Exploiting Cache Traffic Monitoring for Run-Time Race Detection

Jochen Schimmel and Victor Pankratius

Karlsruhe Institute of Technology, IPD
76128 Karlsruhe, Germany
{schimmel,pankratius}@kit.edu

Abstract. Finding and fixing data races is a difficult parallel programming problem, even for experienced programmers. Despite the usage of race detectors at application development time, programmers might not be able to detect all races. Severe damage can be caused after application deployment at clients due to crashes and corrupted data. Run-time race detectors can tackle this problem, but current approaches either slow down application execution by orders of magnitude or require complex hardware. In this paper, we present a new approach to detect and repair races at application run-time. Our approach monitors cache coherency bus traffic for parallel accesses to unprotected shared resources. The technique has low overhead and requires just minor extensions to standard multicore hardware and software to make measurements more accurate. In particular, we exploit synergy effects between data needed for debugging and data made available by standard performance analysis hardware. We demonstrate feasibility and effectiveness using a controlled environment with a fully implemented software-based detector that executes real C/C++ applications. Our evaluations include the Helgrind and SPLASH2 benchmarks, as well as 29 representative parallel bug patterns derived from real-world programs. Experiments show that our technique successfully detects and automatically heals common race patterns, while the cache message overhead increases on average by just 0.2%.

1 Introduction

Race conditions are frequent parallel programming errors that are difficult to detect even for experts. Unfortunately, a solution to the problem of finding all races in arbitrary parallel programs is equivalent to the halting problem [5]. Race detection tools thus have no other choice than use heuristics and accept trade-offs, e.g., in accuracy, false alarm reports, or analysis speed.

A large body of work presents race detectors that are employed during program development [7,8,26], which introduce significant analysis overhead. Application bug reports, however, show very often that racy code might still be present after deployment at clients, which can cause severe damage when crashes corrupt data. This paper tackles this problem and introduces an approach for run-time race detection and automated race healing for production environments. Our

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 15–26, 2011.

detection heuristics focus on speed and on detecting the most common racy patterns due to wrong locking.

Current proposals for run-time race detection [4,11,12,16,20,21,28] typically require specialized hardware. Most standard hardware, however, does not have such costly extensions. The novel extensions proposed in this paper aim to lower the entry barrier and make run-time race detection available in many systems. Our key idea exploits synergy effects between hardware used for performance monitoring and hardware needed for run-time race detection. Moreover, our extensions can be used for more accurate performance monitoring if run-time race detection is not required.

We introduce TachoRace, a novel light-weight race detector that leverages data from hardware performance counters in multicore processors for data race detection. We track down events in the first-level cache of each core and automatically heal races with a new cache protocol extension. We validate the proposed hardware extensions in a controlled environment based on a simulator using PIN [13]. TachoRace executes real binary programs, simulates caches, cache protocols, and performance counters. This infrastructure allows us to precisely quantify TachoRace's effectiveness for race detection as well as performance overhead.

The paper is organized as follows. Section 2 introduces our assumptions and requirements. Section 3 presents the principles of cache traffic monitoring for race detection and healing. Section 5 shows detailed evaluations. Section 6 discusses related work. Section 7 provides a conclusion.

2 Assumptions and Requirements

2.1 Software

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 and on errors resulting from incorrect lock usage. As a race detection in general is equivalent to solving the halting problem [5], our approach specializes on finding races caused by wrong locking, i.e., patterns (b) and (c) in Figure 1, and a generalization of these patterns (e.g., with more than two threads or several locks). Previous work [22] shows that these error patterns are representative for frequent errors in practice.

