ATDetector: Improving the Accuracy of a Commercial Data Race Detector by Identifying Address Transfer

Jiaqi Zhang[1], Weiwei Xiong[2], Yang Liu[1], Soyeon Park[1], Yuanyuan Zhou[1], Zhiqiang Ma[3]

[1]Dept of Computer Science, University of California San Diego, La Jolla, CA 92093, {jiz013,yal036, soyeon, yyzhou}@cs.ucsd.edu
[2]Dept of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL 61801, wxiong2@cs.illinois.edu
[3]Software and Services Group, Intel Corporation, Champaign, IL 61820, zhiqiang.ma@intel.com

ABSTRACT

In order to take advantage of multi-core hardware, more and more applications are becoming multi-threaded. Unfortunately concurrent programs are prone to bugs, such as data races. Recently much work has been devoted to detecting data races in multi-threaded programs. Most tools, however, require the accurate knowledge of synchronizations in the program, and may otherwise suffer from false positives in race detection, limiting their usability. To address this problem, some tools such as Intel® Inspector provide mechanisms for suppressing false positives and/or annotating synchronizations not automatically recognized by the tools. However, they require users' input or even changes of the source code.

We took a different approach to address this problem. More specifically, we first used a state-of-the-art commercial data race detector, namely Intel® Inspector on 17 applications of various types including 5 servers, 5 client/desktop applications, and 7 scientific ones, without utilizing any suppression or annotation mechanisms provided by the product that need users' input. We examined a total of 1420 false data races and identified two major root causes including **address transfer**, where one thread passes memory address to another thread. We found more than 62% false data races were caused by address transfer. Based on this observation, we designed and implemented an algorithm that automatically identify address transfer and use the information to prune the false data races. Our evaluation with 8 real-world applications shows that it can effectively prune all false data races caused by unrecognized address transfers, without eliminating any true data race that was originally reported.

Categories and Subject Descriptors:
D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids*; D.2.4 [**Software Engineering**]: Software/Program Verification – *Reliability*

General Terms:
Reliability

Key Words:
Concurrency Bug, Data Race, False Positive

MICRO'11 December 3-7, 2011, Porto Alegre, Brazil

1. INTRODUCTION

1.1 Motivation

Multi-threaded programs are becoming more and more prevalent due to the reality of multi-core hardware. Unfortunately, concurrent programming is well known to be difficult and may easily introduce bugs. One notorious type of concurrency bugs is data race, which occurs when multiple threads access the same memory location without proper synchronization, and at least one of them writes that memory location. Data races have caused severe disasters in real world, such as the Northeastern electricity blackout [20] and radiation overdose [28].

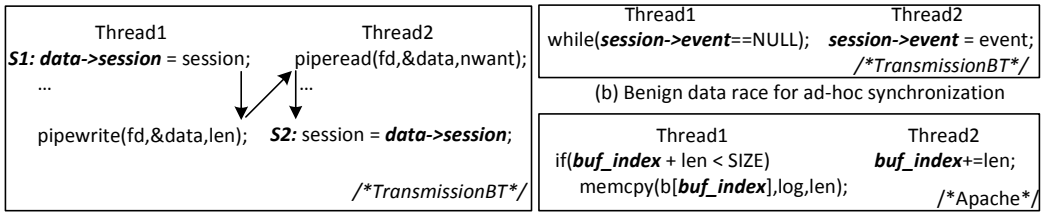
Various solutions have been proposed to detect data races. They are mostly based on one of the two classic algorithms, the happens-before algorithm [32, 26, 27, 7, 22] and the lockset algorithm [35, 11, 30], or a hybrid of the two [17, 12, 40]. The former claims two conflicting memory accesses as a data race if they are not ordered by synchronization operations, and is usually implemented as dynamic tools such as Intel® Inspector (hereinafter referred to as "Inspector") [15] and Valgrind [3] , whereas the latter reports a data race if there is no common lock held to protect conflicting accesses to a shared memory location, and is implemented as both static [11, 25, 29] and dynamic tools [35]. In order to overcome the large overhead associated with software-only implementations, hardware designs have also been proposed: hardware happens-before bug detectors [32, 26, 27, 7] leverage existing cache coherence protocol, and a lock-set data race detector [30] makes use of the hardware bloom filters.

Unfortunately, most existing data race detectors, including even commercial strength race detectors such as Inspector , require accurate knowledge of synchronizations in the program, and may otherwise report false races (referred to as *Not-A-Race* in this paper for clarification) or benign races, of which the definitions are explained in Table 1. The former is not a race but is incorrectly reported when a detector fails to recognize certain happens-before relations in a program. For example, in Figure 1(a), the address of `data` is transferred from thread 1 to thread 2 and then used by thread 2 to conduct a task written in `data`. Because of the causal relationship between S2 and the address transfer process, the two shared accesses at S1 and S2 would never race with each other, yet may be reported as a race if a detector fails to recognize the implicit happens-before relation. On the other hand, a benign race is indeed a data race but is not a bug, i.e. the data race is intended or allowed as shown in the example in Figure 1(b).

Table 2 shows the results from Inspector by using 5 real-world applications including Apache, Berkeley-DB, etc. It shows that the accuracy in detecting harmful races is only around 2%. Interestingly, Not-A-Race is the most dominant source of the inaccuracy,

Class	Definition
Not-A-Race	It is NOT a true data race but reported by a data race detector. If the detector is dynamic, it is usually introduced by non-recognized happens-before relation guaranteed by a program (e.g., through synchronization).
Benign data race	It is a true data race satisfying the data race definition, but does not affect the program correctness.
Harmful data race	It satisfies the data race definition and results in incorrect program behavior once it is exposed.
* data race definition: more than one threads access the same memory location without proper synchronization, and at least one of them is a write.	

Table 1: ***The cases reported by existing data race detectors*****. Besides harmful data races, many existing tools report both Not-A-Race and benign data races as well.**



(a) **Not-A-Race**: the order of S1->S2 is guaranteed by tansferring *data*'s address for task dispatching

(b) Benign data race for ad-hoc synchronization

(c) Harmful data races (a bug) that result in Apache crash

Figure 1: ***Real-world cases reported by the Intel® Inspector [15].*** **The highlighted memory accesses are reported as data races. Although (a) is *not* a data race, and (b) does *not* affect execution correctness, both are reported.**

Apps.	Total	**Not-A Race**	Benign Race	Harmful Race
Apache	592	555	30	7
Berkeley-DB	686	601	65	20
Ocean	116	94	22	0
Asterisk	29	13	13	3
Qt	30	26	3	1
Total	1453	**1289 (88.7%)**	133 (9.2%)	31 (2.1%)

Table 2: ***The accuracy of Inspector*****. Inspector provides mechanisms to leverage users' input for annotating synchronizations not automatically recognized or suppressing false positives, but we didn't use them in our experiments since domain knowledge or extra effort from the users are required. For classification, we manually checked the cases reported by Inspector. Since it is not always clear whether a reported race is benign or harmful due to the lack of program knowledge, we conservatively count benign races only when it is certain that both possible orders of two race instructions will produce exactly the same execution state, following the idea of an existing automatic benign data race classifier [33].**

accounting for 36%–94% of the total reports. In order to separate harmful data races from Not-A-Race reports and benign races, developers have to manually check all the reports, which is tedious and very time consuming. As a result, the wide use of data race detectors has been limited in practice [4].

