



City Research Online

City St George's, University of London

Citation: Galiev, R., Spear, M. & Aksenov, V. (2024). The Next 700 Benchmarking Frameworks for Concurrent Data Structures. In: UNSPECIFIED (pp. 1-9). ACM. ISBN 979-8-4007-0670-7 doi: 10.1145/3663338.3665827

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/36659/>

Link to published version: <https://doi.org/10.1145/3663338.3665827>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).

The Next 700 Benchmarking Frameworks for Concurrent Data Structures

Ravil Galiev
ravil.galiev.2000@gmail.com
ITMO University
Russia

Michael Spear
spear@cse.lehigh.edu
Lehigh University
USA

Vitaly Aksenov
aksenov.vitaly@gmail.com
City, University of London
UK

ABSTRACT

Concurrent data structures are essential building blocks for applications in almost every domain. However, differences among domains make it difficult to share results. Some testing frameworks model specific workloads, while others emphasize stress tests, but there is no easy way to evaluate if a new data structure will accelerate an application based only on such benchmarks. A confounding factor is the huge space of configuration options.

We address these issues with a new benchmarking tool for concurrent data structures in Java and C++. Our tool emphasizes a declarative description of workloads, a modular approach to defining components, and heterogeneity throughout.

As preliminary evidence of the effectiveness of our tool, we show that it can implement important and realistic workloads. In doing so, it contradicts prior assumptions, by showing that each of the three most popular trees written in Java can outperform the others.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Computing methodologies** → **Concurrent computing methodologies**.

KEYWORDS

concurrency, benchmark, data structures

ACM Reference Format:

Ravil Galiev, Michael Spear, and Vitaly Aksenov. 2018. The Next 700 Benchmarking Frameworks for Concurrent Data Structures. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Modern applications process large volumes of data in sophisticated ways. Due to the volume of data being processed, the structures used for storing and organizing that data must be able to scale under high levels of concurrency. At the same time, the tremendous diversity across application domains means that subtle differences

in design and implementation can translate to significant differences in performance.

Consequently, it is difficult to translate research into practice, even when data structure source code is open-source. Data structures are often tightly integrated into their applications. Thus, replacing one data structure with another in a production system is not trivial, even when the data structures have the same API and use the same run-time environment (such as safe memory reclamation). It is not until after the integration is complete that any performance evaluation is possible. If the result is unfavorable, the programmer must discern whether the problem is with the data structure or how it was integrated into the application.

Unfortunately, it is also difficult to make comparisons *among* data structures produced by researchers. Such data structures matter to the Operating Systems, Programming Languages, Architecture, Database, Cloud, HPC, and Theory communities. These communities have fundamentally different workloads and execution environments. Usually, concurrent data structures are tightly integrated into their benchmark harnesses, and comparing research data structures from different communities incurs the same integration overheads, and the same challenges when evaluating an unfavorable outcome.

To make a proper comparison, it is essential to incorporate and evaluate data structures in a synthetic environment that can emulate the sorts of workloads of interest to the respective communities. We state six reasonable requirements for a desired benchmarking suite:

- (1) **Modularity**: it should be easy to add new workloads and to reuse existing workloads / code;
- (2) **Duration**: workloads should be able to run for a fixed number of operations, a fixed time, or infinitely;
- (3) **Heterogeneity**: the benchmark should be aware of all threads, even internal modification threads [8]; and should allow different operation mixes among threads;
- (4) **Realism**: the behavior of the benchmark should mirror real-world workloads, and be predictive of the behavior that an application would experience. In particular, this must include a robust understanding of skew, as it is an important object of study [2, 3];
- (5) **Exploration**: it must be easy to vary parameters of the workload and data structure; and, finally,
- (6) **Portability**: it should be easy to test the same workloads in different programming languages.

Our work is a step toward satisfying these requirements. We introduce a new benchmarking framework that is declarative and emphasizes heterogeneity. Our approach to heterogeneity allows the creation of novel thread interactions with a data structure,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 ACM.

ACM ISBN 978-1-4503-XXXX-X/18/06

<https://doi.org/XXXXXXX.XXXXXXX>

while our declarative design ensures reproducibility and enables communities to distribute workloads that are representative of their applications' needs.

