Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support

Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran
Koç University, Istanbul
{hmatar, ikuru, stasiran}@ku.edu.tr

Roman Dementiev
Intel, Munich
roman.dementiev@intel.com

Abstract

It is typical for state-of-the-art dynamic race detection algorithms for C programs to slow down an application by a large factor. Our measurements indicate that a significant portion of this slowdown is due to additional lock-based synchronization performed by instrumentation code. This synchronization is necessary to ensure atomic update of analysis state. We present the first precise race detection tool that improves race-detection slowdown by using commercial hardware transactional memory support to synchronize analysis and program data. By careful choice of transaction sizes, we obtain noteworthy speedups over lock-based protection of race analysis metadata.

1. Introduction

Data races in concurrent programs are widely considered harmful because they can result in non-deterministic results for read accesses. Most of the time, this non-determinism is unintentional and can lead to unanticipated program behavior. Race conditions are also often symptomatic of higher-level logical errors, such as atomicity violations or poor program structure. Race detection tools are therefore important debugging aids and have been widely studied.

Dynamic race detection techniques monitor program executions and produce warnings about potential or actual race conditions. A precise race-detection algorithm tracks the happens-before relation as defined by the language specification and signals an error if and only if two accesses to a memory location not ordered by this relation are detected. Precise race-detection tools are desirable since they neither miss races nor produce false alarms. Unfortunately, they significantly slow down the application they are working on, which limits their use in practice. This paper is on accelerating precise dynamic race detection for C programs.

Dynamic race detection requires a program to be instrumented either statically or dynamically so that the race detection algorithm can be notified of the program's memory accesses and synchronization operations (*application events*). Precise dynamic race detection algorithms keep a significant amount of information, e.g. per-thread and per-variable vector clocks, locksets, that records the history of an execution. At each state of the execution, this *analysis state'* must be consistent with the program state and the execution being monitored. The analysis state update associated with each application event must be atomic with respect to other analysis state accesses. Race detection tools ensure this by instrumenting the program and performing additional synchronization. Our measurements indicate that a lock-based implementation of a state-of-the-art race detection algorithm spends more than half of its time in these additional synchronization operations. This *instrumentation synchronization* is a primary slowdown factor for precise race-detectors.

In this work, we take a pragmatic approach to combat instrumentation synchronization slowdown by using hardware transactional memory support, now commercially-available in commodity processors (Intel ®Transactional Synchronization Extensions, abbreviated as Intel ®TSX in the rest of the paper, in the 4th generation Intel ®Core microarchitecture, [9]). This approach could not have been widely used until recently. Our preliminary experiments indicate that hardware transactional memory for instrumentation synchronization can reduce the race detection algorithm slowdown. Currently, for C programs, slowdowns of 100X-200X are common (See, e.g., [24]).

In our approach, we start with an application program that uses conventional synchronization primitives, i.e., does not make use of transactional memory. We use hardware transactional memory support to execute each application event atomically with the corresponding race-detection algorithm processing. We accomplish this by ensuring that each execution of a race analysis function runs inside a hardware transaction delineated by the XBEGIN and XEND Intel TSX instructions.

This transactional instrumentation has a runtime overhead typically similar to the latency of atomic lock instructions and might add noticeably to slowdown for very short transactions [8]. In order to amortize this overhead, we experimented with longer transactions that span multiple application events. This approach is sound since application events remain atomic with the corresponding analysis state updates. By introducing randomization into the selection of transaction boundaries, we can ensure that every execution of the original program remains a possible execution of the instrumented program.

To compare instrumentation synchronization via Intel TSX hardware transactional support to lock-based instrumentation synchronization, we experimented with precise race detection using the *lock-based* implementation of the FastTrack algorithm ([4, 5], the fastest precise race-detection algorithm in the literature) using both approaches. We used five benchmarks from the SPLASH-2x benchmark set. Our results indicate that the Intel TSX-based approach is faster than the lock-based approach for large enough transaction sizes.

Our novel contributions are

- the first precise race-detection approach built on commercially-available hardware transactional synchronization support, therefore, practically usable widely,
- a demonstration that the use of transactional synchronization can result in significant speedups in race detection compared to earlier lock-based approaches,
- the use of coarser-grain instrumentation (longer instrumentation transactions) and an investigation of both lock-based and Intel TSX-based race detection performance with this feature.