To improve the accuracy of data race detection, there have been two main approaches: separating benign races from harmful ones [33, 18], and relying on manual annotation for synchronizations [3, 15, 1, 2]. In the first approach, [33] identifies benign data races by trying to flip the execution order of a race pair and comparing the two execution results. [18] uses heuristics to identify certain code patterns that may introduce benign data races. Since their target is limited to benign data races, these methods *are not very helpful in identifying Not-A-Race* among the reports. Some tools including Inspector adopt the second approach by providing mechanisms to leverage users' input to annotate synchronizations that are not automatically recognized. However the annotation can be time consuming and may need source code change which requires the developers to put extra effort on it. Recently a few solutions [38, 16] have been proposed to automatically identify ad hoc synchronizations (e.g., a while-flag). They can help eliminate benign data races used in ad hoc synchronization such as the one shown in Figure 1(b), and some of Not-A-Race caused by the ad hoc synchronization. However, our study reveals that such type of Not-A-Race is not the dominant factor (details in Table 3 in Section 2.1).

1.2 Contribution

This paper first studies why an existing data race detector generates Not-A-Race reports. Our study makes a unique observation of **address transfer**, which contributes to a majority of Not-A-Race reports. Based on this observation, the second part of our study builds a tool called **ATDetector** to effectively identify address transfer related to data race candidates and eliminate them from the data race reports. As a result, we improve the accuracy of Inspector by eliminating *all* Not-A-Race reports caused by unrecognized address transfer.

Unique Observation of Address Transfer: In multi-threaded applications, there are two common ways for multiple threads to access shared data. The memory address may be globally known to all threads from the beginning, and thus they can immediately access the data with its own knowledge of the location. Alternatively, the location to access may be made known to one thread by another at a certain execution point. In this case, one thread explicitly passes a memory address to another thread to give the location information. We refer to this as **address transfer**. We further identified three use cases of address transfer: (i)Task dispatching, (ii)Memory recycling, and (iii) Producer-consumer communication for data sharing. Their details are in Section 2.2.

Address transfer implicitly introduces a happens-before relation between the sender's and the receiver's memory accesses to the

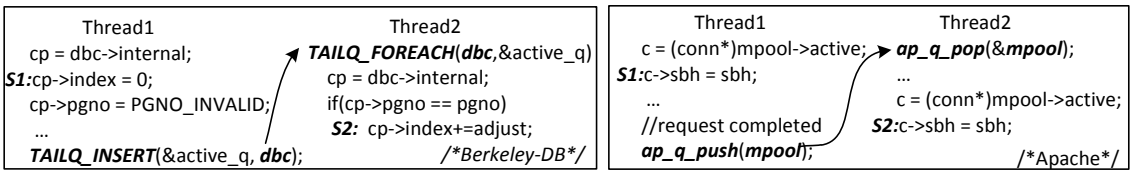


Figure 2: ***The real-world examples of address transfer.*** **The addresses transferred are highlighted. In addition to these two, another use case of address transfer is task dispatching, which is shown in Figure 1(a). Due to the happens-before relation introduced by address transfer, S1 and S2 never race each other.**

transferred address. For instance, in all the three examples in Figure 1(a) and Figure 2, the memory access at S1 by thread 1 always happens before the same memory access at S2 by thread 2. Since S2 uses the received address either directly or indirectly, it causally depends on the address transfer process. Again, the address transfer can happen only after S1.

Importantly, however, existing data race detectors, including commercial strength tools such as Inspector, may not be aware of the happens-before relations introduced by address transfer. It is because address transfer is often implemented with application-specific methods, such as lock-free data structure (e.g., the Apache example in Figure 3(a)), customized locks (e.g., the Berkley-DB example in Figure 3(b)), system calls (e.g., TransmissionBT example in Figure 1(a)), and other synchronization primitives recently developed such as transactional memory (e.g., Intruder example in Figure 3(c)). In general, existing data race detectors recognize only standardized synchronization primitives (e.g., pthread_mutex_lock). Consequently, they incorrectly report the pairs of S1 and S2 in all the examples as data races, while all of them are Not-A-Race.

Specifically, according to our investigation of the results from Inspector with 17 diverse open-source applications, the majority of Not-A-Race reports (62% of 1420) are caused by the unrecognized address transfer (the details are shown in Table 3).

Address Transfer Detection (ATDetector): Based on the observation of address transfer, we propose a method called *ATDetector* that automatically identifies address transfer and uses the information to prune Not-A-Race reports. *ATDetector* focuses on identifying only those address transfers that are related to data race candidates collected by data race detectors.

Given a data race candidate and its memory address, starting from the later race instruction candidate, *ATDetector* backtracks all the relevant address propagation paths within a thread and also across multiple threads. Meanwhile, it collects all potential address receiving and sending sites, which can be shared memory read/write or certain system calls. In this way, we can recognize address transfer regardless of the customized way of implementation. Finally, if one of the address transfer candidates indeed breaks the potential race condition by enforcing happens-before relation, the data race candidate is pruned out.

To implement this idea, *ATDetector* conducts binary instrumentation with Pin [9] to trace memory and register accesses, and performs online dependency analysis with the traces when it encounters potential race candidates.

Improving the accuracy of Inspector, a state-of-the-art commercial data race detector: We used *ATDetector* to improve the accuracy of Inspector by eliminating Not-A-Race reports caused by unrecognized address transfer without any users' input. Since the source code of Inspector is not available, instead of integrating *ATDetector* to the tool, we run it by feeding Inspector's data race reports to *ATDetector* as inputs.

We evaluated *ATDetector* with 8 diverse open-source server, desktop, and scientific programs, including Berkeley-DB, Apache, TransmissionBT, etc. As a result, *ATDetector* can help eliminate **all** of the Not-A-Race reports caused by unrecognized address transfer, which are 62% of the total Not-A-Race reports from Inspector. More importantly, *ATDetector* does **not** eliminate any true data races that are originally reported. It has a modest memory overhead ranging from 2KB to 2MB per thread for most of the evaluated applications.

