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 the 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 set of recommendations for best practices in the design and usage of concurrent data structure microbenchmarks and explore advanced features in the Setbench microbenchmark.

1 Introduction

The execution efficiency of highly parallelizable data structures for concurrent access has received significant attention over the past decade. An extensive variety of data structures have appeared, with a particular focus on data structures implementing concurrent sets (CSets) [9, 11, 21, 37, 51]. CSets have applications in many areas including distributed systems, database design, and multicore computing. 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. A CSet data structure refers to the implementation of a CSet. 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 platforms for microbenchmarking exist to support CSet research. The accuracy and reliability of performance results generated from microbenchmark experiments is fundamental to concurrent data structure research. Researchers must be able to assess the performance benefit vs loss of varying concurrent implementation strategies and their overall impact on performance. Microbenchmarks are also an important tool for comparative performance analysis between different implementations of CSets. While CSet implementations have been well studied [3, 9, 11, 37], the popular microbenchmarks used to evaluate them have not been scrutinized to the same degree. Microbenchmarking idiosyncrasies exist that can significantly distort performance results. The goal of our work is to better understand the role of microbenchmark design in performance results and attempt to minimize factors present within the microbenchmark that misrepresent performance.

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 Ascylib [11], Setbench [8], and Synchrobench [14] 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. [37] 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. We found discrepancies in microbenchmark implementation leading to CSet data structures underperforming in one microbenchmark and over performing in another. We found one popular microbenchmark duplicating the entire benchmark code for each CSet implementation. This renders the code highly prone to errors related to updates or modifications to the benchmark, and may inevitably result in reporting skewed experimental results. 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 performed successive updates to two of the microbenchmarks, adjusting where errors or discrepancies were discovered, until performance is approximately equalized (Figure 1(b)). In this work, we discuss the primary factors leading to microbenchmark performance variance and provide a set of recommended best practices for microbenchmark experiment design.

Microbenchmarks rely heavily on pseudo random number generators (PRNGs) to generate randomized keys and/or select randomized operations on a CSet. In this work, in addition to investigating microbenchmark design differences, we delve into a deeper investigation of PRNG usage in microbenchmarks. Deleterious interactions between a PRNG and a microbenchmark that uses it can go undetected for years. We present examples where (mis)use of PRNGs can cause substantial performance anomalies and generate misleading results. We illustrate how using a problematic PRNG can lead to an inversion of throughput results on pairs of CSet data structures. We discuss the pitfalls of common PRNG usage in microbenchmark experiments. Our experiments are limited to concurrent tree data structures that were present in the three microbenchmarks in our study. We believe the lessons learned in our investigations related to microbenchmark design apply broadly to the experimental process and are not limited to specific CSet implementations. We leave the investigation of microbenchmark performance on varying CSet implementations for future work.
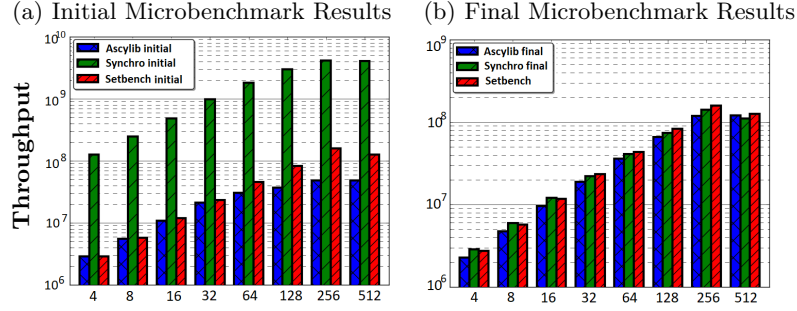
(a) Initial Microbenchmark Results   (b) Final Microbenchmark Results

◼ **Figure 1** Throughput results across three microbenchmarks, Ascylib, Synchrobench and Setbench executing on a 256-core system testing the lock-free BST [37]. Thread count is displayed on the x-axis, y-axis is a logarithmic scale. Figure (a) results from unmodified microbenchmarks (as written by their authors). Figure (b) results for modified versions of Synchrobench and Ascylib correcting for pitfalls in microbenchmark design.

Contributions: In this work, we perform the first rigorous comparative analysis of three widely used microbenchmarks for CSet performance evaluation. We present an overview of related work in Section 3. The three microbenchmarks evaluated in our work are described in detail in Section 4. We investigate the source of performance differences reported by each microbenchmark when testing equivalent tree-based data structures in Sections 4 and 5. We study the role of memory reclamation and its impact on CSet performance in Section 5. In Section 6 we investigate commonly employed methods of fast random number generation and the pitfalls of each. We describe a set of derived recommendations for best practice in microbenchmark design at the ends of Sections 4, 5 and 6. Additional recommendations for further improvements in microbenchmark design are discussed in Section 7. Advanced features of the Setbench microbenchmark are described in Section 7.2 followed by concluding remarks in Section 8. In the next section, we begin with a background on the principles of microbenchmark design with concrete examples from the Setbench microbenchmark.

2 Background

In this work we test three concurrent synthetic microbenchmarks, Setbench [8], Ascylib [11], and Synchrobench [14] for high speed CSet analysis. The key properties of each microbenchmark are summarized in Table 1. The microbenchmarks report the total operations per second performed on the CSet by n threads based on a specified workload. In particular, we study data structures that implement *sets* of keys and provide operations to *search*, *insert* or *delete* a key. The microbenchmarks allow users to specify parameters that include the number of threads (`t`), the experiment duration (`d`), the update rate (`u`), and the key range (`r`) contained within the set (i.e. [1, 200,000]).

Evaluating performance operations on an initially empty data structure will generate results that are misleading and not representative of average performance on a non-empty CSet. Therefore, microbenchmarks typically prefill the CSet before the experiment begins to contain a subset of keys less than or equal to the total range. The prefill size may be specified by the user as the *initial* (`i`) prefill amount, or the microbenchmark may decide the prefill size using its own algorithm. For a duration `d`, a microbenchmark runs in an experiment loop where `n` threads are assigned keys from the specified key range based on a random uniform distribution, though other distributions are also possible. Threads perform a combination of search or update operations based on experimental parameters. For example, if the specified update rate (`u`) is *10%*, the search rate is *90%*. The microbenchmark either randomly splits

Benchmark Properties	Prefill Size	Threads used to Prefill	Prefill Ops	PRNG for key generation	PRNG for update choice	
Ascylib	half-full	single/n	inserts	✓	✓	
Setbench	steady state[1]	n	ins/dels	✓	✓	
Synchrobench	half-full	single	inserts	✓		
Benchmark Properties	Centralized Test loop	Test file per DS	Range Queries Available	Effective Upd Option	Thread Pinning	Unique Features
Ascylib	✓	✓		✓	✓	*2
Setbench	✓		✓		✓	*3
Synchrobench		✓		✓	✓	*4

Table 1 Summary of properties within each microbenchmark. 1: steady state depends on experiment parameters. Setbench allows varying insert and delete ratios; this determines what the data structure will fill to in steady state. (*2: performance tracking. *3: statistics tracking, performance recording, automated graph generation, range query searches, varying distributions of keys/operations are possible, independent insert and delete rates possible. *4: track effective updates, alternating updates possible.)

the update rate across insert and delete operations, or employs its own algorithm to attempt to divide insert and delete operations equally. Microbenchmarks may offer the ability to specify independent insert and delete rates. This is discussed further in Section 4.3.

2.1   Microbenchmark Setup

We use the *Setbench* microbenchmark as a case study to explain some underlying design principles in concurrent microbenchmarks. A typical Setbench experiment involves **n** threads accessing a CSet for a fixed duration. During this time, each thread performs search or update operations that are chosen according to a specified probability distribution on keys randomly drawn from another probability distribution over a *fixed* key range. For example, threads might choose an *operation* to perform *uniformly* (1/3rd insert, 1/3rd delete and 1/3rd search operations), and then choose a *key* to insert, delete or search for from a *Zipfian* distribution. Each thread has a PRNG object, and the same object is used to select a random operation and generate random keys.