Run-time race detection can be made more accurate by annotating which variable a lock should protect. Our approach introduces the `lock_annotate` language extension to let programmers specify the relationship between a lock and a locked element. 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. Here is an example in C:

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

```
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

Fig. 1. Examples for lock usage patterns; (a)–(c) are incorrect programs

Earlier studies [18,19] have shown that parallel programs typically have only a few lines of code containing synchronization constructs, so the expected number of annotations is small. We remark that even languages like Java with block-oriented synchronization keywords provide explicit locks for performance reasons [10], so the aforementioned locking error patterns can also occur in Java.

2.2 Hardware

We detect data races by observing cache bus traffic while applications are running. This can already be done on existing processors, but unfortunately the lack of measurement precision requires hardware extensions. Our specific intention was to envisage extensions that don't require a radically different hardware infrastructure, so they can eventually be available in standard processors. We thus build on cache coherency protocol information that we gather from state-of-the-art hardware performance counters. Our extensions can be used for more accurate performance monitoring if race detection is not needed.

Reading cache coherency protocol data through performance counters incurs almost no run-time overhead compared to the overhead introduced by other race detectors [8,26]. Among others, we employ the `CMP_SNOOP` performance counter [6] to monitor Modified/Exclusive/Shared/Invalid (MESI) messages and count the number of cache lines requested by processor cores.

Current processor performance counters don't provide yet all necessary functionality for race detection. For example, counters on Intel processors don't provide the memory addresses of accesses causing cache events, or filters for events on a range of memory addresses. We thus implemented a software simulator using PIN [13] to validate our technique. Our approach introduces additional debug registers attached to each core, which each consist of one memory address field and one size field (in bytes) for a shared data element. The registers are used to configure a performance counter to only count events with accesses to a specified memory location; the data size – if greater than zero – expands a

filter to a contiguous memory region. For now, we assume that threads are not migrated among cores. As a proof of concept, we evaluate a hardware configuration in which each core has one additional debug register, and assume that the number of registers suffices to supervise programs with a reasonable number of locks. The debug register's address field contains the starting address of a lock-protected data element; the size field can be used to monitor accesses to contiguous data structures such as objects or arrays. Debug registers can be initialized in a transparent way by extending *lock* and *unlock* constructs in libraries such as Pthreads.

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 code with races that actually never occur.

3 Monitoring Cache Traffic to Detect and Heal Races

3.1 Race Detection

TachoRace's principle for run-time race detection is based on inference from observed cache traffic. As an example of how it works, let's assume a dual core machine on which thread T1 executes on core C1 and thread T2 on core C2. Suppose that a programmer forgot to acquire a lock and protect variable x, as in thread 2 in Figure 1 (b). T1 enters the critical section to increment x. At this point, the address of x is stored in the debug register of C1 and a corresponding performance counter is initialized on C1 to count all MESI events accessing this address. T1 loads x from main memory into its local cache, and increments it. Now if T2 attempts to increment x simultaneously, it has to fetch x in a similar way and issue MESI messages on the bus. These messages are registered at C1, which increments the performance counter for access to x address. The new counter value greater than one indicates an incorrect usage of locks, because no other thread should have been allowed to access x.