As for workloads, we re-implemented several of them from well-known benchmarking frameworks and introduced several new. We present a set of experiments that distinguish three concurrent search trees' performance on six synthetic workloads inspired by the real-world behaviours and built using the presented building blocks/entities. We find that despite the similarities among the trees, each is clearly "best" for one of the workloads. This separation makes an existential case for the need for a community effort.

Roadmap. In Section 2, we discuss existing benchmarks. In Section 3, we discuss the design of our framework and its implementation. In Section 4, we described the versions of entities we implemented. In Section 5, we showed that the implemented entities are already powerful and can be used to construct various workloads leading to different performance relations of state-of-the-art binary tree maps on Java. Finally, we conclude in Section 6.

2 EXISTING BENCHMARKS FOR CONCURRENT DATA STRUCTURES

In our work, we focus on data structures that provide a map interface: get, insert, and remove operations. They are usually implemented as trees, lists, and hash tables. The below discussion generally is also applicable to benchmarks for other data structures with other interfaces, such as queues, priority queues, and stacks.

2.1 Microbenchmarks

The most common approach for evaluating concurrent maps is stress-test microbenchmarks. Two of the most popular are Synchrobench [9] and Setbench [6]. Figure 1 depicts their design: each thread executes the same loop (ThreadLoop), which employs a per-thread pseudorandom number generator (PRNG) to select the type of operation to execute, and the operands to that operation. In Synchrobench, keys are drawn from a uniform distribution. Setbench supports uniform and Zipfian [11] distributions, which can simulate some keys being "hot". Both restrict keys and values to the size of a machine word (e.g., 64 bits).

Stress test microbenchmarks measure the scalability of the data structure when it is under heavy load. While stress tests are an important tool for data structure evaluation, they are not without flaws.

First, for large key ranges, a uniform distribution means that operations rarely contend, even at high thread counts. Second, the lack of non-data structure work, coupled with small key and value sizes, means that essential components of the data structure may never be evicted from the cache. This behavior can encourage unrealistic throughput expectations (e.g., when the top few levels of a tree remains resident in all caches at all times). Third, balanced mixes of insert and remove operations can suppress essential behaviors, like tree rebalancing. Lastly, adding new workloads often requires changes across the code base.

We can conclude that these benchmark suites satisfy **Exploration** and **Duration** well and partly satisfy **Portability** (e.g., Synchrobench has Java and C++ versions). However, unfortunately,

they do not satisfy **Modularity**, **Realism**, and **Heterogeneity** goals.

Nevertheless, we built our suite for Java on Synchrobench and for C++ on Setbench.

2.2 Common Benchmark Suites

In contrast to microbenchmarks, the YCSB! [7] and TPC benchmarks [1] aim to model specific workloads. On the one hand, they provide more **Realism**; on the other, they afford less ability to adjust parameters and explore the low-level aspects of a data structure's behavior, making harder to achieve **Exploration**.

YCSB is targeted primarily at cloud-scale data storage systems, but its workloads can be extracted and applied directly to data structures. The tested data structures should support five operation: get / insert / remove / read-modify-write of a single element, and scan, which reads several consecutive elements.

From these it composes six default workloads, which vary in terms of the distribution (skewed based on a Zipfian distribution or a notion of "recency") and the proportion of each of the six operations. For scan, there is an additional distribution and a target number of elements to read. The official YCSB workloads all consider a 1KB value, consisting of 10 equal-sized fields, and require each operation to read all 10 fields:

- update heavy (A): 50% update / 50% get, Zipf;
- read heavy (B): 5% update / 95% get, Zipf;
- read only (C): 100% get, Zipf;
- read latest (D): 5% insert / 95% get, custom "recency" distribution;
- scan workload (E): 95% scan (target 100 elements) / 5% insert, Zipf;
- RMW workload (F): 50% read-modify-write / 50% get, Zipf.

TPC is a suite of suites. Its workloads are more complicated than YCSB workloads, modeling transactions on data warehouses. Of particular interest to our goals are on-line transaction processing (TPC-C) and decision support (TPC-H) [1]. TPC-C is more popular among concurrent data structure researchers, but fundamentally it is only five transaction types over nine table types, and the tables grow over time. Great effort has gone into ensuring this is a realistic workload, but it is not instructive for data structures used in garbage collectors, middleware, operating systems, etc.