To ensure that an experiment measures performance as it would in the steady state (after the experiment has been running for a sufficiently long time), performance measurements are not taken until the data structure is warmed up by performing insertions and deletions until the CSet converges to approximately its steady state. This step is called prefilling. If the key range is $[1, 10^6]$, and threads do 50% insertions and 50% deletions, then the size of the CSet in steady state will be approximately 500,000 (half full). Different microbenchmarks will employ varying methods of prefilling the data structure prior to experimental evaluation. This is discussed in the next section. In this work, we evaluate performance results across three microbenchmarks and analyse the underlying subtleties in microbenchmark design which lead to varying performance on equivalent CSet data structures. An example of this is illustrated in Figure 1(a) where microbenchmark experiments are performed on the lock-free BST by Natarajan et al. [37]. The initial comparative throughput results are very different. We apply successive modifications to the microbenchmarks where required in an attempt to minimize large performance gaps. This process is outlined step-by-step in Section 4.

Experiments performed in this work execute on a dual socket, AMD EPYC 7662 processor with 256 logical cores and 256 GB of RAM. DRAM is equally divided across two NUMA nodes. We use a scalable memory allocator, *jemalloc* [13], to prevent memory allocation bottlenecks. Each microbenchmark employs its own PRNG for generating random numbers

during the experiment loop. This is discussed further in Section 4. We test key ranges between 2000 and 2 million keys using thread counts of up to 512, which gives an indication of the effects of oversubscribing the cores. All figures in Sections 4 and 5 in this work are displayed using a logarithmic y-axis in order to allow visual comparison between algorithms with large differences.

3 Related Work

There are previous efforts in the literature to better understand the underlying structure and design of benchmarks used to evaluate concurrent applications. In their work on the comparative evaluation of transactional memory (TM) systems, Nguyen et al. discuss the unexpected low performance results observed when using benchmarks to evaluate various hardware transactional memory (HTM) and software transactional memory (STM) systems [38]. They argue that the observed limited performance results are a consequence of the programming model and data structure design used within the benchmarks and are not necessarily indicative of true performance results of the TM systems themselves. In related work by Ruan et al. [41], the STAMP benchmark suite [33] used for evaluating transactional memory was identified as being out-of-date. The authors present several suggested modifications to the benchmark suite to boost the reliability of performance results for more accurate TM evaluation. McSherry et al. discuss the COST (Configuration that Outperforms a Single Thread) [32] associated with scaling applications to support multi-threaded execution, and the need to measure performance gains without rewarding the substantial overhead costs of parallelization.

Recent microbenchmarks exist that were not tested in our work, such as the Synch framework [21] for concurrent data structures evaluation. We leave this for future study. There has been some prior investigation of microbenchmark design for concurrent data structure performance evaluation. Microbenchmark experiments executing a search-only workload on CSets have been tested in previous work by Arbel et al. [3]. They considered differences in concurrent tree implementations of CSets and their impact on performance. It was discovered that subtle differences in concurrent tree implementations can play a pivotal role in microbenchmark performance results. Our work concentrates on the impact of *microbenchmark implementation differences* on CSet data structure performance for workloads that include updates. Mytkowicz et al. in their work, "*Producing Wrong Data Without Doing Anything Obviously Wrong!*" [34], illustrate how subtle changes to an experiment's setup can lead to enormous performance differences and ultimately to incorrect conclusions. Tim Harris' presentation, "*Benchmarking Concurrent Data Structures*" [17], is closely related to our work. Harris explains the need for sound experimental methodology in performance evaluation tools and discusses some noted pitfalls in the Synchrobench microbenchmark in [18]. Important considerations in the design of good concurrent data structure experiments have been previously discussed in seminars presented by Trevor Brown [7]. Brown discusses subtle aspects of microbenchmark testing configurations and underlying memory and thread distributions that can play a crucial role in performance results. This is discussed further in Section 7.1. In our work, we provide an investigative approach to microbenchmark design by comparing design strategies employed in three popular microbenchmarks.

4 Comparison Lock-Free Binary Search Tree

As mentioned above, a key performance indicator in the evaluation of CSet data structure performance is the total number of operations per second (throughput). This is computed by
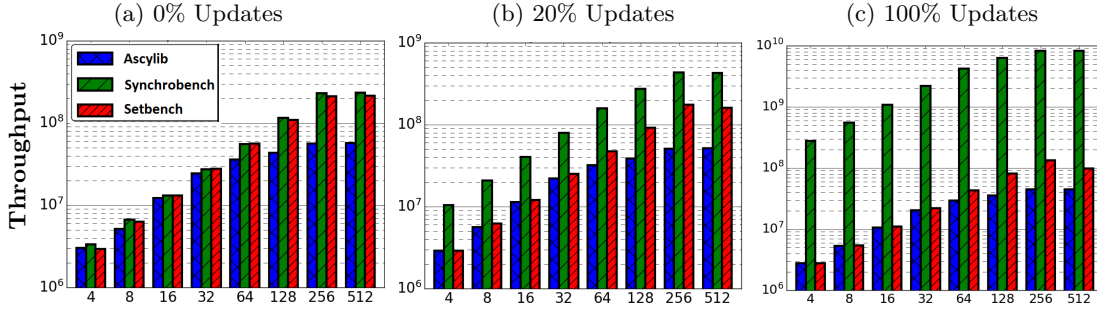
Figure 2 Initial throughput comparisons across three unmodified microbenchmarks testing the lock-free BST [37] on varying update rates.

summing the total number of operations performed per thread and dividing by the duration of the experiment. A key indicator of memory reclamation efficiency is the *maximum resident memory* occupied in RAM by the microbenchmark program during the duration of the experiment. We use this measure to evaluate the memory reclamation capabilities of each microbenchmark in Section 5. We perform a comparative study on the lock-free BST data structure by Natarajan et al. [37] which implements a CSet. The lock-free BST stores keys in leaf nodes; internal nodes contain repeated leaf values to provide direction for searches. Not all microbenchmarks implement the lock-free BST with memory reclamation. Therefore, our initial comparisons turn memory reclamation off. Table 1 describes the properties of each microbenchmark tested in this work.

4.1 Synchrobench

The Synchrobench synthetic microbenchmark allows the evaluation of popular C++ and Java-based CSet implementations. Synchrobench is a popular microbenchmark used for performance evaluation of CSet data structures [48, 5, 11, 14, 16, 47, 50]. Synchrobench allows users to specify an *alternate option* (-A) or an *effective option* (-f) as input parameters to the microbenchmark. The -A option can be used to force threads to alternate between a key being inserted and the same key being deleted. The -f option sets total throughput calculations to count failed update operations as search operations and not as update operations. We do not use either of these options in our experiments. Synchrobench performs single threaded prefilling with insert-only operations. Each data structure directory contains a test file (`test.c`) that runs the basic test loop of the microbenchmark, performing a timed search/update workload on the CSet. In our evaluation of Synchrobench, we found the repetition of the *test.c* file in each data structure directory. This is discussed further in Section 4.5.1. Synchrobench allows users to specify a single update rate that is divided between insert and delete operations, though the division is not necessarily equal. This is discussed further in Section 4.5.4.

4.2 Ascylib

The Ascylib synthetic microbenchmark is another microbenchmark used to compare performance of concurrent data structures [3, 11, 12, 14, 23, 40, 54]. Ascylib also performs an initial prefilling step using single threaded insert-only operations. However, Ascylib has a setting to allow multi-threaded prefilling using insert-only operations. The range and initial values are updated to the closest power of two. This is a necessary condition for the Ascylib test algorithm to generate randomly distributed keys. The experiment testing algorithm

Synchro Version	Synchro	Synchro1	Synchro2	Synchro3	Synchro4	Synchro5	Synchro'
insert & delete		✓	✓	✓	✓	✓	
random seeds/thread			✓	✓	✓	✓	
MM3 RNG				✓	✓	✓	
randomized updates					✓	✓	
common DS impl						✓	✓
Ascylib Version	Ascylib	Ascylib1	Ascylib2	Ascylib3	Ascylib'		
disable thread pin		✓	✓	✓			
MM3 RNG			✓	✓			
common DS impl				✓	✓		

Table 2 Summary of Synchrobench and Ascylib modifications tested in this work. Ascylib required fewer changes, Setbench did not require any modifications for the comparative experiments performed in Section 4. The original installed implementations are labelled Synchro and Ascylib without a version number. Ascylib' and Synchro' are versions of each microbenchmark where only the lock-free BST implementation is modified (imported from Setbench).