2. ADDRESS TRANSFER

This section studies the reasons an existing data race detector introduces Not-A-Race reports. Our study makes a unique observation of memory address passing among multiple threads, which is referred to as *address transfer*. The observation includes (i) unrecognized address transfer is the major cause of Not-A-Race reports (Section 2.1); (ii) address transfer is commonly used for task dispatching, memory recycling, and producer-consumer communications (Section 2.2); (iii) address transfer is implemented with various customized methods, such as lock-free structure (Section 2.3).

Our study is conducted by using Intel® Inspector [15], and 17 open-source multi-threaded programs. To understand the Not-A-Race reports and address transfer, we manually examined a total of 1775 cases reported by Inspector, including 1420 Not-A-Race reports.

2.1 Causes of Not-A-Race Reports

As shown in Table 2, majority(88%) of the cases reported by Inspector are Not-A-Race reports. This problem becomes more severe in some large server applications such as Apache and Berkeley-DB. Therefore, it is critical to eliminate Not-A-Race reports to improve the accuracy of data race detection.

Table 3 further shows the three causes of Not-A-Race reports:

(1) Address transfer: Instead of being a global variable, data can alternatively be shared by passing its address from one thread to another. We refer to the process of passing memory address as *address transfer*. For example, in Figure 1(a), thread 1 dispatches a new task to thread 2 by transferring the address of `data`, which describes a new task. Other uses of address transfer are explained in Section 2.2.

Similar to lock synchronization, address transfer implicitly introduces a happens-before relation between the address sending thread's and the receiving thread's accesses to the same memory location, which is mistakenly reported as a data race. It guarantees that the sender's memory access happens before the receiver's access to the same memory location. The address of the memory accessed may be either exactly the same as the transferred address, or calculated from the transferred address, as shown in Figure 1(a). In this example, S1 is guaranteed to happen before S2 due to the `data`'s address transfer, and the accessed memory address (ad-

	Apps.	Not-A-Race	**Address Transfer**	Customized Synchronization	Others
Server	Berkeley-DB	601	272	329	0
	Apache	555	555	0	0
	Squid	2	0	0	2
	Asterisk	13	3	10	0
	FreeSwitch	1	0	1	0
Desktop	tcmalloc	64	27	37	0
	HandBrake	2	0	0	2
	TransmissionBT	8	7	1	0
	Qt	26	0	26	0
	WxdFast	2	0	2	0
Scientific	intruder	14	8	6	0
	bayes	4	3	1	0
	genome	6	3	3	0
	labyrinth	4	2	2	0
	yada	6	3	3	0
	Ocean	94	0	94	0
	Radix	18	0	18	0
	Total	1420	**883(62.2%)**	533(37.5%)	4(0.3%)

Table 3: The causes of Not-A-Race reports from Intel Inspector.

dress of `data->session`) is calculated from the transferred address (address of `data`).

Unfortunately, most of the address transfers examined are implemented with various customized methods (details are in Section 2.3), and existing data race detectors including Inspector mainly focus on standardized synchronization primitives such as POSIX pthread_mutex_lock, in order to recognize happens-before relation or/and lock-set. Because of this, it results in many(62% of 1420) Not-A-Race reports, as shown in Table 3.

(2) Customized Synchronization: Programmers often implement their own methods of synchronization instead of using the ones provided by the standard libraries. It can be due to many reasons such as performance or convenience [38]. This is especially true in server applications. The synchronization can be a customized lock or an ad-hoc synchronization such as a while loop waiting for an event from the remote site (e.g., the while loop in Figure 1(b) waits until `session->event` is set).

Typical data race detectors only recognize standard primitives such as those from POSIX, and cannot deal with customized synchronizations without user annotations. This is the same in Inspector, and it results in 37.5% of Not-A-Race reports. A few previous works including our own [38] deal with the problem of ad-hoc synchronizations. However, it is less significant compared with Address Transfer and is not the focus of this paper.

(3) Others : For conditional shared data access, a special type of order synchronization can be implemented with a flag and an if-statement checking it. Specifically, it guarantees that certain memory accesses happen only if another access to the same memory from a different thread already happened. Otherwise it skips the memory access without waiting for the remote peer. Inspector fails to recognize such condition checking and generates a few(0.3% of 1420) Not-A-Race reports.

2.2 Use Cases of Address Transfer

In general, there are two ways for a thread to get the memory address of shared data. First, a memory address can be global (e.g., global variables) so that every thread has access to them during its lifetime. In this case, a thread can immediately access the data without communicating with other threads. Alternatively, the addresses of shared data may only be made known to one thread by another at a specific execution point of the program. For example, a thread may produce data for other threads to consume by explicitly passing the location information of the data first. Similarly, address transfer can be used to pass the address of a memory pool that is freed by one thread and later acquired by another thread.

Specifically, from Not-A-Race reports of Inspector, we observed the following three major use cases of address transfer in the 17 real-world multi-threaded programs.

(1) Task dispatching: Especially in multi-threaded server applications, it is common for one thread to create tasks and dispatch them to other threads to execute. The address of a task structure can be either directly transferred to a specific worker thread or passed through a shared task queue. The TransmissionBT example in Figure 1(a) shows that thread 1 directly dispatches the task produced to thread 2 via transferring the address of the task structure, `data`.

(2) Customized memory recycling: Many applications implement their own memory allocators for efficiency. It is possible for a thread to reuse a chunk of memory just returned by another thread to a shared memory pool. In the example in Figure 2(b), after thread 1 ends its task using `mpool`, the address of `mpool` is inserted to a global queue. Then thread 2 reuses the pool by fetching the address from the queue.

(3) Producer-consumer communications for data sharing: In some cases, multi-threaded programs need to dynamically manage shared objects by adjusting the data structure when encountering new program states or input, and the shared objects are passed in a producer-consumer manner. For instance, in the Berkeley-DB example in Figure 2(a), all active cursors are organized as B-Tree by using `index` field. Thread 1 creates the database cursor `dbc` and inserts its address into a shared queue. Thread 2 iterates all the active cursors and adjusts `index` to re-organize the B-Tree. The address of `index` is derived from the address of `dbc` transferred via the shared queue.

