Appears in the Conf on **Parallel Architectures and Compilation Techniques (PACT), Sep 2009**

# StealthTest: Low Overhead Online Software Testing using Transactional Memory

Jayaram Bobba[1], Weiwei Xiong[2], Luke Yen[1,3], Mark D. Hill[1], and David A. Wood[1]

[1]University of Wisconsin-Madison
{bobba,lyen,markhill,david}@cs.wisc.edu

[2]University of Illinois at Urbana-Champaign
wxiong2@cs.uiuc.edu

[3]Advanced Micro Devices

***Abstract*—*Software testing is hard. The emergence of multicore architectures and the proliferation of bug-prone multithreaded software makes testing even harder. To this end, researchers have proposed methods to continue testing software after deployment, e.g., in vivo (IV) testing and Delta Execution (DE) patch testing. These on-line techniques typically fork new processes to hide the functional impact of testing. Unfortunately, the high overhead of fork() significantly degrades performance.*

*To reduce the performance overhead of online testing, we adapt transactional memory (TM) to isolate the functional effects of tests with minimal impact on system performance. We propose StealthTest, an interface that exposes TM transactions as the key mechanism for executing tests. It allows online tests to work on a consistent view of memory without the overhead of creating new processes. Moreover, explicitly aborting the test transaction after it is done guarantees that its changes are invisible to the rest of the system. Thus, StealthTest promises transparent online testing.*

*We demonstrate two test frameworks utilizing StealthTest — StealthIV and StealthDE — that improve on previously proposed fork-based in vivo testing and Delta Execution frameworks respectively. Implementing StealthTest on top of three software TM (STM) systems — TL2 STM, Intel STM and a Pin-based STM— we demonstrate that StealthTest-based frameworks can (a) run a wide range of online tests and (b) execute many more tests with low overhead.*

*StealthTest provides another motivation for efficient TM implementations by extending TM's applicability to a critical software engineering challenge.*

***Keywords*—*Testing;Transactional Memory;Online Testing;Delta Execution***

## I. Introduction

Software testing is a critical component of the software engineering process. Despite studies showing that testing accounts for more than half the software development cost [11], software bugs persist. A NIST study [33] indicates that software bugs cost the U.S. economy an estimated $60 billion annually. The emergence of multicore architectures and the concomitant push toward concurrent multithreaded software compounds the complexity of the testing process by significantly increasing a program's possible execution paths. New techniques are needed to make software testing tractable and effective for real-world software development.

One promising approach is on-line testing [18,26,28,39], where tests are run on deployed software systems. By running tests across a wider range of scenarios, these approaches can test a much larger and more diverse set of executions compared to traditional lab-based testing. Safe online testing must be transparent to the system's execution (i.e., not alter program state and correctness). Even if a test leads to corrupted state, the transparency property prevents the application from observing it.

Currently, providing functional transparency for online tests could incur significant overheads [5,39], since systems rely on application memory snapshotting via process forking. Our results (Section III(A)) confirm that using `fork()` for online testing can reduce an apache web server's maximum throughput by as much as 40%. In our view, software customers are not likely to tolerate significant performance artifacts for activities traditionally done during development. Thus, *widespread adoption of online testing depends upon mechanisms that provide both functional and performance transparency.*

To enable transparent low-overhead online testing, this paper proposes ***StealthTest***. StealthTest uses and extends *T*ransactional *M*emory (TM) [13,17] mechanisms to transparently execute online tests. A transaction is a block of code that appears to execute atomically, i.e., it appears to either execute in its entirety at one instant in time or not at all. StealthTest is based on the following key insight: *A TM transaction that always aborts provides a transparent mechanism for executing online tests*. Test frameworks can enclose and execute online tests within a TM transaction that is explicitly aborted (i.e., rolled back) at the end of the test. Transactions promise both functional transparency—via atomicity—and performance transparency—because they are more light-weight than `fork()`.

Not all TM systems provide sufficient support for StealthTest. We identify four requirements that StealthTest imposes on underlying TM systems: strong atomicity [4], *out-of-band* communication mechanisms, flexible conflict resolution and good performance. While hardware support is not a functional requirement, we believe that limited hardware support (e.g., bounded

hardware TM) may prove necessary to achieve good performance. To support online tests targeted toward identifying atomicity violations, we propose an optional TM extension inspired by degree 2 consistency in databases [8].

To demonstrate StealthTest we apply it to two existing online testing frameworks that previously relied on forking threads: *in vivo* testing (reviewed in Section II(A)) and Delta Execution (Section II(B)). *StealthIV* and *StealthDE* are new StealthTest-based frameworks for *in vivo* testing and Delta Execution, respectively.

We implement StealthTest prototypes on top of three software TM (STM) systems—TL2, Intel STM and a Pin-based TM. We use the software systems to study StealthIV and StealthDE with larger programs from BugBench and STAMP application suites and a set of large workloads from the original Delta Execution paper. We demonstrate that our StealthTest-based frameworks (a) can run a wide range of online tests and (b) execute many more tests with low overhead.

This paper makes three contributions:

- We develop StealthTest, the first transparent, low-overhead online testing interface that uses TM.
- We identify TM properties needed to support online testing (e.g., strong atomicity, out-of-band communication, and flexible conflict resolution).
- We demonstrate StealthTest-based implementations of two existing online testing frameworks—*In vivo* testing and Delta Execution.

StealthTest provides another motivation for efficient TM implementations by extending TM's applicability to a critical software engineering challenge.

We next review *In vivo* testing and Delta Execution (Section II) and then present new work on StealthTest (Section III), StealthIV (Section IV), StealthDE (Section V) and their evaluation (Section VI).