2. Dynamic Race Detection Overview

2.1 Preliminaries

In this section, we present a simple formal model of multithreaded program executions that is sufficient to describe our technique.

Variables and Actions. Threads in a program execute actions from the following categories:

- *Data variable accesses*: $\mathsf{read}(t, x, v)$ by thread t reads the current value v of a data variable x, and $\mathsf{write}(t, x, v)$ by thread t writes the value v to x.
- *Synchronization operations*: These are operations such as lock acquisitions and releases, thread fork and join operations, barriers. We denote these operations with $\mathsf{syncOp}(t, o, \mathtt{opn})$, where thread t performs synchronization operation $\mathtt{opn}$ on synchronization variable o.

We refer to data variable accesses and synchronization operations collectively as *application events* or *application actions.*

Multithreaded Executions. An execution E is represented by a tuple $E = \langle Tid, Act, \xrightarrow{po}., \xrightarrow{so}.\rangle$.

- Tid is the set of identifiers for threads involved in the execution.
- Act is the set of actions that occur in this execution. $Act|_t$ is the set of actions performed by $t \in Tid$, and $Act|_x$ (resp. $Act|_o$) are the sets of actions performed on data variable x (resp. synchronization variable o).
- $\xrightarrow{po}_t$ is the program order (observed execution order) per thread t. For each thread t, $\xrightarrow{po}_t$ is a total order over $Act|_t$ and gives in which order the actions were issued to execute.
- $\xrightarrow{so}_o$ is the synchronization order per synchronization variable. For each o, $\xrightarrow{so}_o$ is a total order over $Act|_o$.

The *happens-before* order $\xrightarrow{hb}$ on the execution E is induced by the program and synchronization orders.

Data Races. Two data variable accesses are called *conflicting* if they refer to the same shared data variable and at least one of them is a write access. Formally, an execution $E = \langle Tid, Act, \xrightarrow{po}., \xrightarrow{so}.\rangle$ contains a race condition if there are two conflicting actions, $\alpha, \beta \in Act|_x$ accessing a data variable x, such that neither $\alpha \xrightarrow{hb} \beta$ nor $\beta \xrightarrow{hb} \alpha$ holds. Conversely, the execution is *race free* if every pair of conflicting accesses to a data variable are ordered by happens-before.

Precise dynamic race detection algorithms detect, for each access by each thread t to a data variable x, whether it is ordered by the happens-before relation with respect to all accesses to x by all other threads.

2.2 FastTrack Algorithm Overview

Since the focus of this paper is accelerating the FastTrack algorithm using hardware-supported transactional synchronization, we only describe the specifics of FastTrack relevant to this purpose. The correctness of FastTrack and and its performance were studied in earlier work [4].

Conceptually, a data race-detection algorithm such as FastTrack examines an execution of a multi-threaded program and identifies the happens-before edges between data accesses. An implementation of FastTrack augments the program with monitoring state variables. These variables, collectively referred to as *analysis state* for FastTrack, are

- a vector clock C_t for each thread t,
- a vector clock L_m for each lock m,
- a read vector clock R_x for each address (data variable) x that keeps track of the history of read accesses to x,
- a write vector clock W_x for each address (data variable) x that keeps track of the history of write accesses to x,

Implementations of FastTrack must ensure that the analysis state update corresponding to each application event is performed atomically with respect to other analysis state accesses. For purposes of illustration, let $\textsc{FTDataAcc}(\mathsf{read}(t, x, v))$ and $\textsc{FTDataAcc}(\mathsf{write}(t, x, v))$ denote the implementation of FastTrack analysis state update rules and race detection for read and write accesses. Similarly, let $\textsc{FTSyncOp}(\mathsf{syncOp}(t, o, \mathtt{opn}))$ denote the analysis state update for a synchronization event. Correct implementation of FastTrack requires that for each $\mathsf{read}(t, x, v)$ access, the execution of the associated $\textsc{FTDataAcc}(\mathsf{read}(t, x, v))$ be performed atomically. The same constraint applies to $\mathsf{write}(t, x, v)$ and $\mathsf{syncOp}(t, o, \mathtt{opn})$ and the corresponding executions of $\textsc{FTDataAcc}(\mathsf{write}(t, x, v))$ and $\textsc{FTSyncOp}(\mathsf{syncOp}(t, o, \mathtt{opn}))$.