2.3 Address Transfer Implementation

Address transfer needs a synchronized communication between the address sender and the receiver, as other data sharing. However, according to our investigation of the data race reports, in many cases (62% of all Not-A-Race reports in Table 3), address transfer is implemented without standard synchronization primitives such

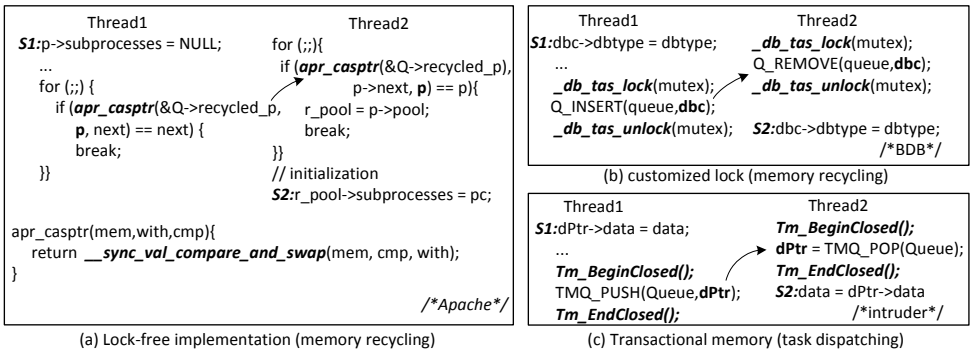


Figure 3: ***Address transfer implementations that result in Not-A-Race reports.*** **Arrow shows the address transfer direction. Inspector fails to recognize such address transfers, and reports each pair of S1 and S2 as a data race, which are incorrect.**

as pthread_mutex_lock. and thus becomes problematic for existing data race detectors. Specifically, the followings discuss the diverse customized methods implementing address transfer, with real-world examples in Figure 3.

(1) Lock-free implementation: Programmers often do address transfer with their own lock-free structures. One popular way is to use atomic read and write operations with a shared queue. For example, Figure 3(a) shows Apache's address transfer for memory recycling. By using the gcc's builtin function for atomic compare and swap, thread 1 posts an address of available memory location `p` to `recycled_p`, and thread 2 reuses it by reading the address atomically.

Note that while atomic operations can be used to implement synchronizations, they are not synchronizations themselves and can be used for other purposes. Therefore it is hard to automatically detect lock-free structures, though the atomic operations themselves are not difficult to identify.

(2) Customized synchronizations: Some synchronization primitives used for address transfer are customized ones that are specifically implemented, and are difficult for typical data race detectors to recognize [38] without users' inputs. The Berkeley-DB example in Figure 3(b) shows that `dbc`'s address is transferred via a shared queue which is protected by a customized lock primitive, `_db_tas_lock`, and thus cannot be recognized.

(3) New parallel programming models: With the recent development in diverse concurrent programming models such as transactional memory [37, 10] and Intel Thread Building Blocks [14], new synchronization primitives are used to implement address transfer. Figure 3(c) shows the implementation with transactional memory.

(4) System calls: Address transfer can also be done using certain types of system calls such as read() and write(). The TransmissionBT example in Figure1(a) uses the read/write system calls to transfer the address of `data` via a pipe. Without being aware of such system calls, data race detectors may fail to recognize the related address transfer.

3. ADDRESS TRANSFER DETECTION

3.1 Overview

Since address transfer severely affects the accuracy of data race detectors, it would be beneficial to identify them. Manually annotating address transfer is tedious and error-prone, as discussed in Section 1. Therefore, we develop a tool called *ATDetector* to dynamically identify the address transfer related to data race candidates , and eliminate the Not-A-Race reports. It can be integrated with a data race detector to filter out Not-A-Race reports automatically. Alternatively, as a stand-alone detector, it can also take data race candidates as input, and validates them by running the application and monitoring its execution.

As we have observed from real-world address transfer cases in Section 2.3, an address transfer can be implemented in various ways, e.g., using lock-free structure, which is not easy to automatically identify. Therefore, *ATDetector* does *not* try to identify such synchronizations used to implement address transfer. Instead, it tries to detect the dependency between the memory address involved in a race candidate and the transferred address. *ATDetector* performs backtracking[36, 6] on all relevant address propagation paths both intra- and inter-thread, trying to locate the source of the accessed memory address. Once the address transfer is identified along the path, the data race candidate is determined to be Not-A-Race and pruned. The backtracking process is shown in Figure 4. This approach makes the identification independent of how address transfer is synchronized.

However, it is not trivial to explore address propagation paths due to the following reasons. First, the memory address used in data race candidates may not be exactly the same as the address transferred. The address accessed could be calculated from the address transferred with other values. Second, address transfers may be implemented using system calls as well as shared memory reads/writes. Third, address transfers can be performed across more than two threads.

To address the above challenges, *ATDetector* conducts the followings: (i) binary analysis to extract memory address, registers in an instruction, and dependency between the operands; (ii) based on the extracted information, building traces for both memory and registers; (iii) dependency analysis on the traces in order to backtrack from *instr2* to the address receiving site (Figure 4); (iv) leveraging the semantic knowledge of certain system calls, library calls, and instructions that affect the propagation path of the address.

Figure 5 shows the detection steps. *ATDetector* first identifies all the possible candidates of address receiving sites (i.e., ATr set) using the dependency analysis explained above. Then, for each ATr, the tool finds its remote peer (i.e., ATs), which is an address sending site that either writes to the same shared memory or is a system call corresponding to ATr. Finally, it checks whether the ATs happens later than *instr*1 on the same thread. If not, *ATDetector* moves on to check another ATr from the ATr set. Otherwise, a valid address transfer is identified and the inspected data race candidate is determined to be Not-A-Race. In addition, to handle the case where more than two threads are involved in the address transfer, we recursively perform the similar steps by treating an intermediate

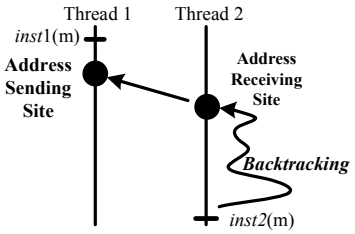


Figure 4: ***Backtracking for address transfer detection. inst*1 and *inst*2 access memory location *m*, which depends on the transferred address.**

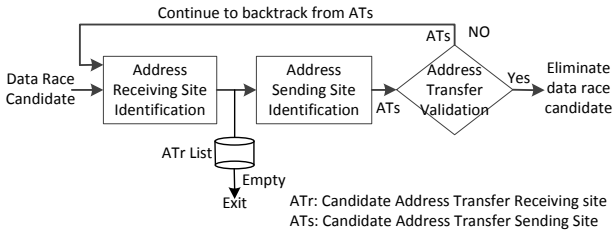


Figure 5: ***Overview of address transfer detection steps. ATDetector* tracks every identified ATs until a valid address transfer is located, or all of the tracking paths are terminated**

ATs as the second race (inst2 in Figure 4) instruction candidate and begins a new detection from it.

