transactions may still be executing user code, and are sometimes referred to as “zombie transactions”.
 - A `Committed` transaction has already committed successfully. If `IsRunning` is true, the transaction may not have completed post-commit cleanup operations, such as executing deferred writes. Still, the logical values of all locations it has written already reflect the new written values.

⁶All usage examples are from a prototype integrated with `dbx`.

- An **Aborted** transaction has failed to commit successfully. If **IsRunning** is true, the transaction may not have completed some post-abort cleanup operations, such as undoing in-place writes.⁷
- The **IsWaiting** flag reports whether the transaction is blocked by the underlying TM, waiting for another transaction. If so, the address for which it is waiting can be provided. Different TM implementations may have blocked transactions in any one of the above states. For example, an **Active** transaction may be blocked at an access to one location, while it is being rendered **Invalid** by a past access to a different location. (The **IsRunning** flag for a blocked transaction must be true, though.)

Note that even though the library conveys transaction status via these three fields, a debugger may display it differently. For example, in Figure 1, the debugger does not display the **IsRunning** value; instead, it uses **Committed** and **Committing** to distinguish a completed transaction that committed (thread **t@2**) from a running transaction that is logically committed but has not yet completed (thread **t@5**).

D. Read, Writes, Conflicts, and Coverage

A transaction **T** *covers* a location **L** if a write access to **L** by another transaction would cause a conflict with **T**. A transaction covers all locations it has read or written. A *false conflict* occurs when the transaction covers locations it has not explicitly accessed. This typically happens when the underlying TM maps the locations to the same “protection unit” as that of accessed locations. Coverage can be refined to *read* coverage, *write* coverage, or *both*.⁸ Distinguishing coverage from access is helpful for distinguishing true and false data conflicts.

For a location **L** and transaction **T**, the library supports checking whether **T** accessed **L**, whether **T** covers **L**, and if, so, for reading, writing, or both. Access and coverage information is recorded when the access occurs in the user’s code, and not when the transaction physically writes the location, or acquires ownership of it (perhaps by acquiring a corresponding lock). The latter events might occur long after the actual access if the underlying TM uses deferred updates or lazy acquisition. Reporting accesses as they happen in the user’s code is easier for the user to understand, and is independent of the underlying TM.

⁷The library is unaware whether the underlying TM uses deferred or in-place writes. It simply reports the state and whether the transaction is running; the user can decide whether a particular state is interesting based on TM-specific knowledge.

⁸We distinguish write coverage from both because some TMs allow transactions that wrote to the same location to commit as long as neither read.

```
> tx coverage &g_list.Head->next->key
Tid  TxID & Status  Coverage Type  Written  Read
t@2  2595.0 Active  Read & Write  Yes     Yes
t@6   1.0 Active  Read & Write  Yes     No
t@4  2023.0 Active  Read & Write  No     No
```

Fig. 2. Coverage for a given variable

```
> printConf
Tid  TxID & Status  Coverage Type  Written  Read
t@2  2595.0 Active  Write          Yes
t@6   1.0 Active  Write          Yes
t@4  2023.0 Active  Write          Yes
=====
0x10019dc48
Tid  TxID & Status  Coverage Type  Written  Read
t@2  2595.0 Active  Read & Write  Yes     Yes
t@6   1.0 Active  Read & Write  Yes     No
t@4  2023.0 Active  Read & Write  Yes     No
=====
0x10018bde0
```

Fig. 3. Full conflicts report

Figure 2 shows how the debugger can use this information to provide full access and coverage information (by all transactions) for the **key** field in a linked list node. In this example, **t@2** has read and written the **key** field, while **t@6** has only written it, but still covers it for both reading and writing, meaning that it read another location that maps to the same “protection unit” by the underlying TM (a false conflict). On the other hand, **t@4** did not access **key** at all, but covers it anyway for both reading and writing (another false conflict). Note that all transactions still have **Active** status, which is possible with an STM that uses lazy acquisition, since none of the owners has started to commit.