Summarizing, YCSB and TPC score high on **Realism** and partially on **Portability** requirements, but low on the other four: on **Modularity** it is not easy to add a new workload, on **Duration** they allow for only a fixed number of operations, on **Heterogeneity** all threads execute the same code, and, finally, on **Exploration** there are complex dependencies among parameters that make it easy to create an invalid experiment.

3 SOFTWARE DESIGN

Figure 2 shows our benchmark design that extends the traditional one in two dimensions. First, we introduce **Modularity**: we delegate the argument choices of operations in a ThreadLoop to a special subcomponents named "argument stream" (light blue box) consisting of one ArgsGenerator and multiple Distribution and DataMap entities. Second, we can express thread heterogeneity by

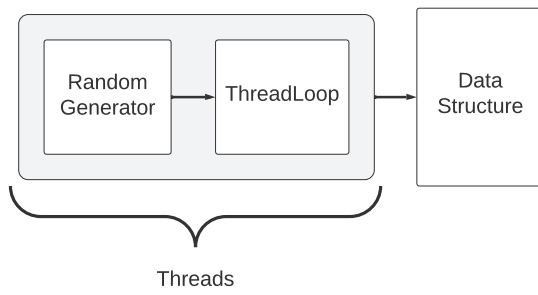


Figure 1: Structure of Syncrobench and Setbench

allowing each thread to use its own ThreadLoop which can use several argument streams.

By assigning each thread the same ThreadLoop with a single argument stream using a single PRNG, our design simply emulates a traditional stress-test microbenchmark on Figure 1. Whereas the older design hard-codes the thread's interaction with the data structure in the thread loop, and only relies on the PRNG to provide a stream of pseudo-random numbers that it can use to modulate its operation mix and choose key/value arguments, our design allows for more logic: the thread loop decides which operations to execute while asking different streams what arguments to use with a chosen operation. In effect, the logic of choosing the arguments from the traditional thread loop migrates to the argument stream. To fully model the older behavior, the argument stream can be customized from command-line arguments, while now we allow to configure the parameters using JSON format. By the same logic, our new design with the specifically implemented submodules can model the six workloads of YCSB, or the behavior of TPC-C.

Of course, in emulating the old design, a benchmark will not offer good **Modularity**, **Heterogeneity**, or **Exploration**. We now discuss how to use the new structure to build better benchmarks. Our focus is on the robustness provided by alternate argument stream objects.

In Figure 2, a workload is built from ThreadLoops with a bunch of argument stream from subcomponents: a Distribution, a DataMap, and an ArgsGenerator. Each entity performs only its own specific task, and requests the necessary information from other entities. Since every entity implementation is interchangeable, this design improves **Modularity**. Note that for simplicity, we depict a design in the shape of a tree. However, it is possible for a ThreadLoop to share a PRNG, DataMap, or Distribution among its argument streams, and even with argument streams of a different ThreadLoop.

Finally, to address the **Duration** requirement, ThreadLoops have a termination condition, implemented as StopCondition object. This condition can be the end of the allocated time, the completion of a certain number of operations, or some user-defined property.

3.1 Overview of an Execution

The benchmark initializes a separate ThreadLoop for each thread. Each ThreadLoop runs until the StopCondition is satisfied. In each iteration, the ThreadLoop chooses an operation and the argument stream generates its arguments. The ThreadLoop then executes the operation on the data structure. Each instance of ArgsGenerator can use several Distributions and DataMaps to generate keys for

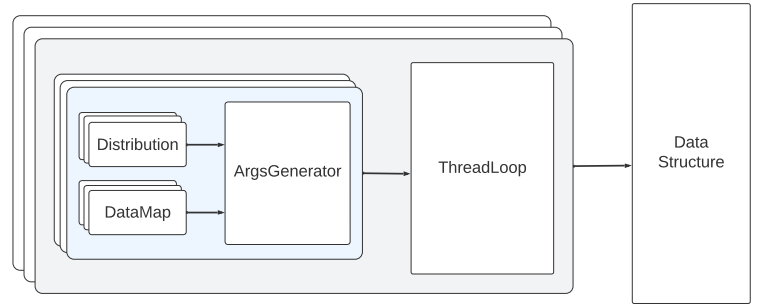


Figure 2: Structure of our benchmark

each type of operation. By default, it takes a random variable from a chosen Distribution and converts it into a key using a chosen DataMap.