## II. Motivation and Background

To demonstrate the utility of StealthTest, we apply it to two existing online testing frameworks—*in vivo* testing and Delta Execution. We present a brief overview.

### A. Online Bug Detection & In Vivo Testing

**Motivation.** In "Why Do Computers Stop and What Can Be Done About It" [9], Jim Gray writes:

> "If you consider an industrial software system which has gone through structured design, design reviews, quality assurance, alpha test, beta test, and months or years of production, then most of the "hard" software bugs, ones that always fail on retry, are gone. The residual bugs are rare cases, typically related to strange hardware conditions (rare or transient device fault), limit conditions (out of storage, counter overflow, lost interrupt, etc...) or race conditions (forgetting to request a semaphore)."

It is widely understood that deployed software systems contain bugs. These bugs are often present because they manifest either in rare conditions that cannot be economically simulated during development or because the developer does not fully understand the software or its deployment environment. Figure 1 (top) presents an illustrative example of an actual bug detected in deployed software. It arises due to incorrect ordering of operations as part of an object deletion routine. The mutex protecting the lock object is incorrectly destroyed *before* the lock is removed from a list. In the interval between mutex destruction and lock removal, a concurrent thread could access the list and find the lock to be still present. Accessing the mutex would lead to a crash.

It has been over two decades since Jim Gray's seminal paper, "Why Do Computers Stop..." [9]. Yet deployed software continues to suffer from the same kinds of *soft* bugs identified in his paper. The emergence of multicore processors and the projected prominence of multithreaded programming makes identification of these soft bugs much tougher, fueling the need for novel approaches to bug detection.

***In vivo* Testing.** *In vivo* (IV) testing [26] is a new software engineering proposal for supporting continued testing on deployed software platforms with two key features. First, it runs a set of pre-determined tests on deployed software at random times under conditions that cannot be either easily anticipated or recreated in a lab. Second, it does not impact the reliability of deployed software since it isolates the functional effects of a test from the rest of the application. As an example of the benefits of IV testing, consider the software engineering goal of ensuring that an object is never accessed after it is destroyed. Developers could deploy an IV test that walks through the list of objects checking consistency of each of an object's fields. Should the test fail and crash, a bug report can be sent home, but the deployed system operation will not be affected. More-

**Buggy Code**(In **mysys/thr_lock.c**):
(http://bugs.mysql.com/bug.php?id=36579)

```
void thr_lock_delete(THR_LOCK *lock) {
  ...
  pthread_mutex_destroy(&lock->mutex);
  ...
  thr_lock_thread_list = list_delete(
      thr_lock_thread_list, &lock->list);
  ...
}
```

**Data Consistency Check:**

```
void thr_check_lock_list() {
  ...
  for (list=thr_lock_thread_list;list;
      list=list_rest(list)){
    THR_LOCK *lock=(THR_LOCK*)list->data;
    thr_check_lock(lock);
  }
}
```

**Figure 1. *In vivo* Testing**

over, even after the bug is fixed, the test can be run as an effective regression test to detect future recurrences. Figure 1 (bottom) shows such a simple IV test that could exercise the previously discussed bug leading to a transparent test failure.

**ForkIV: A Fork-based *In vivo* Testing.** Murphy et al. [26] implement a test framework, ForkIV, that runs IV tests at random times during an application's execution. It uses `fork()` to create a new child process, giving the test its own snapshot of memory isolated from the parent. Upon test completion, ForkIV records the test result in a log and discards the process. Developers analyze the logs off-line to detect any bugs. ForkIV can also run tests over a set of deployed instances of the application [5] in order to either decrease the number of tests run per deployed instance of an application or increase the total number of tests run on an application.

### *B. Online Patch Validation & Delta Execution*

**Motivation.** When bugs are detected after system deployment, developers typically ship software patches to allow end-users to fix them in the field. However, since patches may introduce new bugs, sophisticated end-users generally avoid deploying 'unvalidated' patches [3].

**Delta Execution.** Delta Execution [39] is a novel approach to online patch validation based on the empirical observation that an unpatched execution is often identical to the patched execution. Hence, when the two executions are identical, DE logically runs them both in a single physical instance (*merged execution*). It splits and runs distinct physical instances (*split execution*) only when the executions diverge in either the code they run or the data they access. After a certain time, the two executions can be merged back since they execute the same code accessing identical state. At such a merge point, DE merges the two physical processes back into a single process while recording differences in their state.

DE incurs lower overhead compared to traditional online validation techniques which use two separate instances (e.g., two distinct processes) leading to redundant execution. It can also provide more accurate validation by decreasing sources of non-determinism that cause gratuitous differences [39].

**ForkDE: A Fork-based Delta Execution.** Tucek et al. [39] propose an initial implementation (ForkDE) based on process fork, as illustrated in Figure 2 (a). ForkDE starts execution as a single process, running until it reaches patched code (*delta code*). At this split point, ForkDE forks off a child process containing an identical copy of the program state. While the child process executes patched code, the parent executes the original code. After executing delta code, the two processes rendezvous. ForkDE logically merges the executions into the parent process and discards the child process. As part of the merge, it saves the differences in program state (*delta data*) between the two executions and protects those memory locations using `mprotect()`. A future access to these locations during merged execution will cause a trap to ForkDE which then re-initiates split execution while ensuring that each execution gets to see its own program state. Note that, in this case, the two executions could execute the same code. Split execution continues until both executions run the same code while accessing identical program state.