In order to compare the timing of address transfer related events, *ATDetector* maintains a global timestamp that ticks at each shared memory access as well as each invocation of potential address transfer related system calls such as `read` and `write`.

3.2 Address Receiving Site Identification

In this step, *ATDetector* tries to find all possible address receiving sites that a given memory address depends on. Figure 6 describes the process of address receiving site identification. The input of this process is the memory access instruction that contains the target memory address . In the first iteration, the input is the second instruction involved in the candidate data race pair. *ATDetector* then extracts the relevant register to be tracked. The memory access instruction usually utilizes two registers, one for memory address calculation, referred to as memory base register, the other for storing the value written or read from the address. For the first iteration, we get the memory base register as it is the starting point of our analysis. For the later instructions on the tracking path, we extract the register used in the source operand in order to backtrack the propagation path by following the dependency.

After the register is identified, *ATDetector* searches backward along the register trace and locates the last change of this register. Then it decides the next step based on the source of register modification. The source can be of several types:

Local variable: the simplest case is a register changed by reading from a local variable, via which the memory address is propagated. In this case, *ATDetector* further identifies the last change to this local variable by searching backward the memory access trace. The newly identified change record contains the register which holds the value to be written to the local variable, and the register becomes the next relevant register and participates in the next iteration.

Shared memory: if the source is shared memory, then it is already a candidate address receiving site, and we put it to the output list. However, the tracking process does not terminate here because we need to find all candidate address receiving sites. To do this, *ATDetector* extracts the memory base register of this instruction and continues the local tracking.

Others: there can be other sources of registers along the local tracking process, such as library calls and arithmetic computation. *ATDetector* addresses each of them specifically as follows:

Library calls and system calls affect the detection in two ways: they are actually used to perform the transfer as the example in Figure1(a), or they are in the address flow paths. *ATDetector* deals with these function calls as long as it knows their semantics. For example, if the function can be a data receiver such as "read", it outputs the callsites as ATr. On the other hand, if the function does no communication but just passes on the data between the arguments and return value, such as `memcpy`, *ATDetector* continues the local tracking by attaining the source arguments. Semantics of more library and system calls can be added to *ATDetector* on demand.

Arithmetic computations may be also used to generate memory addresses, especially in the case of arrays or complex data structures. For example, the `LEA` and `ADD` instructions are often used in this case. *ATDetector* deals with them by forking the tracking path according to the operands, and tracks each of the source operands.

The backtracking process terminates when (1)a valid address transfer that orders the two instructions of a given data race candidate is identified, or (2)no more record can be found from the trace, or (3)the backtracking encounters certain function calls, or (4)the record has a timestamp smaller than the first access in the race candidate. Among them, conditions (1) and (2) are intuitive. For (3), some function calls can also terminate the backtracking such as `malloc` and `read`, since their return values do not depend on previous data flow. And for (4), this is because it is impossible to locate a valid address transfer if the current instruction already happened before the first instruction in the race candidate.

Note that in the backward tracking based on dynamic execution trace, inter-procedure analysis is guaranteed automatically. This is because *ATDetector* observes the actual execution sequence of instructions, and it does not need to care about whether the modifications are made inside some routines or not.

3.3 Address Sending Site Identification

After the candidate address receiving sites are identified, *ATDetector* attempts to find the corresponding address sending site for each receiving site. The challenge of this process comes from the wide variety of address transfer implementations. If the address receiving site is a shared memory read, *ATDetector* simply checks the memory trace and looks for the last memory write to the shared memory. However, if the candidate receiving site is the read end of a system call, *ATDetector* needs to consult the knowledge base to get the semantics of the functions involved in the transfer. For example, if the receiving site is a call to *read()* that reads a previously created pipe, as in Figure1(a), *ATDetector* has to pair it with the corresponding *write()* call which is the sending site candidate. Based on the semantics of pipe, it first finds the matching file descriptors that form the pipe, and then instead of finding the latest writing site, *ATDetector* searches the execution trace and finds the writing end based on the syscall invocation order.

3.4 Address Transfer Candidate Validation

Once the candidate address sending site is also identified, *ATDetector* checks if the pair enforces a happens-before relation between the candidate race instructions. Specifically, it checks (i) whether the first access involved in the data race report has a smaller timestamp (happens earlier) than the identified address sending site, and

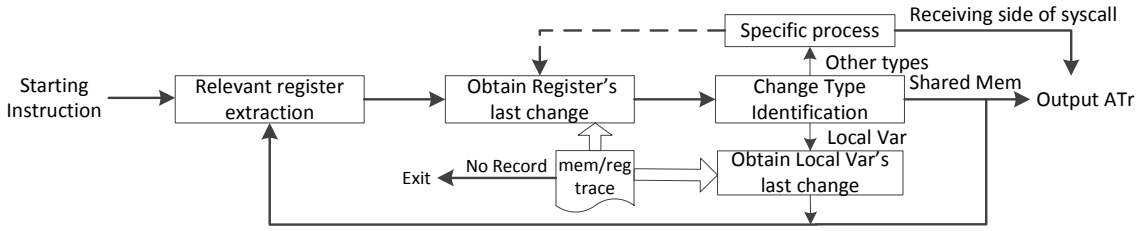


Figure 6: *The process of address receiving site identification.* based on the memory and register trace, *ATDetector* backtracks the address propagation path in order to locate potential address receiving sites. When an ATr is output, it continues to track from the ATr in order to locate further possible receiving sites. The termination conditions are discussed in Section 3.2

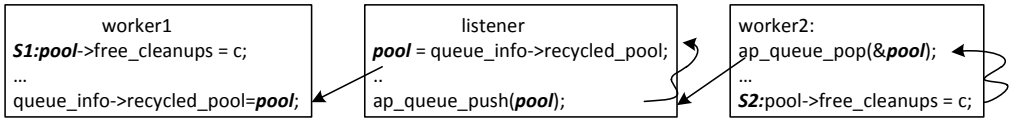


Figure 7: Multiple threads involved in the address transfer

(ii) both of the first access and the address sending site are from the same thread. These two conditions guarantee that the first access happens before the address transfer. Since the backtracking step explained in earlier sections naturally reveals that the second access happens after the address transfer, satisfying the two conditions above finally guarantees that the given two accesses have happens-before relation, so it is a Not-A-Race pair.

If the first condition is not satisfied, *ATDetector* drops the address transfer candidate. If the second one is not satisfied, the candidate sending site is either in the local thread where the backtracking started, or another thread. *ATDetector* needs to continue to backtrack from the identified candidate sending site in both cases. In the first case, the tracking is sill on the same thread and needs to continue looking for further possible address transferring sites. In the second case, our observation is that address may be transferred between more than 2 threads. Figure 7 shows such an example, where tracking continues in the listener thread.

