Brief Announcement: Performance Anomalies in Concurrent Data Structure Microbenchmarks

Rosina F. Kharal
University of Waterloo, Canada

Trevor Brown
University of Waterloo, Canada

Abstract

Recent decades have witnessed a surge in the development of concurrent data structures with an increasing interest in data structures implementing concurrent sets (CSets). Microbenchmarking tools are frequently utilized to evaluate and compare performance differences across concurrent data structures. The underlying structure and design of the microbenchmarks themselves can play a hidden but influential role in performance results. However, the impact of microbenchmark design has not been well investigated. In this work, we illustrate instances where concurrent data structure performance results reported by a microbenchmark can vary 10-100x depending on the microbenchmark implementation details. We investigate factors leading to performance variance across three popular microbenchmarks and outline cases in which flawed microbenchmark design can lead to an inversion of performance results between two concurrent data structure implementations. We further derive a prescriptive approach for best practices in the design and utilization of concurrent data structure microbenchmarks.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies

Keywords and phrases concurrent microbenchmarks, concurrent data structures, high performance simulations, PRNGs

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.45

Related Version *Full Version*: `https://arxiv.org/abs/2208.08469`

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Program Grant: RGPIN-2019-04227, and the University of Waterloo.

1 Introduction

An extensive variety of concurrent data structures have appeared over the past decade, with a particular focus on data structures implementing concurrent sets (CSets). A CSet is an abstract data type (ADT) which stores keys and provides three primary operations on keys: search, insert, and delete. Insert and delete operations modify the CSet and are called *update* operations. There are numerous concurrent data structures that can be used to implement CSets, including trees, skip-lists, and linked-lists. Microbenchmarks are commonly used to evaluate the performance of CSet data structures, essentially performing a stress test on the CSet across varying search/update workloads and thread counts. A typical microbenchmark runs an experimental loop bombarding the CSet with randomized operations performed by threads until the duration of the experiment expires. *Throughput*, number of operations performed by a CSet, is a key performance metric.

Multiple microbenchmarks exist to support CSet research. While CSet implementations have been well studied [1-3], the popular microbenchmarks used to evaluate them have not been scrutinized to the same degree. Microbenchmarking idiosyncrasies exist that can significantly distort performance results across data structures.

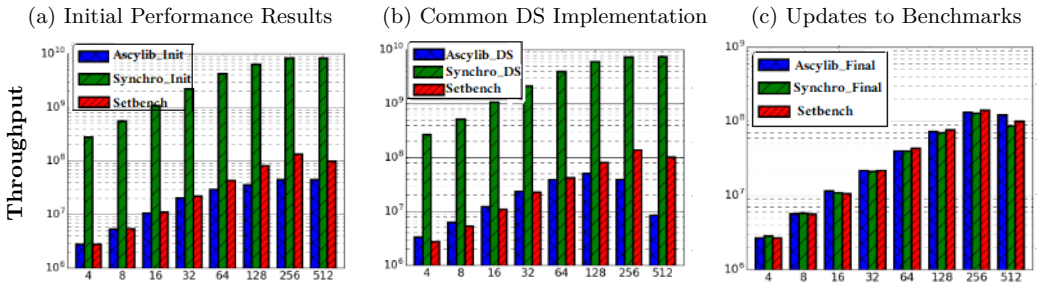

Figure 1 Throughput results across three microbenchmarks, Ascylib, Synchrobench and Setbench executing on a 256-core system testing the lock-free BST [5]. Results displayed on a logarithmic y-axis. Figure (a) results from unmodified microbenchmarks (as written by their authors). Figure (b) equalize the data structure (DS) implementations only. Ascylib_DS and Synchro_DS are updated with the imported lock-free BST implementation from Setbench. Figure (c) results for modified versions of Synchrobench and Ascylib correcting for pitfalls in microbenchmark design.