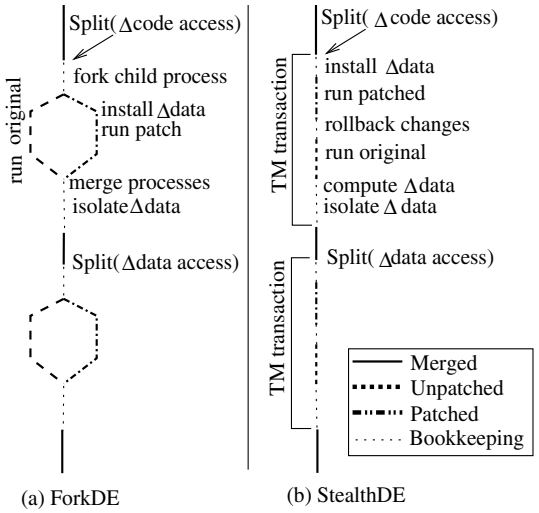


**Figure 2. Delta Execution Implementations**

## III. StealthTest: Using TM for Online Testing

This section shows that Transactional Memory is an enabling mechanism for low-overhead online testing. It then presents the StealthTest interface and discusses requirements on underlying TM implementations.

### *A. Motivation*

ForkIV and ForkDE take big strides toward online testing by proposing solutions that provide transparent test execution in a deployed environment. However, the use of process forking for ensuring functional transparency leads to an important limitation.

**Fork Overhead.** Forking new processes to run tests is a high-overhead operation. Importantly, the overhead gets even worse in multithreaded applications where all threads need to be stopped to take an *atomic* snapshot of memory. Figure 3 shows the throughput of an apache web server as the request rate varies. With no testing (ForkIV(0%)), response rate scales linearly up to 1400 requests/sec. With full testing (ForkIV(100%), wherein every request runs an IV test), the response rate scales only to 800 requests/sec. Other lines show how less frequent testing results in intermediate throughput values. Clearly, the performance degradation of substantial testing is unacceptable in deployed applications.

Moreover, forking a process might not be safe at certain points in an application's execution. In some implementations, a fork operation can cause blocking system

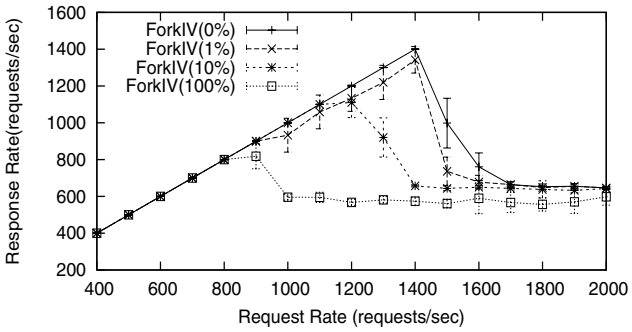


**Figure 3. Fork Performance Overhead**

calls to return with an `EINTR` error leading to unpredictable application behavior [30].

### *B. New Mechanism for Online Testing*

What does it take to execute tests transparently? We identify two requirements for a mechanism. First, during test execution, the mechanism needs to ***isolate*** the changes made by the test from the application. Second, if the test has made changes to application state, then the mechanism needs to ***roll back*** these changes at the end of the test, so that the application does not see the modified state. We observe that Transactional Memory provides both these capabilities and hence is a good candidate for further exploration.

Transactional Memory [13,17] is a language model that attempts to simplify concurrent programming. It allows a programmer to specify *transactions* — blocks of code that are guaranteed to execute atomically in a global serial order (i.e., the execution is *serializable*) by the underlying TM system. In order to operate efficiently, TM systems execute a transaction concurrently with the rest of the application code, dynamically isolating the data accessed by the transaction during its execution. In case of a conflict (i.e., the transaction cannot be *serialized*) during its execution, TM systems can *abort* the transaction, roll back its changes and flexibly re-execute the transaction at a later point. Thus, we make the observation that **TM transactions could provide an efficient transparent enclosure for executing online tests.** An online test placed within a transaction will be isolated from the application by the TM system and an explicit rollback at the end guarantees test transparency.

**No Fork Overhead.** TM operations are light-weight compared to process forking. A TM system does not need to halt all application threads to begin a transaction. Neither does it need to duplicate address spaces via page-table manipulations and incur copy-on-write overheads. As our evaluation (Section VI) shows, TM systems can provide low-overhead isolation and rollback.

### *C. StealthTest Interface*

StealthTest (Table 1) is an interface that TM systems should provide in order to support online test frameworks. The first set of five functions enable test frameworks to enclose tests within transactions and use out-of-band communication to analyze test executions. They are standard procedures that are either supported or can be supported by most TM systems. They can be used for enclosing code in transactions (`ST_[begin|commit]_transaction`), for explicit aborts (`ST_abort_transaction`) and for specifying non-transactional actions within a transaction (`ST_[begin|end]_escape`).

The next three functions, which rely on TM's *version management* [24] mechanisms, allow test frameworks to analyze the modifications performed by various code/test sequences. `ST_get_new` requests the underlying TM system to provide the set of memory modifications (e.g., set of <address, value> pairs) done by a transaction thus far. `ST_get_old` returns the old (pre-transaction) values corresponding to the locations modified by a transaction. `ST_undo` unrolls the modifications done by a transaction without releasing transactional isolation on them.

The final three functions enable test frameworks to set up fine-grain memory protection using TM conflict detection mechanisms [24]. `ST_protect_set` and `ST_protect_clear` request that the TM system protect and unprotect, respectively, certain memory locations from future memory accesses. Any access to a protected location from outside a StealthTest transaction causes a callback to a conflict handler (registered using `ST_register_conflict_callback`).

