Brief Announcement: Efficient Access History for Race Detection

Brief Announcement: Efficient Access History for Race Detection

Yifan Xu
Washington University in St. Louis
St. Louis, MO, USA
xuyifan@wustl.edu

Anchengcheng Zhou
Washington University in St. Louis
St. Louis, MO, USA
ann.zhou@wustl.edu

Grace Q. Yin
Massachusetts Institute of Technology
Cambridge, MA, USA
graceyin@mit.edu

Kunal Agrawal
Washington University in St. Louis
St. Louis, MO, USA
kunal@wustl.edu

I-Ting Angelina Lee
Washington University in St. Louis
St. Louis, MO, USA
angelee@wustl.edu

Tao B. Schardl
Massachusetts Institute of Technology
Cambridge, MA, USA
neboat@mit.edu

Abstract

While there has been extensive research on race-detection algorithms for task-parallel programs, most of this research has focused on optimizing a particular component, namely, ***reachability analysis***, which checks whether two instructions are logically in parallel. Little attention has been paid to the other important component, the ***access history***, which stores all memory locations previous instructions have accessed. In theory, the access-history component adds no asymptotic overhead; however, in practice, it is often the most expensive component of race detection since it is queried and (possibly) updated at each memory access. We optimize this component based on the observation that, typically, strands within parallel programs access contiguous blocks of memory. Therefore, instead of maintaining the access history at the granularity of individual memory locations, we maintain it at the granularity of these (varying size) intervals. To enable this access history, we propose (1) compiler and runtime mechanisms that allow us to efficiently collect these intervals and (2) a tree-based access-history data structure that allows updates and queries at interval granularity. The resulting tool can race-detect fork-join code with amortized constant overhead, assuming the number of intervals is small compared to the total work of the computation.

CCS Concepts

• **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Software testing and debugging**.

Keywords

determinacy race; race detection; shadow memory; access history; treap

SPAA '21, July 6–8, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8070-6/21/07.
https://doi.org/10.1145/3409964.3461825

ACM Reference Format:
Yifan Xu, Anchengcheng Zhou, Grace Q. Yin, Kunal Agrawal, I-Ting Angelina Lee, and Tao B. Schardl. 2021. Brief Announcement: Efficient Access History for Race Detection. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3409964.3461825

1 Introduction

A ***determinacy race*** [3] (or a ***general race*** [7]), occurs when two or more logically parallel instructions access the same memory location in a ***conflicting*** way, i.e., at least one of the accesses is a write. In the context of the task-parallel programming, determinacy races are often considered bugs since they can lead to nondeterministic behaviors.

Researchers have proposed several algorithms for detecting determinacy races in task-parallel code [1–6, 8, 10, 12–15]. Most of this work focuses on detecting races "on the fly" as program executes a particular input. These algorithms provide the strong guarantee that the race detector has no false positives and, if the race detector does not find a race, then the program has no races for that input. Such on-the-fly race detectors consist of two important components: (1) a ***reachability-analysis*** component that determines whether two ***strands*** — sequences of instructions containing no parallel control — are logically in parallel with each other, and (2) an ***access-history*** (also called shadow-memory) component that records (possibly a subset of) strands that have accessed a given memory location in the past. When a strand s accesses a particular memory location x, the race detector first queries the access history to find prior strands that have accessed x in a conflicting way. Next, the race detector queries the reachability component to determine if any of the strands with conflicting accesses is logically in parallel with the current strand s. If so, a race is reported. If not, the access history is updated with this new strand s so future strands can detect races.

The prior work on race detection has primarily focused on designing data structures and/or runtime mechanisms for maintaining the reachability component in a provably and practically efficient manner. In contrast, the access history has received little attention. Most prior race detectors maintain an access history by using an optimized hashmap to maintain the mapping from each memory address to previous accesses, which allows for (amortized) constant-time insertions and queries from the access history. In

practice, however, the management of access history often incurs much higher overhead than the reachability component does.

	# accesses $\times 10^6$		# intervals $\times 10^6$	
	read	*write*	*read*	*write*
chol	1466.0	671.2	2.1	0.7
fft	2013.9	1400.9	325.4	16.3
heat	5274.3	1053.8	2.2	1.0
mmul	17712.5	536.9	33.6	8.4
sort	693.7	535.1	1.3	0.2
stra	3173.5	342.0	2.1	0.8
straz	3814.0	216.4	4.5	1.7

Figure 1: Number of memory locations and intervals accessed, on the order of millions.