(*test_simple.c*) is also repeated in each data structure directory. However, the main test loop is implemented in one common macro and is shared across each CSet data structure implemented in Ascylib. The update rate is randomly distributed among insert and delete operations and updates are not required to be effective. Ascylib allows additional user inputs to define profiling parameters which are not tested in this work. Additional properties of Ascylib can be seen in Table 1.

4.3 Setbench

The Setbench synthetic microbenchmark is another benchmarking tool employed in concurrent data structure literature [3, 8, 9, 10, 24, 42]. Setbench employs a directory structure per CSet implementation. However, each CSet utilizes a single experiment test loop via an adapter class which imports each specific CSet implementation into the main experimental algorithm. This allows a single point of update for the testing algorithm and avoids software update errors. Setbench allows specification of independent insert and delete rates. Setbench uses per thread PRNGs initialized with unique seeds. Although Setbench has multiple choices of PRNGs, we employed the *murmurhash3* (MM3) [1] PRNG for comparative microbenchmark experiments in this section of our work. Setbench employs multi-threaded prefilling using randomized insert and delete operations. The benefits of this are discussed in Section 4.8. We delve into further details regarding the Setbench microbenchmark in Section 7.2.

4.4 Throughput Comparisons

We test the initial installed implementations of the three aforementioned microbenchmarks in order to compare performance results on the lock-free BST data structure. To standardize experiments across the three microbenchmarks, we performed single threaded prefilling using insert-only operations to reach a start state where the data structure contains exactly half of the keys from the specified input range. Memory reclamation was turned off in all microbenchmarks. Attempted updates and effective updates are both counted towards the total operation throughput. We examine throughput results for experiments running for 20 seconds with update rates varying from 0% to 100% and a specified range of 2 million keys unless stated otherwise. Enforcing the range to a power of 2 is turned off in Ascylib experiments to match the other microbenchmarks. Initial results across the three microbenchmarks can be seen in Figure 2 where throughput values are displayed on
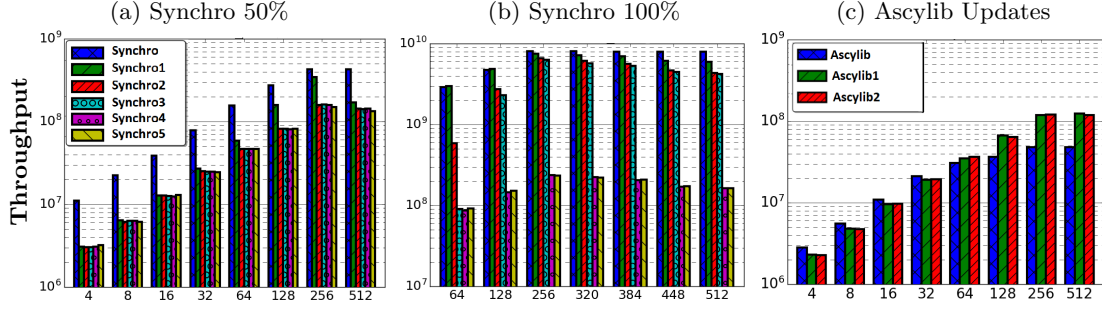
Figure 3 Throughput results for successive modifications to Synchrobench (a), (b) and successive modifications to Ascylib (c). Synchro1 to Synchro4 involve updates to the Synchrobench microbenchmark design. Synchro5 updates the data structure implementation to that of Setbench. Figure (a) uses a key range of 2 million. Figure (b) uses a key range of 20,000 keys and displays the impact of successive modifications to Synchrobench at a 100% update rate.

a logarithmic y-axis. We observe a range of varying performance results on the lock-free BST across the three microbenchmarks. In particular, across all experiments, Synchrobench throughput results are one to two orders of magnitude higher than Setbench or Ascylib. Ascylib results are notably lower than those of Setbench and Synchrobench. We also observe that Ascylib throughput results tend to plateau at about 128 threads and do not indicate growth as is expected and seen with Setbench and Synchrobench. We investigate further to understand the role of individual microbenchmark design on performance results.

4.5 Performance Factors: Synchrobench

Further investigation is required to understand the underlying causes of comparatively spiked performance results from the Synchrobench microbenchmark seen in Figure 2. In the following set of experiments we aim to equalize the performance results of Setbench and Synchrobench through various adjustments made to Synchrobench where errors or bugs were discovered. We modify the original installation of Synchrobench and title each updated version as `SynchroX`, where X is the adjustment number. With each successive modification, for both Ascylib and Synchrobench, all previous modifications are maintained unless stated otherwise. A summary of modifications performed in our work are listed in Table 2.

4.5.1 Missing Insertions

Synchrobench utilizes a file (`test.c`) in each data structure implementation in order to run the microbenchmark experiment loop. Each thread executes in the loop for the duration of the experiment, and all threads are joined prior to termination. Insert operations occur only following a successful delete operation which indicates success by setting a variable `last` to -1. This value is checked on the next update operation; if `last` is negative an insert operation will proceed. However, the `test.c` file in the lock-free BST directory contained a bug in which `last` was an unsigned type and could never take on negative values. As a result, all experiments on the lock-free BST were performing update operations comprised of deletions and never insertions. The data structure is initially prefilled to half of the specified range, but following prefilling, insert operations never take place due to this particular bug in the experiment loop. Delete-only update operations generate notably higher throughput results since the data structure becomes empty very quickly; essentially all operations reduce to searches as the duration of the experiment increases. Upon correction of this bug, throughput

(a) Results Compare Final   (b) Add DS Common   (c) Only DS Common
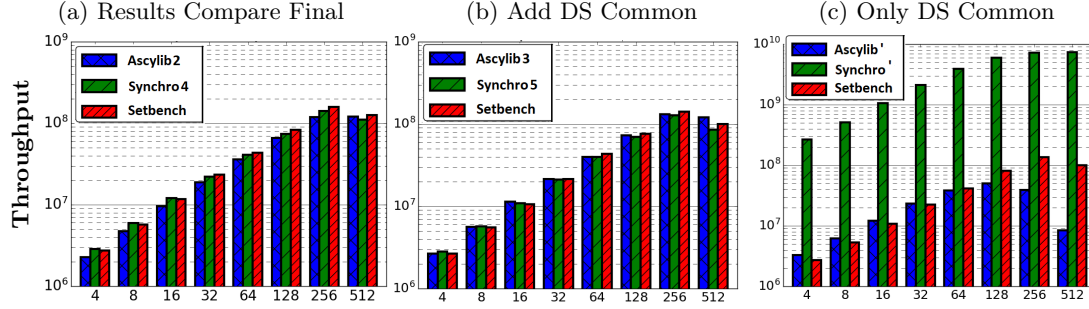
■ **Figure 4** (a) Throughput results display the final comparison across three microbenchmarks with all successive modifications (Synchro4 and Ascylib2). Figure (b) tests Synchro5 and Ascylib3 which maintain all microbenchmark changes and also update the data structure (DS) implementation to that of Setbench. Figure (c) tests Ascylib' and Synchro' which do not contain any modifications to the microbenchmarks and only equalize the DS implementation.

results lowered significantly. Performance results of progressive adjustments to Synchrobench are illustrated in Figure 3. This adjustment was the first of a series of modifications made to the original version of Synchrobench for the lock-free BST and is labelled `Synchro1` in the figure. There is a drop in throughput from the original installation of `Synchro` to `Synchro1`. We note that missing insert operations in the experiment loop was not a common occurrence in other data structure directories of Synchrobench.

