Karlsruhe Reports in Informatics 2010,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

TachoRace: Exploiting Performance Counters for Run-Time Race Detection

Jochen Schimmel, Victor Pankratius

2010

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

TachoRace: Exploiting Performance Counters for Run-Time Race Detection

Jochen Schimmel
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
schimmel@ipd.uka.de

Victor Pankratius
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
pankratius@ipd.uka.de

ABSTRACT

Fixing data races is a difficult parallel programming problem, even for experienced programmers. At the moment, dynamic race detectors are frequently used because they find races more reliably than other approaches; however, the dynamic approach significantly influences application behavior during debugging because all thread's memory accesses need to be monitored. Despite using such detectors at application development time, complex parallel applications may manifest existing races only after deployment at customers, leading to crashes and corrupted data. Addressing these problems, we present TachoRace, a run-time detector using hardware performance counters in a novel way for identification of data races and inconsistent locking. Our low-overhead technique monitors cache coherency bus traffic to detect parallel accesses to unprotected shared resources and helps resolve conflicts transparently, thus making it appealing for production environments. Contrary to other run-time race detectors, TachoRace mostly builds upon existing hardware and makes a novel proposal to use synergy effects between hardware needed for debugging and hardware needed for performance analysis. As a proof of concept, we fully implemented the TachoRace detector as a software simulator; we evaluated it by executing real C/C++ applications from the Helgrind and SPLASH2 benchmark suites, including 29 representative parallel bug patterns derived from real-world programs. Experiments showed that TachoRace did not report false positive races and only missed one race pattern that could not be found by design. The average traffic overhead introduced by TachoRace to detect and automatically fix occurring races was just 0.2%.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.5 [**Software Engineering**]: Error handling and recovery

Technical Report 2010-01
Karlsruhe Institute of Technology, Germany
Institute for Program Structures and Data Organization (IPD)
Multicore Sofware Engineering Young Investigator Group
April 7, 2010

Keywords

Debugging, Race Conditions, Performance Counters, Multicore

1. INTRODUCTION

Multicore processors are standard in every PC and many programmers need to write multithreaded shared memory programs. Race conditions are among the most frequently introduced parallel programming errors, and might go unnoticed due to rare manifestation; even experienced programmers have difficulties finding races. Unfortunately, the exact solution to the problem of finding all races in arbitrary parallel programs is equivalent to the halting problem [6]. Thus, race detectors have to rely on heuristics; this means that no detector will ever be perfect and that there is no way around trade-offs.

During the software development process, race detectors are typically employed in the implementation and testing phase. Complex multithreaded programs, however, might still contain races after being deployed at the customer's site, causing a program to behave in unexpected ways and produce incorrect results. Overlooking races during development can easily happen due to program complexity and the inherent imperfection of race detectors. In the long run, it is thus important to also address the race detection and prevention issues at run-time, i.e., during productive execution of an application. This improves software reliability on desktop PCs and servers and is also useful in scenarios with multicore embedded systems where software updates can't be applied too frequently.

Although run-time race detection approaches have been proposed in the past [5, 13, 16, 20, 24, 25, 34] they typically require significant system resources or specialized hardware extensions that are too expensive for most everyday scenarios. Our work makes a completely novel proposal to reduce such cost, basically by using performance monitoring hardware for bug detection and making just a few incremental extensions to standard hardware. In addition, we build upon synergy effects that were not exploited in the past: if our proposed hardware extensions are not needed for run-time race detection, they can instead be used for more accurate performance monitoring.

This paper introduces TachoRace, a novel light-weight race detector that leverages data from hardware performance counters (which are available in many modern processors) for data race detection. Such counters (resembling "tachometers") were originally designed for performance analysis and are used by tools such as Intel VTune [8], but this paper

will show how they can be used for debugging and on-the-fly race detection. We use performance counters to track down events in the first-level cache of the cores of a multi-core processor and prevent races if unsynchronized accesses are detected. TachoRace effectively identifies and corrects races occurring due to wrong locking. It is not designed for situations in which locking is incorrectly not done at all; it also does not correct races that actually do not occur.

Our initial experiments revealed that not all processors have equally good performance counter support; collected data can be noisy because it can't be gathered selectively enough. This motivated us – for a more general validation – to build a simulator using the PIN framework [17]. TachoRace executes binary programs and simulates caches, cache protocols, and performance counters. Based on simulations, we also tested novel hardware extensions and evaluated their effectiveness for software debugging. This paper presents two run-time race detection strategies that differ in precision and the required level of hardware and software support.

The paper is organized as follows. Section 2 presents data race patterns that can be detected by TachoRace. Section 3 discusses requirements for run-time race detection. Section 4 shows how TachoRace detects races using hardware performance counters. Section 5 introduces an extension to automatically prevent races as soon as conflicting accesses are detected. Section 6 describes TachoRace's implementation, and Section 7 shows the results from experiments. Section 8 contrasts TachoRace to related work. Section 9 discusses open issues and future work. The paper provides a conclusion in Section 10.

2. COMMON DATA RACE PATTERNS

A data race occurs when two threads simultaneously access the same memory location without synchronization, and at least one of them performs a write operation. This work focuses on locks as a means for synchronization, motivated by the observation that large amounts of code as well as many programmers currently use locks.

Figure 1 shows four simplified but common code patterns using two threads, three of which are racy. Previous work has shown that these error patterns are representative for errors occurring frequently in practice, so they need special attention [26]. In the Figure, each thread increments the global shared variable `x`. In Figure 1(a), there is no locking at all. Although the "`++`" operator seems to be atomic, it actually executes a read and a write operation, so `x`'s value could be 1 or 2, depending on the thread execution order. Most programmers are aware of such problems and intend to introduce a protective lock. A common mistake, however, is shown in Figure 1(b): a missing lock in the second thread, i.e., not all necessary places are protected. User studies have also shown that programmers tend to forget such locks for read operations (even though concurrent writes might occur), wrongly assuming that reads don't need protection [22]. Another common mistake is to secure accesses to the shared variable by different locks, as shown in Figure 1(c). This can easily happen in large programs with code distributed in many different files. Our approach is designed to find races caused by wrong locking, i.e., patterns as in (b) and (c). It also works for a generalization of these cases (e.g., with more than two threads or several locks).

2.1 Lock - Locked Element - Relation

Programming languages that allow the explicit definition of locks often do not allow the expression of a connection between a lock and the data protected by this lock; this semantic relationship only exists in the developer's mind. Nevertheless, experienced programmers write self-documenting code including this information in comments, variable names, or conventions on data structures. For example, consider in C++ test case number 301 from the Helgrind Test Suite [30]:

```
int var;    /*GUARDED_BY(mu1)*/
Mutex mu1;  /*This Mutex guards var*/
```

Another example uses a naming scheme:

```
int var;   Mutex mu_var;
```

The Linux Kernel illustrates structural conventions; the lock of the corresponding locked element is located in the first field of a structure:

```
struct rq {
  /* runqueue lock: */ spinlock_t lock;
  /* nr_running and cpu_load should*/
  /*...remote CPUs use both these ...  */
  unsigned long nr_running;  ...
}
```

Even though Java or C# have the *synchronized* keyword that helps determine the lock-locked element relationship, programmers in practice mostly use explicit locks due to performance reasons (see [12] for a performance comparison).