3.5 Implementation

ATDetector is implemented using Pin[9]. Pin provides a comprehensive runtime inspection platform to analyze a program's behavior, and plenty of APIs to inspect each instruction. *ATDetector* records the necessary traces by instrumenting memory accessing instructions as well as instructions and system/library calls that could affect the address propagation, as described in Section 3.2 and 3.3. *ATDetector* leverages the instruction inspection APIs to analyze binary code. However, it is difficult to accurately decide whether a memory is shared or not. Therefore *ATDetector* conservatively treats all heap memory as potentially shared. Heap memory can be recognized by checking if the address falls into thread stack address range.

Since we do not have the source code of Inspector and can not directly integrate *ATDetector* with it, we use its data race report as the input. The program runs under *ATDetector* and when a candidate data race is triggered, *ATDetector* starts the address transfer detection. If an address transfer is detected, the candidate race is classified as a Not-A-Race report.

In order to traceback the execution and to find how the memory address propagates, *ATDetector* records every instruction that may lead to data passing. A typical data race detector keeps the record of memory accesses, which can be optionally utilized by *ATDetector* when tracing back the memory modifications. However, a unique additional requirement in *ATDetector* is the trace of the register modifications. For example, *ATDetector* needs to record instructions that affect the address propagation path, such as `MOV`, `ADD`, and `LEA`, etc. Besides memory and register trace, *ATDetector* also needs to record the relevant system calls and library calls as described earlier.

The trace is maintained in a per-thread manner to reduce the contention. As the program runs, the trace may grow larger than the memory capacity especially when the trace size is not limited. *ATDetector* periodically flushes the traces to disks, and fetches them when needed during address transfer detection.

3.6 Memory Usage

A critical concern for tools that perform runtime analysis is the memory overhead. If too much memory is consumed by the tool itself, the target application may suffer from extremely low performance (frequent page-swaps). This is especially true when it needs to record the execution trace, which is usually very large and needs to be minimized or compressed for high performance [36, 24, 39].

An important observation of address transfer is that ***the address receiving site and the second memory access show strong spatial locality with regard to the instructions***. For example, in the Berkeley-DB code shown in Figure2(a), the program immediately accesses the data after taking from the list. Other examples in Figure 1(a), Figure 2, and Figure 3 share the same characteristic. Based on this observation, we can limit the amount of memory for recording the register trace from the address receiving site to the memory accessing instruction without significant accuracy loss. *ATDetector* allows users to specify the amount of memory for recording, and the sensitivity of the trade-off between memory consumption and detection effectiveness is shown in Section 4.3.

Note that loops may hurt the memory efficiency by producing large amount of memory accessing records. Currently *ATDetector* detects and skips simple spin loops. Complex loops can also be detected by building the control flow graph in the pre-execution phase[16]. We leave it to the future work.

Currently *ATDetector* uses the same-sized trace buffer for each thread. However, different threads may require different amount of space for trace, depending on their tasks. Even though our evaluation results show that *ATDetector* has moderate memory overhead, using simple optimization techniques like dynamically adapting buffer size will help further reduce the overhead.

apps	Not-A-Race related to Address Transfer		True Race	
	Reported by Inspector	**Pruned by *ATDetector***	Reported by Inspector	**Pruned by *ATDetector***
Apache	531	531	37	0
Berkeley-DB	247	247	74	0
Asterisk	3	3	18	0
TransmissionBT	8	8	8	0
tcmalloc	25	25	4	0
intruder	8	8	9	0
genome	3	3	1	0
labyrinth	2	2	6	0
Total	827	**827(100%)**	157	**0(0%)**

Table 5: *Effectiveness of ATDetector in pruning out Inspector's Not-A-Race reports related to address transfer*. Note that the number of such reports from Inspector is different from Table 3 (i.e., "Address Transfer" column), since some data race candidates reported by Inspector are not even appeared in *ATDetector* 's run.

Apps	Apache	Berkeley-DB	Asterisk	TransmissionBT	tcmalloc	intruder	genome	labyrinth
trace size	1.6MB	40MB	2KB	2MB	8KB	2KB	2KB	2KB

Table 6: *The size of register traces per thread*. It shows *ATDetector* 's memory overhead when it prunes all Not-A-Race reports caused by address transfer.

4. EVALUATION

4.1 Experimental Methodology

In our experiments, we evaluate the accuracy of *ATDetector* using Inspector's data race reports on 8 real-world applications including 3 server applications, 2 desktop application, and 3 scientific benchmarks using transactional memory from STAMP [23]. Table 4 shows the description of each application. Since the source code of Inspector is not available for us, it is not possible to directly integrate our proposed idea to Inspector. Therefore, we first run Inspector alone to get data race reports, then separately run *ATDetector* with the data race candidates as its inputs. The application inputs are the same in both phases.

4.2 Effectiveness of ATDetector

Table 5 shows the effectiveness of *ATDetector* in pruning Not-A-Race reports that are caused by unrecognized address transfer. First, *ATDetector* successfully eliminated **all** of the Not-A-Race reports from Inspector, as long as the data race candidates are encountered during *ATDetector* 's standalone execution. Please note that, due to some timing issues, the data race candidates encountered during Inspector's detection may not all appear again in *ATDetector* 's runs. It is not an issue originated from *ATDetector* , and can be avoided by integrating our idea directly into a data race detector, or by dynamically enforcing the target data race candidates to execute during a standalone run, just like many testing tools [34, 21]. Second, *ATDetector* eliminates **no** true data races that are originally reported by Inspector.

The above results indicate that *ATDetector* is highly accurate to identify address transfer, and is thus practical to improve widely-used commercial data race detectors.