4.5.2 Thread PRNG seeds

The test algorithm (`test.c`) for the lock-free BST data structure did not assign each thread a unique initialization seed for use in the PRNG employed to generate random keys. Having a PRNG initialized with the same seed per thread resulted in threads utilizing the same set of keys for search/update operations, resulting in an overall high throughput. As the duration of the experiment and the number of threads increase, updates are again essentially reduced to search operations due to other threads having previously completed the requested operation on the given key. Inserts fail because the key is already there, deletes fail because the key was removed by another thread. With `Synchro2` we correct this problem with the addition of randomly generated seeds to initialize each thread's PRNG. The impact of this update can be seen more prominently in Figure 3(b) where there is a drop in throughput with Synchro2 on 100% updates operating on a 20,000 key range. This is not so visible when the key range is much larger. At a key range of 2 million, the dominant overhead in operations is traversing a large tree; therefore, we see less variation in throughput from *Synchro2* to `Synchro5` in Figure 3(a). The probability of contention on the same set of keys is lower at 2 million keys, therefore, the impact of `Synchro2` is more prominent in smaller key ranges.

4.5.3 Standardized PRNG

As discussed earlier, Synchrobench utilizes a standard built in C++ PRNG, `rand()` to supply randomly generated keys. Setbench and Ascylib use XOR shifting based PRNGs. The Synchrobench microbenchmark is adapted to support the XOR-shift based PRNG employed in Setbench (MM3). This adjustment is labeled `Synchro3` in Figure 3. The adjustment does slightly reduce overall throughput as MM3 uses a slightly more complicated random number generation algorithm, using multiply and XOR-shifts, than what was previously employed in Synchrobench.

4.5.4   Effective Insert and Delete Operations

In attempt to equally distribute insert and delete operations across threads, Synchrobench uses an effective update strategy. Effective updates require threads to perform one type of update successfully before the other type of update is attempted. For example, a thread must perform and insert operation that successfully modifies the data structure before it can attempt a delete operation. This is considered an *effective* update, an approach we found to offer no tangible benefit and can be unforgiving of data structure specific bugs. Effective updates should not be confused with the -f (effective) option. The -f option in Synchrobench controls only how failed update operations will count towards total throughput, but an effective update strategy for insertions and deletions is used regardless.

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. Essentially, a number of search operations are inserted in between insert and delete operations, thereby inflating the total number of operations. The implementation of the lock-free BST in Synchrobench has a known concurrency bug contained in the original algorithm [3]; modified nodes are not always correctly updated in the tree. The requirement for effective updates in the experiment can generate results which erroneously indicate performance gains in the presence of errors in the implementation. The approach followed in Setbench is to randomize insert and delete operations using per thread PRNGs. This will generate more accurate performance results in spite of possible errors in the implementation. This adjustment is added to Synchrobench and is labelled `Synchro4`.

It may also be noted that a *checksum validation* step would prove beneficial in Synchrobench to catch data structure related concurrency bugs. A checksum validation verifies that 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. Incidentally, the implementation of the lock-free BST in Synchrobench was failing checksum validation. `Synchro4` is the final correction to the Synchrobench microbenchmark design. The data structure specific concurrency bug is updated in the next modification.

4.5.5   Equalizing the Lock-Free BST Implementation

The final update to Synchrobench is a modification of the data structure implementation and equalizing the three microbenchmarks to use the lock-free BST implementation provided in Setbench. The Setbench implementation corrects the concurrency bug and adds checksum validation, which does not exist in the other microbenchmarks. This adjustment is labeled `Synchro5`. We do not see a large difference in performance from `Synchro4` to `Synchro5` in Figure 3(b), which highlights the need for randomized insert and delete operations in concurrent microbenchmark experiments. By employing a randomized update operation assignment, we mitigate the impact of concurrency bugs on overall CSet data structure performance. We also assess an implementation of Synchrobench, `Synchro'`(Synchro prime), with the imported lock-free BST implementation from Setbench which does not include any modifications to the Synchrobench microbenchmark given in `Synchro1` to `Synchro4`. This comparison is given in Figure 4.

4.6   Performance Factors: Ascylib

The Ascylib microbenchmark test algorithm and underlying default settings lead to a few factors that impact performance results on the lock-free BST. Each successive modification to Ascylib is labeled `AscylibX`.

4.6.1   Thread Pinning

The Ascylib general installation enables thread pinning by default. With further investigation, we found that built-in thread pinning settings were under utilizing the 256 available cores during experimentation. Ascylib captures the underlying core and NUMA node count at compile time; we updated build settings to ensure the correct number of cores were detected. Although the Ascylib build displays that the correct number of cores have been detected, we found the Ascylib throughput results in Figure 2 were based on under 50% core utilization. The default settings were unable to utilize the full set of cores. To remove the underlying thread pinning settings, and disable thread pinning entirely, we recompiled with SET_CPU=0. This adjustment is labelled `Ascylib1`. Results for `Ascylib1` indicate full core utilization and improve performance in Figure 3(c). A user that is unaware of Ascylib's default setting may unknowingly generate misleading results. Rather than modifying the three benchmarks to perform identical thread pinning, we disabled thread pinning in all three for consistency. This is perhaps not ideal for microbenchmark experiments. Recommendations for thread pinning in microbenchmark experiments are discussed further in Section 7.

4.6.2   Standardized PRNG

As was the case with the Synchrobench microbenchmark, we use the same PRNG across all three microbenchmarks. Ascylib is updated to use the MM3 PRNG employed in Setbench and Synchrobench labelled `Ascylib2`. We do not see a significant change in performance between `Ascylib1` and `Ascylib2`, as the original PRNG algorithm used in Ascylib was also based on XOR-shifts [30].

4.6.3   Equalizing the Lock-Free BST Implementation

Last, for a comparison that evaluates a standard data structure implementation on each microbenchmark, we implement the lock-free BST implementation from Setbench into Ascylib. This is labelled `Ascylib3`. `Ascylib3` maintains all previous benchmark adjustments whereas `Ascylib'` only updates the common data structure implementation from Setbench into the original installation of Ascylib (Table 2).

4.7   Final Comparisons

The final comparative results following successive modifications to Ascylib and Synchrobench are given in Figure 4(a), which tests `Ascylib2`, `Synchro4` and the original Setbench implementation. These implementations use the built in data structures of each microbenchmark while adjusting for microbenchmark design differences in an attempt to equalize the throughput results. We have achieved throughput results that are fairly consistent across microbenchmarks. There are slight discrepancies in throughput results, but these are not nearly as drastic as the performance differences across the original implementations of Ascylib and Synchrobench in Figure 2. Additional final comparisons are provided in Figure 4(b) and (c), which also equalize the lock-free BST implementation across all microbenchmarks on a 100% update workload across 2 million keys. Figure 4(b) includes all microbenchmark modifications for both Synchrobench and Ascylib, whereas Figure 4(c) does not include any microbenchmark modifications from the original installed versions of Synchrobench and Ascylib. The results in Figure 4(c) illustrate the variations in throughput that occur on account of microbenchmark implementation differences. We see that once microbenchmark idiosyncrasies have been ironed out in (a) and (b), the performance results are much more

consistent. This highlights again the crucial role of microbenchmark design in the performance of CSet data structures.

4.8   Microbenchmark Design Considerations

In this section we investigated microbenchmark idiosyncrasies between three microbenchmarks. We performed successive modifications to two of the microbenchmarks to account for design differences. During our experiments, we discovered the following factors in microbenchmark design which lead to the greatest impact on 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) Microbenchmarks use a variety of techniques for splitting the update rate between insert and delete operations. Recommended practice is to randomly distribute update operations between inserts and deletes using per thread PRNGs. (3) Synchrobench introduced a setting to enforce effective updates. We note in Section 4.5.4, effective updates unnecessarily inflate throughput results and are not recommended. (4) 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. In our work, adding checksum validation assisted in discovering microbenchmark and data structure implementation errors.

Prefilling a CSet prior to running the microbenchmark experiment is also an important design consideration. Although experiments in this section used insert-only prefilling, we recommend against this for CSet microbenchmark experiments. (5) Data structure prefilling should occur through (a) randomized insert and delete operations, and (b) using the same **n** threads that will be employed during the measured portion of experiments. This will generate a more realistic configuration of a concurrent data structure in steady state as opposed to a data structure prefilled using single-threaded insert-only operations. Single-threaded prefilling will result in memory allocation specific to one thread's NUMA node. This will results in memory access latency for threads on different NUMA nodes during the measured portion of experiments. Using **n** threads to perform prefilling will disperse memory allocation across additional NUMA nodes. **N**-threaded prefilling with randomized insert and delete operations is used in Setbench as mentioned previously. We discuss additional considerations in microbenchmark design and provide further recommendations in Section 7. In the next section, we experiment with memory reclamation in microbenchmarks and evaluate its impact on performance.