2.2 Consequences for Run-Time Race Detection

As will be shown later, knowing the lock-locked element relationship can dramatically improve the precision of run-time race detection. TachoRace introduces a single language extension, `lock_annotate`, to make this relationship (that is often visible anyway) explicit. For illustration, consider a pattern protecting the variable account by `acc_lock`:

```
int account = 0; Lock acc_lock;
/*acc_lock protects account*/
lock_annotate(&acc_lock, &account, sizeof(account));
```

The `lock_annotate` construct registers the address of the lock, the address of the locked element and the locked element's size. The locked element can be composed of other elements that are contiguously stored in memory. The lock annotation construct can be implemented as a native language construct, as a runtime library function, or as an operating system call.

Are annotations worth the effort? Earlier studies [22, 23] have shown that parallel programs typically have only a few lines of code containing synchronization constructs, so the expected number of annotations is small. Moreover, developers document the relationship between locks and locked objects in their code anyway. It thus is natural to standardize the expression of such relationships to improve parallel program understanding. Even though similar annotations were proposed in earlier work [32], they were not designed for automated on-the-fly race detection.

Section 4.2 will show that TachoRace also works without the proposed lock annotations, but in a less accurate way if the lock-locked element relationship has to be inferred.

```
int x = 0;

void Thread1_inc(){
  x++;
}

void Thread2_inc(){
  x++;
}
```
(a) No Locking

```
int x = 0;
Lock lock_x;

void Thread1_inc(){
  lock(lock_x);
  x++;
  unlock(lock_x);
}

void Thread2_inc(){
  x++;
}
```
(b) Inconsistent Locking

```
int x = 0;
int y = 0;
Lock lock_x;
Lock lock_y;

void Thread1_inc(){
  lock(lock_x);
  x++;
  unlock(lock_x);
}

void Thread2_inc() {
  lock(lock_y);
  x++;
  unlock(lock_y);
}
```
(c) Wrong Locking

```
int x = 0;
Lock lock_x;

void Thread1_inc(){
  locka(lock_x);
  x++;
  unlock(lock_x);
}

void Thread2_inc()
{
  lock(lock_x);
  x++;
  unlock(lock_x);
}
```
(d) Correct Locking

Figure 1: Examples (a)–(c) show some common patterns of incorrect lock usage that can be detected by TachoRace.

3. REQUIREMENTS FOR RUN-TIME RACE DETECTION

Traditional dynamic race detectors [31, 10, 9] log and analyze all thread's memory accesses. This introduces significant run-time and memory overhead. This is inappropriate for run-time race detection: we need low overhead, small amounts of data to analyze, and support for long-running applications. We also need to avoid halting a program when races are detected, that is, we expect a fault-tolerant behavior in which the race is automatically prevented or fixed.

To satisfy these requirements, we designed TachoRace as a light-weight approach. It detects data races in running applications in a novel way by observing bus traffic (i.e., cache coherency protocol information) between level-one caches of processor cores. We avoided too complex hardware modifications so that current processor hardware can be incrementally improved at cost that is lower than that of heavily specialized extensions. TachoRace thus builds on cache coherency protocol information gathered from state-of-the-art hardware performance counters. For race prevention at runtime, Section 5 discusses a simple but effective extension of the MESI [7] protocol.

3.1 Exploiting Hardware Performance Counters

Hardware performance counters (HPCs) are integrated in standard processors to measure a variety of run-time statistics, such as memory accesses or prefetching information. Only a few instructions are needed to read a hardware performance counter on an Intel Pentium or Intel multicore processor: one instruction defines the metric, and other instructions are used to start, stop or reset the counter. Reading cache coherency protocol data through performance counters incurs almost no run-time overhead compared to the overhead introduced by other race detectors [31, 10]. Among others, we use for example the `CMP_SNOOP` performance counter [7] to monitor MESI messages on the bus; this counter counts the number of cache lines requested by processor cores.

Unfortunately, current processors' performance counter capabilities vary widely. Counters on Intel processors that are configured to observe cache events neither provide the memory addresses of accesses causing these events, nor can they filter events on a range of memory addresses. Counters on other architectures such as the IBM Power5 processor are more advanced [29], but also insufficient in this respect. To reliably conclude which particular memory access was responsible for the generation of a particular event, current hardware needs extensions to track the addresses related to memory accesses in cache events. As this data cannot be gathered from existing processors, we built a simulator to validate our approach.

3.2 Hardware Extensions

Performance counters in current processors are not selective enough and need better event filters. As shown in Figure 2, our approach introduces just one additional debug register attached to each core, which consists of one memory address field and one size field (in bytes) for a shared data element. This register is used to configure a performance counter to only count events with accesses to the specified memory location; the data size – if greater than zero – expands the filter to a contiguous memory region.

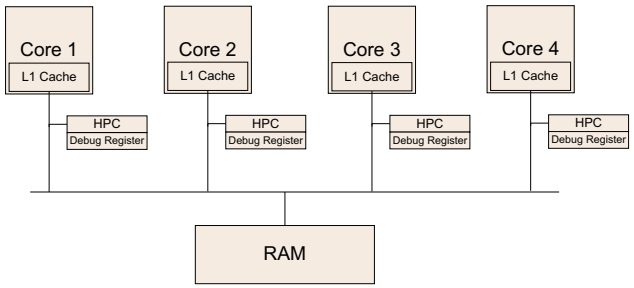

Figure 2: Data bus with attached hardware performance counters. Each counter has a debug filter register.

The information stored in debug registers represents the lock-locked element relationship discussed in Section 2.1. The debug register's address field contains the starting address of a locked data element; the size field can be used

to monitor contiguous data structures such as objects or arrays. Debug registers can be used in two different race detection modes. One mode requires program annotation, the other does not, as presented next.

4. RACE DETECTION WITH CACHE PERFORMANCE COUNTERS

We present two approaches that are both implemented in TachoRace. The first approach requires program annotations to make race detection more accurate. The second approach does not require any program annotations at all, but is less accurate.

4.1 Lock Annotation Mode

In this mode, critical sections' locks are annotated by a programmer, as discussed in Section 2.2.

Figure 3 illustrates TachoRace's rationale for run-time race detection. Suppose that a programmer forgot a lock, as in thread 2 in Figure 1 (b). In step (1), thread 1 running on core one enters the critical section to increment x, and the address of x is stored in core one's debug register. In addition, a corresponding performance counter is initialized on core one to count all other core's MESI events accessing this address. Thread 1 loads x from main memory into its local cache, and increments it safely in step (2); the MESI events generated on the bus are visible to all cores. If thread 2 attempts to increment x at the same time (step 3), it has to fetch x in the same way by getting the value from main memory to its local cache, also generating MESI messages on the bus. These messages also reach core one and increment its counter; this indicates an incorrect usage of locks, as no other thread should have been allowed to access x while thread 1 held the lock.