**Table 1. StealthTest Interface**

| Function | Description |
|---|---|
| **`ST_[begin\|commit]_transaction`**`()` | Begin/End a TM transaction |
| **`ST_abort_transaction`**`(UINT status)` | Explicitly abort a TM transaction |
| **`ST_[begin\|end]_escape`**`()` | Begin/End an escape action |
| **`ST_get_new`**`(map<ADDR,UINT>& mod_map)` | Return modifications done by transaction |
| **`ST_get_old`**`(map<ADDR,UINT>& old_map)` | Return old values for modified locations |
| **`ST_undo`**`()` | Rollback modifications done by the transaction |
| **`ST_protect_set`**`(vector<ADDR> addr_set)` | Protect addresses in addr_set |
| **`ST_protect_clear`**`(vector<ADDR> addr_set)` | Clear protection on addresses in addr_set |
| **`ST_register_conflict_callback`**`(FN_PTR conflict_fn)` | Register callback for conflict on protected location |

We have designed StealthTest to support online testing using the examples of *In vivo* testing and Delta Execution patch testing. Nevertheless, we expect—but cannot prove—that StealthTest can be applied or extended to support other online testing frameworks as they emerge.

### D. StealthTest Requirements on TM Systems

In addition to requiring access to some internal TM state/mechanisms, StealthTest makes four demands on an underlying TM implementation.

**Strong Atomicity.** *To guarantee test transparency, non-transactional memory accesses (e.g., the application code) should not be able to access data accessed by the test transaction.* This property is provided by strongly atomic TM systems [4], because they detect conflicts even on accesses made by non-transactional threads. All current HTMs [17], some hybrid TM systems [23] and some STMs [1,36] are strongly atomic. Weakly atomic systems are being scrutinized, however, for other reasons (e.g., privatization [36]).

**Flexible Conflict Resolution.** *Test transactions should not indefinitely slow the execution of application threads.* Thus, TM conflict resolution policies should favor the application over tests. Fortunately, many TM systems permit flexible conflict resolution policies [34] and hence can install a policy where the test threads get the lowest priority during conflict resolution.

**Communication From Within Transactions.** *Test transactions must be able to communicate failure (and success) to the external world prior to aborting and discarding their state changes.* Thus, TM systems should support an out-of-band mechanism for making state changes that are not isolated/rolled back. While some existing systems already have such mechanisms (e.g., escape actions [25], external actions [10], `tm_pure` [16]), other systems can easily incorporate them.

**Competitive Performance.** *To achieve competitive performance, TM implementations must have low overhead, especially for non-transactional code while preserving strong atomicity.* This requirement is met by most current HTM and HybridTM systems, but not (yet) by STMs.

### E. Discussion

Beyond the requirements discussed above, StealthTest benefits from additional TM properties discussed below.

**Hardware Support for StealthTest.** Implementations of StealthTest benefit from hardware support for at least two reasons. First, hardware support in the form of bounded HTM systems could enable efficient strongly atomic TM systems. Second, fine-grained protection calls (e.g., `ST_protect_*`) could benefit significantly from hardware support if the amount of protected data is large and distributed over multiple pages.

```
test();  ➞
```

```
ST_begin_transaction();
 try {
   test();
   ST_begin_escape();
    fprintf(log,"...",success);
   ST_end_escape();
 } catch/except () {
   ST_begin_escape();
    fprintf(log,"...",fail);
   ST_end_escape();
 }
ST_abort_transaction(NO_RETRY);
```

**Figure 4. StealthIV Test Transformation**

**Interaction of Locks & StealthTest Transactions.** Application code (including test transactions) might acquire and release conventional locks. Using a TM-agnostic lock library could lead to deadlocks and livelocks [40]. Correct operation requires either dynamically inserted TM-aware wrappers around lock operations or a TM-aware lock library.

**System Calls within StealthTest Transactions.** Support for system calls within transactions [2,25] is an area of active research. Empirical evidence suggests that many system calls can be supported from within transactions. Nevertheless, some system calls cannot be executed from within a transaction making them ineligible for use within StealthTest tests.

The TM-based mechanisms used by StealthTest are complementary to fork-based mechanisms examined in prior work. Hence, in the (rare) case that StealthTest does not support an online test, we envision falling back to inefficient, but more robust, fork-based mechanisms.

## IV. StealthIV: A StealthTest-based Framework for *In vivo* Testing

StealthIV is a new *in vivo* test framework that leverages StealthTest. It uses compiler and library support to act as a transformation layer operating in between the application developer and the underlying TM system by enclosing tests within TM transactions.

### A. An Operational View

StealthIV performs two main tasks—transforming tests into transactions and providing infrastructure for launching and executing IV tests.

**Test Transformation.** Developers write a set of IV tests for various modules within the program. During compilation, StealthIV wraps the tests with transactions provided by the underlying TM system (as shown in Figure 4). The test result is then written to a separate log using out-of-band TM mechanisms, to ensure that the test result is not lost when the transaction aborts. Finally, the test transaction is explicitly aborted.

**Launching and Executing Tests.** StealthIV also provides the bookkeeping infrastructure for tests. At application initialization, it creates a separate pool of threads for running IV tests. StealthIV launches tests at pro-

**Buggy Code** (In **java.lang.StringBuffer**):

```
public synchronized
StringBuffer append(StringBuffer sb) {
  int len = sb.length();
  ...//other threads may change sb.length()
  sb.getChars(0, len, value, count);
  ...
}
```

Other threads might modify `sb.length()` between execution of two statements. Could lead to an exception when `sb.getChars()` sees incorrect length (`len`).