We have implemented FastTrack for C programs, closely following the implementation presented in [4, 5]. We use `pthreads` locks to ensure atomicity of analysis function executions. Our implementation operates on a per-address granularity. In order to ensure the atomicity requirements outlined in the paragraph above, we provide two mechanisms.

- In *coarse-grain locking* mode, we use a single lock to protect *all* analysis state. This lock is acquired by each execution of $\textsc{FTDataAcc}(\mathsf{write}(t, x, v))$, $\textsc{FTDataAcc}(\mathsf{read}(t, x, v))$ and $\textsc{FTSyncOp}(\mathsf{syncOp}(t, o, \mathtt{opn}))$ before any analysis state is accessed and released after the last access to analysis state.
- In *fine-grain locking* mode, we use per-address, per-lock and per-thread locks to ensure the atomicity requirements. For each analysis state variable associated with an address, lock or thread, its lock is acquired by $\textsc{FTDataAcc}(\mathsf{write}(t, x, v))$, $\textsc{FTDataAcc}(\mathsf{read}(t, x, v))$ and $\textsc{FTSyncOp}(\mathsf{syncOp}(t, o, \mathtt{opn}))$ before the analysis variable is accessed for the first time and released after it is accessed for the last time.

We refer to the additional locks added by our instrumentation and the FastTrack implementation as *analysis locks* in order to distinguish them from locks that exist in the original application program. We experimented with both coarse-grain and fine-grain analysis lock modes in order to provide a fair comparison with the optimistic concurrency control provided by the Intel TSX instructions.

2.3 Hardware-Supported Transactional Synchronization

Transactional memory simplifies concurrent programming by allowing a sequence of instructions to execute atomically. Hardware transactional memory support is now available on mainstream Intel CPUs. In this study, we make use of hardware transactional memory support on Intel processors through Intel Transactional Synchronization Extensions (Intel®TSX) instructions. Intel® TSX debuted in June 2013 in Intel microprocessors based on the Haswell microarchitecture.

For processors with Intel® TSX support, code blocks can be designated for transactional execution using Restricted Transactional Memory (RTM) instructions. Intel® TSX enables optimistic execution of transactional code blocks. The transactional memory hardware monitors threads for conflicting memory accesses and aborts and rolls back transactions that cannot be successfully completed. This ensures that transactional code is executed atomically and that transactional blocks are serializable. Intel TSX implements a best-effort Hardware Transactional Memory support not providing a guarantee that any particular transaction will successfully commit. Therefore programmers need to implement a fall-back handler not using transactions. For this purpose Intel TSX provides a mechanism to specify a software handler for aborted transactions.

RTM provides three instructions XBEGIN, XEND and XABORT. The XBEGIN and XEND instructions mark the start and the end of a transactional code block; the XABORT instruction explicitly aborts a transaction. Transaction failure redirects the processor to the fallback code path specified as an argument to the XBEGIN instruction, with the abort status returned in the EAX register. In this study, we use RTM to protect blocks of code executing application events and the corresponding race detection algorithm functions. We also make use of the XTEST instruction. The return value of this instruction indicates whether the logical core is executing a transactional block.

3. Our Approach

3.1 Overview

The key idea in our approach is to use Intel® TSX hardware transactional synchronization support rather than locks to provide synchronization for FastTrack analysis state variables. Roughly speaking, we partition the sequence of instructions of each application thread into blocks, where each block is executed as a hardware transaction. A new transaction begins as soon as one ends, thus, a program that uses conventional synchronization is transformed so that all shared memory accesses and synchronization operations only appear within transactional blocks. I/O operations, system calls, and, when possible without restricting the length of the transaction, non-shared variable accesses and application lock acquires and releases are left outside transactional blocks. This is somewhat unconventional use of hardware transactional memory. Normally, within a hardware transaction, other synchronization constructs such as locks would not be used by an application. Here, only the additional FastTrack locks are replaced by Intel® TSX transactions. We refer to this latter approach as *TSX-based FastTrack* as opposed to *lock-based FastTrack* implementations.