As processor documentation shows [7], starting and stopping hardware counters causes very low overhead. Configuring the counters also has a low overhead, which makes TachoRace applicable in a real-time context.

How to fill the debug registers? The *lock* and *unlock* operations in libraries such as Pthreads can be extended to configure the registers and start/stop the counters in a way that is not visible to the user. In our simulator, we fully implemented the register configuration in this way using the PIN [17] framework. We remark that the principle presented here can be extended to protect nested locks, provided that additional debug registers are available on each core. We successfully validated this extension in our simulator, as described later.

4.2 Annotation-Free Mode

In situations where annotations are not available, the lock-locked element relationship has to be inferred. Firstly, *lock* and *unlock* operations need to be trapped to dynamically add the set-up of performance counters and the debug register initialization. Secondly, in a block enclosed by a *lock* and *unlock* operation, it has to be inferred which data elements were intended to be protected by that particular lock. The exact information which memory location actually needs the lock is not available due to missing annotations.

The answer to this question is not straightforward in practice because programmers (and especially inexperienced programmers) may be conservative and include in a critical section more data elements than necessary (especially thread-local ones). Monitoring uncritical elements that will never conflict with other threads does not help finding races. TachoRace therefore follows a best-effort approach and randomly picks an element in the critical section when it is entered, assuming that this is one that must be protected; the probability of an element being picked is the inverse of the total number of data elements in the critical section. TachoRace then assigns the memory address of this element to the debug register to monitor accesses. The remaining steps of the race detection work as described in Section 4.1.

Obviously, this strategy has the drawback to pick the wrong data element to monitor, while a race manifests on the memory address of another data element that is not being monitored. The probability of picking the right element has to be multiplied by the probability of observing a thread schedule that leads to a race. However, as this debugging approach works at run-time, TachoRace assumes that malign data races will eventually be observed, provided that the application executes the critical section frequently enough. Then, the longer the execution time of the application, the higher the chances of detecting races. In addition, studies [23, 22] have shown that most of the critical sections in parallel programs are short, which means that there is a good chance of randomly picking the "right" data elements. The effectiveness of this approach can also be improved by adding more debug registers, so that more memory locations can be monitored simultaneously.

Although the annotation-free mode is not as effective as the mode requiring annotations, it still helps finding races at run-time with a relatively low overhead. By contrast, dynamic race detectors like [31, 9] are better at detecting races without program annotations, but the required monitoring overhead is intolerable for on-the-fly race detection at run-time (see Section 8). TachoRace thus tries to balance these trade-offs. Without run-time race detection, existing races would be missed, and developers would have to rely on users to report strange program behavior so they can trace the symptoms back to the racy origins.

4.3 A Remark on Inconsistent Locking

Our conflict detection scheme generally works when locks are used inconsistently, which leads to races in patterns like Figure 2 (b)–(d). However, we also detect situations in which multiple read accesses to a locked element occur inconsistently; for example, thread one acquires a lock before accessing variable x (for read access only), while thread two simultaneously reads x without acquiring the lock. Technically, this is not a race, but points to a potential mistake, so TachoRace can inform the developer about this.

5. RACE PREVENTION AT RUN-TIME

We also extended the race detection approach to allow "race healing" during program execution, which means that TachoRace not only detects conflicts causing races, but also prevents them.

5.1 How it works

In principle, TachoRace avoids races by modifying in real-time thread access schedules, i.e., a bad schedule that leads to a race is modified by issuing messages on the bus that delay the executions of other cores' threads causing the conflicts. We realized race prevention by extending the MESI protocol with just two new Inter Processor Interrupt (IPI) messages: "RaceWait" and "RaceContinue".

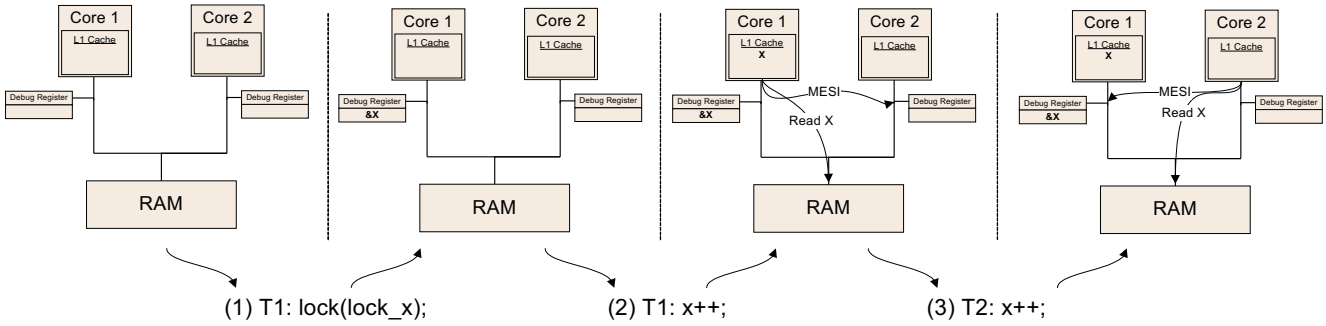

Figure 3: Detecting a data race: The code from figure 1 (b) is executed, thread 1 is runnign on core 1, thread 2 on core 2.

The working principle is illustrated in Figure 4; the left side of this Figure shows what would happen with the traditional MESI protocol, and the right side shows TachoRaces' extended protocol.

Initially (in step A1 and B1), Figure 4 depicts a situation in which core one acquired a lock on x and manipulated x's value, so x is available in core one's cache. In TachoRace, the address of the locked data element is additionally stored in core one's debug register. When core two attempts to read x from main memory, these requests become visible on the bus. In step B2, core one owning the data notices the potentially conflicting request because it listened to bus traffic, and issues a "RaceWait" message notifying of the conflicting access. In step B3, core two receives the "RaceWait" message and blocks the execution of its thread until it receives a "RaceContinue" message in step B4. The "RaceContinue" message is issued when core one's thread executes its *unlock* operation. Finally, core two's thread resumes execution in step B5 and re-issues the reading operation on x.

By design, data races are only detected and "healed" if they indeed occur, that is, a potentially racy pattern in the code that happens to be executed in a thread interleaving that does not cause a conflict, is ignored.

We implemented this approach in our simulator; we show later on benchmark programs in Section 6 that it indeed works. We chose to modify the MESI protocol due to several reasons: (1) many processors (e.g., Intel Core 2 or AMD Athlon) implement this protocol; (2) it is likely more economical to introduce simple hardware extensions rather than propose entirely different hardware.

5.2 Handling Deadlocks

We have shown that TachoRace can resolve races at runtime. There is a special case, however, in which a resolution is impossible and where TachoRace would introduce a deadlock trying to resolve a conflict. We elaborate on this problem next.