Apps.	LOC	Description
Apache	248K	Apache web server
Berkeley-DB	388K	Database library
Asterisk	530K	Telephony server
TransmissionBT	96K	Bit-torrent cliente
tcmalloc	20K	Memory allocator
intruder-STAMP	1.3K	Network intrusion detector
genome-STAMP	1.3K	Gene sequencing algorithm
labyrinth-STAMP	1K	Maze routing algorithm

Table 4: Applications used in the evaluation

4.3 Memory Overhead

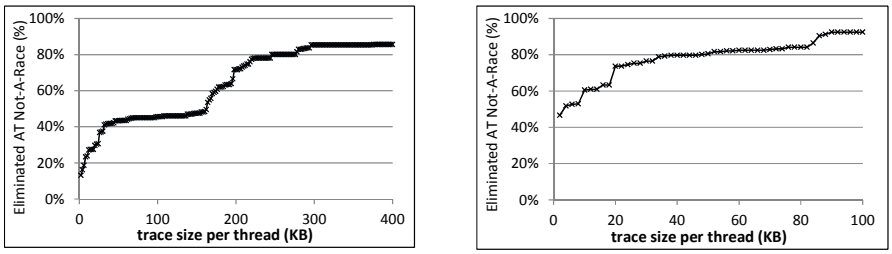
The memory overhead may be concerned especially due to tracing register writes, as discussed in Section 3.5. Table 6 shows the maximum size of register trace per thread. It shows that the size varies from 2KB to 40MB, depending on applications. But except for Berkeley-DB, most applications use modest amount of memory during the entire detection run.

Since some applications such as Berkeley-DB require a large amount of trace buffers , the size of memory used for trace should be limited and is better to be adjustable by users. To further study the sensitivity of the algorithm to the size of register trace, we measure *ATDetector* 's pruning rates by changing the size of register trace buffer with Apache and Berkeley-DB, which are the top two applications showing high number of Not-A-Races related to address transfer.

As the result shown in Figure 8, even with the small size of trace buffers, *ATDetector* can show high pruning effectiveness. In the case of Apache, with only a 160KB trace buffer per thread, *ATDetector* can still prune around 50% of the total targets. The pruning rate eventually reaches 98% on 800KB, and then 100% on 1.6MB, which are not shown in this graph due to space limit. In the case of Berkeley-DB, the pruning rate increases even faster at the beginning: 90% of the total targets are eliminated only with 100KB trace buffer per thread.

This results reveal that there exists a strong spatial locality of the instructions while backtracking to address transfer sites. It is intuitive in that, after a thread receives a memory address, it usually accesses the memory right away, as we discussed in Section 3.6.

Nevertheless, there are some extreme cases where *ATDetector* 's backtracking requires a large register trace buffer. For instance, once an Apache thread receives the address of memory pool, long time later, the thread can do a final access to the memory to cleanup the memory pool, happening to be a race candidate. To identify the



(a)trace size sensitivity of Apache

(b)trace size sensitivity of Berkeley-DB

Figure 8: ***The sensitivity to the size of register trace buffer.*** **Due to the space limit, we are not showing all x-axis. At the maximum trace size shown in Table 6, the graph will eventually reach 100%, meaning that *ATDetector* can eliminate all the Not-A-Race reports related to address transfer.**

```
inst 15d4e1 <libdb-4.8.so> at thread3 (start_thread)->...->3fcda(__bam_search)
inst 159542 <libdb-4.8.so> at thread2 (start_thread)->...->15d4bd(__memp_fget)
                                                              Data Race Candidate Info
--Begin Tracking--
change stack var at b1dfe81c is inst 15916f via reg 17  <__memp_alloc> <libdb-4.8.so>, step 1
...                                                                  Tracking
changing of register 17 at instruction 12fb4e is global <__env_alloc> <libdb-4.8.so>, step 11   Process
global mem change to b399e020 is at instruction 130493 <__env_size_insert> <libdb-4.8.so> by thread 2
--Address Transfer identified between inst 15d4e1@thread3 and inst 159542@thread2 Because of
Provider: 130493@thread2 (start_thread)->...->13029b(__env_alloc_free)
                                                              Address Transfer Info
Receiver: 12fb4e@thread3 (start_thread)->...->15d4bd(__memp_fget)
```

Figure 9: ***ATDetector 's detailed output to aid user inspection. ATDetector*** **'s output includes three sections, each of which describes the data race candidate, the tracking process explaining the dependency between the accessed address and the transferred address, and the identified address transfer sending/receiving sites. It reports the relative virtual address of each instruction for portability.**

address transfer for memory recycling, *ATDetector* needs to keep all the register traces collected for the long duration.

4.4 Output to Aid Inspection

For each eliminated Not-A-Race report, *ATDetector* outputs the detailed information in order to help users understand why the pruned pair is not a race, and how the memory address is passed from one thread to the other. Figure 9 shows the *ATDetector* 's output for one Not-A-Race report prunned on Berkeley-DB.

4.5 Result Discussion

As an automatic tool, *ATDetector* efficiently helps avoid human mistakes in reasoning about data races, especially for complicated multi-thread programs. In our study, there are several cases where we have mistakenly determined as true races because of some "common senses" such as "if one object is the last one accessed in a critical section, it is very likely to be protected by the surrounding lock [11]". There is one case in Berkeley-DB where the write to a global variable is the last statement in a critical region, and the read is not protected. Therefore we identify it as a true data race. During the experiment, however, *ATDetector* reports all of its dynamic instances as Not-A-Race. By checking the output as Figure 9, we find it is actually well ordered by address transfer. And the lock surrounding the write is meant to protect other fields of the same data structure. Without knowing this, one may fix the problem by adding a lock to the read, and thus degrade the software performance by adding potential contention.

Similar to all the dynamic program analysis tools, *ATDetector* analyzes the program behavior based on a specific execution, and does not consider the different behaviors in the others. Therefore it is possible that some instances of a candidate data race are ordered by address transfer but others are not. An instance is the recycled data structure may contain pointers to the same globally shared variable which might be accessed simultaneously by multiple threads and thus forms true race. This is a well-known limitation of happens-before relation based analysis, and would not be prevented even if the programmers manually and correctly annotate the synchronization points. In order not to prune true races, *ATDetector* only reports address transfer if all the dynamic instances of a candidate race in the execution can be ordered. Our experiments show that there are some true races that can be ordered by address transfer in certain instances. However none of them has all the instances detected as Not-A-Race in our study.

5. RELATED WORK

Data race detection and its accuracy Data race detection has been under intensive study for a long time, and there are a lot of work using both software and hardware to perform the detection [11, 35, 15, 30, 7, 32]. More discussion is in Section 1.1.

In order to improve the accuracy of the data race detectors, some tools require users' annotation in order to mark the customized happens-before relation in the programs [15, 3, 1, 2], and some others classify benign data races from harmful data races [33, 18].

The detection of ad-hoc synchronization [38, 16] can help automatically eliminate Not-A-Race reports by providing one type of originally missing happens-before relation. However it only accounts for small portion of all the Not-A-Race reports. See Section 2.1 for details. RaceFuzzer [21] uses the reported data race information as a heuristic to test "error-prone" schedules in multi-threaded programs. It produces the scenarios of real data races by controlling the scheduling randomly. However it suffers from prunning large amount of real races (as much as 50%) as the schedules randomly produced by their scheme cannot cover all bug-triggering interleaving especially when the data race set is large.