The debugger can also use this functionality to display access and coverage information by the current debugged transaction and for each variable in its variables/watch window. Also, if the transaction’s status indicates it is blocked, the function can find the blocking transaction.

In addition to access and coverage information for a given address, the library provides the ability to iterate through the set of all locations a transaction has accessed, either for read or for write. The iterator provides the address and value written (or read) for each location. This high-level interface hides whether the underlying TM implementation maintains explicit write sets, or instead

uses an undo log.

The debugger can use the write set iterator not only to present the user with the set of written locations and values, but also to provide information on all locations currently involved in conflicts, as illustrated in Figure 3. Since each address involved in a conflict must be written by some transaction, the debugger can build a list of potentially conflicted addresses by iterating through the write sets of all running transactions, and then checking (using the coverage query functionality) which of these addresses is covered by more than one transaction.

Finally, the library provides access and coverage information for the latest transaction of each thread regardless of its status, even if the transaction has already completed. The debugger can choose, for each of its commands, whether to take into account already completed transactions. (For example, the user may be interested in seeing the set of writes done by the previous, committed transaction, but may not be interested in seeing conflicts with these writes.)

E. Sub-word Accesses

An interesting question is the granularity the runtime TM provides with respect to non-transactional accesses. Almost all STM runtimes disallow concurrent transactional and non-transactional access to the same location, but the notion of a “location” varies from one STM to another. In many STMs, a location is a single or double word. Thus, *if a local variable that is accessed non-transactionally happens to lie in the same physical word with a transactional variable, the system may function incorrectly*. Some other STM runtimes, like SkySTM [14], provide sub-word granularity: if a transaction writes a certain byte in a word, neighboring bytes can be written non-transactionally at the same time. To handle such differences in granularity, the write set iterator also provides a mask showing which bytes are affected. (For word-granularity TMs, this mask will always be all ones.)

Note that the mask information can be used when checking whether a variable was accessed by some transaction. This way, in the above example where a variable v that is accessed non-transactionally is in the same physical word with a variable that is accessed by some transaction, the debugger will show that v is accessed transactionally (thus indicating a bug) if and only if the underlying TM does not provide sub-word granularity with respect to non-transactional writes.

F. Usage Example: Viewing Data

When examining data, we believe that the debugger should show the user the data from the point of view of

the debugged program, or more precisely, of the debugged thread. The debugger can easily do so by providing the user with the logical value if the debugged transaction has not written the examined location, or with the value returned by the write set iterator otherwise. We denote this as the *transactional view* of the data.

Note that when the underlying TM provides sub-word granularity with respect to non-transactional writes, it is possible that only some of the bytes of an examined location were written by the debugged transaction, in which case the debugger may need to combine some bytes of the value returned by the write set iterator, with bytes of the logical value. For example, suppose that we have an array of Booleans:

```
bool flag[#threads]
```

Let Tx1 be a transaction that writes `flag[1]` transactionally, Tx2 a transaction that writes `flag[2]` transactionally and T3 be a thread that writes `flag[3]` non-transactionally. Suppose the user asks to see the value of the double word containing flags 0-7, at a time when Tx1 is the debugged transaction that has not yet committed, Tx2 is committed but has not yet completed a deferred update to `flag[2]`, and T3 has already written its flag in the array. The presented value is assembled from the write sets of Tx1 and Tx2, as well as the memory updated in place by T3; the library builds the logical value from the combination of the value in Tx2’s write set and the in-memory value, and the debugger combines the result with the value from the write set of the debugged transaction (Tx1).

Local variables present a slightly different challenge. Some TMs provide methods for a transaction to write local variables without paying the overhead of synchronization, but nevertheless guaranteeing that changes will be undone if the transaction aborts. Isolation for these local writes is usually not enforced because it is assumed that they are not accessed by other threads. At this point, the library provides no special support for local writes beyond the isolation guarantees provided by the underlying TM. A user who examines a local variable accessed by another transaction may observe the other transaction’s ongoing modifications.