5.2.1 An Example

Let's consider the following code:

```
Lock A; /*locks X*/
Lock B; /*locks Y*/ int X,Y;
lock_annotate(&A, &X, sizeof(X));
lock_annotate(&B, &Y, sizeof(Y));
void Thread_One() {Lock(&A); X++; Y++; Unlock(&A);}
void Thread_Two() {Lock(&B); Y++; X++; Unlock(&B);}
```

Thread one acquires lock A, securing access to variable X, while thread two acquires lock B, securing access to variable Y. In the critical sections, both threads concurrently write access the variable that is locked by the other thread, so two data races are overlapping. TachoRace detects both races and responds by sending a RaceWait message to the conflicting thread for each data race. Both threads are stopped and wait for a RaceContinue message from the other thread, forming a deadlock. In more complex scenarios, more than one thread can be involved in a deadlock when unsynchronized variable accesses interleave with RaceWait messages.

The inherent problem is that TachoRace does not know the programmer's intention. The program is simply wrong. Automatic race repair avoids the race, but leads to a deadlock; the programmers is the only one to fix it. Nevertheless, TachoRace detects and reacts to such deadlocks.

5.2.2 Deadlock Detection

As an arbitrary number of threads may participate in a deadlock, we need to check for deadlocks whenever a core waiting for a RaceContinue message emits a RaceWait message. This could potentially close a circle of waiting threads. TachoRace checks for deadlocks as follows (see Figure 5): If a thread 1 pauses another thread, say 2, by sending a RaceWait message while 1 is paused itself, it broadcasts a DeadlockCheck message on the bus. This message contains the information which thread has sent the initial RaceWait message to thread 1, e.g., thread 2. All other threads read the DeadlockCheck message by snooping the bus and check if the attached thread ID matches themselves. Thread 2 thus detects that it has blocked thread 1. If thread 2 has not received a RaceWait message before, it posts a NoDeadlock message because a circular wait is no longer possible; this concludes the protocol and resumes program execution. If thread 2 has been paused by a RaceWait message before, it broadcasts a DeadlockCheck message itself, including the id of the thread that has blocked thread 2. Using this pattern, a thread will eventually either send a NoDeadlock message or the RaceWait chain exploration will reach thread 1 again (with a message containing thread 1's id), the one initially starting the deadlock checking. In the latter case, thread 1 will issue a DeadlockFound message, signaling a deadlock.

Even though checking for deadlocks is complex and may contain many IPI-messages at protocol level, it does not harm TachoRaces' ability to be used at runtime. As TachoRace only emits messages when a data race manifests itself

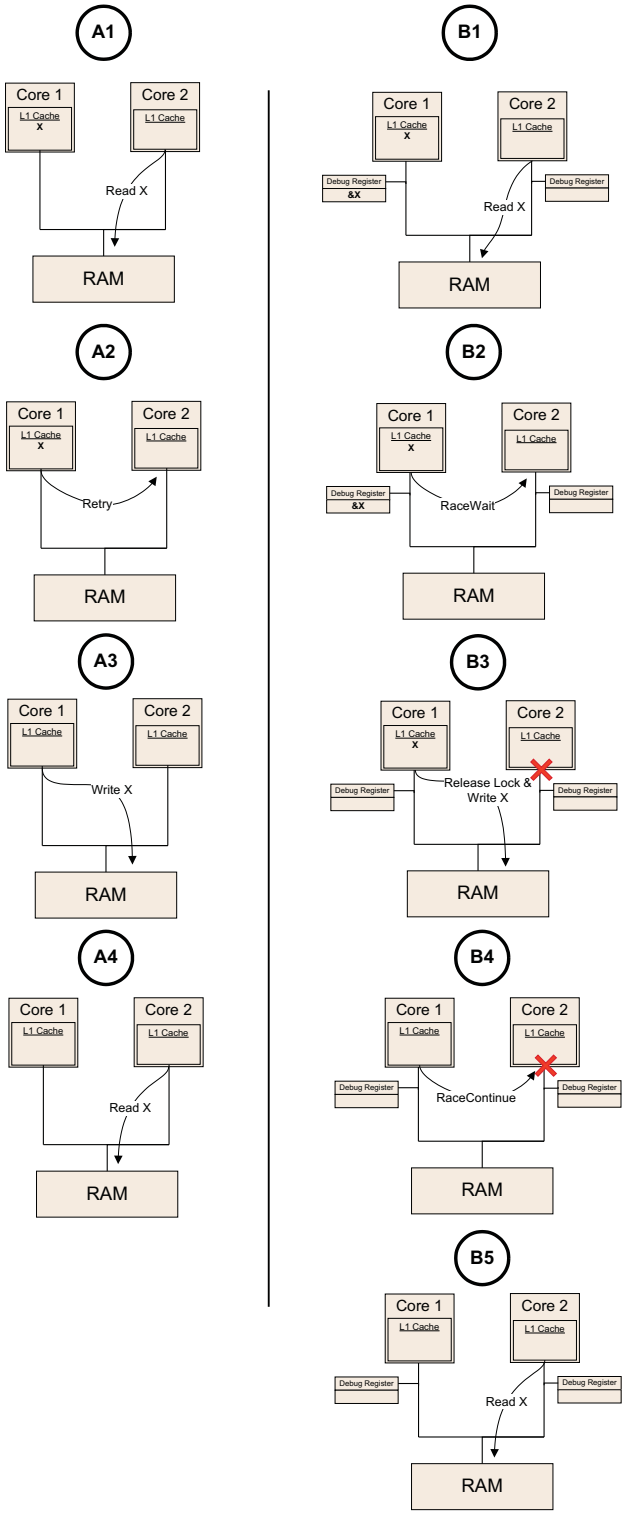

Figure 4: Illustration of the race prevention strategy using RaceWait and RaceContinue messages.

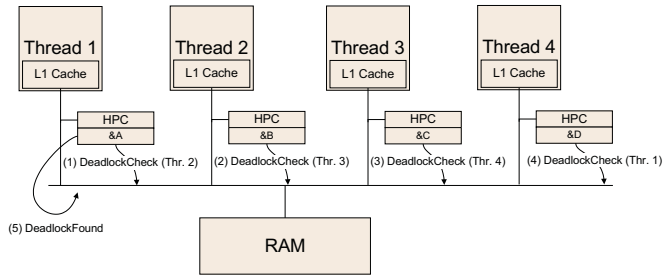

Figure 5: MESI messages sent during deadlock detection.

in an undesired memory ordering, chances for any message at all are low. Deadlocks are only checked when at least two different data races manifest themselves at the same time and the same threads are part of both races. Chances for such an error are very small. In fact, it never occurred in our experiments. Furthermore, [15] found in real-world applications that deadlocks with more than two participating threads are rare.

5.2.3 Reacting to Deadlocks

In general, two strategies are sensible to react in such situations: (1) send RaceContinue messages to all threads participating in the deadlock; (2) Throw a specific exception to be handled by the application or halt the application otherwise.

In the first case, the data race is accepted and the application continues execution, but the error is logged for the application's developers. In the second case, the developer can provide error handling code at development time, for example in form of a C++ exception handler, to recover the application or stop it gracefully. Processor exceptions are an elegant implementation (they are raised for example in case of unresolvable errors, such as division by zero or wrong pointers in protected memory mode [7]).