TachoRace detects and heals data races only when they occur. Our conflict detection scheme targets inconsistent lock usage as in patterns like Figure 1 (b) and (c). However, we also handle situations in which multiple read accesses to a locked element occur inconsistently; for example, one thread acquires a lock before accessing variable x (for read access only), while another thread simultaneously reads x without acquiring the lock. Technically, this is not a race, but points to a potential error, and TachoRace reports a warning.

3.2 Race Healing

TachoRace heals races by modifying conflicting thread access schedules in real-time. Messages that delay the execution of other cores that cause a conflict are then issued on the bus. We extend the MESI protocol by five Inter-Processor Interrupt (IPI) messages: "RaceWait", "RaceContinue", "DeadlockCheck", "NoDeadlock", and "DeadlockFound".

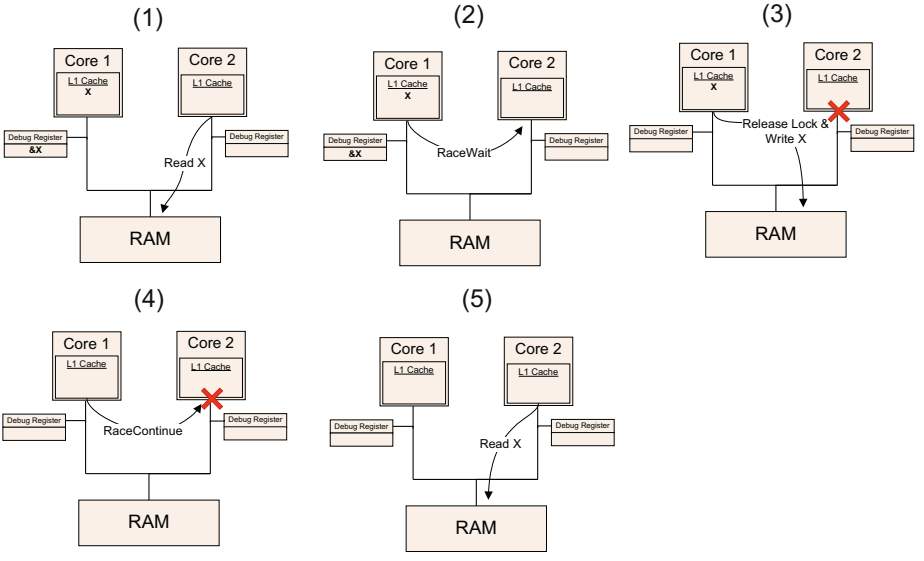

Fig. 2. Illustration of the race prevention strategy issuing RaceWait and RaceContinue messages

Figure 2 illustrates our race healing protocol. In step (1), core one acquired a lock on x and manipulated x's value, so x is available in core one's cache. TachoRace stores the address of the locked data element in core one's debug register. When core two attempts to read x from main memory, the new requests are visible on the bus. In step (2), core one detects the potentially conflicting request by listening to bus traffic and issues a "RaceWait" message. In step (3), core two receives "RaceWait" and blocks the execution of its thread until it receives "RaceContinue" in step (4). The "RaceContinue" message is issued when core one's thread executes its *unlock* operation. Finally, core two's thread resumes execution in step (5) and re-issues the reading operation on x.

Due to space limitations, we have to omit details on correctness checking of the protocol and describe how TachoRace handles some special cases. A special case is trapped when automatic race repair avoids a race, but leads to a deadlock. This can happen when two data races overlap; for example, one thread acquires lock A, securing access to variable X, while thread two acquires lock B, securing access to variable Y. If both threads concurrently write on the variable that is locked by the other thread, TachoRace would send a "RaceWait" message to each thread, but no one will issue a "RaceContinue" message. The inherent problem is that TachoRace does not know the programmer's intention in a program that is simply wrong. We thus check for deadlocks whenever a core waiting for a "RaceContinue" message emits a "RaceWait" message.

In brief, such rare deadlock situations are handled as follows. If a thread issues "RaceWait" and pauses, it broadcasts on the bus a "DeadlockCheck" message identifying the thread that has sent the initial "RaceWait" message. Snooping

the bus, each active thread checks if the message matches itself. In a match, the respective thread broadcasts a "DeadlockCheck" message if it has been paused by "RaceWait", again identifying the thread that sent "RaceWait". A thread will eventually either send a "NoDeadlock" message or the "RaceWait" chain exploration will reach the first thread, the one that initiated deadlock checking. This thread issues a "DeadlockFound" message, so TachoRace can report the incident. We remark that additional messages to handle deadlocks hardly influence average application performance. The reason is that the described chain of events rarely occurs.

4 The Detector

Current multicore processors do not have a debug register like the one proposed in this paper, but it is required to implement our run-time race detection. To validate TachoRace, we thus developed a hardware and cache simulator based on PIN [13], 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 different cache levels. Every cache level can be individually configured to be shared among certain 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 [6], 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 instruction 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. We don't simulate the operating system, interrupts, or traps, so we can ensure that 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.

5 Evaluation

5.1 Setup

We evaluate the effectiveness of TachoRace's on-the-fly race detection at run-time. As it is the first detector of its kind, we compare its results with Helgrind [26] (an open-source detector) and Intel's commercial Thread Checker [7]. We use two well-known benchmarks: The Helgrind race detection unit tests [25] and SPLASH2 [27].