2 Microbenchmark Idiosyncrasies

When testing a CSet implementation on three different microbenchmarks with identical parameters, one would expect to observe similar performance results within a reasonable margin of error. However, we found 10-100x performance differences on the same CSet data structure tested across the Setbench [2], Ascylib [3], and Synchrobench [4] microbenchmarks. These microbenchmarks are often employed for evaluation of high performance CSets.

In Figure 1(a) we observe a range of varying performance results on the popular lock-free BST by Natarajan et al. [5] across the three microbenchmarks displayed using a logarithmic y-axis in order to capture wide performance gaps on a single scale. We performed a systematic review of the design intricacies within each microbenchmark. Our investigations led to the discovery that seemingly minor differences in the architecture and experimental design of a microbenchmark can cause a 10-100x performance boost erroneously indicating high performance of the data structure when the underlying cause is the microbenchmark itself. We perform successive updates to two of the microbenchmarks adjusting where errors or discrepancies were discovered until performance is approximately equalized (Figure 1(c)). In previous work by Arbel et al. [1], it was noted that *data structure implementation* differences can account for varying performance results in microbenchmark experimentation. We adjusted each microbenchmark to use a common lock-free BST implementation (Figure 1(b)) and still observed large performance gaps. Adjustments to the microbenchmark design were necessary to equalize results.

During our investigations, we found the following factors have the greatest impact on microbenchmark performance: (1) Repeated benchmark code is prone to error. In Synchrobench where the algorithm running performance experiments is duplicated for each data structure, errors in the algorithm led Synchrobench results to exceed other microbenchmarks by 100x. The microbenchmark testing algorithm should exist in one centralized location and provide easy adaptation to new data structures. (2) Pseudo random number generators (PRNGs) are typically used to generate random keys and operations. The way that a PRNG is integrated into the microbenchmark can play a significant role in experimental results. We investigate in detail in the full paper. (3) Microbenchmarks use a variety of techniques for splitting the update rate between insert and delete operations. For example, alternating between update operations, or flipping a biased coin to decide if

the next update will be an insert or delete. This has a non-trivial impact on performance. Recommended practice is to randomly distribute update operations between insert and delete operations using per thread PRNGs. (4) Synchrobench introduced a setting to enforce a specified rate of *effective* updates. An update operation is considered *effective* if it successfully modifies the CSet. Enforcing effective updates is problematic because, for example, in an almost full data structure, to perform an effective insert, one may need to repeatedly attempt to insert many random keys until one succeeds. The attempts leading up to the successful insert are essentially searches (which are faster than updates), and they are counted towards throughput, inflating performance. (5) Memory reclamation can play an influential role in performance results. The Ascylib microbenchmark memory reclamation algorithm is leaking memory at higher thread counts. This may render some microbenchmark experiments impracticable due to growth in memory usage. (6) Our recommended best practice for microbenchmark design includes strategies to detect and mitigate errors in the microbenchmark. We certainly recommend a checksum validation in microbenchmark experiments: the sum of keys inserted minus the sum of keys deleted into the CSet *during* an experiment should equal the final sum of keys contained in the CSet *following* the experiment. In our work, adding checksum validation assisted in discovering microbenchmark and data structure implementation errors. (7) We recommend a data structure prefilling step that includes randomized insert and delete operations. This generates a prefilled CSet data structure with a more realistic configuration, as opposed to a CSet that is produced by insert-only operations.

2.1 PRNG Usage in Concurrent Microbenchmarks

PRNGs are heavily relied upon in microbenchmarks to generate randomized keys and/or select randomized operations on a CSet. Researchers may be tempted to pre-generate a large array of random numbers prior to an experiment, thereby moving the cost of generating high quality randomness into the unmeasured setup phase of the experiment. We found this approach to be counter productive due to the impact on processor caching. In our full paper, we examine various PRNG methodologies and make practical recommendations for generating fast, high quality randomness, using hardware support where available.

References

1 Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference*, 2018.

2 Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

3 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

4 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015.

5 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014.