6. IMPLEMENTATION

Current multicore processors do not have a debug register like the one proposed in this paper, but which is required to implement our run-time race detection. To validate TachoRace, we thus developed a hardware and cache simulator based on PIN [17], which is capable of executing real, multithreaded binary applications. TachoRace runs on Windows and Linux.

TachoRace can be configured to use a wide range of cache architectures that may have a different number of cores and cache levels. Every cache level can be individually configured to be shared among selected cores. For example, it is easy to model Intel's Core 2 Quad Q6600 processor, where each pair of cores share a common L2 cache, whereas every single core has a private L1 cache. TachoRace can even use a different cache coherence protocol on each cache level. We implemented the widely-used MESI protocol [7], but our simulator can be easily extended to use MSI or MOESI [1]. We also support the Least-Recently-Used replacement strategy and an adjustable cache line size. Each cache level can be configured to be fully associative, set associative, or n-way associative. TachoRace does not consider prefetching. All caches contain data only, as in most architectures in-

Test case no.	Helgrind test case ID	Error Class	Description	(1)	(2a)	(2b)	(3a)	(3b)	(4a)	(4b)
					Offline (slow)		Offline (slow)		Run-Time (fast)	
1	1	A	write vs. write, no locking	1	1	0	1	0	0	0
2	3	D	correct synchronisation with locks and signals	0	0	0	0	0	0	0
3	4	D	correct sync., producer/consumer-pattern	0	0	0	0	0	0	0
4	6	D	correct sync. with locks and signals	0	0	0	0	0	0	0
5	8	D	correct sync. with thread-joining	0	0	0	0	0	0	0
6	9	A	read vs. write without locking	1	1	0	1	0	0	0
7	11	D	two worker threads, sync. with locks and signals	0	1	1	0	0	0	0
8	12	D	producer/consumer-pattern with mutexes	0	1	1	0	0	0	0
9	13	D	mutex-synchronisation	0	1	1	0	0	0	0
10	15	D	mutex-synchronisation, three threads	0	0	0	0	0	0	0
11	20	A	wrong sync. using timeouts	1	1	0	1	0	0	0
12	32	D	sync. with thread-joining and mutex	0	1	1	0	0	0	0
13	47	B	read vs. write with wrongly used mutex	1	1	0	1	0	1	0
14	50	B	read vs. write with wrongly used mutex	1	1	0	1	0	1	1
15	52	B	wrong signal-based sync.	1	1	0	1	0	1	0
16	55	D	correct sync. with locks	0	1	1	1	1	0	0
17	56	A	four threads, no sync. on global variable	1	1	0	1	0	0	0
18	64	A	producer/consumer-pattern with unsync. thread	1	1	0	0	0	0	0
19	65	C	producer/consumer-pattern with wrong locking	1	1	0	1	0	1	1
20	68	B	correct write, unlocked read on glob. var.	1	1	0	1	0	1	1
21	69	C	1 reader, 3 writer, wrong mutex usage	1	1	0	1	0	1	1
22	128	C	incrementing using wrong mutex	1	1	0	1	0	1	0
23	146	C	3 workers, 4 global variables, wrong mutex	1	1	0	1	0	1	1
24	301	C	2 mutexes used wrongly	1	1	0	1	0	1	1
25	302	C	2 workers, using wrong mutex	1	1	0	1	0	1	0
26	305	B	4 workers, inconsistent locking	1	1	0	1	0	1	1
27	306	B	3 workers, third without sync.	1	1	0	1	0	1	0
28	310	C	3 workers, one uses wrong mutex	1	1	0	1	0	1	1
29	311	C	4 threads, thread 4 uses wrong mutex	1	1	0	1	0	1	1
			#races detected	19	24		19		14	9
			total number of false positives			5		1	0	0

Table 1: Detection results for general bug patterns. Test cases are classified according to patterns (a)–(d) shown in Fig. 1. Colums: (1) Test case contains data race; (2a) Race detected offline by Helgrind; (2b) False positive reported by Helgrind; (3a) Race detected offline by Intel ThreadChecker; (3b) False positive reported by Intel ThreadChecker; (4a) Race detected and corrected at run-time by TachoRace in annotation mode; (4b) Race detected and corrected by TachoRace's annotation-free mode.

struction data is read-only; instruction caches are not modeled.

As a proof of concept, the implementation is based on the following model: The processor contains $n \geq 1$ processing cores, each of which has its own level one cache. Higher cache levels and main memory are shared among x cores (e.g., with $x = 2$ for Intel's Q6600). A program has a maximum of n threads, each of which is attached to one distinct core, and threads are not migrated from one core to another core. If there are more cores available than threads, the redundant cores remain idle. Only one parallel program is running at a time. Another program only starts when the previous program has finished, excluding scheduling overlaps. Threads can be deliberately paused and resumed to achieve different thread interleavings. Neither an operating system nor interrupts or traps are simulated, so the currently executing program is the only one to cause caching activity.

We remark that the restrictions in the simulation environment were chosen to create a controlled environment that cleanly demonstrates that the results are due to the race detection approach, and not due to other factors or noise. As TachoRace uses concrete hardware memory addresses, it also works when threads from different processes incorrectly access a shared resource.

7. EVALUATION

We evaluate the effectiveness of TachoRace's on-the-fly race detection at run-time and compare the results with Helgrind [31] (and open-source detector) and Intel's commercial Thread Checker [9], which are both dynamic race detectors widely used in application development.

7.1 Benchmarks

We used two well-known benchmarks: The Helgrind race detection unit tests and SPLASH2 [33]. The Helgrind unit tests [30] consist of small programs implementing common parallel error patterns that are taken from real-world programs, and other test cases (not related to parallelism) to evaluate the tool. SPLASH2 consists of several realistic parallel applications in which we introduced races by removing locks or incorrectly placing locks. For the evaluation of TachoRace's annotation mode, all lock declarations were annotated as described in Section 4.1.

The Helgrind unit tests consist of more than 50 different parallel programs. We selected appropriate test cases that were designed for race detection, resulting in a subset of 29 executable test cases. Each test creates several threads and executes a small piece of code that either has a data race or implements correct code that might look like a race to a race detection tool. Table 1 shows an overview of our test cases; out of 29 cases, 4 implement one racy pattern with no locking at all (which TachoRace cannot detect by design),

Testcase	SPLASH2 App	File	Code Lines	Effect	Testtype	Result
1	cholesky	malloc.C	141 - 145	program crashes	true-positive	detected
2	cholesky	malloc.C	150 - 188	not visible	true-positive	detected
3	cholesky	malloc.C	198 - 204	not visible	true-positive	detected
4	cholesky	malloc.C	277 - 279	not visible	true-positive	detected
5	cholesky	mf.C	109 - 126	not visible	true-positive	detected
6	cholesky	mf.C	148 - 162	not visible	true-positive	detected
7	cholesky	solve.C	329 - 332	all PIDs contain same number	true-positive	detected
8	cholesky	solve.C	349 - 360	not visible	true-positive	detected
9	cholesky	solve.C	372 - 382	not visible	true-positive	detected
10	water-nsquared	interf.C	145 - 151	not visible	true-positive	detected
11	water-nsquared	intraf.C	133 - 137	not visible	true-positive	detected
12	raytrace	shade.C	200 - 205	not visible	true-positive	detected
13	raytrace	shade.C	279 - 283	not visible	true-positive	detected

Table 2: Errors Built Into The SPLASH2 Benchmark.