The Helgrind unit tests consist of more than 50 parallel programs. We select the tests designed for race detection, i.e., a subset of 29 executable programs. All lock declarations are annotated as described earlier. 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. Table 1 shows an overview: out of 29 cases, 4 have a racy pattern with no locking at all (which TachoRace cannot detect by design), 10 are synchronized correctly, and the other 15 use locks incorrectly. In addition, we include the following applications from SPLASH2: "cholesky" (a numerical application), "water-nsquared" (a physical simulation), and "raytrace" (a parallel raytracer). We seeded 13 data races into these applications (see Table 2), using the patterns shown in Figure 1 (b) and (c) by randomly deleting pairs of lock acquisitions and releases. Some of the races influenced the progress of the applications and even crashed them when the race occurred. TachoRace proved to heal the races and prevent the crashes in all of these cases.

5.2 Results

Table 1 shows effectiveness results. TachoRace finds all races in all fourteen test cases that use locks. It is successful on all of the test cases where it should find a race. TachoRace did not report false positives and resolved all detected conflicts at run-time by delaying the execution of the malicious threads, so the programs were able to produce correct outputs. Races in the five remaining racy programs are not detected, as TachoRace was not designed to find races in programs that do not use any locks at all. In an additional stress-test, we inserted sleep statements at key positions in the parallel program's code to cause different thread schedule interleavings, and TachoRace's was able to detect the same races. We encountered no situation where TachoRace's race healing produced a deadlock.

By contrast, Helgrind incorrectly reported 5 race-free test cases to contain races. Intel's Thread Checker reported just one false positive, but its overhead lead to application execution time slowdowns of up to 3324x (!). Such huge slowdowns are not unusual for dynamic detectors that check for races at each memory access.

Table 3 shows efficiency results. The message overhead introduced by RaceWait/ RaceContinue messages and by counter accesses is low, compared to regular MESI messages that would have been on the bus without TachoRace. On

average, TachoRace introduces just 0.2% more messages. The introduced slowdowns for application execution are minor; the exact slowdown depends on the specific hardware, however, assuming $10ns$ for message handling (see [24] for clock cycle estimations) and counter access, yields for all 29 test cases a median application slowdown of $50ns$. Even the maximum slowdown of $10ms$ (e.g., for test case 20) corresponds to an application run-time slowdown of 0.002%. By contrast, measured overhead with Intel's Thread Checker is significantly higher for all test cases. On an Intel Quadcore machine running Ubuntu Linux 9.1, we obtained a median slowdown of 77 times the application execution time, and a maximum slowdown 3324 times (e.g., for test case 20).

Table 1. Detection results for general bug patterns for Helgrind, Intel Thread Checker, and TachoRace

Test case no.	Helgrind test case ID	Error Class acc. Fig. 1	Description	Has Race?	Helgrind Found?	Helgrind False Alarm	Intel TC Found?	Intel TC False Alarm	Tacho-Race Found & Healed
1	1	a	write vs. write, no locking	1	1	0	1	0	0
2	3	d	correct synchronization with locks and signals	0	0	0	0	0	0
3	4	d	correct sync., producer/consumer-pattern	0	0	0	0	0	0
4	6	d	correct sync. with locks and signals	0	0	0	0	0	0
5	8	d	correct sync. with thread-joining	0	0	0	0	0	0
6	9	a	read vs. write without locking	1	1	0	1	0	0
7	11	d	two worker threads, sync. with locks and signals	0	1	1	0	0	0
8	12	d	producer/consumer-pattern with mutexes	0	1	1	0	0	0
9	13	d	mutex-synchronization	0	1	1	0	0	0
10	15	d	mutex-synchronization, three threads	0	0	0	0	0	0
11	20	a	wrong sync. using timeouts	1	1	0	1	0	0
12	32	d	sync. with thread-joining and mutex	0	1	1	0	0	0
13	47	b	read vs. write with incorrectly used mutex	1	1	0	1	0	1
14	50	b	read vs. write with incorrectly used mutex	1	1	0	1	0	1
15	52	b	wrong signal-based synchronization	1	1	0	1	0	1
16	55	d	correct synchronization with locks	0	1	1	1	1	0
17	56	a	four threads, no sync. on global variable	1	1	0	1	0	0