**Figure 5. Why Atomic Tests are insufficient**

grammer-specified instrumentation points or at randomly picked locations in the execution. On reaching a test launch point, StealthIV picks a random test and dispatches it to one of the test threads.

### *B. Detecting Atomicity Violation Bugs*

As defined, both ForkIV and StealthIV execute tests atomically. While clean, this behavior prevents IV tests from exercising and discovering *atomicity-violation* bugs (e.g., the bug in Figure 5, as reported by Flanagan et al. [7]) that manifest only when the test thread interacts with one or more remote threads [20]. In order to handle atomicity-violation tests, we need to relax the isolation characteristics of IV tests to allow remote threads to modify data that has been read by a test.

**Degree-2 Transactions.** We propose optionally enhancing underlying TM systems with *Degree-2* transactions. These are code sequences for which the TM system isolates write operations but not read operations in a manner inspired by Degree-2 consistency in databases [8]. StealthIV then wraps each atomicity-violation test in a Degree-2 transaction. When such a test runs, it can detect an atomicity violation if it reads a datum twice: before and after an application thread writes the datum. Since Degree-2 transactions still isolate their own writes, atomicity violations caused by a non-atomic test do not corrupt application state.

We see three ways to support Degree-2 transactions. First, StealthIV could enclose every read of a Degree-2 transaction in a non-transactional 'escape' action. While inelegant, this solution works without modifying existing TM systems. Second, StealthIV could leverage *early release* [12], a technique to remove isolation for a memory location accessed within a transaction. Finally, a TM system could directly support Degree-2 transactions by exposing a mechanism to disable read isolation.

## V. StealthDE: A StealthTest-based Framework for Delta Execution

StealthDE combines dynamic instrumentation with StealthTest to form a new Delta Execution framework.

### *A. An Operational View*

Figure 2 (b) gives a high-level view of StealthDE operation. The two key operations in Delta Execution

```
get_lock(split_exec_mutex);
ST_begin_transaction();
  copy_delta_data(Δdata);
  patched_execution();
  ST_begin_escape();
    ST_get_new(&patch_mod);
    ST_undo();
  ST_end_escape();
  original_execution();
  ST_begin_escape();
    ST_get_new(&orig_mod);
    ST_get_old(&old_values);
    generate_delta_data(patch_mod,
      orig_mod,old_values,&newΔdata);
    ST_protect_set(newΔdata.keys());
  ST_end_escape();
ST_commit_transaction();
oldΔ = get_diff(Δdata,newΔdata);
ST_protect_clear(oldΔ);
Δdata = newΔdata; //set delta data
release_lock(split_exec_mutex);
```

**Figure 6. Delta Execution using StealthTest**

are tracking *delta state* (i.e., different program state between original and patched executions) and setting up split execution.

**Identifying Access to Delta State.** Similar to ForkDE, StealthDE identifies delta code by dynamically instrumenting the merged execution to detect access to patched code. It then replaces the code location with the DE code sequence shown in Figure 6, where `patched_execution` runs the patched code while `original_execution` runs the original code. StealthDE monitors delta data created during split execution using `ST_protect_*` calls. As a result, it receives a conflict callback from the TM system when delta data is accessed. It can then dynamically insert the DE code sequence in Figure 6. Note that in this case, the two executions might execute identical code images. Similar to ForkDE, we heuristically attempt to merge executions at every function return when the nesting level is less than the level at the start of split execution.

**Split Execution.** Unlike ForkDE, StealthDE runs the two executions (original and patched) sequentially using TM transactions to monitor and unroll the changes made by the patched execution. As we show in Section VI(D), sequentially running split execution is acceptable since the time spent on forking and merging processes greatly exceeds the time spent in split execution. Patched execution begins by copying the delta data corresponding to its execution into program memory. At the end of patched execution, StealthDE notes all the changes done before rolling them back. It then executes the original code. At the end of original execution, the changes made by the two executions are compared to generate and save the new delta data (`generate_delta_data`). These locations are then isolated for detecting future accesses. Finally, StealthDE completes split execution by commiting the transaction and moves on to merged execu-

tion. For simplicity, we currently support only one atomic split execution at a time. If this becomes a bottleneck, we could enhance StealthDE to support simultaneous split executions.

### *B. Advantages over ForkDE*

In addition to potential performance benefits due to the use of transactions in place of process forking, StealthDE offers other advantages compared to ForkDE.

**Multi-threading.** With standard process forking, only the thread initiating fork is replicated on a child process. If non-duplicated threads are allowed to continue execution on the parent process, ForkDE would spuriously identify all the changes performed by them as delta data leading to unnecessary split/merges and validation failures. ForkDE addresses this issue by simply stopping all the other threads during split execution. StealthDE, by virtue of its atomic split execution, can provide merged execution for the rest of threads while providing split execution for the thread accessing delta state. Thus, threads do not have to be explicitly stopped. Threads that conflict with split execution can either be stalled or be allowed to proceed after aborting the split execution.

**System State.** Process forking cannot duplicate all program state associated with an application (e.g., PID). Since StealthDE does not create a new process, system state is identical during split and merged executions.

**DeltaData Granularity.** ForkDE tracks delta data at page-granularity, whereas StealthDE can track it at a much finer (e.g., word or cache-block) granularity. Thus, StealthDE reduces unnecessary split/merges caused by "false conflict" accesses to non-delta data.