We use the term *transactional block* to refer to a sequence of actions between an XBEGIN and XEND executed by a single thread. When using Intel® TSX instructions, we ensure the atomicity of each FastTrack function execution, as required for correct dynamic race detection. Given the instruction stream of a thread, we place pairs of XBEGIN and XEND instructions to partition the stream into transactional blocks and make sure that race detection code associated with each application event is run within the same transactional block.

To the best of our knowledge, for previous lock-based implementations of FastTrack, locks are acquired prior to, and released immediately after the corresponding FastTrack race checks and analysis state updates are performed. To provide a fair comparison with Intel® TSX-based FastTrack and to explore whether performance improves, we experiment with FastTrack implementations in which a single global lock is used to protect a code block that covers several application accesses and synchronization operations and the corresponding FastTrack function executions. In order to have a general term for the lock- or Intel® TSX transaction-protected code blocks, we use the term *analysis block.*

The selection of the length of transactional or analysis blocks must take into account the following concerns:

1. Hardware transactional memory limits the number of addresses that can be accessed within a transaction. If this limit is exceeded, the transaction is aborted.
2. There is execution time overhead associated with transactional instrumentation. (We perform experiments to measure this overhead). In order to minimize application slowdown due to race detection, this overhead must be amortized by using longer transactions.
3. In the instrumented program, thread interleavings that involve a context switch inside a transactional block are not explored. In other words, for longer transactions, the instrumented program allows only a subset of the thread interleavings of the original program. Shorter transactions preserve more of the interleavings of the original program.

In future work, we plan to eliminate concern (iii) by introducing randomization into the choice of beginning and end points for transactional and analysis blocks.

3.2 The Implementation

Dynamic vs. Static Instrumentation: In future work, we plan to dynamically instrument x86 binaries in order to partition each thread's instruction stream into transactional blocks. While we have this capability implemented, we had implementation difficulties while integrating PIN instrumentation and the use of Intel® TSX instructions. In this work, our goal is to provide a proof-of-concept implementation of the idea outlined and demonstrate that commercially-available hardware transactional memory support can reduce application slowdown due to additional race detection algorithm synchronization. We therefore implemented the following approach for comparison, emulating dynamic instrumentation of binaries.

Both for lock-based and Intel® TSX-based FastTrack implementations, we manually instrumented the C source code of the benchmark applications being studied and inserted (i) calls to FastTrack functions after each memory access or synchronization access, and (ii) calls for acquiring and releasing analysis locks and calls to XBEGIN and XEND as described in earlier sections. By doing so, we emulated what PIN dynamic instrumentation would have accomplished with some additional runtime overhead, for each approach. In our experimental comparisons, we contrast lock-based and Intel® TSX-based FastTrack implementations with the same transactional or lock-protected blocks.

Fall-back to Lock-Based FastTrack: A transactional block may fail to commit despite repeated re-tries for reasons including but not limited to buffer overflows and presence of instructions that are "unfriendly" with Intel® TSX in transactional blocks. To be able to continue dynamic race detection using FastTrack in these cases, during Intel® TSX-based FastTrack, we revert to using lock-based FastTrack if a transactional block repeatedly aborts.

Constraints on Transaction Boundaries: In the following cases, constraints on hardware transactional memory support forces or makes it more desirable for a transactional block to be ended or a new one started: system call instructions, input/output actions, calls to standard memory allocators, and thread forks and joins.

Barriers: Barriers are constructed from other synchronization primitives, and, one way of handling them in a dynamic race detector is to instrument the barrier implementation. However, as in FastTrack, there is a more efficient alternative, which is to treat the release from the barrier of each thread as a higher-level synchronization operation. We employ this approach as it is sound and provides better performance.

4. Experimental Evaluation

We compare the performance of lock-based and Intel® TSX-based implementations of the FastTrack algorithm for C programs. We explore the design space for both approaches. In order to provide as fair a comparison as possible, we and strive to make fair comparisons, for instance, by using the same code blocks for protection using locks vs. protection using Intel® TSX transactions.