Finally, while we believe that by default the debugger should provide the users with the transactional view of the data they examine, it is sometimes useful to override this functionality and examine the actual physical values in memory. Privatization [16] is one such example [16], when a thread uses a transaction to isolate a region of memory, rendering it inaccessible to other threads, and then accesses it non-transactionally. If, when accessed non-transactionally, the logical values in the buffer differ from those in memory, perhaps because some transaction is still in the process of writing committed values to the buffer,

then this non-transactional access is unsafe.⁹ Comparing the transactional and the physical views of the buffer’s memory will reveal such an unsafe access.

G. Transactional Events

Transactional memory introduces new events that the user may want to track or to trigger breakpoints on. Figure 4 shows the four basic event types that can be monitored by `tm_db`. When an event is triggered, information is reported to the user in the form of *reporting parameters*. For example, all events report which thread triggered the event. `TxAbort` reports why the transaction aborted: whether a conflict was triggered with earlier reads or with earlier writes; whether the abort is due to a timeout waiting to access an address (which address?), or did the transaction explicitly abort itself. `TxAbortOther` reports which other thread was aborted, and why (type of conflict).

One of the goals of `tm_db` is to introduce minimal intervention until an event of interest occurs. This is especially important when debugging multithreaded programs, where any intervention may cause a bug to disappear. One way to achieve this goal is to *filter* events in the TM runtime: the debugged thread is not stopped until the event happens with the specified value of the *monitoring parameters*. For example, stop only when a transaction of a given thread is aborted, or aborted for a specific reason. Figure 4 shows the supported monitoring parameters for each event type.

Finally, sometimes it makes sense not to halt a triggering transaction immediately, but rather to wait until the transaction is committed or aborted. For example, when a transaction aborts another, stopping immediately may cause the aborting transaction itself to be aborted, if there is a long delay between when the triggering thread is stopped and the time that the other threads are stopped. Also, deferring the breakpoint may reveal a more complete picture of the transaction that triggered the event. On the other hand, some data not accessed by the transaction may be out of date. For these reasons, `TxAbortOther` reporting can be either immediate or deferred.

H. Scopes

A monitoring *scope* is another runtime filtering tool provided to reduce the number of times a debugged thread needs to be halted. With scopes, a particular event can be monitored for a specific duration. In particular, a thread `T` can monitor an event `E` in one of three scopes: in the *default scope*, occurrences of `E` by any of `T`’s transactions are reported; in the *until-success scope*, only occurrences

⁹Such an unsafe access may happen either because the underlying TM does not provide appropriate implicit privatization support, or because the program’s logic to isolate the buffer is flawed.

of `E` by `T`’s current *logical* transaction are reported; and, in the *transaction scope*, only occurrences of `E` by `T`’s current *physical* transaction are reported. Monitoring of `E` is automatically canceled when thread exits the monitoring scope, without stopping the debugged program.

Here are some usage examples. Once the debugger has stopped inside a transaction, we can resume execution and stop again if and when the debugged transaction aborts (monitoring `TxAbort` in transactional scope). Or, by using both monitoring parameters and scopes, we can stop if the logical transaction aborts involuntarily (monitoring `TxAbort` for any reason but self-abort in the *until-success scope*). Scopes are useful also within the debugger itself. For example by monitoring `TxAbort` in transactional scope, the debugger can notify the user if the debugged transaction is aborted when leaving the lexical scope by “stepping up” from a function to its caller.

In the future, scopes could be combined with additional support to allow placing monitoring commands *in the user program* to provide powerful debugging capabilities. For example, if we want to stop only if an execution of a given atomic block aborts, we can add a command to monitor `TxAbort` in the current thread, *until-success scope*, at the beginning of that atomic block. We can further narrow the monitoring by restricting the reason for the abort (which is a monitoring parameter of the `TxAbort` event). We can provide delayed breakpoints [2], stopping when and if a transaction that executed a given instruction commits, without stopping when the instruction is executed. This is done by adding a command to monitor `TxCommit` for the current thread in transactional scope right after the instruction in the program code. We can also support assertions, tests that an atomic block does not violate an invariant, by asserting the invariant inside the atomic block, and if violated, execute a command to monitor `TxCommit` for the current thread in transactional scope. The invariant is evaluated as part of the transaction, and is thus consistent with the committed transaction’s view of the memory.