## VI. Evaluation

To evaluate StealthTest, we modify existing TM systems to support the StealthTest interface and then build StealthIV and StealthDE on top of them. We first describe the workloads and TM systems and then evaluate StealthIV and StealthDE. We show that both can effectively support online testing with low overhead.

### *A. Workloads*

**StealthIV Workloads.** In order to evaluate the effectiveness of StealthIV on real bugs, we choose BugBench [19], a set of real applications that each have at least one documented bug. We obtained 10 programs in a pre-release package from the authors. Table 2 gives some important characteristics of these programs (as reported by the original authors). Note that some of the programs contain atomicity bugs that require the use of Non-Atomic tests (Section IV(B)) for detection.

The BugBench suite specifies bug-triggering inputs but not standard inputs which prevents us from comparing performance against ForkIV. Toward this goal, we use a set of four multithreaded workloads from the STAMP application suite [22] that come with standardized inputs and unit tests that could be run as IV tests.

**Table 2. BugBench Applications**

| Name | Description | Size (LOC) | Bug Type | Error Detect |
|---|---|---|---|---|
| NCOM | file compress | 1.9K | StackSmash | Yes |
| POLY | file "unixier" | 0.7K | StackSmash & BufferOverflow | Yes |
| GZIP | file compress | 8.2K | BufferOverflow | Yes |
| MAN | documentation | 4.7K | BufferOverflow | Yes |
| BC | calculator | 17.0K | BufferOverflow | Yes |
| HTPD1 | web server | 224K | Atomicity | Yes |
| SQUD | proxy cache | 93.5K | BufferOverflow | Possible |
| CVS | version control | 114.5K | DoubleFree | Possible |
| MSQL2 | DBMS | 514K | Atomicity | Possible |
| MSQL3 | DBMS | 1028K | Atomicity | Possible |

**StealthDE Workloads.** In order to evaluate StealthDE, we use the same set of applications and real software patches used in the original ForkDE study (Table 3). The inputs to the program are chosen to exercise code paths that had been patched while still being realistic. Server loads are tuned towards maximum throughput.

### *B. TM Systems*

As noted in Section III(D), TM systems that support StealthTest need to satisfy four requirements—strong atomicity, flexible conflict resolution, out-of-band communication mechanisms, and competitive performance. Currently only hardware TM systems promise to provide all the above properties, but these systems are not widely deployed and HTM simulators are slow.

Thus, to evaluate StealthTest for large real applications, we choose two software TM systems (for StealthIV) and a TM emulator (for StealthDE). While these systems either do not support strong atomicity or could incur high performance overheads; they do allow an evaluation of StealthTest using realistic applications.

### *StealthIV Implementations*

StealthIV requires only a subset of the StealthTest interface that is supported by most TM systems. We choose two different STMs for building it.

**TL2 STM [6] .** TL2 is a library-based TM system that provides a TM with lazy version management and commit-time conflict detection. We use the x86 port of TL2 provided with the STAMP application suite and use the existing interface to build *StealthIV_TL2*. Since library-based systems require manual insertion of TM hooks for every transactional access, we use it only for STAMP applications that come with the hooks in place.

**Intel STM [16].** The Intel STM system is a language-based TM system consisting of a C++ compiler (Prototype Edition 2.0) and an associated set of libraries providing TM functionality. We build *StealthIV_ICC* using the TM system's language extensions: `__tm_atomic` for specifying StealthTest transactions and `tm_pure` procedures for out-of-band communication. The Intel

STM system provides stable language-level support for working with the larger BugBench applications.

The STMs are installed on a quad-core workstation running RedHat Enterprise Linux 5. Note that these systems, as well as all publicly available STM systems, are weakly atomic. As discussed in Section III(D), certain tests run with these systems might violate transparency making our implementation less robust. However, we do not observe such violations in our evaluation.

### *StealthDE Implementation*

In order to implement and evaluate StealthDE, we require open access to the TM system and language support. Since neither of the above TM systems provide both these features, we look elsewhere.

**Pin-based TM Emulation**[21]**.** Pin is a dynamic instrumentation tool that allows flexible insertion of callback functions into an executing binary image. In order to study StealthDE, we extend a Pin-based strongly atomic TM emulator [31] to support all the features required by StealthTest and then build *StealthDE* on top of it. Programmers provide StealthDE with a patch description file that identifies regions in the execution binary where patched code differs from original code. StealthDE dynamically intercepts execution when it reaches one of these delta code regions. It then automatically injects the code shown in Figure 6 into the dynamic instruction stream to initiate delta execution. *All* memory accesses are similarly intercepted to check for transactional conflicts. Like ForkDE, our implementation supports elimination of dead stack values from delta data. However, it does not yet support the 'shadow memory' technique used by ForkDE to avoid turning internal memory allocation structures into delta data. It also has limited TM system call support. StealthDE runs on a 2-way SMP with 2.4 GHz Pentium 4 CPUs and 2.5 GB of memory.

### *C. StealthIV Results*

**Does StealthTest facilitate effective IV testing?** We develop IV tests for detecting the known bugs in BugBench programs and run them using StealthIV_ICC.[1]

We find that IV tests run using StealthTest are able to capture the errors caused due to 6 out of the 10 bugs in the BugBench suite (right-most column in Table 2). With regards to the rest of the bugs (in SQUD, CVS, MSQL2 and MSQL3), it is possible to develop IV tests for exercising them. However, the code surrounding the bugs in these programs makes library and system calls. The Intel STM system does not yet support rollback and isolation of these operations. Hence, we are unable to run tests (as noted in Section III(E)) targeting the region surrounding the bug. As future TM systems add support for isolating and rolling back system call activity, we believe it will become possible to detect the bugs in these four programs. Note that ForkIV would also not support these tests since it too does not currently isolate and unroll library/system calls.