18	64	a	producer/consumer-pattern with unsync. thread	1	1	0	0	0	0
19	65	c	producer/consumer-pattern with wrong locking	1	1	0	1	0	1
20	68	b	correct write, unlocked read on glob. var.	1	1	0	1	0	1
21	69	c	1 reader, 3 writer, incorrect mutex usage	1	1	0	1	0	1
22	128	c	incrementing using wrong mutex	1	1	0	1	0	1
23	146	c	3 workers, 4 global variables, wrong mutex	1	1	0	1	0	1
24	301	c	2 mutexes used incorrectly	1	1	0	1	0	1
25	302	c	2 workers, using wrong mutex	1	1	0	1	0	1
26	305	b	4 workers, inconsistent locking	1	1	0	1	0	1
27	306	b	3 workers, third without sync.	1	1	0	1	0	1
28	310	c	3 workers, one uses wrong mutex	1	1	0	1	0	1
29	311	c	4 threads, thread 4 uses wrong mutex	1	1	0	1	0	1
			Number of races detected	19	24		19		14
			Total number of false positive alarms			5		1	0

Table 2. Errors seeded in the SPLASH2 benchmark

Test	SPLASH2 App	File	Code Lines	Effect	Testtype	TachoRace
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	wrong PIDs	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 3. The message traffic overhead introduced by TachoRace is low, compared to the regular MESI traffic

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

6 Related Work

We refer to [23] for details on our alternative approach that does not require lock annotations, but which is less accurate.

On-the-fly race detection schemes typically require specialized hardware [29]. TachoRace is the first approach alleviate this problem by exploiting synergies

between hardware required for debugging and hardware required for performance monitoring.

Some Transactional Memory approaches have been extended to detect races. For example [4] require additional registers at the granularity level of cache lines; by contrast, TachoRace works at the granularity level of memory addresses and avoids false sharing problems. *ToleRace* [22] detects patterns as shown in Figure 1, but operates on copies of shared variables and introduces additional overhead.

BugNet [15] works as an application-level debugging aid and introduces hardware extensions for event capturing. The *FastTrack* [3] dynamic detector has lightweight vector clocks but still incurs average program execution slowdown of 8.5x, which is inappropriate for online race detection detection. [21] works at the granularity of memory pages and requires lock annotations. Differing from our approach [21] require page copies containing the locked data elements as soon as a critical section is entered, which leads to high overhead and more memory consumption. *Light64* [16] introduces one additional register per core and requires each program to be executed several times, so the tool can compare data changes to detect races; such repetitions are not required for TachoRace. The detector of [20] has a lazy release consistency memory model and a theoretically exponential overhead. Programs are slowed down by a factor of 200%, and the approach has been demonstrated to work just on two out of four tested programs. *Isolator* [21] 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 and ensures that incorrectly synchronized threads do not interfere in correctly synchronized parts of the program. Isolator does not provide a detailed solution for cases in which Isolator's repair attempts would introduce deadlocks.

Contest [9] introduces sleep statements into multithreaded programs to alter buggy schedules. Healing has been demonstrated just for one bug pattern (load-store bugs), and slowdowns can be up to 3.75x. *AVIO* [11] proposes cache coherence hardware extensions to detect atomicity violations, but requires multiple program runs (some of which need to be correct) so the tool can infer invariants. *Colorama* [2] proposes hardware extensions to automatically infer critical sections, but creates additional memory overhead and even introduces races if the inference mechanism does not make correct predictions; this cannot happen with TachoRace. *Atom-Aid* [12] dynamically reduces the probability that atomicity violations can manifest; by contrast, TachoRace repairs races when they occur. *Autolocker* [14] 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. In contrast to TachoRace, Autolocker may refuse to execute certain programs.

Hard [28] introduces a hardware implementation of the lock-set algorithm, but in contrast to TachoRace, it does not heal races. The hybrid dynamic race detection approach in [17] combines lock-set and happens-before-based detection to improve accuracy, but has slowdowns by orders of magnitude.