While support for placing such monitoring commands in the user code may not be available in commercial debuggers in the near future, we may be able to experiment with such advanced debugging features by providing a runtime extension to `tm_db` that allows the programmer to manually introduce such monitoring commands in their code, and run the modified program in the debugger.

III. Debugging Infrastructure Design

In this section, we describe how we designed `tm_db` to provide debuggers with a TM-independent interface for transactional debugging, and TM designers with a simple interface for transactional debugging support.

Two processes are involved in debugging a program: the

Event name	Trigger	Reporting	Monitoring	Report time
TxBegin	Tx begins	Reporting thread Tx Type: first, retry	Reporting thread	immediate
TxCommit	Tx committed	Reporting thread	Reporting thread	immediate
TxAbort	Tx aborted	Reporting thread, Abort reason: - reads-invalid, - writes-invalid, - timeout, - self-abort	Reporting thread, Abort reason	immediate
TxAbortOther	Tx aborts another	Reporting thread, Aborted thread, Conflict type: - read-write, - write-read, - write-write	Reporting thread, Aborted thread, Conflict type	immediate or deferred

Fig. 4. Transactional Events

debugger process, that runs the debugger code, and the *target* process, that runs the executable being debugged. These processes run in different spaces: the debugger cannot simply access variables and addresses in the target space through pointers, but must rather use some kind of inter-process communication.

The debugger exports the ability to access the target process space via the `proc_service` interface, defined as part of OpenSolaris™ [17]. This interface provides basic accessibility to a process’s space, including the ability to look up symbols (e.g. finding the address of a global variable), and to read and write memory and registers.

The `proc_service` (or a similar) interface is implemented by debuggers such as `dbx`, `TotalView®`, and `gdb` to provide external libraries, like `libthread_db` and `librtld_db`, the ability to access the target process. In this way, the debugger can out-source the implementation of some debugging functionality to external helper libraries.

A. Design

Figure 5 shows the structure of our solution.

We implemented `tm_db` as an external helper library that lives in the debugger process space, and provides the debugger with the functionality described earlier. It uses the `proc_service` interface for access to the target process, similarly to other external debugger helper libraries, such as `libthread_db` for functionality related to threads, and `librtld_db` for functionality related to the run-time linker. Our library can thus be used by any debugger that implements a `proc_service` provider.

A *Remote Debugging Module* (RDM) is the part of the debugging solution that depends on the particular TM run-

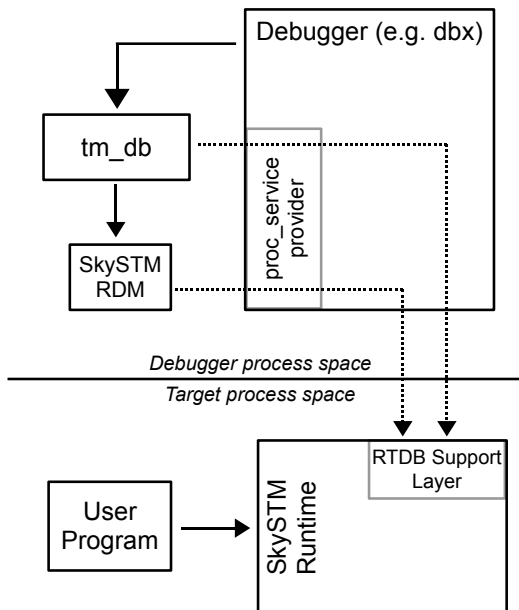


Fig. 5. Debugging with `tm_db`

time. Each TM runtime that supports debugging provides its own RDM that lives in the debugger process space, and accesses the target space through the `proc_service` interface to provide debugging support for that particular TM runtime. When the debugger starts debugging a program, `tm_db` accesses the debugged process to determine whether the program is using a TM runtime that supports debugging, and if so, to dynamically load the appropriate RDM. The RDM provides TM designers a well-defined

interface for providing debugging support for their TM.