To that end, we propose mechanisms to speed up sequential race detectors for task-parallel code, focusing on optimizing the access history component. The key observation is as follows: For many task-parallel programs, a single strand typically performs many accesses to contiguous memory locations. We shall refer to a range of contiguous memory accessed by the same strand as an ***interval***.

Figure 1 shows the number of distinct (four-byte) memory words read/written and the number of intervals read/written for the tested benchmarks indicating that the number of intervals can be several magnitudes smaller than the number of memory words. If we manage the access history at the granularity of intervals instead of memory words, we can reduce both the time overhead and memory footprint of the access history.

Given this observation, we optimize the access history as follows. Instead of checking races at every memory access, we wait until the end of a strand and check for races on all accesses performed by the strand at this point, at the interval granularity, thereby reducing the number of queries and updates to the access history.

We propose two advances to enable such an optimization. First, we utilize compiler and runtime coalescing to produce intervals accessed by a given strand. Second, we propose an interval-based data structure to represent the access history in a more compact fashion that allows for efficient updates and queries of intervals. Specifically, given an interval to insert (or query), our proposed data structure can find all overlapping intervals already in the data structure efficiently.

Combining these advances, our race detector provides a provably efficient execution-time bound and incurs substantially lower overhead compared to a race detector that utilizes a traditional hashmap-based access history. Given a computation with T_1 work — the time it takes to execute the computation on one processor — our race detector runs in $O(T_1 + n \lg n)$ time, where n is the number of intervals accessed by the program. If n is small compared to T_1, which is typically the case, then our race detector can race-detect the computation in $O(T_1)$ time, incurring amortized constant overhead. Empirically, our evaluation shows that the optimization reduces the race detection overhead by 76%. In this brief announcement, we summarize these advances and present the empirical evaluation.

2 Overview of Our Design

This section overviews the design of the interval-based race detector. We describe compile-time and runtime coalescing to identify intervals accessed by the program. We overview the interval-based access-history data structure that maintains the access history at interval granularity.

Compile-Time and Runtime Coalescing

We check for races at the end of a strand on all accesses performed by the strand instead of checking races at every memory access. This strategy allows us to perform ***temporal*** and ***spatial*** coalescing. Temporal coalescing removes duplicate accesses: If the strand accesses the same memory location again and again, we only check for races and record this access once at the end of the strand, thereby reducing the number of queries to the access-history and reachability data structures. Spatial coalescing combines contiguous memory accesses within a strand into intervals and invokes the access-history and reachability data structures at the interval granularity.

Our race detector performs coalescing at both compile time and runtime. Some spatial coalescing occurs at compile time when the compiler can statically detect that the memory accesses within a strand are contiguous. Doing so allows the race detector to lower the instrumentation overhead, since instrumentation (i.e., invocations of race-detector operations) occurs at the granularity of intervals as opposed to at every memory access. The compile-time coalescing is necessarily conservative, however, and may miss coalescing opportunities. Our detector checks for additional opportunities for coalescing at runtime. Collectively, compile-time and runtime coalescing allows us to exploit spatial and temporal locality that exist in the code to reduce both instrumentation overhead and calls to reachability and access-history data structures.

Interval-Based Access-History Data Structure

Instead of storing accesses at word granularity, we store them as intervals, i.e., a start address and an end address. Doing so allows the access history to be represented in a more compact fashion, but we need a data structure that allows for efficient updates and queries of intervals. Specifically, given an interval to insert (or query), the we must find all overlapping intervals already in the data structure efficiently.

We use a balanced binary search tree data structure to maintain the access history. (Our implementation uses treaps [9, 11], but any balanced binary search tree would work.) Our construction differs from normal interval trees since it enforces an additional ***non-overlapping*** property, that is, no two intervals within the tree overlap. In particular, the cost of inserting into and querying our data structure for an interval x is $O(h + k)$ where h is the height of the tree and k is the number of intervals that overlap with x. By maintaining a balanced binary search tree such as a treap, our insert and query costs are bounded by $O(\lg n + k)$ (with high probability), where n is the number of intervals in the treap when x is inserted.