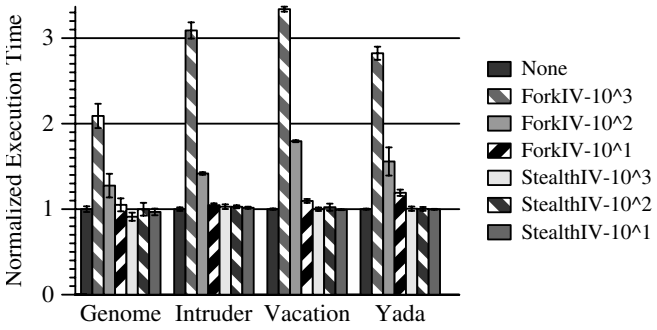


**Figure 7. StealthIV Performance Comparison**

**Does StealthTest provide low overhead IV tests?** We evaluate the STAMP applications on the standard set of *non-simulator* inputs (using 4 threads). We run the provided unit tests as IV tests at varying TPM (tests per minute) using both ForkIV and StealthIV_TL2[2].

Figure 7 shows the execution time for the various runs normalized to runs where no IV tests were run (*None*). Running tests at the rate of 10^3 TPM, ForkIV could incur up to 3.3X degradation in performance (ForkIV-10^3 for Vacation). As expected, the performance overhead of ForkIV decreases as we decrease the frequency of IV tests. In contrast, the performance overhead of StealthIV_TL2 is statistically insignificant in all cases irrespective of frequency of test execution. Thus, we can conclude that StealthIV can execute orders of magnitude more IV tests when compared to ForkIV.

### *D. StealthDE Results*

**Does StealthDE facilitate effective Delta Execution?** We run the patch validation workloads on StealthDE with the same set of applications used by the original authors of ForkDE.

StealthDE runs DE correctly and validates patch executions for 6 of the 10 programs. Of the remaining, ATPhttpd has calls to read and write from network sockets within patched code that cannot be isolated and rolled back by our TM system. Similarly, MySQL5.0, OpenSSL and Squid have memory allocation functions within patched code that are not yet supported. While transactional support for network socket calls in ATPhttpd is tougher to implement, we believe that TM memory allocation support can be implemented [14].

**Does StealthDE provide low-overhead Delta Execution?** Since StealthDE uses emulation, we cannot make direct comparisons with the ForkDE implementation obtained from the original authors. Nevertheless, we present some key measurements (Table 3) that enable a first-order estimation of StealthDE's overheads compared to ForkDE's overheads.

1. We have also run experiments with the Siemens test suite [15] where IV tests were developed without apriori knowledge of bugs. The tests detected 68% of the bugs in these programs.

2. With BugBench applications using StealthIV_ICC, the degradation in performance is also statistically insignificant in all applications except GZIP(~3%).

**Table 3. Patch Validation Workloads and Results**

| Program | Description | ForkDE | | | | StealthDE | |
|---|---|---|---|---|---|---|---|
| | | NumSplits | ΔData (pages) | Fork Overhead(%) | Patch Duration(%) | NumSplits | ΔData(words) |
| Crafty | Chess App | 5 | 0 | 0.1 | <0.1 | 0 | 0 |
| Raytrace | Raytracer | 4 | 1 | 0.2 | 0.5 | 1 | 0 |
| Tar | Archive Util | 184 | 0 | 41 | 7.3 | 184 | 0 |
| Apache1 | Web Server | 25 | 0 | 2.8 | 0.1 | 25 | 0 |
| Apache2 | Web Server | 110 | 1 | 12 | 0.1 | 100 | 0 |
| DNSCache | DNS Cache | 4622 | 0 | 65 | 0.1 | 4409 | 0 |
| ATPhttpd | Web Server | 600 | 0 | 65 | 0.8 | | |
| MySQL5.0 | DB Server | 30 | 21 | 4.7 | 5.0 | | |
| OpenSSL | Security Lib | 100 | 0 | 12 | <0.1 | | |
| Squid | Web Cache | 15 | 0 | 2.9 | 0.2 | | |

We first compare the split execution behaviors of ForkDE and StealthDE by measuring the number of split executions (*Num Splits*) and the average amount of delta data ($\Delta Data$) at end of split execution. We find the behavior to be similar but not exactly identical. The difference primarily arises since the two implementations patch code and track data at different granularities.

Next, we look at first-order differences in performance between the two systems. First, ForkDE incurs a substantial fork overhead that StealthDE avoids. We measure and report the fraction of total execution time spent in forking and merging test processes (*Fork Overhead*). Clearly, fork overhead could be extremely high (up to 65%). Second, unlike ForkDE, StealthDE executes patched and original code sequentially. We find that the fraction of total execution time spent in patched code is fairly small (except in tar where the 7% spent in patched execution is still much less than the 41% fork overhead). Thus, since StealthDE exhibits similar split execution behavior as ForkDE, we expect it to incur much lower overhead compared to ForkDE (except for MySQL where the overheads are similar).

### *E. Discussion*

Our results indicate that StealthTest-based frameworks could incur much lower relative overhead than their fork-based counterparts. However, these results do not reflect either the absolute performance overhead incurred by STMs or the additional overhead imposed by strong atomicity. Both these issues are areas of active research with recent results indicating that strong atomicity overheads could be as small as 10% [35].

StealthTest provides an additional incentive to build efficient TM systems, including hardware support to provide strong atomicity with little overhead.