Also, our design supports pre-filling and warming up the data structure. Initializer threads can run some workload until some condition is satisfied, e.g., the data structure reaches becomes half-filled. Note that both sequential and multithreaded pre-fillings are possible. Due to providing both pre-filling choices, the benchmark can support NUMA systems, where objects/nodes should be distributed evenly over different sockets, which is only achieved by threads *specifically* on those sockets creating/inserting objects [10]. Through **Modularity**, additional warm-up steps are also possible (e.g., to make sure all threads have similar initial cache states). If a user still wants the prefill to be sequential — he needs to set the number of prefilling threads to one.

3.2 Entities

We now describe the entities from Figure 2 in greater detail.

Distributions: The lowest-level entity is the Distribution, which simulates a random variable. It is important to note that it generates some value from a distribution that later is translated into an appropriate key or value by an ArgsGenerator. A Distribution's next() method generates a value within some range.

Default distributions in our benchmark include uniform, Zipfian, and skewed uniform [3]. We also allow the ArgsGenerator to mutate the distribution at run time, e.g., by modifying the range of keys.

DataMaps: The DataMap is used by an ArgsGenerator to translate an “index-key” into a key or value, e.g., via computation and/or table lookups. We provide three implementations: a shuffled array, a hash, and an identity map.

In some sense, an ArgsGenerator asks a Distribution for an index-key and passes it through DataMap to obtain a final key/value argument.

Consider a simple example. Suppose that we want the arguments to be all even integers in the range $2N$ chosen uniformly. An ArgsGenerator asks a uniform Distribution for an element in $[1, N]$ and then a DataMap just multiplies this value by 2.

ArgsGenerators: ArgsGenerators are used to generate arguments for operations. In addition to specializing to the types used by a data structure, an ArgsGenerator can be stateful. This is important when modeling temporal locality, for example, a thread-private ArgsGenerator can hold a knowledge of recently requested keys in order to choose one of them for future operations use.

ThreadLoops: The ThreadLoop chooses the operations by some logic and asks the arguments from some of its argument stream. Threads are not required to use the same ThreadLoop implementation. ThreadLoop has only one main method `step()` that explains how to choose an operation and perform it during the main execution phase. The method `step()` is called in a conditional loop by the `run()` method. A programmer can override `run()` when more complicated logic is needed, for example, a maintenance daemon thread.

ThreadLoop is also a utility class: it calculates different statistics, for example, the number of successful operations. This is important for the benchmark in order to compare different data structures and check their correctness.

3.3 Entities in code

We currently provide Java and C++ versions of our benchmark tool (<https://github.com/Mr-Ravil/Benchmark-v700/> in “java” an “cpp” folders, correspondingly). While the discussion in this section is generic, our implementations use appropriate mechanisms (dynamic class loading, template metaprogramming, build-time code generation) to reduce programmer effort and avoid implementation artifacts that can perturb performance measurement.

When the benchmark starts, it processes the JSON configuration file and initializes threads with the corresponding ThreadLoops. This step uses the builder pattern to reduce complexity of the code. Each ThreadLoop, in turn, builds the entities it uses.

While JSON is the primary configuration mechanism, we also support limited overriding on the command line (e.g., for specifying the number of threads, or changing arguments to a data structure’s constructor).

3.3.1 Implementing a new entity. The implementation of each entity (Distribution, DataMap, etc.) consists of two parts: its logic, and its Builder. For each of these two parts, there exists an interface that indicates which functions are needed to be implemented. To give a sense about the effort involved in creating a new and reusable entity, we present a generic `ExampleEntity` class. The implementation of `ExampleEntity` is pretty simple.

```
class ExampleEntity implements Entity:
    ExampleEntity(/*entityParameters*/) {...}

    /*overriding entity interface functions*/
```

The programmer only needs to override the interface functions of the chosen entity. For example, `ArgsGenerator` interface requires a generator function for each type of an operation: `nextGet()`, `nextInsert()`, and `nextRemove()`.