So far, we have implemented eight RDMs, to support debugging of transactional programs that use one of eight variants of the SkySTM runtime: easy versus lazy acquisition, visible versus invisible reads, and privatization enabled or disabled. This RDM is open-source to help other TM runtime designers provide their own RDMs.

Note that `tm_db` is really a proxy between the debugger and the RDMs. This additional layer allows us to simplify the RDM implementation by moving common TM-related functionality to `tm_db`. For example, as described in Section IV, SkySTM’s RDMs are stateless: they do not store any information about the TM runtime, debugging session, and so on. We can move functionality between the debugger and `tm_db` without affecting all the RDMs. Having the debugger interface to a single module instead of many RDMs should make it easier to deliver and maintain the RDMs as part of the TM runtime they support.

Finally, in the target process space, the TM runtime is augmented with a *runtime debugging* (RTDB) support layer. This layer has two purposes. First, when a program is loaded, it indicates to `tm_db` which RDM, if any, to load. Second, once an RDM is loaded, it provides the information and functionality required to implement the various debugging features. For example, the RDM configures the RTDB support layer with which transactional events to track, and the RTDB support layer is responsible for the runtime filtering and event reporting, and so on.

Our solution defines the interface between the debugger and `tm_db`, and between `tm_db` and the RDMs, but does not constrain the interface between the RDM and the RTDB Support layer. In particular, this interface can be TM dependent. The TM designer is in the best position to pick the right balance between overhead to the runtime and complexity of the RDM.

B. Supporting Partial Functionality

Not all TM runtimes can support all the debugging features described in Section II. For example, many STMs, including SkySTM [14], do not keep track of the exact addresses read by a transaction. With these runtimes, we cannot check whether an address was accessed by a transaction, but we may still be able to check whether that address is covered. As another example, the TL2 STM provides an optimization with which read-only transactions do not keep *any* information about their transactional reads. For these transactions, even read coverage information is not available.

A more interesting example is that of STM runtimes that may not be able to determine whether some transactions have already committed. These STM runtimes typically do not lock their read set and thus cannot guarantee

that the locations they have read are not modified while attempting to commit. Instead, when trying to commit, a transaction must *read-validate* its read set, checking that the locations it read have not changed since the start of the transaction. With such an STM, a transaction takes effect at the beginning of the read validation only if the read validation completes successfully. Thus, if a transaction is stopped during the read validation, it is undetermined whether it has already committed or not.

To help dealing with such missing functionality, the RDM interface provides various error codes to allow an RDM implementation to indicate that a feature is not supported, either temporarily (for example, it cannot determine this transaction’s status at this point), or permanently (it cannot report read values for transactions). TM designers can thus begin with providing an RDM that only supports a particular subset of the debugging features, and add support to more features over time as found necessary.

Finally, note that some optimizations in the TM runtime may be problematic for debugging support. An example is the optimization that avoids read validation at commit time, and instead linearizes a read only transaction at the time of its first read [5]. The problem with such an optimization is that the transaction may be in a “committed” state while executing user code, when the current state of the data might no longer correspond to that seen by the transaction. Thus, the user may stop a transaction, and will not be able to examine the data as seen by the debugged transaction; this is especially problematic if the STM also does not support the iterator over the read addresses and values.

C. The Import Interface

Our solution will be useful only if debuggers can use the library to provide appropriate commands for debugging transactional programs. There is ongoing work on integrating transactional debugging support in `dbx` using `tm_db`; we hope it will be available to the public soon.