5   Memory Reclamation

A key indicator of memory use efficiency of a program is the *maximum resident memory* occupied in RAM by the program during the duration of its execution. The lock-free BST as described by Natarajan et al. [37] does not provide a complete algorithm for memory reclamation during execution. The partial algorithm suggests removing an unbounded number of nodes that are nearby neighbours in the tree pending deletion. Any given thread may proceed to delete and free (unlink) **n** nodes that are in close proximity within the tree. However, the original implementation was leaking memory. Synchrobench does not implement any memory reclamation in its implementation, whereas Ascylib has an added option for garbage collection (GC). The authors of the lock-free BST suggest adding epoch based
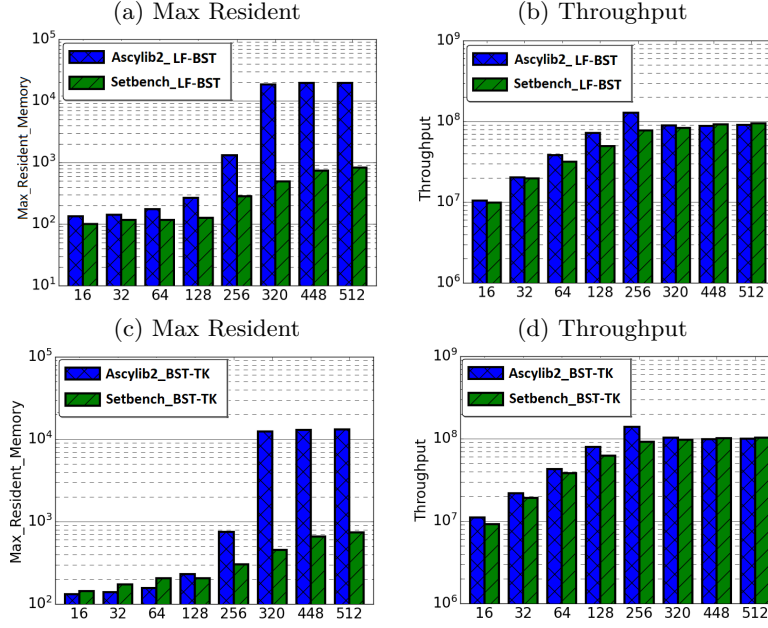
Figure 5 Maximum Resident Memory and Throughput results for Ascylib and Setbench on the lock-free BST ((a), (b)) and BST-TK ((c), (d)). Ascylib implementation contains microbenchmark updates contained in Ascylib2 (Section 4.2).

memory reclamation, but it was not so simple. The memory reclamation algorithm from the original work is updated in the Setbench implementation to correctly reclaim memory [3]. We first compare the memory reclamation implementations in Setbench and Ascylib by setting Ascylib's GC setting to true, and Setbench epoch based reclamation is turned on. We show comparative analysis of results across each microbenchmark in Figure 5. The Ascylib microbenchmark has been updated to `Ascylib2` in order to disable thread pinning and equalize the PRNG utilized in both microbenchmarks. We have ensured all 256 cores are being utilized by Ascylib. Figure 5(a) illustrates differences in each microbenchmark's ability to reclaim memory as the thread count increases and cores are oversubscribed. Ascylib's memory usage surpasses that of Setbench by over one order of magnitude, particularly as the thread count increases. Throughput results are relatively equal, however, the high maximum resident memory values may render Ascylib experiments unfeasible in some settings. We further consider microbenchmark comparisons on the equalized lock-free BST implementation with memory reclamation turned on. We discover similar performance discrepancies to those discussed in Section 4, although the data structure implementation and memory reclamation algorithms are identical across the three microbenchmarks. Performance results continue to show variance until microbenchmark idiosyncrasies are accounted for. Results for these experiments can be seen in Appendix A.1 in the full version of this work [22].

5.1 Setbench/Ascylib BST Ticket

In Section 4 of this work, we examined performance factors for the lock-free BST on three concurrent synthetic microbenchmarks. We noted substantial impacts on performance as a result of microbenchmark implementation intricacies. In this section we investigate performance differences across the BST ticket (BST-TK) CSet data structure as implemented in the Setbench and Ascylib microbenchmarks. The ticket based binary search tree by
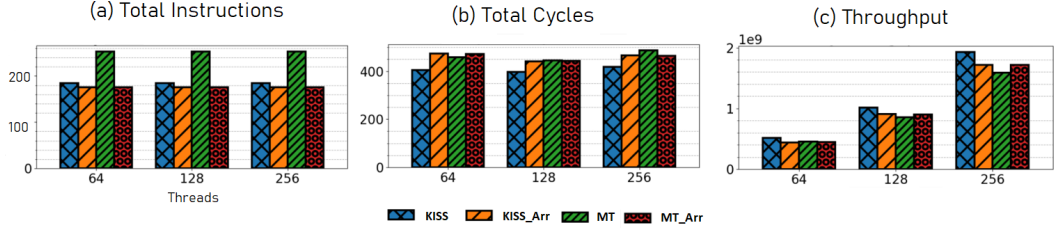
Figure 6 Array Based Pre-generated PRNG vs Non Array based PRNG on a key range of 20 000. (a) Total Instructions per operation (b) Total Cycles per operation (c) Total Throughput per second

Guerraoui et al. [11] appears in both Setbench and Ascylib microbenchmarks; however, it is not implemented in Synchrobench. The BST-TK is an external binary tree where leaf nodes contain the set of keys contained within the data structure. Internal nodes are used for routing and contain locks and a version number. This allows optimistic searches on the tree where concurrency can be verified by the correct version number. Both Ascylib and Setbench implement the BST-TK with memory reclamation. Ascylib has garbage collection (GC) turned on, Setbench performs epoch based reclamation. We observe in Figure 5(d) that throughput results from both microbenchmarks are similar on the BST-TK data structure. In Figure 5(c), we see again the Ascylib microbenchmark has higher memory usage, a greater than one order of magnitude increase over Setbench. This may render Ascylib experiments impractical in some settings and indicates memory is leaking at higher thread counts.

We have seen microbenchmarks can vary greatly in performance and memory usage across two different concurrent data structure implementations. We recommend microbenchmark users investigate overall memory usage in parallel with throughput results in order to get a clear understanding of the role of memory reclamation on the performance of a CSets. Memory may not be leaking necessarily; if the memory reclamation algorithm is simply slow or inefficient, there maybe a tangible impediment on performance.

6   Randomness in Concurrent Microbenchmark Experiments

As we have seen in previous sections, concurrent microbenchmarks rely heavily on PRNGs to generate randomized keys and randomized operations for high performance CSets that can perform potentially billions of operations per second. A fast PRNG is key. In this section we draw our attention to best practices of PRNG usage in concurrent microbenchmark experiments. We limit our attention to non-cryptographic PRNGs due to the speed requirement. It is desirable to utilize a PRNG with low overhead costs to the running experiment. Some microbenchmarks may choose a custom built PRNG, while others may opt for a standard built-in PRNG such as `rand()` used in Synchrobench. Some will pregenerate an array of random numbers (RNs); this allows fast, direct access to a list of RNs and avoids in-place generation costs. If properties of high quality randomness are desired, one may use an architecture specific hardware RNG. We explore the practicality and benefit of these approaches in subsequent sections. The PRNGs tested in this work include commonly used software PRNGs: murmurhash2 (MM2) [45], murmurhash3 (MM3) [1], KISS [6], Mersenne Twister (MT) [31], and MRG [36]. We describe custom hash functions and hardware RNGs in subsequent sections. Experiments in this section were run for durations of 3-5 seconds.
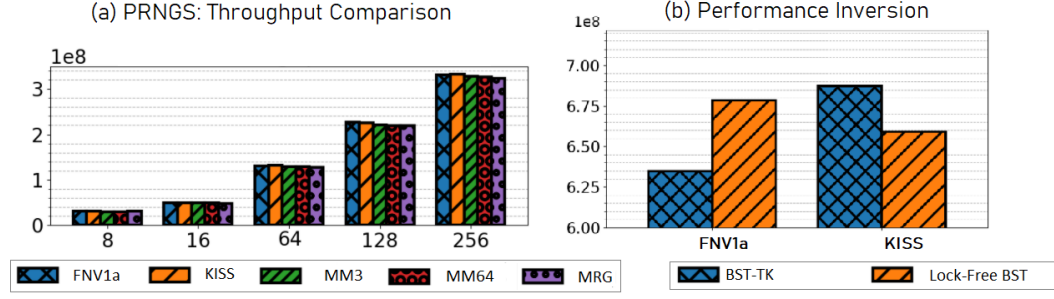
Figure 7 (a) Throughput results comparing FNV1a to other PRNGs; FNV1a algorithm does not indicate any performance anomalies in throughput results. Figure (b) Subtle effect of PRNG usage can remain undetected (4096 key range). There exists a performance inversion when using FNV1a as the per thread PRNG vs KISS.

