Debugging with Transactional Memory

Yossi Lev
Brown University & Sun Microsystems Laboratories

Mark Moir
Sun Microsystems Laboratories

ABSTRACT

Transactional programming promises to substantially simplify the development of correct, scalable, and efficient concurrent programs. Designs for supporting transactional programming using transactional memory implemented in hardware, software, and a mixture of the two have emerged recently. To our knowledge, nobody has yet addressed issues involved with debugging programs executed using transactional memory.

Because transactional memory implementations provide the "illusion" of multiple memory locations changing value atomically, while in fact they do not, there are challenges involved with integrating debuggers with such programs to provide the user with a coherent view of program execution. This paper shows how to overcome these problems by making the debugger interact with transactional memory implementations in a meaningful way. In addition to describing how "standard" debugging functionality can be integrated with transactional memory implementations, we also describe some powerful new debugging mechanisms that are enabled by transactional memory infrastructure. Our description focuses on how to enable debugging in software and hybrid software-hardware transactional memory systems.

1. INTRODUCTION

In concurrent software it is often important to guarantee that one thread cannot observe partial results of an operation being executed by another thread. These guarantees are necessary for practical and productive software development because, without them, it is extremely difficult to reason about the interactions of concurrent threads. In today's software practice, these guarantees are almost always provided by using locks to prevent other threads from accessing the data affected by an ongoing operation. Such use of locks gives rise to a number of well known problems, both in terms of software engineering and in terms of performance.

Transactional memory (TM) [7, 16] allows the programmer to think as if multiple memory locations can be accessed and/or modified in a single atomic step. Thus, in many cases, it is possible to complete an operation with no possibility of another thread observing partial results, even without holding any locks. This significantly simplifies the design of concurrent programs.

Transactional memory can be implemented in hardware [7], with the hardware directly ensuring that a transaction is atomic, or in software [16] that provides the "illusion" that the transaction is atomic, even though in fact it is executed in smaller atomic steps by the underlying hardware. Substantial progress has been made in making software transactional memory (STM) practical recently [2, 3, 6, 10]. Nonetheless, there is a growing consensus that at least some hardware support for transactional memory is desirable, and several proposals for supporting TM in hardware have emerged recently [1, 4, 13]. All existing proposals for implementing TM in hardware *either* impose severe limitations on programmers *or* are too complicated and inflexible to be considered in the near future, and also leave a number of issues unresolved. To address this situation, we have proposed *Hybrid TM* (HyTM) [11], which provides a fully functional STM implementation that can exploit *best-effort* HTM support to boost performance *if* it is available and when it is effective. Kumar et. al [8] have recently made a similar proposal.

To our knowledge, none of the TM designs (HTM, STM, or HyTM) proposed to date addresses the issue of debugging programs that use them. While TM promises to substantially simplify the development of correct concurrent programs, programmers will still need to debug code while it is under development, and therefore it is crucial that we develop robust TM-compatible debugging mechanisms.

Debugging poses challenges for all forms of TM. If HTM is to provide support for debugging, it will be even more complicated than current proposals. STM on the other hand provides the "illusion" that transactions are executed atomically, while in fact they are implemented by a series of smaller steps. If a standard debugger were used with an STM implementation, it would expose this illusion, creating significant confusion for programmers. HyTM is potentially susceptible to both problems. In this paper, we describe a series of mechanisms for supporting debugging in STM and HyTM systems. In keeping with the HyTM philosophy, we do not impose any requirement on HTM support for debugging.

For concreteness we describe the debugging techniques

(c) Sun Microsystems, Inc. 2006. All rights reserved.

in the context of a simple word-based HyTM system, such as described in [11]. In Section 2 we give a brief overview of this HyTM system. In Section 3, we describe several debug modes which will aid in the description of our debugging techniques. Section 4 presents debugging techniques in the following topics:

- Breakpoints in atomic blocks.
- Viewing and modifying variables
- Atomic snapshots
- Watchpoints
- Delayed breakpoints
- Replay debugging

2. A WORD-BASED HYTM SCHEME

2.1 Overview

The HyTM system [11] comprises a compiler, a library for supporting transactions in software, and (optionally) HTM support. Programmers express blocks of code that should (appear to) be executed atomically in some language-specific notation. For concreteness, we assume the following simple notation:

```
atomic {
   ...
   code to be executed atomically
   ...
}
```

For each such atomic block, the compiler produces code to execute the code block atomically using transactional support. A typical HyTM approach is to produce code that attempts to execute the block one or more times using HTM, and if that does not succeed, to repeatedly attempt to do so using the STM library.

The compiler also produces "glue" code that hides this retrying from the programmer, and invokes "contention management" mechanisms [6, 15] when necessary to facilitate progress. Such contention management mechanisms may be implemented, for example, using special methods in the HyTM software library. These methods may make decisions such as whether a transaction that encounters a potential conflict with a concurrent transaction should a) abort itself, b) abort the other transaction, or c) wait for a short time to give the other transaction an opportunity to complete. As we will see, debuggers may need to interact with contention control mechanisms to provide a meaningful experience for users.