The Builder used for creation the entity object should perform three operations: (1) accept raw parameters specified by the users; (2) convert the raw parameters to the inner-representation ones; and (3) create the entity object itself. The first stage is implemented through simple set functions for each raw parameter. We allow to describe the builders of subcomponents in the parameters. The second stage is implemented through the `init()` function, which the programmer needs to override. This stage is necessary so that the Builder does not depend on global parameters, such as key set range, during initialization of entity parameters. Therefore, the user

can use only one Builder object for several test runs with different global parameters. And the last stage is implemented using the `build()` function, where the programmer creates an entity object using the inner-representation parameters. Now, we present a generic `ExampleEntityBuilder` class.

```
ExampleEntityBuilder implements EntityBuilder:
    /*rawParameters*/
    /*finalParameters*/

    ExampleEntityBuilder() {...}

    /*rawParameters setter methods*/

    ExampleEntityBuilder init(/*globalParameters*/):
        /*initializing final parameters
           by processing raw and global parameters*/

    ExampleEntity build():
        /*creating the new instance of ExampleEntity*/
```

Since the Builder after the first stage has all the necessary data and does not depend on global parameters, we can dump it in the JSON format which later can be reused in different benchmark runs.

4 IMPLEMENTED ENTITIES

In this section, we discuss the default set of entities provided by our suite. Our focus is on providing robust support for benchmarking ordered and unordered sets and maps (e.g., key-value data structures). We provide implementations of these entities for both Java and C++ (<https://github.com/Mr-Ravil/Benchmark-v700/> in “java” an “cpp” folders, correspondingly). To make everything convenient and portable, we allow the user to provide the description of the workload with entities using the JSON format which is language independent: the same JSON file sets ups the same workload for both Java and C++ implementations.

4.1 Distributions

Our distributions are compatible with a broad selection of random generators, including those in the standard library (e.g., `rand`, `mt19937`, etc.) and third-party generators (e.g., `xoshiro`).

Uniform Distribution. We support a traditional uniform distribution, as well as a `MutableDistribution` that does not restrict the range of values which can change through time: it generates a random value using `random.nextInt(range)`.

Zipfian Distributions. The Zipfian Distribution is based on Zipf’s law [11].

Skewed Uniform Distribution. This distribution has two parameters: `hotSize` and `hotRatio`. It divides the set of values into hot and cold subsets of sizes `hotSize%` and `100 – hotSize%` of the total, respectively. Then, with probability `hotRatio%` this distribution chooses an element uniformly at random from the hot subset and from the cold subset, otherwise.

4.2 DataMaps

DataMaps can combine lookup tables and code to translate a random value to a datum of the desired type.

Identity DataMap. The Identity DataMap is a simple identity function. To support corner cases in some map implementations, e.g., when 0 is a special non-requested value, this DataMap given i returns $i+1$.

Array DataMap. The Array DataMap creates an array filled with values from the entire range of keys and shuffles them randomly. When asked to translate i , it just returns the corresponding element from the array.

Hash DataMap. The Hash DataMap converts the index to a key using some hash function, for example, a bit shuffling.

4.3 ArgsGenerators

When choosing an argument an ArgsGenerator sends an index, i.e., a generated value by a chosen Distribution, to some chosen DataMap and gets the corresponding key.

Default ArgsGenerator. The Default ArgsGenerator accepts one of the existing distributions and one of the existing DataMaps as input and selects the next key based on this distribution from the chosen DataMap.

Skewed Sets ArgsGenerator. The Skewed Sets ArgsGenerator uses two SkewedUniformDistributions separately for read and update operations, and takes the following parameters:

- $rp\%$ of read operations are performed on a random subset of keys of proportion $rs\%$ where a key is taken uniformly. All other read operations are performed on the rest of the set.
- $wp\%$ of update operations are performed on a random subset of keys of proportion $ws\%$ where a key is taken uniformly. All other update operations are performed on the rest of the set.
- $inter\%$ of keys are in the intersection of the working sets of read and update operations.

The similar skewed read-update workload was used to benchmark the Splay-List [3].

Temporary Skewed Sets ArgsGenerator. The Temporary Skewed ArgsGenerator allows the skew of a generator to change over time, e.g., to model daily fluctuations in search queries. This generator has two types of states:

- k -th *excited* state — the keys are selected using k -th SkewedUniformDistribution, i.e., there is a hot set of keys.
- a *dormant* state — all keys are selected with the UniformDistribution. This state happens between k -th and $k+1$ -th excited states.