6.1 Pre-Generated Array of Random Numbers

One might be tempted to think that the best way to obtain fast, high quality randomness would be to pre-generate a large array of RNs (or one array per thread) before running an experiment. Then, one could employ hardware randomness, or a cryptographic hash function, and push the high cost of generating random numbers into the unmeasured setup phase of the experiment. We tested this method in Setbench with per thread arrays of pre-generated RNs using the KISS and MT PRNGs versus in place RN generation with each algorithm. It is important to note that the pre-generated array approach eliminates the cost of in-place random number generation; during an experiment it is simply a matter of requesting an index into an array to generate the next random. A limitation of an array-based approach is, of course, the array size. It is undesirable to have frequent repetition of RNs during experimentation. We use array sizes of 10 million to generate a large set of RNs. Results in Figure 6(c) indicate the KISS_Array employed during experiments was notably slower than using the KISS algorithm in-place. This is due to the fact that a large array of 10 Million will lead to additional clock cycles generated by cache misses. An algorithm that is relatively fast, such as the KISS PRNG, will not benefit from taking a pre-generated array-based approach. However, a slightly more complex algorithm such as MT with more instructions in the PRNG algorithm, as illustrated in Figure 6(a), can benefit from an array-based approach. The MT_Array has slightly higher throughput results than using MT alone as indicated in (c). However, the benefit is not as striking as one may expect with an array-based PRNG approach. There may be use cases for an array-based PRNG such as requiring a more complicated (exotic) distribution of RNs. In this case, pre-generating RNs in an array may be an effective approach to limiting the overhead of a complex algorithm.

6.2 PRNG Associated Experimental Anomalies

In the search for high-speed generation of RNs, researchers may choose to implement their own PRNGs or use a custom hash function that may not have been well tested for properties consistent with high-quality PRNGs. Prior to this work, the PRNG used in Setbench was FNV1a [26], a fast, non-cryptographic 64-bit hash function. FNV1a is recommended by Lessley et al. as a hash function with *"consistently good performance over all configurations"* [26]. Setbench employed an FNV1a based PRNG that was used to generate both random operations and random keys. However, upon testing single threaded experiments, we noticed that the data structure prefilling step was failing to converge (i.e., it
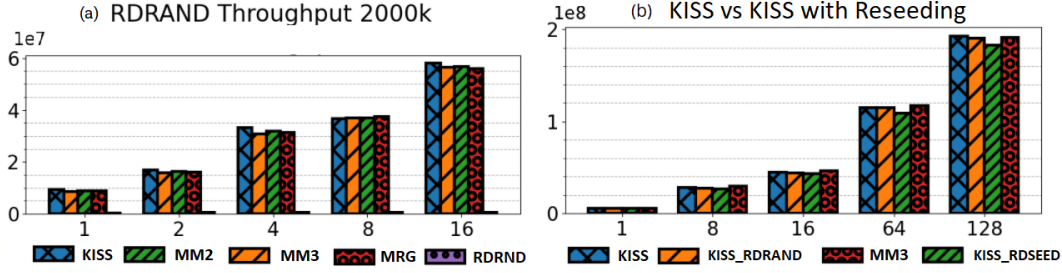
Figure 8 Hardware vs Software PRNGs: Figure (a) RDRAND throughput compared to software PRNGs. Figure (b) Reseeding KISS PRNG with RDSEED (or RDRAND) every 1 million random numbers indicates no strong penalty for reseeding a software PRNG with a hardware RNG.

was non-terminating). Upon further investigation, we found that the FNV1a based PRNG was generating RNs that followed a strict odd-even pattern. That is, after generating an even number, the next number would always be odd, and vice versa. (The initial seed determined whether the first number was even or odd.) During prefilling, a thread uses the first RN to determine the key and the second RN to determine the operation. In this case, the set of keys were always either all odd or all even, leading to an infinite loop when attempting to prefill the data structure to half full. Recall, that Setbench employs both insertions and deletions to prefill the CSet to steady state (half-full in this case). This odd-even pattern can easily be missed in overall data structure performance results. Figure 7(a) illustrates throughput results comparing various PRNGs tested in Setbench. There is no notable indication of threads generating all even or all odd keys from the FNV1a algorithm. As soon as the thread count increases, some threads are generating all even keys, while others are all odd. Setbench prefilling occurs with n threads, so as soon as the thread count increases from 1, the probability of convergence increases. One could imagine this kind of error remaining undetected and having a subtle effect on performance; limiting the set of keys per thread will affect which other threads it could contend with. In addition, it is not sound experimental methodology for a microbenchmark. Second, this undesirable behaviour found in FNV1a can lead to performance inversions when evaluating CSets in a microbenchmark. The impact of the FNV1a based PRNG is more clearly displayed in the results of Figure 7(b) where, given a high insert workload, FNV1a can show a performance inversion of experimental results. The experiment illustrates that the lock-free BST (Natarajan et al.) [37] throughput results are 1.12 times higher than that of the BST-TK (Guerraoui et al.) [11] when Setbench is using FNV1a as its PRNG. However, using another PRNG, such as KISS, we see the results indicate the lock-free BST *underperforms* by a factor of 0.96. This is an approximately 16% performance error leading to an inversion of results that could possibly remain undetected when one concurrent microbenchmark employs a problematic PRNG algorithm such as FNV1a. We implemented a tool, the $n^{th}-bit\ summation\ result$, to assess bitwise randomness in RNs generated by a PRNG. Incidentally, FNV1a also illustrated periodic behaviour in higher order bits. The tool is further discussed in Appendix B in the full version of this work [22].

6.3 Hardware RNG

A search for a high-quality 64-bit PRNG with its own source of entropy led us to an Intel Secure Key instruction, RDRAND, available on Ivy Bridge processors [19]. The RDRAND instruction returns an RN from Intel on-chip RNG hardware. We compare RDRAND with the software based PRNGs to evaluate the suitability of a hardware based PRNG for use

in synthetic microbenchmarks. We have illustrated throughput results in experiments with various PRNGs in Figure 8(a). We can see that the experimental throughput of RDRAND is significantly less compared to software PRNGs. A smaller key range of 2000 keys was necessary to visually illustrate the low throughput results generated when RDRAND is employed in Setbench. The overhead costs of hardware entropy greatly impede the overall performance of an experiment which aims to maximize throughput results. Algorithms such as KISS and MM3 that are computationally fast in nature have much higher throughput results. For concurrent microbenchmark experiments, it is not recommended to use a hardware PRNG alone. During our experimentation, we found that because RDRAND is a significant bottleneck in the benchmark experimental loop, results can appear equal for two CSet data structures that otherwise behave very differently. Although RDRAND is slow, it can be useful as a source of entropy for faster software PRNGs. The idea of periodically reseeding to introduce additional randomness into a PRNG is discussed by Manssen et al. [28] and Dammertz [20]. RDSEED is an Intel Secure Key instruction that complements RDRAND and is used to generate high quality random seeds for seeding PRNGs [44]. RDSEED is slower than RDRAND but is recommended to use for reseeding PRNGs. We tested a hybrid PRNG solution on the KISS algorithm where RDSEED is used for reseeding at intervals of every 1 million RNs (*KISS_RDSEED*). The results in Figure 8(b) indicate comparable throughput results to purely software based PRNGs without reseeding. We compared KISS reseeding with RDSEED to KISS reseeding with RDRAND (*KISS_RDRAND*), and there is a small drop in performance with RDSEED.