Because the above-described approach may result in the concurrent execution of transactions in hardware and in software, we must ensure correct interaction of these transactions. The HyTM approach is to have the compiler emit additional code in the hardware transaction that looks up structures maintained by software transactions in order to detect any potential conflict. In case such a conflict is detected, the hardware transaction is aborted, and is subsequently retried, either in hardware or in software. Below we explain how software transactions provide the illusion of atomicity, and how hardware transactions are augmented to detect potential conflicts with software ones.

2.2 Transactional Execution

As a software transaction executes, it acquires "ownership" of each memory location that it accesses: exclusive ownership in the case of locations modified, and possibly shared ownership in the case of locations read but not modified. This ownership cannot be revoked while the owning transaction is in the `active` state: A second transaction that wishes to acquire exclusive ownership of a location already owned by the first transaction must first *abort* the transaction by changing its status to `aborted`. Furthermore, a location can be modified only by a transaction that owns it. However, rather than modifying the locations directly while executing, the transaction "buffers" its modifications in a "write set". Thus, if a transaction reaches its end without being aborted, then all of the locations it accessed have maintained the same values since they were first accessed. The transaction atomically switches its status from `active` to `committed`, thereby *logically* applying the changes in its write set to the respective memory locations it accessed. Before releasing ownership of the modified locations, the transaction copies back the values from its write set to the respective memory locations so that subsequent transactions acquiring ownership of these locations see the new values.

2.3 Ownership

In the word-based HyTM scheme described here, there is an ownership record (henceforth *orec*) associated with each *transactional location* (i.e., each memory location that can be accessed by a transaction). To avoid the excessive space overhead that would result from dedicating one orec to each transactional location, we instead use a special *orec table*. Each transactional location maps to one orec in the orec table, but multiple locations can map to the same orec. To acquire ownership of a transactional location, a transaction acquires the corresponding orec in the orec table. The details of how ownership is represented and maintained are mostly irrelevant here. We do note, however, that the orec contains an indication of whether it is owned, and if so whether in "read" or "write" mode. These indications are the key to how hardware transactions are augmented to detect conflicts with software ones. For each memory access in an atomic block to be executed by a hardware transaction, the compiler emits additional code for the hardware transaction to lookup the corresponding orec and determine whether there is (potentially) a conflicting software transaction. If so, the hardware transaction simply aborts itself. By storing an indication of whether the orec is owned in read or write mode, we allow a hardware transaction to succeed even if it accesses one or more memory locations in common with one or more concurrent software transactions, provided none of the transactions modifies these locations.

2.4 Atomicity

As described above, the illusion of atomicity is provided by considering the updates made by a transaction to "logically" take effect at the point at which it commits, known as the transaction's *linearization point* [5]. By preventing transactions from observing the values of transactional locations that they do not own, we hide the reality that the changes to these locations are in fact made one by one after the transaction has already committed.

If we use such an STM or HyTM package with a standard debugger, the debugger will not respect these ownership rules. Therefore, for example, it might display a pre-transaction value in one memory location and a post-transaction value in another location that is updated by the same transaction. This would "break" the illusion of atomicity, which would severely undermine the user's ability to reason about the program.

Furthermore, a standard debugger would not deal in meaningful ways with the multiple code paths used to execute transactions in hardware and in software, or library calls for supporting software transactions, contention management, etc. In this paper, we explain how to address all of these issues. We also explain how the infrastructure for STM and HyTM can support some powerful new debugging mechanisms.

3. DEBUG MODES

In this document we will distinguish between three basic debug modes:

- *Unsynchronized Debugging:* In this mode, when a thread stops (when hitting a breakpoint, for example), the rest of the threads keep running.

- *Synchronized Debugging:* if a thread stops the rest of the threads also stop with it. There are two synchronized debugging modes:

  - *Concurrent Stepping:* In this mode, when the user asks the debugger to run one step of a thread, the rest of the threads also run while this step is executed (and stop again when the step is completed, as this is a synchronized debugging mode).

  - *Isolated Stepping:* In this mode, when the user asks the debugger to run one step of a thread, only that thread's step is executed.

For simplicity, we assume that the debugger is attached to only one thread at a time, which we denote as the *debugged thread*. If the debugged thread is in the middle of executing a transaction, we denote this transaction as the *debugged transaction*. When a thread stops at a breakpoint, it automatically becomes the debugged thread. Note that with the synchronized debugging modes, after hitting a breakpoint the user can choose to change the debugged thread, by switching to debug another thread.

4. DEBUGGING TECHNIQUES

4.1 Breakpoints in Atomic Blocks

The ability to stop the execution of a program on a breakpoint and to run a thread step by step is a fundamental feature of any debugger. In a transactional program, a breakpoint will sometimes reside in an atomic block. In this section we describe a technique that enables the debugger to stop and step through such a block in the HyTM system, wherein an atomic block may have at least two implementations, for example, one that uses HTM and another that uses STM.