To support infinite execution, the excited states are chosen in a cyclic manner, i.e., after the latest *excited* state it returns to the first dormant state. This reflects the daily-periodic pattern of requests, e.g., in social networks. The TemporarySkewedArgsGenerator uses the following parameters:

- $state-count$ — the total number of excited states in the workload;
- ht — the default duration of an excited state (the duration is specified in the number of operations);
- rt — the default duration of a dormant state;
- during the i -th excited state, $p_i\%$ of operations are performed on $s_i\%$ of keys, and $100 - p_i\%$ of operations are performed on the rest of the set;

- ht_i — the duration of the i -th excited state (optional);
- rt_i — the duration of the dormant state after i -th excited state (optional).

Creakers and Wave ArgsGenerator. The Creakers and Wave ArgsGenerator models a different sort of temporal locality: recently inserted keys are requested more often, but become obsolete over time. Among other things, this ArgsGenerator can model YCSB workload D. Our idea for the workload is supported by the following observation. For example, musical hits are listened to more often when they have just been released. Then, something new is released and the popularity of the old ones decreases. To simulate this behaviour, the Creakers and Wave ArgsGenerator has an entity Wave.

The Wave is a subset of all keys yielded to an array and specified by a head and a tail. It generates a new key according to the following rules:

- $nextRemove()$ — the Wave returns the key of the current tail and moves/increments the tail by one;
- $nextInsert()$ — select the key next to the head of the Wave, and makes it the new head (or increments the head);
- $nextGet()$ — select a key from the Wave according to some specified distribution.

As a default, Wave uses the Zipfian Distribution for get operations, where the closer the key is to the head, the greater the probability of being selected. Since there is a limit on the range of elements, to make the ArgsGenerator infinite the Wave moves in a cyclic manner over a working set of keys. Besides the Wave there is an entity called Creakers. This is a subset of keys that are requested rarely but permanently. This entity is needed to check how well the data structure copes with such keys in the presence of rapidly growing and equally rapidly discarded keys from the Wave.

The Creakers and Wave ArgsGenerator has the following parameters:

- $cp\%$ of operations are performed on $cs\%$ of keys (Creakers entity), and $100 - cp\%$ are performed on the Wave entity;
- The Wave is initialized with $ws\%$ of keys;
- $c-age$ get operations are performed on the Creakers entity during warmup before the benchmarking to increase their “hotness”;
- $c-distribution$ — a distribution of get requests to keys in the Creakers entity (by default, the Uniform distribution);
- $w-distribution$ — a mutable distribution of get requests to keys in the Wave entity (by default, the Zipfian distribution with $\alpha = 1$).

All threads share the Wave head and tail.

When coupled with the DefaultThreadLoop, Creakers and Wave ArgsGenerator models workload D from YCSB. However, there are differences in favor of our approach. At first, our workload can be infinite, while YCSB is predefined. There is no doubt that one can make YCSB work infinitely long, but the corresponding generator cannot generate a key with a time complexity of $O(1)$ in the runtime. This definitely affects the workload. Secondly, we added Creakers to our workload. Logically, nothing should really change — we just added a small permanent load that always happens to the systems. However, it is a huge change for self-adjusting data structures [2, 3]. The known implementations store a counter for each element, i.e.,

how many times it was an argument. The bigger the counter the faster the element is accessed. The problem occurs when elements are removed and lose their whole counter. Thus, if we consider some infinite execution, Creakers will have the largest counters, while the elements from the Wave will have smaller counters due to the insertion and removals. Obviously, this has a huge impact on the performance of the aforementioned index implementations. Also, our design makes the two key sets explicit, and, thus, produces reliable results regardless of the duration of the experiment.

Leafs Handshake ArgsGenerator. The last presented ArgsGenerator is Leafs Handshake ArgsGenerator. It is stateful and mutable: the key selection for an insert operation depends on the argument of the latest remove operation. Intuitively, the closer the key to the last removed one, the more probability that this key will be selected for an insertion. It accepts the following parameters:

- *get-distribution* — a distribution of keys for a get operation (by default, the Uniform distribution);
- *remove-distribution* — a distribution of keys for a remove operation (by default, the Uniform distribution);
- *insert-distribution* — a mutable distribution of keys for an insert operation (by default, the Zipfian distribution with $\alpha = 1$).