Note, however, that the code interfacing `tm_db` inside `dbx` is not likely to be open source, and thus it will not be possible for researchers to change existing and add new debugging commands.

To address this issue, and to ease gradual integration of transactional debugging support, we plan to provide some functionality using a special `dbx` interface, called the *Import Interface*. The Import Interface enables one to add new commands to `dbx` without changing and recompiling the `dbx` code. This is done by letting an external shared library register new commands and functions to implement them, and then importing the library into `dbx` using the `dbx import` command. Such an external library cannot access the `dbx` internals, but it can use other external libraries like `tm_db`, already loaded in `dbx`. We thus plan

to provide with our `tm_db` library an additional module, to be imported into `dbx` using the import interface, that will provide some additional, experimental debugging features. Because it will be open source, this module will allow researchers to experiment with modifications or addition of new commands, and provide an example for the designers of other debuggers of how to use the `tm_db` library.

IV. Debugging support for an STM runtime

This section describes our experiences implementing RDM and RTDB support for the SkySTM runtime. While these issues are interesting in and of themselves, they also shed light on what it might take to adapt other STMs.

In principle, writing an RDM is a simple matter of writing remote accessor functions for data stored by the TM runtime and the RTDB support. In particular, the RDM uses the `proc_service` interface to poke around the target process's memory. It can locate the (remote) addresses of global variables, read from these addresses, and thus trace pointers in the target process until it reaches its goal.

We designed the interface between the RDM and `tm_db` so that the RDM can be stateless. In particular, the RDM provides `tm_db` with the set of threads that can run transactions, and for each such a thread, it provides a TM-specific key, opaque to `tm_db`, that is passed back to the RDM when `tm_db` queries it for thread-specific information. For SkySTM, we use a pointer to a (remote) thread-private metadata. This way, the RDM does not need to keep track of debugging sessions, because all such state resides in `tm_db`.

Overall, SkySTM supports all of the debugging features described in Section II, except for the ability to iterate through a transaction's read set, or to test whether a particular address was read by it. (Read coverage test is still supported for all transactions.) This support is provided with minimal overhead to the TM runtime: even with a microbenchmark that consists of only very small transactions, the runtime overhead (when the debugger is not attached) is less than 5%.

A. Computing Logical Values

SkySTM, like many other STM runtimes [5], uses deferred updates, meaning that transactional updates to shared memory are buffered in a thread-local buffer, written back after the transaction commits. There is thus a window of time in which the physical values (in memory) of locations written by the transaction do not reflect their logical values. During this time, however, the transaction holds exclusive write ownership of these locations.

Therefore, the SkySTM RDM computes the logical value of a location `L` using the following simple algorithm:

- If no committed transaction holds exclusive write ownership of `L`, return the contents of `L`.
- Otherwise, if `T` is the committed transaction holding write ownership, search `T`'s write set for `L`. If found, return the value from the write set, possibly combined with the actual contents of `L` if `T` did not write all of `L`'s bytes. If not found, `T` covers `L` but has not written it, so return the contents of `L`.

Finally, as noted in Section III-B, it may not always be possible to determine whether the transaction that holds write ownership of `L`, has already committed. In that (extremely rare) case, we simply return a *not currently available* error code. In future releases, we may instead support some kind of *safe-point* mechanism, where the RDM advises the debugger when the transaction is in an "unstable state", and allows the debugger to step it through to a point where the transaction status is unambiguous.

B. Monitoring Events

Event monitoring and filtering in SkySTM is safe and efficient. Monitoring is done on a per-thread basis. Each thread keeps a bit-mask, called the *monitoring command*, indicating which events should be reported. Each combination of an event plus monitoring parameter value (such as "aborted due to invalid reads") has its own bit in the monitoring command.

When an event occurs, a fast-path check tests whether the monitoring command is all-zeros (true whenever monitoring is off, such as when no debugger is attached). Otherwise, the specific bit for the event is checked, and if on, the event is reported. An event is reported by calling an appropriate stub for the event – an empty function in the runtime that the debugger places a breakpoint on. The values for the reporting parameters are passed as arguments to the stub.