Benchmark	Threads	Adresses	Locks	Shared Reads	Shared Writes	Lock Operations	Barrier Operations
Barnes	1	175051	219	63750909	23892727	34490	17
	2	131738	111	63752154	23893192	34500	34
	4	111994	64	63753951	23893815	34564	68
	8	101722	67	63757781	23895013	34674	136
	16	98859	74	63764960	23897144	34904	272
Fft	1	1574923	1	20392461	13566984	2	7
	2	1575946	1	20394519	13566985	4	14
	4	1577992	1	20398635	13566987	8	28
	8	1582084	1	20406867	13566991	16	56
	16	1590268	1	20423331	13566999	32	112
Lu_cb	1	263698	1	103104431	45264525	2	67
	2	263703	1	103129429	45264530	4	134
	4	263713	1	103179425	45264540	8	268
	8	263733	1	103279417	45264560	16	536
	16	263773	1	103479401	45264600	32	1072
Lu_ncb	1	262669	1	98858101	45264525	2	67
	2	262673	1	98858630	45264530	4	134
	4	262681	1	98859688	45264540	8	268
	8	262697	1	98861804	45264560	16	536
	16	262729	1	98866036	45264600	32	1072
Radix	1	2117649	1	13680665	7389189	2	11
	2	2142229	1	13770808	7462916	4	22
	4	2195485	1	13983864	7626754	8	44
	8	2306093	1	14459132	7979006	16	88
	16	2531404	1	15491596	8724470	32	176

Table 1. Concurrency data on benchmarks studied

4.1 Experimental Setup

We performed our experiments on an Intel(R) Haswell Microarchitecture i7-4770 CPU desktop with support for Intel® TSX instructions, 4 cores with hyperthreading, 8GB RAM, and a clock speed of 3.4GHz. The L1 caches were 8-way set-associative, had a size of 32KB, coherency line size of 64. The L2 caches, also 8-way set-associative, were 256KB. We used the Suse11 SP2, x86_64 Linux distribution.

We report preliminary experimental results on five benchmarks from the SPLASH-2x benchmark suite: barnes, fft, lu_cb, lu_ncb, and radix. The characteristics of the benchmarks and the inputs used are shown in Table 1. For each benchmark, as an indication of the application workload, for a 4-thread configuration, we present the number of unique memory addresses accessed, and the total numbers of read and write accesses to these addresses. We also show in the "locks" column the number of unique locks used by the application (not counting analysis locks). The lock operations column shows the number of times application locks were acquired or released. In the last column, we show number of barrier operations executed.

Every running time result reported for each configuration of a benchmark and race detection algorithm is the average of 100 runs. For each benchmark and race detection algorithm configuration pair, we experimented with 1-16 application threads. The inputs used for the benchmark were from the "simsmall" input set.

As explained in Section 3.1, to amortize the cost of starting and ending a Intel® TSX hardware transaction for Intel® TSX-based FastTrack implementations, and the cost of acquiring and releasing analysis locks for the single-lock-based Intel® TSX implementations, we experimented with different transactional (analysis) block sizes. Since we are emulating dynamic instrumentation by static instrumentation of code, we found it convenient to explore transactional block sizes from 1 to 16 application events.

Benchmark	Number of Application Threads				
	1	2	4	8	16
Barnes	69.08%	64.20%	57.06%	45.93%	34.80%
FFT	52.38%	47.06%	49.29%	36.53%	33.33%
Lu_cb	66.144%	63.80%	61.026%	50.91%	48.98%
Lu_ncb	68.21%	63.25%	61.68%	54.26%	51.67%
Radix	35.01%	32.74%	32.96%	29.78%	24.27%

Table 2. Percentage of running time FastTrack spends on instrumentation lock operations.