Program dependency analysis Several works exist for program

dependency analysis such as dynamic dependency profiling, memory tainting, and backtracking. Dynamic dependency profiling [39] builds the program's dependency graph by analyzing the execution traces, and is widely used to automatically parallelize sequential code. SD^3 [24] accelerates this process and reduces the memory overhead by parallelizing the dependency profiling steps and compressing the traces with stride patterns. Memory tainting marks some initial memory locations, and tracks the data flow by tainting the memory that depends on the tainted set [19, 8, 5]. It is often used to detect security attacks. LIFT [13] reduces the overhead by eliminating unnecessary dynamic information flow tracking aggressively, and running instrumented code as short as possible by efficiently switching between the target program and the instrumented code. Backtracking techniques are often used in intrusion analysis [36, 6], and are aimed to detect the sequence of events that lead to certain consequences. When intrusion is detected, it traces back the system logs and tries to locate the origin of the intrusion.

Dynamic dependency analysis is not suitable for *ATDetector* because *ATDetector* does not need the information of whole program dependency. Furthermore, dependency analysis in parallel programs is even more complicated [31]. *ATDetector* cannot utilize memory tainting because it does not know the initial tainting locations to begin with. Compared with backtracking technique, *ATDetector* needs to trace back the program memory and register trace, instead of the system logs used for the intrusion detection purpose.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we studied false data race reports (Not-A-Race reports) from a state-of-the-art commercial data race detector, Intel® Inspector, with 17 applications from various fields. By manually examining all the 1420 Not-A-Race reports, we identified two major sources of Not-A-Race reports: *address transfer* and customized synchronizations. Particularly, the happens-before relations introduced by address transfer are often unrecognized during data race detection due to its customized way of implementation, causing many (62%) Not-A-Race reports. To address this problem, we developed an automatic tool *ATDetector* to identify address transfer, and use it to prune Not-A-Race reports. We evaluated *ATDetector* with Inspector using 8 real-world applications including server, client/desktop, and scientific applications. With modest memory overhead, *ATDetector* could successfully prune all Not-A-Race reports caused by address transfer, without eliminating any true data races originally reported by Inspector.

We plan to extend this work in three ways. First, for more practical use, we plan to integrate our idea into a data race detector and help suppress Not-A-Race reports directly in the race detection process. Second, for more comprehensive characteristic study, we plan to investigate more diverse address transfer cases from other applications. Third, triggered by the high spatial locality related to address transfer and the address use, we plan to see if small hardware support for event tracing can help fast on-the-fly detection.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful suggestions. This research is supported by NSF CNS-0720743 grant, NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career reward), NSF CSR Small 1017784, and Intel grant.

8. REFERENCES

[1] Google thread sanitizer. http://code.google.com/p/data-race-test/wiki/ThreadSanitizer.

[2] Sun studio 12: Thread analyzer user's guide. http://download.oracle.com/docs/cd/E19205-01/index.html.

[3] Valgrind. http://www.valgrind.org.

[4] A.Bessey, K.Block, B.Chelf, and et al. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[5] A.Ho, M.Fetterman, C.Clark, A.Warfield, and S.Hand. Practical taint-based protection using demand emulation. In *EuroSys*, pages 29–41, 2006.

[6] A.Mairh, D.Barik, K.Verma, and D.Jena. Honeypot in network security: a survey. In *ICCCS*, pages 600–605, 2011.

[7] A.Nistor, D.Marinov, and J.Torrellas. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *MICRO*, pages 541–552, 2009.

[8] A.Slowinska and H.Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *EuroSys*, pages 61–74, 2009.

[9] C.K.Luk, R.Cohn, R.Muth, and et al. Pin: Building customized program analysis tool with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[10] C.Rossbach, O.Hoffman, D.Porter, H.Ramadan, A.Bhadari, and E.Witchel. Txlinux: Using and managing hardware transactional memory in the operating system. In *SOSP*, pages 87–102, 2007.

[11] D.Engler and K.Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[12] E.Posniansky and A.Schuster. Efficient on-the-fly race detection in multithreaded c++ programs. In *PPoPP*, pages 179–190, 2003.

[13] F.Qin, C.Wang, Z.Li, and et al. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[14] Intel. Intel threading building blocks. http://threadingbuildingblocks.org.

[15] Intel. Intel inspector xe 2011, 2011. http://software.intel.com/en-us/articles/intel-inspector-xe/.

[16] A. Jannesari and W. F. Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *IPDPS*, 2010.

[17] J.D.Choi, K.Lee, A.Loginov, R.O'Callahan, V.Sarkar, and M.Sridharan. Efficient and precise data race detection for object oriented programs. In *PLDI*, pages 285–297, 2002.

[18] J.Erickson, M.Musuvathi, S.Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[19] J.Newsome and D.Song. Dynamic taint analysisi for automaic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[20] K.Poulsen. Tracking the blackout bug. http://www.securityfocus.com/news/8412, 2007.

[21] K.Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[22] L.Lamport. Time,clocks,and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[23] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp:stanford transactional applications for multi-processing. In *IISWC*, 2008.

[24] M.Kim, H.Kim, and C.K.Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *MICRO*, pages 535–546, 2010.

[25] M.Naik, A.Aiken, and J.Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[26] M.Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *HPCA*, pages 232–243, 2006.

[27] M.Prvulovic and J.Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, 2003.

[28] N.G.Leveson and C.S.Turner. Investigation of the terac-25 accidents. *IEEE Computer*, 26(7):18,41, 1993.

[29] N.Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[30] P.Zhou, R.Teodorescu, and Y.Zhou. Hard: Hardwared assisted lock-set based data race detection. In *HPCA*, pages 121–132, 2007.

[31] R.Chugh, J.Voung, R.Jhala, and S.Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, pages 316–326, 2008.

[32] S.L.Min and J.D.Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, pages 235–244, 1991.

[33] S.Narayanasamy, Z.Wang, J.Tigani, A.Edwards, and B.Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[34] S.Park, S.Lu, and Y.Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[35] S.Savage, M.Burrows, G.Nelson, P.Sobalvarro, and T.Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, 1997.

[36] S.T.King and P.M.Chen. Backtracking intrusions. In *SOSP*, 2003.

[37] U.Aydonat and T.Abdelrahman. Hardware support for relaxed concurrency control in transactional memory system. In *MICRO*, pages 15–26, 2010.

[38] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, pages 163–176, 2010.

[39] X.Zhang and R.Gupta. Whole execution traces. In *MICRO*, 2004.

[40] Y.Yu, T.Rodeheffer, and W.Chen. Racetrack:efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2006.