10 are synchronized correctly, and the others incorrectly use locks.

From SPLASH2, the second benchmark containing larger programs, we selected three applications: "cholesky" (a numerical application), "water-nsquared" (a physical simulation), and "raytrace" (a parallel raytracer). We inserted 13 data races into these applications, with patterns like those shown in Figure 1 (b) and (c), by randomly deleting appropriate pairs of lock acquisitions and releases. The introduced bugs are similar to those shown in the Helgrind Test Suite and are also found in other real-world applications. Table 2 shows a summary of the introduced races (9 in the "cholesky" application, 2 in "water-nsquared", and 2 in "raytrace"). Some of these bugs influenced the progress of the applications and even crashed them when the race condition occurred. In these situations, TachoRace could demonstrate that it was able to detect and prevent the race conditions at runtime and avoid crashes.

7.2 Results

When comparing results, the reader must be aware that we compare online race detection techniques that are less accurate (but usable in real-time) with offline techniques that are more accurate (but slow and usable at development time). This is an inherent tradeoff and the reader should not be tempted to give accuracy results of one technique more weight over the other, because the usage scenarios and technical realizations are different.

Table 1 shows that using the annotation mode, TachoRace finds all races in all test cases that employ locks, so it works for all of the test cases where it should find a race. Out of 29 test cases, 19 contained races. TachoRace found races in 14 cases; the races in the remaining 5 cases were not detected, because they incorrectly did not use any locks at all, and TachoRace was not designed to work in such scenarios. TachoRace did not report false positives and resolved all detected conflicts at run-time by delaying the execution of the malicious thread, so the programs produced correct outputs. As a further stress-test, we inserted additional sleep statements into key positions in the parallel program's code to provoke different thread schedule interleavings. The results emphasize TachoRace's robustness: it still reported and fixed conflicts only when they occurred. No situation was encountered in which TachoRace's race healing lead to a deadlock.

In the class of offline detectors, Helgrind erroneously reported 5 race-free test cases to contain races. Intel's Thread Checker was better and reported just one false positive, but as shown in Table 3, its overhead can lead to slowdowns up to 3324x (!); this is not unusual for offline dynamic detectors.

Table 3 shows that the message overhead introduced by TachoRace through RaceWait / RaceContinue messages and counter access is low. As the concrete message communication and counter access times are machine-dependent (but on average approximately at the nanosecond scale), we considered it more meaningful to list the respective counts obtained by TachoRace. Although Thread Checker and TachoRace work in an entirely different way and overhead comparisons are difficult, we measured Thread Checker's v. 3.1 slowdown for each test case on an Intel Quadcore machine running Ubuntu Linux 9.1, to give the reader an impression of the overhead difference.

For the more complex SPLASH2 applications, TachoRace found all races and automatically fixed them. Without TachoRace, the applications produced incorrect results and even crashed.

The annotation approach is superior to the annotation-free mode, so annotations pay off. To evaluate the annotation-free mode, we executed each test case without annotations ten times. No false positives were reported, however as expected, this mode is more imprecise and reports fewer races than the annotation mode. Out of 14 test cases whose race is detectable by TachoRace, our random approach correctly identified 9 racy cases (64%) within just a few iterations. Though these results are not too bad, the technique seems promising and motivates further research on other random schemes.

7.3 Which Other Error Types Can Be Detected?

In addition to detecting data races on a single variable, TachoRace can also detect inconsistent locking involving several fields in structures, which are in principle atomicity violations. These types of errors are discussed extensively in [14] and called multi-variable access correlations. TachoRace can detect these errors whenever the correlated variables are located in the same structure and the access to the whole structure is protected by a single lock, as in the following example:

```
struct MyStruct {
        int element1;
        long element2;
        double element3;
        double element4;
}
```

We tried out this scenario in a simple test case. The lock protecting an instance of such a structure was annotated to protect the memory region of all fields. Thus, threads trying to access any of those fields without acquiring the lock generated bus events that helped TachoRace detect atomicity violations.

8. RELATED WORK

Past on-the-fly race detection approaches still introduced significant program slowdowns or required specialized hardware that was too expensive or not demanded by the mass market; these could be reasons for under-exploration [35]. TachoRace is the first approach reducing this trade-off by exploiting synergies between hardware required for debugging and hardware required for performance monitoring.

8.1 Approaches related to Transactional Memory

Contrary to the extensions proposed by TachoRace, Transactional Memory extensions used for race detection work in an entirely different way and cannot be used for performance capturing if online bug detection is not needed.

A transactional memory hardware implementation is used in [5] to detect data races. Additional registers at the granularity level of cache lines are introduced; by contrast, TachoRace works at the granularity level of memory addresses and thus avoids false sharing problems. TachoRace also targets hardware extensions that are easier to implement incrementally in current multicore processors.

ToleRace [26] is an online detector that also works on race patterns as shown in Figure 2; in contrast to TachoRace, it is more similar to Transactional Memory and operates on copies of shared variables, thus introducing more overhead.

8.2 Comparison of other detectors and tools

FastTrack [4] is an offline dynamic detector reducing overhead by using a lightweight representation of vector clocks; however, it still incurs an average slowdown of 8.5x that is too slow for online detection.

The approach proposed in [25] works at the granularity of memory pages to enable race detection and fault tolerance; it also requires lock annotations. An important difference to TachoRace is that [25] require page copies containing the locked data elements as soon as a critical section is entered, which causes high overhead and high memory consumption. This is unacceptable during run-time.

Light64 [20] also introduces an additional register per core. Light64 detects races by comparing data changes from repeated program runs. While this is suitable during development time, it is inappropriate for run-time. TachoRace fills this gap by allowing long-running applications to be monitored for races that are fixed when conflicting accesses are detected.

The online race detector of [24] requires no annotations, and programs need to be instrumented just as for dynamic race detectors; this leads to significantly more overhead than in TachoRace. In addition, the technique requires a lazy release consistency memory model and has a theoretically exponential overhead; even with pruning attempts, programs are about 200% slower. The approach worked for just two out of four tested programs.

Isolator [25] dynamically ensures isolation for programs in which some parts correctly obey a locking discipline, while others don't. In contrast to TachoRace, Isolator has a different goal ensuring that correctly synchronized program parts are not interfered by incorrectly synchronized threads. The authors also do not elaborate on a detailed solution for the situation in which Isolator's algorithm introduces deadlocks.