## VII. Related Work

Testing for software bugs is traditionally done during development. As noted earlier, many promising approaches [18,26,28,38,39] extend bug detection to deployed applications. StealthTest attempts to provide a common framework for such approaches using TM as a low-overhead mechanism.

StealthTest leverages the *sandboxing* capability of TM to guarantee test transparency. Other mechanisms that provide transactional isolation and rollback can also permit sandboxing. They range from code-emulation (e.g., STEM [37]), binary translation (e.g., Sprockets [32]), databases ([29]) to hardware thread-level speculation (TLS) (e.g., [27]). These mechanisms have been used in proposals targeted toward various software engineering goals like safety, reliability, security etc. StealthTest differs from the above proposals in its use of TM as a mechanism and its focus on software testing. We believe that for the emerging class of multi-threaded applications, TM can provide a simpler and faster online testing mechanism than existing state-of-the-art. TM transactions can incur much lower performance overhead than code-emulation or binary translation and are easier to use at application level than TLS (which provides only bounded transactions). Moreover, StealthTest does not require a global stop-the-world phase, unlike existing mechanisms [32,37].

## VIII. Summary and Future Work

Testing is and will be an important challenge in software systems development. We propose StealthTest, an interface that enables transparent low-overhead online testing using TM. We demonstrate the utility of StealthTest with two complementary test frameworks for online bug detection and online patch validation.

Future work could include greater system call support to enable more online tests, exploring other testing scenarios (e.g., regression testing), supporting a wider range of online tests (e.g., multi-threaded tests), examining a wider range of test frameworks (e.g., [38,41]) and providing better debugging feedback with StealthTest.

## Acknowledgments

This work is supported in part by the National Science Foundation (NSF), with grants CCR-0324878, CNS-0551401, CNS-0720565, as well as donations from Intel and Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel or Sun Microsystems. Yen was a PhD student at the University of Wisconsin-Madison when this work was performed. Xiong's work is supported in part by NSF grant 0615372 with PI Yuanyuan Zhou.

We thank Piramanayagam Arumuga Nainar, Dan Gibson, Ben Liblit, Shan Lu, Michael Swift, Yuanyuan Zhou and the anonymous reviewers for their comments.

## References


[1] Baugh et al. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[2] Lee Baugh and Craig Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007.

[3] Beattie et al. Timing the Application of Security Patches for Optimal Uptime. In *Proc. of the 16th Systems Administration Conference*, pages 233–242, 2002.

[4] Blundell et al. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), November 2006.

[5] Chu et al. Distributed In Vivo Testing of Software Applications. *Software Testing, Verification and Validation, 2008 1st Intl. Conf. on*, pages 509–512, April 2008.

[6] Dice et al. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, September 2006.

[7] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proc. of The 31st ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 256–267, January 2004.

[8] Gray et al. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems, Elsevier North Holland, New York*, 1975.

[9] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[10] Tim Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[11] Mary J. Harrold. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 61–72, 2000.

[12] Herlihy et al. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

[13] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intnl. Symp. on Computer Architecture*, pages 289–300, May 1993.

[14] Hudson et al. McRT-Malloc: a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management*, June 2006.

[15] Hutchins et al. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th International Conference on Software Engineering*, pages 191–200, 1994.

[16] Intel. Intel C++ STM Compiler, Prototype Edition 2.0. http://softwarecommunity.intel.com/articles/eng/1460.htm.

[17] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[18] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, UC-Berkeley, December 2004.

[19] Lu et al. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[20] Lucia et al. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[21] Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, pages 190–200, June 2005.

[22] Minh et al. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[23] Minh et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[24] Moore et al. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, February 2006.

[25] Moravan et al. Supporting Nested Transactional Memory in LogTM. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, October 2006.

[26] Murphy et al. Towards in vivo testing of software applications. Technical Report cucs-038-07, Columbia University, 2007.

[27] Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 184–196, October 2002.

[28] Orso et al. Gamma System: Continuous Evolution of Software after Deployment. In *Proceedings of the ACM International Symposium on Software Testing and Analysis(ISSTA)*, pages 65–69, 2002.

[29] Roy Osherove. Simplified Database Unit testing using Enterprise Services. http://weblogs.asp.net/rosherove/articles/dbunittesting.aspx, 2004.

[30] Man pages: fork(2). http://docs.sun.com/app/docs/doc/816-5167/fork-2.

[31] Pan et al. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News*, 33(5):45–50, 2005.

[32] Peek et al. Sprockets: safe extensions for distributed file systems. In *Proc. of the 2007 USENIX Annual Technical Conference*, pages 1–14, June 2007.

[33] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, May 2002.

[34] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *24th ACM Symp. on Principles of Distributed Computing*, July 2005.

[35] Schneider et al. Dynamic Optimization for Efficient Strong Atomicity. *SIGPLAN Not.*, 43(10):181–194, 2008.

[36] Shpeisman et al. Enforcing Isolation and Ordering in STM. In *Proc. of the SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, June 2007.

[37] Stelios Sidiroglou. Building a reactive immune system for software services. In *Proc. of the 2005 USENIX Annual Technical Conference*, pages 11–11, June 2005.

[38] Sidiroglou et al. Band-aid patching. In *Proc. of the 3rd workshop on Hot Topics in System Dependability*, 2007.

[39] Tucek et al. Efficient Online Validation with Delta Execution. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[40] Volos et al. Pathological Interaction of Locks with Transactional Memory. In *Proc. of the 3rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2008.

[41] Yabandeh et al. Crystal Ball: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation*, May 2009.