In keeping with the HyTM philosophy, we do not assume that any special debugging capability is provided by the HTM support. Therefore, if the user sets a breakpoint inside an atomic block, in order to debug that atomic block, we must *disable* the code path that attempts to execute this particular atomic block using HTM,[1] thereby forcing it to be executed using STM. If we cannot determine whether a given atomic block contains a breakpoint (for example, in the presence of indirect function calls), we can simply abort the executing hardware transaction when it reaches the breakpoint, eventually causing the atomic block to be executed by a software transaction.

One way to disable the HTM code path is to modify the code for the transaction so that it branches unconditionally to the software path, rather than attempting the hardware transaction. In HyTM schemes in which the decision about whether to try to execute a transaction in hardware or in software is made by a method in the software library, the code can be modified to omit this call and branch directly to the software path. An alternative approach is to provide the debugger with an interface to the software library so that it can instruct the software method to always choose the software path for a given atomic block.

In addition to disabling the hardware path, we must also enable the breakpoint in the software path. This is achieved mostly in the same way that breakpoints are achieved in standard debuggers. However, there are some issues to note.

First, the correspondence between the source code and the STM-based implementation of an atomic block differs from the usual correspondence between source and assembly code: the STM-based implementation uses the STM library functions for read and write operations in the block, and may also use other function calls to correctly manage the atomic block execution. For example, it is sometimes necessary to invoke the STM library method `STM-Validate` in order to verify that the values read by the transaction so far represent a consistent state of the memory. Figure 1 shows an example of an STM-based implementation of a simple atomic block.

The debug information generated by the compiler should reflect this special correspondence to support a meaningful debugging view to users. When the user is stepping in source-level mode, all of these details will be hidden, just as assembly-level instructions are hidden from the user when debugging in source-level mode with

[1]We do not want to disable *all* use of HTM in the program, because we wish to minimize the impact on program timing in order to avoid masking bugs.

```
atomic {
   v = node->next->value;
}

     ⟹

while(true) {
   tid = STM-begin-tran();
   tmp = STM-read(tid, &node);
   if (STM-Validate(tid)) {
      tmp = STM-read(tid, &(tmp->next));
      if (STM-Validate(tid)) {
         tmp2 = STM-read(tid, &(tmp->value));
         STM-write(tid, &v, tmp2);
      }
   }
   if (STM-commit-tran(tid)) break;
}
```

Figure 1: An example of an atomic block and its STM-based implementation.

a standard debugger. However, when the user is stepping in assembly-level mode, all STM function calls are visible to the user, but should be regarded as atomic assembly operations: stepping into these functions should not be allowed.

Another issue is that control may return to the beginning of an atomic block if the transaction implementing it is aborted. Without special care, this may be confusing for the user: it will look like "a step backward". In particular, in response to the user asking to execute a single step in the middle of an atomic block, control may be transferred to the beginning of the atomic block (which might reside in a different function or file). In such cases the debugger may prompt the user with a message indicating that the atomic block execution has been restarted due to an aborted transaction.

Finally, it might be desirable for the debugger to call `STM-Validate` right after it hits a breakpoint, to verify that the transaction can still commit successfully. This is because, with some HyTM implementations, a transaction might continue executing even after it has encountered a conflict that will prevent it from committing successfully. While the HyTM must prevent incorrect behavior (such as dereferencing a null pointer or dividing by zero) in such cases, it does not necessarily prevent a code path from being taken that would not have been taken if the transaction were still "viable". In such cases, it is probably not useful for the user to believe that such a code path was taken, as the transaction will fail and be retried anyway. The debugger can avoid such "false positives" by calling `STM-Validate` after hitting the breakpoint, and ignore the breakpoint if the transaction is no longer viable.

The debugger may also provide a feature that allows the user to abort the debugged transaction, with the option to either retry it from the beginning, or perhaps to skip it altogether and resume execution after the atomic block. Such functionality is straightforward to provide because the compiler already includes code for transferring control for retry or commit, and because most TM implementations provide means for a transaction to explicitly abort itself.

4.1.1 Contention Manager Support

When stepping through an atomic block, it might be useful to change the way in which conflicts are resolved between transactions, for example by making the debugged transaction win any conflict it might have with other transactions. We call such a transaction a *super-transaction*. This feature is crucial for the isolated stepping synchronized debugging mode because the debugged thread takes steps while the rest of the threads are not executing, and therefore there is no point in waiting in case of a conflict with another thread, nor in aborting the debugged transaction. It may also be useful in other debugging modes, because it will avoid the debugged transaction being aborted, causing the "backward-step" phenomenon previously described. This is especially important because the debugged transaction will probably run much slower than other transactions, and therefore is more likely to be aborted.

In some STM and HyTM implementations, particularly those supporting read sharing, orecs indicate only that they are owned in read mode, and do not indicate which transactions own them in that mode (with these implementations, transactions record which locations they have read, and recheck the orecs of all such locations before committing to ensure that none has changed). Supporting the super-transaction with these implementations might seem problematic, since when a transaction would like to get write ownership on an orec currently owned in read mode, it needs to know whether one of readers owning this orec is a super-transaction. One simple solution is to specially mark the orecs of all locations read so far by the debugged transaction upon hitting a breakpoint, and to continue marking orecs newly acquired in read mode as the transaction proceeds. The STM library and/or its contention manager component would then ensure that a transaction never acquires write ownership of an orec that is currently owned by the super-transaction.