Contest [11] presents on-the-fly race healing by introducing sleep and other statements into multithreaded programs to influence the scheduler or introduce synchronization, but the healing has been focused and demonstrated for just one bug pattern (load-store bugs). Slowdown overhead is reported to be dependent on the situation, reaching up to 3.75x.

AVIO [13] also proposes cache coherence hardware extensions to help detecting atomicity violations, but requires multiple program runs (esp. correct runs) to extract invariants that are needed for detection. In contrast to TachoRace, this causes additional training overhead.

Colorama [2] proposes hardware extensions to automatically infer critical sections, but causes additional memory overhead and even introduces races if the inference mechanism does not make correct predictions; this cannot happen with TachoRace.

Atom-Aid [16] dynamically reduces the probability that atomicity violations can manifest. By contrast, TachoRace repairs races when they occur.

Autolocker [18] employs program analysis to find a locking policy that does not lead to race conditions and uses lock annotations similar to TachoRace. However, resource-intensive pointer analysis would have been necessary to detect all accesses to a particular variable. Contrary to TachoRace, Autolocker may refuse executing certain programs.

Hard [34] introduces a hardware implementation of the lock set algorithm, but contrary to TachoRace, does not heal races.

The hybrid dynamic race detection approach in [21] combines lock set and happens-before-based detection to improve accuracy, but has slowdowns by orders of magnitude.

Differing from TachoRace, *BugNet* [19] introduces hardware extensions for a capture-replay approach during production runs, but is intended to be an application-level debugging aid.

9. DISCUSSION

We now discuss some open issues and future extensions. Moreover, we argument that extensions to hardware performance counters are necessary (although they might not be easy to implement).

9.1 Scheduling Interference

A problem not addressed in this paper is that the operating system may interrupt and even migrate an executing thread. The operating system also has to take care that in case of migration, the debug register contents are migrated as well, and ensure that the memory bus is locked.

9.2 Current Hardware Performance Counters Are Noisy

Our initial experiments with hardware performance counters in current multicore processors have shown that these counters are not selective enough. Thus, gathered data can be noisy, because it does not isolate precisely enough the events that are of interest.

We counted coherency protocol events using an Intel Core

Test Case No.	TachoRace Overhead												Counter Accesses	Comparison Thread Checker Overhead (Exectime Slowdown)
	RaceContinue		RaceWait		MESI_Invalidate		MESI_Shared		MESI_Retry		Total Messages			
	#msgs	%	#msgs	%	#msgs	%	#msgs	%	#msgs	%	#msgs	%		
1	0	0	0	0	5070	90.01	529	9.39	34	0.60	5633	100	0	248 x
2	0	0	0	0	4078	92.45	275	6.23	58	1.31	4411	100	0	122 x
3	0	0	0	0	4356	91.76	339	7.14	52	1.10	4747	100	0	1014 x
4	0	0	0	0	6326	95.19	264	3.97	56	0.84	6646	100	0	122 x
5	0	0	0	0	5243	95.34	232	4.22	24	0.44	5499	100	0	1007 x
6	0	0	0	0	24096	98.29	364	1.48	54	0.22	24514	100	0	57 x
7	0	0	0	0	4679	92.14	331	6.52	68	1.34	5078	100	0	62 x
8	0	0	0	0	7041	94.08	383	5.12	60	0.80	7484	100	0	999 x
9	0	0	0	0	5206	92.29	378	6.70	57	1.01	5641	100	0	573 x
10	0	0	0	0	8337	92.94	552	6.15	81	0.90	8970	100	0	1014 x
11	1	0.01	1	0.01	6580	91.22	577	8.00	54	0.75	7213	100	5	106 x
12	0	0	0	0	5722	91.14	497	7.92	59	0.94	6278	100	0	27 x
13	1	0.02	1	0.02	4660	88.29	568	10.76	48	0.91	5278	100	4	6 x
14	1	0.02	1	0.02	6117	94.72	288	4.46	51	0.79	6458	100	81	34 x
15	1	0.01	1	0.01	9067	95.73	346	3.65	56	0.59	9471	100	83	65 x
16	0	0	0	0	6529	94.00	361	5.20	56	0.81	6946	100	0	248 x
17	0	0	0	0	22226	95.19	1054	4.51	69	0.30	23349	100	0	3324 x
18	0	0	0	0	5760	89.16	611	9.46	89	1.38	6460	100	0	13 x
19	2	0.04	2	0.04	4172	84.27	699	14.12	76	1.54	4951	100	131	37 x
20	300	4.02	300	4.02	5045	67.57	777	10.41	1044	13.98	7466	100	464	4 x
21	9	0.13	9	0.13	5998	86.25	771	11.09	167	2.40	6954	100	195	39 x
22	1	0.02	1	0.02	5298	95.00	230	4.12	47	0.84	5577	100	65	114 x
23	1	0.02	1	0.02	4971	84.80	799	13.63	90	1.54	5862	100	322	12 x
24	1	0.02	1	0.02	4058	88.41	487	10.61	43	0.94	4590	100	3	512 x
25	20	0.20	20	0.20	8644	85.61	525	5.20	888	8.79	10097	100	327	0.43 x
26	1	0.02	1	0.02	5086	91.85	364	6.57	85	1.54	5537	100	201	77 x
27	1	0.02	1	0.02	4517	92.77	284	5.83	66	1.36	4869	100	118	222 x
28	1	0.02	1	0.02	4493	87.38	576	11.20	71	1.38	5142	100	129	12 x
29	2	0.02	2	0.02	8179	92.00	627	7.05	80	0.90	8890	100	112	76 x

Table 3: The overhead of additional message traffic introduced by TachoRace (RaceWait and RaceContinue) is low, compared to the overall traffic.

2 Quad CPU running Ubuntu Linux. It was not possible to isolate cache events caused by one particular process on a specific memory location. HPC libraries such as PerfMon2 [3] already control the counters to be active only when a certain process is running. However, this is not sufficient for race detection, as other processes' events may be counted as well.

We modified the Linux kernel to allow only threads of the process of interest to use the CPU cache by dynamically modifying the Page Attribute Table and modifying the Linux page tables. While this allowed us to restrict HPC cache coherency measurements to a single process, it is unrealistic for production environments. The limitation of current hardware performance counters are also criticized in [27, 28].

9.3 How to Protect Critical Sections Containing Non-Contiguous Data

TachoRace requires that the protected data elements are stored in contiguous memory regions. However, if hardware provided enough debug registers, this limitation could be overcome. This can be done by individually assigning one debug register to each contiguous memory chunk.

A typical example are dynamically linked lists or tree structures. As the memory areas belonging to such a data structure change over time, expressing the lock-locked element relationship is not easy. This information, however, is required to identify all memory blocks that need to be stored in the debug registers. This information can be obtained at runtime by introducing an `AddLockedElement` instruction, as in the following code example:

```
LOCK(&lock);
foreach (listelement l of list) {
  AddLockedElement(&lock, &l, sizeof(l));
  process(&l); }
UNLOCK(&lock);
```

This function iterates over an internally linked list and counts the elements. The whole list needs to be locked, so that no insert or delete operations can be performed on the list while counting. The `AddLockedElement` instruction adds the start address of each list node to the debug register before the node is visited. From then on, any access violation to visited nodes will be detected; each node is checked for conflicts as soon as it is visited. That allows changes to parts of the list that have not yet been visited by the count function, offering a higher degree of parallelism than fine-grained locks on each element.