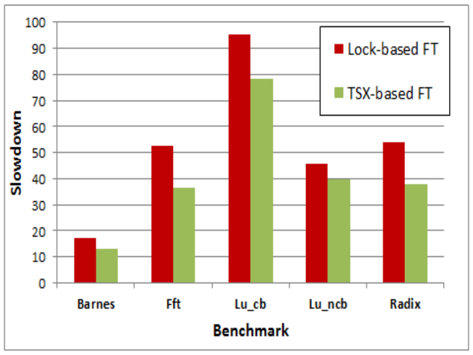

Figure 1. Slowdowns for lock-based vs. Intel® TSX-based FastTrack with 4 application threads

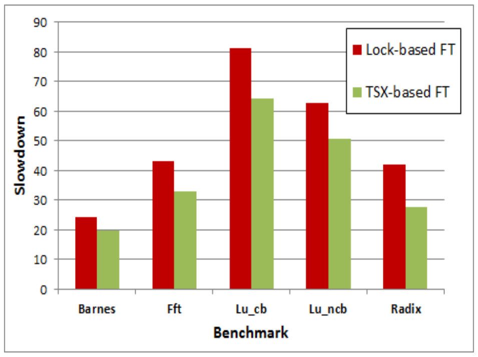

Figure 2. Slowdowns for lock-based vs. Intel® TSX-based FastTrack with 8 application threads

4.2 Findings

We next present the key findings from our preliminary experiments, along with the experimental data that support them.

- **Lock-based FastTrack spends a significant amount of its running time on instrumentation locks** As shown in Table 2, even a per-address fine-grain lock-based implementation of FastTrack spends a significant portion of the running time on operations on instrumentation locks.
- **Intel® TSX-based FastTrack is faster than lock-based FastTrack** In Figures 1 and 2 we present for our benchmarks, run with 4 and 8 threads respectively, the running time comparison between fine-grain lock-based FastTrack (the better performing lock-based option in these cases) and Intel® TSX-based FastTrack. In these comparisons, for the TSX-based implementation analysis block size of up to 16 operations is used. In both the 4 and 8 thread cases, Intel® TSX-based FastTrack is up to 1.4 times faster than lock-based FastTrack.
- **The advantage Intel® TSX-based FastTrack has over coarse-grain lock-based FastTrack increases as the number of application threads increase** In Figures 3 and 4, we plot the speedup Intel® TSX-based FastTrack accomplishes over lock-based FastTrack (single analysis lock and fine-grain analysis locking, respectively) as the number of application threads increase. These graphs show a general trend that, for the coarse-grain version of lock-based FastTrack, up to a point, the speedup of the Intel® TSX-based approach becomes more significant as the number of threads increase. We do not cur-

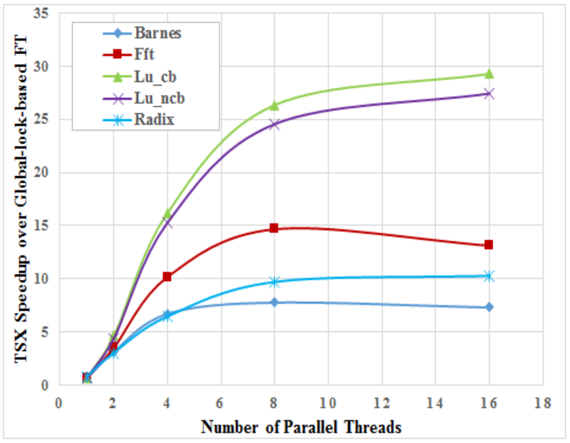

Figure 3. Intel® TSX-based FastTrack speedup over coarse-grain lock-based FastTrack vs number of application threads on a four-core machine. Analysis transactions included up to eight application events each.

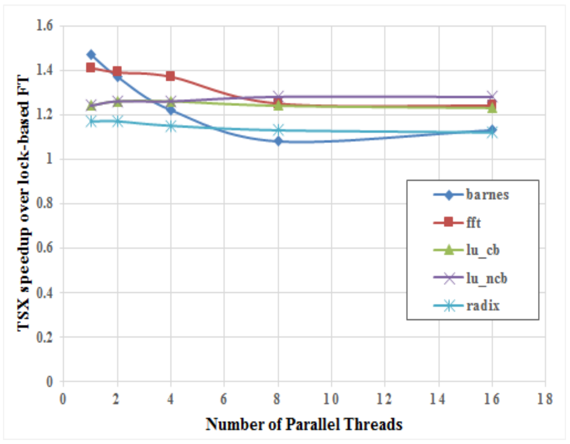

Figure 4. Intel® TSX-based FastTrack speedup over fine-grain lock-based FastTrack vs number of application threads on a four-core machine. Analysis transactions included up to 16 application events each.