To support scopes, the runtime must turn off event monitoring on scope exit. To do that, we keep two additional bit-masks, one which indicates the bits of the monitoring command that should be cleared when the physical transaction ends, and one for those to be cleared when the logical transaction ends. Updating the monitoring command when a transaction ends is thus a simple matter of applying the appropriate bit-mask to the monitoring command, and is only done when some event is monitored in a non-default scope (i.e. one of the masks is non-zero).

V. Future Work and Directions

The `tm_db` library takes a first step toward providing systematic support for debugging transactional programs. It provides only the basic, essential features. We expect that experience debugging real transactional programs will

lead to development of additional debugging features, built on top of the basic library.

One particularly interesting extension would be to move from debugging to profiling. Most TM runtimes provide some system-wide statistics, tracking information like the number of committed and aborted transactions, conflicts, and so on. We are not aware, however, of any program that digests and presents this information to the user in a helpful way, say by displaying which statements and/or addresses caused most of the conflicts, or how many times an execution of a particular atomic block was aborted. We hope to extend the infrastructure described here to provide such profiling information.

Finally, we hope to see more runtime TM systems use `tm_db` to provide debugging support. In particular, we are interested in systems that combine hardware and software support for transactions (for example, SpHT [18]).

VI. Conclusions

We presented a generic debugging framework for transactional programs centered around `tm_db`, an open source library designed to be a common debugging interface for a range of TM runtimes. We discussed the debugging features provided by `tm_db`, why they are essential, and our experience implementing them in a real TM runtime. We hope that our work will lay the base for a transactional debugging standard that will ease the task of designing future TM runtimes, and the task of those who need to make transactional programs work.

Acknowledgments

The development of `tm_db` would not have been possible without the important contribution of many people at Sun Microsystems, particularly Chris Quenelle, Ivan Soleimanipour, and Isaac Chen, who gave valuable advice on making the library useful for a real commercial debugger. We thank the people of the Scalable Synchronization Research Group, for many useful discussions throughout this project. In particular, we would like to thank Mark Moir for co-inventing many of the concepts of transactional debugging [2], and Dan Nussbaum, for the advice on debugging support for SkySTM, and for making open sourcing of this work possible.

References

- [1] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2003, pp. 92–101.
- [2] Y. Lev and M. Moir, "Debugging with transactional memory," in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.
- [3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 227–238.
- [4] S. Dolev, D. Hendler, and A. Suissa, "Car-stm: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, August 2008, pp. 125–134.
- [5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *DISC*, 2006, pp. 194–208.
- [6] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrst-stm: a high performance software transactional memory system for a multi-core runtime," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 187–197.
- [7] J. R. Larus and R. Rajwar, *Transactional Memory*, 2006.
- [8] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive software transactional memory," in *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005, earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.
- [9] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.
- [10] —, "Automatic data partitioning in software transactional memories," in *20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.
- [11] F. Tappa, C. Wang, J. R. Goodman, and M. Moir, "NZTM: Nonblocking, zero-indirection transactional memory," in *Workshop on Transactional Computing (TRANSACT)*, 2007.
- [12] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 261–272.
- [13] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer, "TMunit: Testing Transactional Memories," 2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09), Raleigh, North Carolina, USA, February 15 2009.
- [14] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," 2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09), Raleigh, North Carolina, USA, February 15 2009.
- [15] I. Sun Microsystems, *Sun Studio 12: Debugging a Program With dbx*, Sun Microsystems, Inc., 4150 Network Circle Santa Clara, CA 95054 U.S.A.
- [16] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott, "Privatization techniques for software transactional memory," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2007, pp. 338–339.
- [17] N. A. Solter, J. Jelinek, and D. Miner, *OpenSolaris Bible*. Wiley, 2009, sBN-10: 0470385480.
- [18] Y. Lev and J.-W. Maessen, "Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 197–206.