3 Empirical Evaluation

We have empirically evaluated our race detector and the impact of the optimizations across seven standard task-parallel benchmarks: Cholesky decomposition (`chol`), parallel merge sort (`sort`), fast-fourier transform (`fft`), heat diffusion simulation on a 2D grid (`heat`), matrix multiplication (`mmul`), and two versions of Strassen's algorithm for matrix multiplication (`stra` and `straz`, which differ in that `straz` uses a Morton Z-layout).

	base	*vanilla*		*compiler*		*comp+rts*		*treap*	
chol	0.61	84.66	(138.79×)	82.87	(135.85×)	26.73	(43.82×)	19.43	(31.85×)
fft	13.55	488.19	(36.03×)	368.76	(27.21×)	304.92	(22.50×)	539.42	(39.81×)
heat	4.36	367.24	(84.23×)	326.03	(74.78×)	144.43	(33.13×)	23.42	(5.37×)
mmul	8.11	355.66	(43.85×)	345.08	(42.55×)	218.78	(26.98×)	223.66	(27.58×)
sort	3.39	72.27	(21.32×)	69.39	(20.47×)	40.61	(11.98×)	15.90	(4.69×)
stra	1.49	423.43	(284.18×)	414.52	(278.20×)	96.30	(64.63×)	38.57	(25.89×)
straz	1.54	244.54	(158.79×)	244.36	(158.68×)	100.15	(65.03×)	52.16	(33.87×)

Figure 2: Execution times (in seconds) and overheads of different versions of the race detector compared to the baseline (i.e., no race detection).

We run the benchmarks with four versions of the race detector to tease out the impact of each optimization: (1) ***vanilla*** employs a hashmap, implemented as an optimized two-level page-table, to manage the access history with no coalescing; (2) ***compiler*** introduces compile-time coalescing and uses the same hashmap to manage the access history as in vanilla; (3) ***comp+rts*** includes both compile-time and runtime coalescing but still uses the same hashmap to manage the access history; and (4) ***treap*** includes both coalescing and uses the treap data structure to manage the access history.

These different versions allow us to gauge the impact of each optimization. By comparing vanilla and compiler, we gauge how much instrumentation overhead is reduced. By comparing compiler and comp+rts, we gauge how much overhead full coalescing (i.e., including runtime coalescing) reduces. By comparing comp+rts and treap, we measure the impact of using a treap instead of a hashmap, which incurs higher overhead per operation but reaps the full benefit of coalescing.

Figure 2 shows the race-detection overhead of each version of the detector compared to the ***baseline*** execution time, which performs no race detection. For most benchmarks, each additional optimization brings some benefit to the overhead reduction, leading to the final result, where treap incurs on average (geometric mean) of 18.61× overhead, much less than that of vanilla, 78.13×. The only exceptions are `mmul` and `fft`, whose overheads increase slightly from comp+rts to treap. This increase is because these two benchmarks have both more intervals and smaller interval sizes, while treap really shines when the interval sizes are large.

4 Concluding Remarks

We have presented an optimization of the access history to speed up a sequential race detector for task-parallel programs. One of the future directions is to investigate how to race detect in parallel using our optimized access history. The main problem with parallelizing the access history is to handle concurrent accesses to the binary search tree data structure while still maintaining efficiency, both theoretically and in practice, which seems challenging in the face of possible contention.

Acknowledgments

This work was supported in part by the National Science Foundation under grant numbers CCF-1733873, CCF-1910568, CCF-1943456, CCF-1533644, CCF-1725647 and CCF-1439062; and by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New Orleans, Louisiana.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

[3] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–11.

[4] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

[5] Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[6] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

[7] Robert H. B. Netzer and Barton P. Miller. 1992. What are Race Conditions? *ACM Letters on Programming Languages and Systems* 1, 1 (March 1992), 74–88.

[8] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

[9] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. In *ALGORITHMICA*. 540–545.

[10] Rishi Surendran and Vivek Sarkar. 2016. *Dynamic Determinacy Race Detection for Task Parallelism with Futures*. Springer International Publishing, Cham, 368–385.

[11] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. 2018. Zip Trees. arXiv:1806.06726 [cs.DS]

[12] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Asilomar State Beach, CA, USA, 83–94.

[13] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2019. Efficient Race Detection with Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, Washington, District of Columbia, 340–354.

[14] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-dimensional Dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vienna, Austria, 368–380.

[15] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel Determinacy Race Detection for Futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. ACM, San Diego, California, 217–231.