4.1.2 Switching between Debugged Threads

When stopping at a breakpoint, the thread that hit that breakpoint automatically becomes the debugged thread. In some cases though, the user would like to switch to debug another thread after the debugger has stopped on the breakpoint. This is particularly useful when using the isolated steps synchronized debugging mode, because in this case the user has total control over all the threads, and can therefore simulate complicated scenarios of interaction between the threads by taking a few steps with each thread separately.

There are a few issues to consider when switching between debugged threads. The first has to do with hardware transactions when using HyTM: it might be that the new debugged thread is in the middle of executing the HTM-based implementation of an atomic block. Depending on the HTM implementation, attaching the debugger to such a thread may cause the hardware transaction to abort. Moreover, because HTM is

not assumed to provide any specific support for debugging, we will often want to abort the hardware transaction anyway, and restart the atomic block's execution using the STM-based implementation.

Again, depending on the HTM support available, various alternatives may be available, and it may be useful to allow users to choose between such alternatives, either through configuration settings, or each time the decision is to be made. Possible actions include:

1. Switch to the new thread aborting its transaction
2. Switch to the new thread but only after it has completed (successfully or otherwise) the transaction (this might be implemented for example by appropriate placement of additional breakpoints).
3. Cancel and stay with the old debugged thread.

Another issue to consider is the combination of the super-transaction feature and the ability to switch the debugged thread. Generally it makes sense to have only one super-transaction at a time. If the user switches between threads, it is probably desirable to change the previously debugged transaction back to be a regular transaction, and make the new debugged transaction a super-transaction. As described above, this may require unmarking all orecs owned in read mode by the old debugged transaction, and marking those of the new one.

4.2 Viewing and Modifying Variables

Another fundamental feature supported by all debuggers is the ability to view and modify variables when the debugger stops execution of the program. The user provides a variable name or a memory address, and the debugger displays the value stored there and may also allow the user to change this value. As explained earlier, in various TM implementations, particularly those based on STM or HyTM approaches, the current logical value of the address or variable may differ from the value stored in it. In such cases, the debugger cannot determine a variable's value by simply reading the value of the variable from memory. The situation is even worse with value modifications: in this case, simply writing a new value to the specified variable may violate the atomicity of transactions currently accessing it. In this section we explain how the debugger can view and modify data in a TM-based system despite these challenges.

The key idea is to access variables that may be accessed by transactions using the TM implementation, rather than directly, in order to avoid the above-described problems. However, there are several important issues to consider in deciding whether to access a variable using a transaction, and if so, with which transaction.

First, the debugged program may contain *transactional* variables that should be accessed using TM and *nontransactional* variables that can be accessed directly using conventional techniques. A variety of techniques for distinguishing these variables exist, including type-based rules enforced by the compiler, as well as dynamic techniques that determine and possibly change the status of a variable (transactional or nontransactional) at runtime (for example, [9]). Whichever technique is used in a particular system, the debugger must be designed to take the technique into account and access variables using the appropriate method. In particular, the debugger should always use transactions to access transactional variables, and nontransactional variables can be accessed as in a standard debugger.[2]

For transactional variables, one option is for the debugger to get or set the variable value by executing a "mini-transaction"—that is, a transaction that consists of the single variable access. The mini-transaction might be executed as a hardware transaction or as a software transaction, or it may follow the HyTM approach of attempting to execute it in hardware, but retrying as a software transaction if the hardware transaction fails to commit or detects a conflict with a software transaction.

If, however, the debugger has stopped in the middle of an atomic block execution, and the variable to be accessed has already been accessed by the debugged transaction, then it is often desirable to access the specified variable from the debugged transaction's "point of view". For example, if the debugged transaction has written a value to the variable, then the user may desire to see the value it has stored, even though the transaction has not yet committed, and therefore this value is not (yet) the value of the variable being examined. Similarly, if the user requests to modify the value of a variable that has been accessed by the debugged transaction, then it may be desirable for this modification to be part of the effect of the transaction when it commits. To support this behavior, the variable can be accessed in the context of the debugged transaction simply by calling the appropriate library function. (We note that it is straightforward to extend existing HyTM and STM implementations to support functionality that determines whether a particular variable has been modified by a particular transaction.)

Note that it is still better to access variables that were not accessed by the debugged transaction using mini-transactions and not the debugged transaction itself. This is because accessing such variables using the debugged transaction increases the set of locations that the transaction is accessing, thereby making it more likely to abort due to a conflict with another transaction.

In general, it is preferable that actions of the debugger have minimal impact on normal program execution. For example, we would prefer to avoid aborting transactions of the debugged program in order to display values of variables to the user. However, we must preserve the atomicity of program transactions. In some cases, it may be necessary to abort a program transaction in order to service the user's request. For example, if the user requests to modify a value that has been accessed by an existing program transaction, then the mini-transaction used to effect this modification may conflict with that program transaction. Furthermore,

[2]In some TM systems, accessing a nontransactional variable using a transaction will not result in incorrect behavior, in which case we can choose to access all variables with transactions.

some STM and HyTM implementations are susceptible to *false conflicts* in which two transactions conflict even though they do not access any variables in common.