rently understand why the speedup that Intel® TSX-based FastTrack has over fine-grain lock-based FastTrack diminishes as the number of application threads grows beyond the number of cores but plan to investigate this issue in future work. For machines with more cores, we expect the speedup of Intel® TSX-based FastTrack to be more pronounced as locking is known to become more expensive on machines with higher number of cores.

- **For larger transactional/analysis blocks, the win that Intel® TSX has over coarse-grain lock-based FT improves.** In Figure 5, we present how the speedup of Intel® TSX-based FastTrack over fine-grain lock-based FastTrack varies as a function of the average analysis block size. Since the coarse-grain lock-based FastTrack's performance is worse than that of the fine-grain version, we only carried out the comparison between Intel® TSX-based FastTrack for different block sizes and fine-grain lock-based FastTrack.

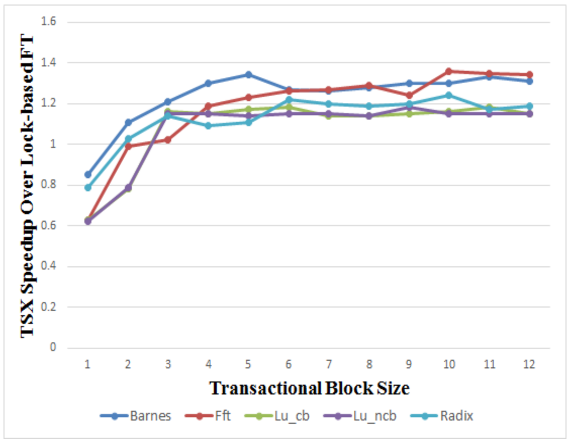

Figure 5. Intel® TSX-based FastTrack speedup over lock-based FastTrack as a function of the analysis block size. 4 application threads were used.

Although Intel® TSX in the 4th generation Intel ® Core ™ microarchitecture was mainly designed to implement lock elision [13] for critical sections encountered in ordinary multi-threaded software considered together, our preliminary findings demonstrate that it can efficiently accelerate dynamic race detection.

5. Related work

We present the first fully-precise happens-before race detector supported by commercially-available hardware transactional memory. This provides significant acceleration for dynamic race detection without requring any custom hardware. Our approach can accomodate equally well other race detection algorithms. It can also be combined with any sampling mechanism if further reduction of runtime overhead at the expense of precision is desired.

There is a large body of work on dynamic race detection. Among precise dynamic race detection algorithms, the one we build on, FastTrack [4] is among the ones with best performance. Dynamic race detection significantly slows down an application. To reduce this slowdown, a variety of techniques have been explored. Some approaches improve performance by sacrificing precision, i.e., missing some races. They accomplish this by sampling the accesses performed [18–20]. Speeding up race detection and/or replay by parallelization has also been explored in the literature. [21, 22]. Others (e.g., [16, 23]) make use of custom hardware to accelerate race detection and similar parallel program monitoring techniques.

Olszewski et al. [24] present a technique built using a hybrid of existing hardware features and dynamic binary rewriting for accelerating dynamic analyses, including the FastTrack race detection algorithm. In Aikido, a custom hypervisor is used to detect memory accesses and modest speedups of FastTrack is accomplished on most benchmarks, with the exception of a factor of six speedup on the raytracer benchmark. Aikido addresses race detection slowdown due to the detection of the memory accesses whereas our approach addresses race detection slowdown due to analysis data synchronization. Thus, the two approaches are complementary.

[25] investigate ways of using transactional memory in existing lock-based applications with a focus on programmability. Similar to our work, they allow lock operations within transactional blocks. They find that, in a majority of cases, had TM been used to implement the original program, the concurrency bugs would have been avoided. They also find a significant number of cases in which

the use of TM is not sufficient to fix an existing concurrency bug. While we, similarly to this work, intend to pursue race avoidance using TM in the future, the focus of this paper is to use race conditions as concurrency bug symptoms and to accelerate their dynamic detection as a debugging aid.