6.4 PRNG Recommendations

Massively parallel, high throughput experiments require billions of random numbers to be generated per second, which pushes the limits on PRNGs of our time. Some important points to consider for PRNG usage in microbenchmarks: (1) Hardware RNGs provide an external source of entropy, however, they are impractical for use in high speed concurrent microbenchmark experiments as the performance results are greatly impeded by RN generation time. (2) A pre-generated array of RNs is also counterproductive due to penalties associated with cache access. A pre-generated array of RNs may be useful if the PRNG algorithm is complex and in-place RN generation is too expensive. (3) For synthetic microbenchmark experiments we recommend two PRNG instances per thread; one for generating random values during the experiment and one for injecting new entropy into the first PRNG (periodic reseeding). If periodic reseeding is used every 1 million keys, there is a low, intangible impact on performance. If RDSEED or RDRAND are not available, we recommend using a high quality cryptographic PRNG for the 2nd PRNG. (4) Experiments that rely on bitwise randomness in RNs should first examine the set of generated numbers for periodic behaviour across various bit positions. (5) Last, in an era where data structures are performing billions of operations per second, we also think it's important to use PRNGs with at least 64 bits of state to avoid repeating the same sequence of generated keys on a time scale of seconds.

7   Towards Better Microbenchmarks

In Section 4.8, 5.1 and 6.4 we gave some recommendations for good benchmark design that were informed by our study of Ascylib, Setbench and Synchrobench. Setbench was designed with many of those recommendations in mind, and underwent relatively few changes as a result of our study. In this section, we give some additional recommendations and highlight features of Setbench that promote high quality experiments.

7.1   Additional Recommendations

More expressive ADTs

Today, many CSet data structures support range query operations and other interesting operations such as clone and size, and benchmarks should consider including support for them. It is not necessary for every operation to be implemented by every data structure, but providing a framework for additional operations to be included in experiments may encourage research in this direction.

Similarly, we encourage support for maps (also called dictionaries), which associate a value with each key, and support for large and/or variable-sized keys and values. This could encourage evaluations that span data structures published in distributed computing venues and those published in database and data management venues (e.g., [29, 27, 4, 25])—evaluations that are desperately needed in our opinion.

Starvation-aware experiments

Note, however, that some care is needed in experimenting with range queries, and any other types of long running operations that are prone to starvation. Consider a workload where threads perform, say, 49% insert, 49% delete and 2% range queries spanning the entire range of keys contained in the data structure. One would expect such range queries to be starved by updates, but in practice we find they are not! The trick is that *each* thread will perform only so many updates before performing a range query. So, if all range queries are perpetually starved, while updates succeed, eventually *all* threads will be executing range queries, and they will all succeed in a batch. This behaviour makes starvation seem like less of a problem than it might be in the real world, where there might *never* be a time when there are no updates in progress. Experiments involving starvation prone operations should *expose* the effects of starvation, possibly by allowing groups of threads to be dedicated to starvation-prone and non-starvation-prone operations respectively (see, e.g., [2]).

Pinning threads

In Section 4.6.1, we disabled thread pinning in all microbenchmarks in order to have consistency across all experiments. We recommend pinning threads to improve consistency of experiments, so for example, when you run 48 threads on a system with four 48-thread sockets, your threads run on a predictable set of cores, rather than, e.g., being clustered on one socket in one execution, and spread across three sockets in another. Additionally, thread pinning should be used to clearly expose the performance impact of hyper threading and the effects on non-uniform memory architectures in performance graphs. Thread pinning in benchmarks has been discussed in more detail by Gramoli et al. [14, 15] and Brown [7].

Non-uniform key distributions

Benchmarks should also consider incorporating various distribution generators for keys and values, rather than limiting experiments to uniform randomness. Researchers should consider using Zipfian, binomial, exponential or other skewed distributions in their experiments [35]. Distribution generators should be implemented efficiently, and sanity checks should be performed to ensure that the rate of key/value generation is not a bottleneck.

Uniform memory reclamation

Research in safe memory reclamation for CSets has consistently demonstrated that CSet performance can depend heavily on the algorithm for reclaiming memory (see, e.g., [10, 52, 46, 43, 39]). For this reason, memory should be reclaimed similarly across all data structures evaluated. In some cases, ad-hoc memory reclamation is tightly integrated in

a CSet, but benchmarks should offer a fast, easy-to-use memory reclamation algorithm to promote uniformity wherever possible.

Performance tools

Benchmarks should make a best effort attempt to *automatically* gather lightweight systems-level performance data, such as cache misses per operation, total cycles per operation, and peak memory usage. We suggest incorporating a library for performance monitoring such as the Performance Application Programming Interface (PAPI) [49]. We think it is crucial that these measurements are not only automatically gathered, but automatically *visualized*. Ideally, graphs for CSet throughput results and for systems level performance monitoring would be produced by default, at the same time, and would be visible in the same place. "Easy to check" is good. "Difficult to ignore" is better.

7.2   Benchmarking Advances in Setbench

Setbench was specifically designed to address all of the recommendations above, featuring a 256-bit PRNG, range query support (with support for independent range query threads and update threads), the ability to specify thread pinning policies at the command line, fast Zipfian and Uniform key distributions, uniform epoch based memory reclamation, and integration with a rudimentary implementation [53] of TPC-C and YCSB application benchmarks. It also includes a large collection of powerful tools for debugging, running experiments and analyzing performance, as well as automatic containerization for artifact evaluation.

Collecting user defined statistics

Debugging and performance analysis are extremely time consuming, and often researchers are limited in how much investigation they can do by the time it takes to modify their code to record specific events in their data structure. These events can be quite varied.

For example, one might want to answer a simple question like: in a lock-free algorithm, *how often do threads help complete other threads' operations?* Or, in an algorithm that uses epoch based memory reclamation, where objects are reclaimed in batches, one might want to answer a much more complex question—*how to produce a logarithmic histogram showing the distribution of the sizes of the first 10,000 batches reclaimed by each thread in an execution.* Setbench's global stats library (**gstats**) makes it fast and easy to explore such questions.

To emphasize how easy gstats makes this, to implement the latter, one would first create a gstats statistic that is accessible globally (throughout all files in the entire benchmark), by adding the following code to a file in Setbench called `define_global_statistics.h`:

```
gstats_handle_stat(LONG_LONG, epoch_batch_size, 10000, \
    { gstats_output_item(PRINT_HISTOGRAM_LOG, NONE, FULL_DATA) })  \
```

In essence, this efficiently allocates global per-thread arrays of 10,000 elements, and specifies that their contents should be used to build a logarithmic histogram. Whenever a thread `T` reclaims a batch of size `n`, it can append the batch size `n` to its array by invoking:

```
GSTATS_APPEND(T, epoch_batch_size, n);
```

These simple modifications result in the following new output when the benchmark is run:

```
log_histogram_of_none_epoch_batch_size_full_data=[...]
        [2^00, 2^01]: 71905
        (2^01, 2^02]: 206257
        (2^02, 2^03]: 307829
        (2^03, 2^04]: 469972
        [...] // output truncated to save space
```

Furthermore, scripts are included to plot bar graphs and line graphs from any data collected with gstats. In this case, assuming the output above is in `data.txt`, one would simply run: `trial_to_plot.sh data.txt epoch_batch_size`, which would create a PNG file.

Running Experiments and Plotting Results

Setbench also offers a powerful suite of Python scripts for running experiments and automatically plotting their results. Example run scripts that are suitable for CSet research are included. They produce MatPlotLib graphs of throughput *and many systems level performance metrics*, such as L3 cache misses per CSet operation, cycles per operation, and peak memory usage. Scripts are also available for several papers published by our group. The development of these scripts focused on conciseness, expressiveness and flexibility, and the scripts could be adapted to drive completely different benchmarks in different domains.