In case the mini-transaction used to implement a user request does conflict with a program transaction, several alternatives are possible. We might choose either to abort the program transaction, or to wait for it to complete (in appropriate debugging modes), or to abandon the attempted modification. These choices may be controlled by preferences configured by the user, or by prompting the user to decide between them when the situation arises. In the latter case, various information may be provided to the user, such as which program transaction is involved, what variable is causing the conflict (or an indication that it is a false conflict), etc.

In some cases, the STM may provide special-purpose methods for supporting mini-transactions for debugging. For example, if all threads are stopped, then the debugger can modify a variable that is not being accessed by any transaction without acquiring ownership of its associated orec. Therefore in this case, if the STM implementation can tell the debugger whether a given variable is being accessed by a transaction, then the debugger can avoid acquiring ownership and aborting another transaction due to a false conflict.

4.2.1 Adding and Removing a Variable from the Transaction's Access Set

As described in the previous section, it is often preferable to access variables that do not conflict with the debugged transaction using independent mini-transactions. In some cases, however, it may be useful to allow the user to access a variable as part of the debugged transaction even if the transaction did not previously access that variable. This way, the transaction would commit only if the variable viewed does not change before the transaction attempts to commit, and any modifications requested by the user would commit only if the debugged transaction commits. This approach provides the user with the ability to "augment" the transaction with additional memory locations.

Moreover, some TM implementations support *early-release* functionality [6]: with early-release, the programmer can decide to discard any previous accesses done to a variable by the transaction, thereby avoiding subsequent conflicts with other transactions that modify the released variable. If early-release is supported by the TM implementation, the debugger can also support removing a variable from the debugged-transaction's access set.

4.2.2 Displaying the pre-transaction value of the debugged transaction

Although when debugging an atomic block the user would usually prefer to see variables as they would be seen by the debugged transaction, in some cases it might be useful to see the value as it was before the transaction began (note that since the debugged transaction has not committed yet, this *pre-transaction* value is the current logical value of the variable, as may be seen by other threads). Some STM implementations can easily provide such functionality because they record the value of all variables accessed by a transaction the first time they are accessed. In other STM implementations, the pre-transaction value is kept in the variable itself until the transaction commits, and can thus be read directly from the variable. In such systems, the debugger can display the pre-transaction value of a variable (as well as the regular value seen by the debugged transaction).

4.2.3 Getting values from conflicting transactions

In some cases, it is possible to determine the logical value of a variable even if it is currently being modified by another transaction. As described above, it may be possible for the debugger to get the pre-transaction value of a variable accessed by a transaction. If the debugger can determine that the conflicting transaction's linearization point has not passed, then it can display the pre-transaction value to the user. How such a determination can be made depends on the particular STM implementation, but in many cases this is not difficult.

Another potentially useful piece of information we can get from the transaction that owns the variable the user is trying to view is the *tentative value* of that variable—that is, the value as seen by the transaction that owns the variable. Specifically, the debugger can inform the user that the variable is currently accessed by a software transaction, and give the user both the current logical value of the variable (that is, its pre-transaction value), and its tentative value (which will be the the variable's value when and if the transaction commits successfully).

4.3 Atomic Snapshots

The debugger can allow the user to define an *atomic group* of variables to be read and/or modified atomically. Such a feature provides a powerful debugging capability that is not available in standard debuggers: the ability to get a consistent view of multiple variables even in unsynchronized debug mode, when threads are running and potentially modifying these variables. (It can also be used with synchronized debugging when combined with the delayed breakpoint feature; see Section 4.5.)

Implementing atomic groups using TM is simply done by accessing all variables in the group using one transaction. The variables in the group are read using a single transaction. As for modifications, when the user modifies a variable in an atomic group, the modification does not take effect until the user asks to commit all modifications to the group, at which point the debugger begins a transaction that executes these modifications atomically. The transactions can be managed by HTM, STM or HyTM.

Note that the displayed values of the group's variables may not be their true value at the point the user tries to modify them. We can extend this feature with a *compare-and-swap* option, which modifies the values of the group's variables only if they contain the previously displayed values. This can be done by beginning a transaction that first rereads all the group's variables and compares them to the previously presented values (saved by the debugger), and only if these values all match, applies the modifications using the same transaction. If some of the values did change, the new values

can be displayed.

Finally, the debugger may use a similar approach when displaying a compound structure, to guarantee that it displays a consistent view of that structure. Suppose, for example, that the user views a linked list, starting at the head node and expanding it node-by-node. Because in unsynchronized debugging mode the list might change while being viewed, reading it node-by-node might display an inconsistent view of the list. The debugger can use a transaction to re-read the nodes leading to the node the user has just expanded, thereby avoiding such inconsistency.

4.4 Watchpoints

Many debuggers support *watchpoint* functionality, allowing a user to instruct the debugger to stop when a particular memory location or variable is modified. More sophisticated watchpoints, called *conditional watchpoints*, can also specify that the debugger should stop only when a certain predicate holds (for example, that the variable value is bigger than some number).