In addition to our own work on using software transactional memory for accelerating race detection [10], the research closest to ours in existing literature is by Gupta et al. [26]. Authors present RaceTM, a tool using which race detection for an application that uses transactional memory to protect portions of the code is considered. Non-transactional portions of code are wrapped in "debug transactions" and conflicts between debug transactions are reported as race conditions and conflicts between a debug transaction and a transaction in the original program are flagged as potential race conditions. For debug transactions, the rollback mechanism of the hardware transactional memory is disabled. Our work is distinguished from RaceTM in the following ways.

- Differently from RaceTM, we provide sound, precise happens-before race detection. In RaceTM, a race is informally defined as conflicting accesses to the same memory location by two threads, with the restriction that the conflicting accesses be close to each other in time. This definition of races may miss many actual races that take place in a given execution, for instance, if two debug transactions do not overlap in time, racy accesses contained within these transactions will not be detected.
- RaceTM disables debug transactions during lock-protected regions and barriers. There may be races in lock or barrier-protected code, which would be missed by RaceTM but would be handled precisely in our approach.

6. Conclusion and Future Work

We present the first precise dynamic race detection approach that makes use of hardware transactional synchronization support to reduce application slowdown. Our preliminary experiments show noteworthy speedups.

References

[1] D. L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004.

[2] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proc. 1st Workshop on Architectural and system support for improving software dependability*, ASID '06, pages 63–65.

[3] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*, pages 245–255.

[4] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *PLDI '09*, 44:121–133, June 2009.

[5] RoadRunner Lock-based FastTrack Implementation https://github.com/stephenfreund/RoadRunner/blob/master/src/tools /fasttrack/FastTrackTool.java

[6] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93*, pages 289–300.

[8] Intel. Chapter 8: Intel transactional synchronization extensions. http://software.intel.com/sites/default/files/m/9/2/3/41604.

[9] Intel. Intel architecture instruction set extensions programming reference with intel tsx. *http://download-software.intel.com/sites/default/files/319433-014.pdf.*

[10] I. Kuru, H. Matar, A. Cristal, G. Kestor, and O. Unsal. PaRV: Parallelizing runtime detection and prevention of concurrency errors. In *Runtime Verification*, volume 7687 of *LNCS*, pages 42–47, 2013.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI '05*, 40:190–200, June 2005.

[12] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *International Journal on Software Tools for Technology Transfer*, 14(3):291–305, 2012.

[13] R. Rajwar and J. R. Goodman Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution In *MICRO 34*, pages 294–305, 2001.

[14] D. Sánchez, J. L. Aragón, and J. M. García. A log-based redundant architecture for reliable parallel computation. In *HiPC*, pages 1–10. IEEE, 2010.

[15] A. H. Vineeth Mekkat and A. Zhai. Accelerating data race detection utilizing on-chip data-parallel cores. In *Intl. Conf. on Runtime Verification (RV), 2013*, INRIA Rennes, France, 24-27 September 2013.

[16] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS '10*, pages 271–284.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95*, pages 24–36.

[18] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proc. of the Workshop on Binary Instrumentation and Applications (WBIA '09).* pages 62-71. 2009

[19] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *(ICSE '11).* ACM, New York, NY, USA, pages 401-410. 2011.

[20] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *(OSDI'10).* USENIX Association, Berkeley, CA, USA, 1-16. 2010

[21] U. C. Bekar, T. Elmas, S. Okur, and S. Tasiran. Kuda: Gpu accelerated split race checker. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, London, England, UK, March 2012.

[22] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS XVI*, SIGPLAN Not. 46, 3 (March 2011), pages 15-26.

[23] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07).* pages 121-132.

[24] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: accelerating shared data dynamic analyses. In *(ASPLOS XVII).* ACM, New York, NY, USA, 173-184.

[25] H. Volos, A. .J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. In *(ASPLOS XVII).* pages 211-222.

[26] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Roetteler. RaceTM: detecting data races using transactional memory. In *Symposium on Parallelism in algorithms and architectures (SPAA '08).* pages 104-106.