At a high level, to use these scripts, one defines a sequence of experimental *parameters*, and for each parameter, one specifies a list of values the parameter should take on. One then specifies a *run command* for the benchmark, and specifies how the parameters should be supplied to the run command. The command is run for each combination of parameters, and the output of each run is stored in an individual file. The scripts then process each file, and extract lines of the form "NAME=DATA" to produce columns in a **sqlite database**. As part of this process, data is *validated* according to user specified rules such as (`'total_throughput'`, `is_positive`) or (`'validate_result'`, `is_equal('success')`). Failed validation causes (colourful!) warnings to be emitted, and warnings can also be queried later from the sqlite database.

The scripts expose functions for easily producing plots (bars, lines, histograms and heatmaps) from the sqlite database simply by specifying which columns of data should be used for the x-axis and y-axis. Additional columns can be specified and graphs will be produced for every combination of values in these columns. Filters can also be specified to add to the SQL WHERE clauses in the queries that underpin plot generation.

In short, a single command `run_experiment.py [your_experiment.py]`, depending on its arguments, can compile (`-c`), run (`-r`), create the sqlite database (`-d`), produce graphs (`-g`) and create an HTML website (`-w`) organizing them into sections for convenient viewing. Clicking a graph on the website drills down to the rows of data the graph was built from, and clicking a row shows the raw text output for that run. A generated example website can be viewed at: `https://cs.uwaterloo.ca/~t35brown/setbench_example_www`. Results in the sqlite database can also be queried conveniently from the command line using SQL (e.g., `run_experiment.py your_experiment.py -q "select * from data"`). A wide range of additional capabilities are documented in extensive Jupyter notebook tutorials.

8   Conclusions

We hope this work encourages further research into how best to design benchmarks for concurrent data structures. Setbench was carefully designed to mitigate many of the problems we are aware of, but there are surely more benchmarking pitfalls yet to be discovered in this area. We also encourage researchers to try using Setbench for their own experiments, because its features make it much easier to drill down to the root causes of performance anomalies. After designing an algorithm, proving correctness, and implementing it, there is often little time left to do systems level performance analysis. Our hope is that by improving tools and automating the collection and graphing of key performance metrics, we can improve the quality of experiments without unduly burdening researchers in this area.

References

1   Austin Appleby. Murmurhash3, 2012. URL: `https://github.com/aappleby/smhasher/wiki/MurmurHash3`.

2   Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3178487.3178489`.

3   Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, 2018.

4   Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

5   Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.

6   Séamus Brannigan, Neil Smyth, Tobias Oder, Felipe Valencia, Elizabeth O'Sullivan, Tim Güneysu, and Francesco Regazzoni. An investigation of sources of randomness within discrete gaussian sampling. *Cryptology ePrint Archive*, 2017.

7   Trevor Brown. Good data structure experiments are r.a.r.e, 2017. URL: `https://www.youtube.com/watch?v=x6HaBcRJHFY`.

8   Trevor Brown. Powerful tools for data structure experiments in c++, 2021. URL: `https://gitlab.com/trbot86/setbench`.

9   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*, pages 276–291, 2020.

10   Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

11   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*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2694344.2694359`.

12   Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ascy-compliant concurrent search data structures. Technical report, EPFL Infoscience, 2014.

13   J. Evans. Scalable memory allocation using jemalloc. 2011. URL: `https://www.facebook.com/notes/10158791475077200/`.

14   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 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015.

15   Vincent Gramoli. The information needed for reproducing shared memory experiments. In *European conference on parallel processing*, pages 596–608. Springer, 2016.

16   Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with optik. *ACM SIGPLAN Notices*, 51(8):1–12, 2016.

17   Tim Harris. Benchmarking concurrent data structures. 2016. URL: `https://timharris.uk/misc/2017-tpc.pdf`.

18   Tim Harris. Do not believe everything you read in the papers. 2016. URL: `https://timharris.uk/misc/2016-nicta.pdf`.

19 Gael Hofemeier and Robert Chesebrough. Introduction to intel aes-ni and intel secure key instructions. *Intel, White Paper*, 62, 2012.

20 John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. Random number generators for massively parallel simulations on gpu, 1975. `doi:10.1109/SFCS.1975.23`.

21 Nikolaos D Kallimanis. Synch: A framework for concurrent data-structures and benchmarks. *arXiv preprint arXiv:2103.16182*, 2021.

22 Rosina Kharal and Trevor Brown. Performance anomalies in concurrent data structure microbenchmarks, 2022. URL: `https://arxiv.org/abs/2208.08469`, `doi:10.48550/ARXIV.2208.08469`.

23 Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.

24 Petr Kuznetsov and LTCI INFRES. Refining concurrency for perfomance. 2017.

25 Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

26 Brenton Lessley, Kenneth Moreland, Matthew Larsen, and Hank Childs. Techniques for data-parallel searching for duplicate elements. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 1–5, 2017. `doi:10.1109/LDAV.2017.8231845`.

27 Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

28 M. Manssen, M. Weigel, and A. K. Hartmann. Random number generators for massively parallel simulations on gpu. *The European Physical Journal Special Topics*, 210(1):53–71, Aug 2012. URL: `http://dx.doi.org/10.1140/epjst/e2012-01637-8`, `doi:10.1140/epjst/e2012-01637-8`.

29 Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

30 George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8:1–6, 2003.

31 Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, jan 1998. `doi:10.1145/272991.272995`.

32 Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

33 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

34 Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

35 N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan. Chapter 3 - discrete lifetime models. In N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan, editors, *Reliability Modelling and Analysis in Discrete Time*, pages 107–173. Academic Press, Boston, 2018. URL: `https://www.sciencedirect.com/science/article/pii/B9780128019139000038`, `doi:https://doi.org/10.1016/B978-0-12-801913-9.00003-8`.

36 Morita Naoyuki. Pseudo random number generator with mrg (multiple recursive generator), 2020. URL: `https://www.schneier.com/blog/archives/2008/05/random_number_b.html`.

37 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, pages 317–328, 2014. URL: `http://doi.acm.org/10.1145/2555243.2555256`, `doi:10.1145/2555243.2555256`.

38 Donald Nguyen and Keshav Pingali. What scalable programs need from transactional memory. *SIGPLAN Not.*, 52(4):105–118, apr 2017. `doi:10.1145/3093336.3037750`.

39   Ruslan Nikolaev and Binoy Ravindran. Brief announcement: Crystalline: Fast and memory
         efficient wait-free reclamation. 2021.

40   Javier Picorel, Djordje Jevdjic, and Babak Falsafi. Near-memory address translation. In *2017
         26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*,
         pages 303–317. Ieee, 2017.

41   Wenjia Ruan, Yujie Liu, and Michael Spear. Stamp need not be considered harmful. In *Ninth
         ACM SIGPLAN Workshop on Transactional Computing*, 2014.

42   Tomer Shanny and Adam Morrison. Occualizer: Optimistic concurrent search trees from
         sequential code. In *16th USENIX Symposium on Operating Systems Design and Implementation
         (OSDI 22)*, pages 321–337, 2022.

43   Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings
         of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–445,
         2021.

44   Thomas Shrimpton and R Seth Terashima. A provable-security analysis of intel's secure key
         rng. In *Annual International Conference on the Theory and Applications of Cryptographic
         Techniques*, pages 77–100. Springer, 2015.

45   Hardy-Francis Simon. Murmurhash2, 2010, 2010. URL: `https://simonhf.wordpress.com/
         2010/09/25/murmurhash160/`.

46   Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In
         *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel
         Programming*, pages 175–190, 2021.

47   Ajay Singh and Sathya Peri. *Efficient means of Achieving Composability using Transactional
         Memory*. PhD thesis, Indian Institute of Technology Hyderabad, 2017.

48   Ajay Singh, Sathya Peri, G Monika, and Anila Kumari. Performance comparison of various stm
         concurrency control protocols using synchrobench. In *2017 National Conference on Parallel
         Computing Technologies (PARCOMPTECH)*, pages 1–7. IEEE, 2017.

49   Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data
         with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E.
         Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg,
         2010. Springer Berlin Heidelberg.

50   Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky,
         and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings
         of the 2018 International Conference on Management of Data*, pages 473–488, 2018.

51   Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-
         based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.

52   Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-
         based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.

53   Xiangyao Yu. *An evaluation of concurrency control with one thousand cores*. PhD thesis,
         Massachusetts Institute of Technology, 2015.

54   Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient
         lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26,
         2019.