Watchpoints are sometimes implemented using specific hardware support, called hw-breakpoints. If no hw-breakpoint support is available, some debuggers implement watchpoints in software, by executing the program step-by-step and checking the value of the watched variable(s) after each step, which results in executing the program hundreds of times slower than normal.

We describe here how to exploit TM infrastructure to stop on any modification or even a read access to a transactional variable. The idea is simple: because the TM implementation needs to keep track of which transactions access which memory locations, we can use this tracking mechanism to detect accesses to specific locations. Particularly, with the HyTM implementation described in Section 2, we can mark the orec that corresponds to the memory location we would like to watch, and invoke the debugger whenever a transaction gets ownership of such an orec. In the hardware code path, when checking an orec for a possible conflict with a software transaction, we can also check for a watchpoint indication on that orec. Depending on the particular hardware TM support available, it may or may not be possible to transfer control to the debugger while keeping the transaction viable. If not, it may be necessary to abort the hardware transaction and retry the transaction in software.

The debugger can mark an orec with either a *stop-on-read* or *stop-on-write* marking. With the first marking, the debugger is invoked whenever a transaction gets read ownership of that orec (note that some TM implementations allow multiple transactions to concurrently own an orec in read mode), and with the latter, it is invoked only when a transaction gets write ownership of that orec. When invoked, the debugger should first check whether the accessed variable is one of the watchpoint's variables (multiple memory locations may be mapped to the same orec). If so, then the debugger should stop, or, in the case of a conditional watchpoint, evaluate a predicate to decide whether to stop.

Stopping the program upon access to a watchpoint variable can be done in one of two ways:

1. *Immediate-Stop:* The debugger can be invoked immediately when the variable is accessed. While this gives the user control at the first time the variable is accessed, it has some disadvantages:
   - The first value written by the transaction to the variable may not be the actual value finally written by the transaction: the transaction may later change the value written to this variable, or abort without modifying the variable at all. In many cases, the user would not care about these intermediate values of the variable, or about accesses done by transactions that do not eventually commit.
   - Most STMs do not reacquire ownership of a location if the transaction modifies it multiple times. Therefore, if we stop execution only when the orec is first acquired, we may miss subsequent modifications that establish the predicate we are attempting to detect.
2. *Stop-on-Commit:* This option overcomes the problems of the immediate-stop approach, by delaying the stopping to the point when the transaction commits. That is, instead of invoking the debugger whenever a marked orec is acquired by a transaction, we invoke it when a transaction that owns the orec commits; this can be achieved for example by recording an indication that the transaction has acquired a marked orec when it does so, and then invoking the debugger upon commit if this indication is set. That way the user sees the value actually written to the variable, since at that point no other transaction can abort the triggering transaction anymore. While this approach has many advantages over the immediate-stop approach, it also has the disadvantage that the debugger will never stop on an aborted transaction that tried to modify the variable, which in some cases might be desirable for example when chasing a slippery bug that rarely occurs. Therefore, it may be desirable to support both options, and allow the user to choose between them. Also, when using the stop-on-commit approach, the user cannot see how exactly the written value was calculated by the transaction, although this problem can be mitigated by the replay debugging technique describes in Section 4.6.

While the above description assumes a TM implementation that uses orecs, the techniques we propose are also applicable to other TM approaches. For example, in object-based TM implementations like the one by Herlihy et. al. [6], we can stop on any access to an object since any such access requires opening the object first, so we can change the method used for opening an object to check whether a watchpoint was set on that object. This might be optimized by recording an indication in an object header or handle that a watchpoint has been set on that object.

4.4.1 Dynamic Watchpoints

In some cases, the user may want to put a watchpoint on a field whose location may dynamically change. Suppose, for example, that the user is debugging a linked list implementation, and wishes to stop whenever some transaction accesses the value in the first node of the list, or when some predicate involving this value is satisfied. The challenge is that the address of the field storing the value in the first node of the list is indicated by `head->value`, and this address changes when `head` is changed, for example when inserting or removing the first node in the list. In this case, the address of the variable being watched changes. We denote this type of a watchpoint as a *dynamic watchpoint*.

We can implement a dynamic watchpoint on `head->value` as follows: when the user asks to put a watchpoint on `head->value`, the debugger puts a regular watchpoint on the current address of `head->value`, and a special debugger-watchpoint on the address of `head`. The debugger-watchpoint on `head` is special in the sense that it does not give the control to the user when `head` is accessed: instead, the debugger cancels the previous watchpoint on `head->value` at that point, and puts a new watchpoint on the new location of `head->value`. That is, the debugger uses the debugger-watchpoint on `head` to detect when the address of the field the user asked to watch is changed, and changes the watchpoint on that field accordingly.

4.4.2 Multi-Variable Conditional Watchpoints

Watching multiple variables together may also be useful when the user would like to condition the watchpoint on more than one variable: for example, to stop only if the sum of two variables is greater than some value. We denote such a watchpoint as a *multi-variable conditional-watchpoint*. With such a watchpoint, the user asks the debugger to stop on the first memory modification that satisfies the predicate.