7 Conclusion

Online race detectors serve as the last safety net to prevent parallel programming bugs in applications deployed at clients from causing greater damage. The required hardware, however, is typically specialized and expensive, which makes it unlikely that it will become available in everyone's multicore system. The tradeoff approach proposed in this paper alleviates this problem by exploiting synergy effects between hardware needed for performance monitoring and hardware needed for online race detection. TachoRace not only detects common race patterns, but also automatically fixes races while programs execute. In the long run, the ideas presented in this paper can bring us closer to making on-the-fly race detection available on every multicore desktop.

Acknowledgements. We thank the Excellence Inititative and the Landesstiftung Baden-Württemberg for their support. Many thanks also to Sebastian Crüger for his support during implementation.

References

1. AMD. Amd64 architecture programmer's manual (September 2007), `http://www.amd.com`
2. Ceze, L., et al.: Colorama: Architectural support for data-centric synchronization. In: Proc. IEEE HPCA 2007, pp. 133–144 (2007)
3. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Proc. PLDI 2009, pp. 121–133. ACM, New York (2009)
4. Gupta, S., et al.: Using hardware transactional memory for data race detection. In: Proc. IEEE IPDPS 2009, pp. 1–11 (2009)
5. Helmbold, D.P., McDowell, C.E.: 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)
6. Intel. Intel 64 and IA-32 architectures software developer's manual (December 2009), `www.intel.com`
7. Intel. Intel thread checker v.3.1 (2011), `http://software.intel.com`
8. Jannesari, A., et al.: Helgrind+: An efficient dynamic race detector. In: Proc. IEEE IPDPS 2009 (2009)
9. Krena, B., et al.: Healing data races on-the-fly. In: Proc. ACM PADTAD 2007, pp. 54–64 (2007)
10. Lea, D.: The java.util. concurrent synchronizer framework. Sci. Comp. Prog 58(3) (2005)
11. Lu, S., et al.: Avio: detecting atomicity violations via access interleaving invariants. In: Proc. ASPLOS-XII, pp. 37–48. ACM, New York (2006)
12. Lucia, B., et al.: Atom-aid: Detecting and surviving atomicity violations. In: Proc. ISCA 2008, pp. 277–288. ACM, New York (2008)
13. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. PLDI 2005, pp. 190–200. ACM, New York (2005)
14. McCloskey, B., et al.: Autolocker: synchronization inference for atomic sections. In: Proc. POPL 2006. ACM, New York (2006)

15. Narayanasamy, S., et al.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: Proc. ISCA 2005, pp. 284–295. ACM, New York (2005)
16. Nistor, A., et al.: Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In: Proc. IEEE MICRO 2009 (2009)
17. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. In: Proc. PPoPP 2003. ACM, New York (2003)
18. Pankratius, V., Jannesari, A., Tichy, W.: Parallelizing bzip2: A case study in multicore software engineering. IEEE Software 26(6), 70–77 (2009)
19. Pankratius, V., Adl-Tabatabai, A.-R.: A Study of Transactional Memory vs. Locks in Practice. In: Proc. SPAA 2011. ACM, New York (2011)
20. Perkovic, D., Keleher, P.J.: Online data-race detection via coherency guarantees. In: Proc. OSDI 1996. USENIX (1996)
21. Rajamani, S., et al.: Isolator: dynamically ensuring isolation in concurrent programs. In: Proc. ASPLOS 2009. ACM, New York (2009)
22. Ratanaworabhan, P., et al.: Detecting and tolerating asymmetric races. In: Proc. PPoPP 2009. ACM, New York (2009)
23. Schimmel, J., Pankratius, V.: TachoRace: Exploiting Performance Counters for Run-Time Race Detection. Technical Report 2010-01, Karlsruhe Institute of Technology, Germany (April 2010)
24. Slater, R., Tibrewala, N.: Optimizing the mesi cache coherence protocol for multithreaded applications on small symmetric multiprocessor systems (1998), `http://tibrewala.net/papers/mesi98`
25. Valgrind-project. Data-race-test:test suite for helgrind, a data race detector (2008)
26. Valgrind-project. Helgrind: a data-race detector (2011), `http://valgrind.org`
27. Woo, S., et al.: The SPLASH-2 programs: characterization and methodological considerations. In: Proc. ISCA 1995. ACM, New York (1995)
28. Zhou, P., et al.: Hard: Hardware-assisted lockset-based race detection. In: Proc. IEEE HPCA 2007, pp. 121–132 (2007)
29. Zhou, Y., Torrellas, J.: Deploying architectural support for software defect detection in future processors. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)