9.4 Scalability

Our implementation is aimed at demonstrating the feasibility of the concepts behind TachoRace and shows that we can exploit synergies between bug finding and performance monitoring. In its current form, the implementation works with a number of debug registers that is less than or equal to the number of program locks. However, our approach can be extended in several ways to cope with situations where the number of available registers is too small, which will be explored in future work. For example, one possible strategy is to prioritize which locks are being monitored by TachoRace. Another strategy could use virtualization to swap locks in debug registers in a round-robin fashion, so every critical section will be monitored in the long run.

9.5 Lessons Learned for Programming Language Extensions

Despite language constructs in modern languages such as"synchronized" in Java that aim to make the parallel programmer's life easier, many programmers prefer using locks due to performance (see also study of [12]). We thus cannot ignore locks and have to make life easier for programmers who choose to use locks. As the evaluation shows, lock annotations significantly improve the effectiveness of run-time race detection. Programming languages therefore need to offer intuitive ways of expressing the lock-locked element relationship mentioned in Section 2.1. For example, a useful language extension for Visual Basic could look like this:

```
Option StrictLocking

Dim x as Object
Dim y as Integer
Dim xy_lock As Lock Locking x,y
```

Options in Visual Basic enforce a specific way of coding; `Option Strict` requires the developer to explicitly declare every variable's type. `Option StrictLocking` can be used to mandatorily require additional information for locks. A lock declaration extended by the `Locking`-keyword additionally specifies the element to be protected by the lock. If a lock secures more than one data element, a comma-separated notation could be used. This scheme may be extended to other synchronization mechanisms such as signals. In addition, the compiler may use this information to contiguously store x and y in memory.

10. CONCLUSION

The approach presented in this paper is the first to propose the exploitation of synergy effects between run-time bug detection and performance monitoring, using modified hardware performance counters to detect data races. TachoRace is also capable of preventing races as soon as conflicting accesses are detected. The approach has a low overhead, which makes it applicable at run-time in novel scenarios, for example after applications have been deployed at customer sites. This is a significant achievement improving state-of-the-art, as available dynamic race detectors considerably slow down application execution by orders of magnitude. The results validate that TachoRace's underlying principle works. Experiments show that TachoRace effectively finds and fixes races in applications and works on a variety of general bug patterns. This opens the door for new directions on how to make the execution of parallel programs more reliable in production environments.

About the authors

Jochen Schimmel is a PhD student in the "Multicore Software Engineering" young investigator group at the Karlsruhe Institute of Technology, Germany. His research focuses on parallel program debugging.

Dr. Victor Pankratius heads the "Multicore Software Engineering" young investigator group at the Karlsruhe Institute of Technology, Germany. His current research concentrates on how to make parallel programming easier for the average programmer. His work on multicore software engineering covers a range of research topics including empirical studies, auto-tuning, language design, and debugging. Contact him at http://www.victorpankratius.org

Acknowledgements

We would like to thank the LANDESSTIFTUNG Baden-Württemberg for the financial support of this research project as part of the Elite Program for Postdocs. We also thank the Excellence Initiative at the Karlsruhe Institute of Technology for funding and Sebastian Crüger for his support during TachoRace's implementation.

11. REFERENCES

[1] AMD. Amd64 architecture programmer's manual. http://www.amd.com, September 2007.

[2] L. Ceze et al. Colorama: Architectural support for data-centric synchronization. In *Proc. HPCA '07*, pages 133–144, Washington, DC, USA, 2007. IEEE Computer Society.

[3] S. Eranian. The perfmon2 project. http://perfmon2.sourceforge.net.

[4] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proc. PLDI '09*, pages 121–133. ACM, 2009.

[5] S. Gupta et al. Using hardware transactional memory for data race detection. In *IPDPS*, pages 1–11, May 2009.

[6] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical report, University of California at Santa Cruz, UCSC-CRL-94-35, Santa Cruz, CA, USA, September 28 1994.

[7] Intel. Intel 64 and ia-32 architectures software developer's manual. http://www.intel.com, December 2009.

[8] Intel. Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/, 2009.

[9] Intel. Intel thread checker v.3.1. http://software.intel.com, 2010.

[10] A. Jannesari et al. Helgrind+: An efficient dynamic race detector. *IEEE IPDPS*, 2009.

[11] B. Krena et al. Healing data races on-the-fly. In *Proc. ACM PADTAD '07*, pages 54–64, 2007.

[12] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comp. Prog.*, 58(3), 2005.

[13] S. Lu et al. Avio: detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS-XII*, pages 37–48, New York, NY, USA, 2006. ACM.

[14] S. Lu et al. Muvi: automatically inferring multi-variable access correlations and detecting

related semantic and concurrency bugs. In *SOSP '07*, pages 103–116, New York, NY, USA, 2007. ACM.

[15] S. Lu et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS XIII*, pages 329–339, New York, NY, USA, 2008. ACM.

[16] B. Lucia et al. Atom-aid: Detecting and surviving atomicity violations. In *Proc. ISCA '08*, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[17] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.

[18] B. McCloskey et al. Autolocker: synchronization inference for atomic sections. In *Proc. POPL '06*. ACM, 2006.

[19] S. Narayanasamy et al. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proc. ISCA '05*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[20] A. Nistor et al. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *MICRO'09*, 2009.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. PPoPP '03*. ACM, 2003.

[22] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical report, IPD, University of Karlsruhe, Germany, 2009.

[23] V. Pankratius, A. Jannesari, and W. Tichy. Parallelizing bzip2: A case study in multicore software engineering. *Software, IEEE*, 26(6):70–77, Nov.-Dec. 2009.

[24] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *Proc. OSDI '96*, 1996.

[25] S. Rajamani et al. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09*, 2009.

[26] P. Ratanaworabhan et al. Detecting and tolerating asymmetric races. In *Proc. PPoPP '09*, 2009.

[27] F. T. Schneider et al. Online optimizations driven by hardware performance monitoring. In *PLDI '07*, 2007.

[28] A. Shye et al. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05*, 2005.

[29] D. K. Tam et al. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09*, pages 121–132, New York, NY, USA, 2009. ACM.

[30] Valgrind-project. Data-race-test:test suite for helgrind, a data race detector, 2008.

[31] Valgrind-project. Helgrind: a data-race detector. [Online]. http://valgrind.org, 2010.

[32] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, pages 334–345, New York, NY, USA, 2006. ACM.

[33] S. Woo et al. The splash-2 programs: characterization and methodological considerations. In *Proc. ISCA'95*.

[34] P. Zhou et al. Hard: Hardware-assisted lockset-based race detection. In *Proc. HPCA '07*, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Y. Zhou and J. Torrellas. Deploying architectural support for software defect detection in future processors. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.