To implement a multi-variable conditional watchpoint, the debugger can place a watchpoint on each of the variables, and evaluate the predicate whenever one of these variables is modified. We denote by the *triggering transaction* the transaction that caused the predicate evaluation to be invoked. One issue to be considered is that evaluating the predicate requires accessing the other watched variables. This can be done as follows:

- The debugger uses the stop-on-commit approach, so that when a transaction that modifies any of the predicate variables commits, we stop execution either before or after the transaction commits. In either case, we ensure that the transaction still has ownership of all of the orecs it accessed, and we ensure that these ownerships are not revoked by any other threads that continue to run, for example by making the triggering transaction a super-transaction.
- When evaluating the predicate, the debugger distinguishes between two kinds of variables: ones that were accessed by the triggering transaction, which we denote as *triggering variables*, and the rest which we denote as *external variables*. External variables might be accessed by using the stopped transaction, or by using another transaction initiated by the debugger. In the latter case, because the triggering transaction is stopped and retains ownership of the orecs it accessed while the new transaction that evaluates the external variables executes, the specified condition can be evaluated atomically.
- While reading the external variables, conflicts with other transactions that access these variables may occur. One option is to simply abort the conflicting transaction. However, this may be undesirable, because we may prefer that the debugger has minimal impact on program execution. As discussed in Section 4.2.2, it is possible in some cases to determine the pre-transaction value for the watched variable without aborting the transaction that is accessing it.

4.5 Delayed Breakpoints

Stopping at a breakpoint and running the program step-by-step affects the behavior of the program, and particularly the timing of interactions between the threads. Placing a breakpoint inside an atomic block may result in even more severe side-effects, because the behavior of atomic blocks may be very sensitive to timing modifications since they may be aborted by concurrent conflicting transactions. These effects may make it difficult to reproduce a bug scenario.

To exploit the benefits of breakpoint debugging while attempting to minimize such effects, we suggest the *delayed breakpoint* mechanism. A delayed breakpoint is a breakpoint in an atomic block that does not stop the execution of the program until the transaction implementing the atomic block *commits*. To support delayed breakpoints, rather than stopping program execution when an instruction marked as a delayed breakpoint is executed, we merely set a flag that indicates that the transaction has hit a delayed breakpoint, and resume execution. Later, upon committing, we stop the program execution if this indication is set. Besides the advantage of impacting execution timing less, this technique also avoids stopping execution in the case that a transaction executes a breakpoint instruction, but then aborts (either explicitly or due to a conflict with another transaction). In many cases, it will be preferable to only stop at a breakpoint in a transaction that subsequently commits.

One simple type of a delayed breakpoint stops on the instruction *following* the atomic block if the transaction implementing the atomic block hit the breakpoint instruction in the atomic block. This kind of delayed breakpoint can be implemented even when the transaction executing the atomic block is done using HTM. The debugger simply replaces the breakpoint-instruction in the HTM-based implementation to branch to a piece of code that executes that instruction, and raises a flag indicating that the execution should stop on the instruction following the atomic block. This simple approach has the disadvantage that the values written by the atomic block may have already been changed by other threads when execution stops, so the user may see

a state of the world that differs from the state when the breakpoint instruction was hit. Moreover, if the transaction is executed in hardware, then unless there is specific hardware support for this purpose, the user would not be able to get any information about the transaction execution (like which values were read/written, etc.).

On the other hand, if the atomic block is executed by a software transaction, we can have a more powerful type of a delayed breakpoint, which stops *at the commit point* of the executing transaction. More precisely, the debugger tries to stop at a point during the commit operation of that transaction in which the transaction is guaranteed to commit successfully, but that no other transaction has seen its effects on memory. This can be done by having the commit operation check the flag that indicates if a delayed-breakpoint placed in the atomic block was hit by the transaction, and if so do the following:

1. Make the transaction a super-transaction (see Section 4.1.1 for details).
2. Validate the transaction. That is, make sure that the transaction can commit. If validation fails, abort the transaction, fail the commit operation, and resume execution.
3. Give control to the user.
4. When the user asks to continue execution, commit the transaction. Note that, depending on how super-transactions are supported, a lightweight commit may be applicable here if we can be sure that the transaction cannot be aborted after becoming a super-transaction.

The idea behind the above procedure is simple: Guarantee that all future conflicts will be resolved in favor of the transaction that hit the breakpoint, check that the transaction can still commit, and then give control to the user, who can subsequently decide to commit the transaction.

At Step 3 the debugger stops the execution of the commit operation and gives control to the user. This is the point where the user gets to know that a committed execution of the atomic block has hit the delayed breakpoint. At that point, the user can view various variables, including those accessed by the transaction, to try to understand the effect of that execution. In Section 4.6, we describe other techniques that can give the user more information about the committed transaction's execution at that point.

4.5.1 Combining with Atomic Groups

One disadvantage of using a delayed breakpoint is that if the user views variables *not* accessed by the transaction, the values seen are at the time the debugger stops rather than the time of the breakpoint-instruction execution. Therefore, it may be useful to combine the delayed breakpoint mechanism with the atomic group feature (Section 4.3): with this combination, the user can associate with the delayed breakpoint an atomic group of variables whose values should be recorded when the delayed breakpoint instruction is executed. When the delayed breakpoint instruction is hit, besides triggering a breakpoint at the end of the transaction, the debugger gets the atomic group's value (as described in Section 4.3), and presents it to the user when it later stops in the transaction's commit phase.

4.6 Replay Debugging for Atomic Blocks

It is useful to be able to determine how the program reached a breakpoint. *Replay debugging* has been suggested in a variety of contexts to support such functionality, and support ranging from special hardware to user libraries have been proposed (see [12, 14] for two recent examples). Replay debugging for multithreaded concurrent applications generally requires logging that can add significant overhead. In this section, we explain how STM infrastructure can be exploited to support replaying atomic blocks, without the need for additional logging. We also explain how the user can experiment with alternative executions of the atomic block by modifying data and even commit an alternative execution instead of the original one. To our knowledge, previous replay debugging proposals do not include such functionality.

The idea behind our replay debugging technique is to exploit the fact that the behavior of most atomic blocks is uniquely determined by the values it reads from memory[3]. Some STM implementations record values read by the transaction in a readset. Others preserve these values in memory until the transaction commits, at which point the values may be overwritten by new values written by the transaction. In either case, if we modify the STM to allow the debugger access to this information, then the debugger can reconstruct execution of the transaction, as explained in more detail below:

- The debugger maintains its own write-set for the transaction. This is necessary to allow the debugger to determine the values returned by reads from locations that the transaction has previously written. The replay begins with an empty write set.
- The replay procedure starts from the beginning of the debugged atomic block, and executes all instructions that are not STM-library function calls as usual.
- The replay procedure ignores all STM library function calls except the ones that implement the transactional read/write operations.
- When the replay procedure reaches a transactional write operation, it writes the value in the write set maintained by the debugger.
- When the replay procedure reaches a transactional read operation, it first searches the write set maintained by the debugger. If a value for the address

[3]We call such atomic blocks *transactionally deterministic*. While the techniques described in this section may be useful even for blocks that the compiler cannot prove are transactionally deterministic, in this case the user should be informed that the displayed execution might not be identical to the one that triggered the breakpoint.

being read is there, this is the value read by the transactional read operation. Otherwise, the original value read by the transaction is used (acquired from the readset or from memory, depending on the STM implementation).

Because the debugged transaction retains ownership of orecs it acquired during the original execution, memory locations it accesses cannot change during replaying, so the replayed execution is faithful to the original.

Replay debugging functionality can be combined with various other features we have described. For example, by combining replay debugging with the delayed breakpoint feature described in Section 4.5, we can create the illusion that control has stopped inside an atomic block, although it has actually already run to its commit point. Then, the replay functionality allows the user to step through the remainder of the atomic block before committing it. It is even possible to allow experimentation with alternative executions of a debugged atomic block, for example by changing values it reads or writes. In some cases, we may wish to do so without affecting the actual program execution. In other cases, we may prefer to change the actual execution, and subsequently resume normal debugging. One way to handle the latter case is to abort the current transaction *without releasing orecs*, and replay it up to the point at which the user wishes to change something. This way, we guarantee that the transaction will reexecute up to this point identically to how it did in the first place.

Combining replay debugging with other debugger features we have proposed can support a rather powerful debugging environment for transactional programs.

Acknowledgements

We thank Maurice Herlihy for suggesting the ability to see a transaction's tentative values (Section 4.2.3).

5. REFERENCES

[1] Ananian, C. S., Asanović, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)* (San Franscisco, California, Feb. 2005), pp. 316–327.

[2] Ananian, C. S., and Rinard, M. Efficient object-based software transactions. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)* (Oct. 2005).

[3] Fraser, K. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[4] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2004, p. 102.

[5] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (January 1991), 124–149.

[6] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (Jul 2003), pp. 92–101.

[7] Herlihy, M., and Moss, J. Transactional memory: Architectural support for lock-free data structures. Tech. Rep. CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.

[8] Kumar, S., Chu, M., Hughes, C., Kundu, P., and Nguyen, A. Hybrid transactional memory. In *Preceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006).

[9] Lev, Y., and Maessen, J. Towards a safer interaction with transactional memory by tracking object visibility. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)* (Oct. 2005).

[10] Marathe, V. J., Scherer III, W. N., and Scott, M. L. Adaptive software transactional memory. Tech. rep., Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May 2005.

[11] Moir, M. Hybrid transactional memory, Jul 2005. http://www.cs.wisc.edu/trans-memory/misc-papers/moir:hybrid-tm:tr:2005.pdf.

[12] Narayanasamy, S., Pokam, G., and Calder, B. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 284–295.

[13] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 494–505.

[14] Saito, Y. Jockey: A user-space library for record-replay debugging. Technical Report HP-2006-46, HP Laboratories, Palo Alto, CA, March 2005.

[15] Scherer, W., and Scott, M. Advanced contention management for dynamic software transactional memory. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing* (2005).

[16] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (February 1997), 99–116.