The ArgsGenerator gets the arguments from the range $[1, N]$ and then transforms them to the final arguments using a DataMap. By default, the ArgsGenerator chooses get and remove arguments uniformly at random from the whole range. For insert, it just maintains the last removed element r , chooses the side from $[r-1, 1]$ and $[r+1, N]$ with probability $1/2$, and returns an element from the chosen side using *insert-distribution*.

4.4 ThreadLoops

Default ThreadLoop. The Default ThreadLoop selects the next operation with some fixed probability. It accepts the following parameters:

- $ui\%$ of operations are insert operations;
- $ue\%$ of operations are remove operations;
- while $100 - ui - ue\%$ of operations are get operations.

Temporary Operations ThreadLoop. The Temporary Operations ThreadLoop selects the next operation depending on a time interval. It accepts the following parameters:

- *temp-oper-count* — a number of different intervals;
- ot_i — the duration of the i -th interval, as a number of operations;
- $ui_i\%$ of operations are insert operations during the i -th interval;
- $ue_i\%$ of operations are remove operations during the i -th interval;

5 SEPARATION OF BINARY SEARCH TREES

In this section, we use the Java version of our benchmark. We consider three commonly-known binary search tree (BST) implementations. We find that with our workloads, each can outperform the others. This result affirms the importance of more robust benchmarking for concurrent data structures.

5.1 Implementations

We take three “fastest” binary search tree (BST) implementations written in Java for Sychrobench:

- A BCCO BST by Bronson et al. [5].
- A Contention-Friendly (CF) BST by Gramoli et al. [8].
- A Concurrency-Optimal (CO) BST by Aksenov et al. [4].

All of them are partially-external and the main difference between them is how they handle physical removal and rotation. In the BCCO-BST by Bronson et al. [5] a working thread always removes nodes physically and rotates subtrees, if necessary. In the CF-BST by Gramoli et al. [8], a working thread makes only logical removals. However, there is a special daemon thread responsible for physical removals and rotations. In the CO-BST by Aksenov et al. [4], a thread performs physical removal immediately but does not perform rotations at all.

Prior papers using the workloads from Sychrobench, e.g., [4], suggest that CO-BST is superior, CF is the runner-up, and BCCO is worst. Obviously, it should not be the case on all possible workloads because: 1) CO does not make rotations at all, leading to longer traversals; 2) CF should perform worse on the larger trees, since the daemon thread cannot catch up with changes. We show that it is indeed the case: depending on the workload we can see different relative performance.

5.2 Experimental Results

We evaluated the trees on a system with two Intel Xeon Gold 6240R CPUs (48 cores total). We show results starting from 16 working threads, since the high number of threads is our focus. Each plot has three graphs: blue circles (CO), green triangles (CF), and red crosses (BCCO). The X-axis represents the thread count, and the Y-axis represents the throughput (operations/second). Each point in plots is the average of ten 10-seconds trials.

5.2.1 Uniform and Zipfian Workloads. Uniform and Zipfian Workloads are the most common in scientific publications. We run them with the following parameters: range is 10^4 (for Uniform) and 10^5 (for Zipfian) and update ratio is 5%. The insert ratio is equal to the remove ratio. Both these workloads are widely accepted as “realistic”. For example, allocators typically have uniform access patterns, while social networks exhibit Zipfian patterns. Unsurprisingly, Figures 3 and 4 show the same trend, with CO outperforming CF, and CF outperforming BCCO.

5.2.2 Infinite Leafs Handshake Workload. Since BCCO-BST always rotates and removes physically, one expects it to be too far behind in terms of throughput. But that does not mean that it works badly on any workload. We present the Infinite Leafs Handshake Workload. It is based on the Leafs Handshake ArgsGenerator and the Temporary Operations ThreadLoop presented in Section 4. It has three time intervals: 1) the filling interval — there are more insertions than removal; 2) the read interval — there are only read operations; 3) the cleaning interval — there are more removals than insertions.

During the filling interval, two new neighbors are inserted for one removed node. During the reading interval, threads read keys uniformly at random. During the cleaning interval, the working threads remove nodes uniformly at random from the tree to restore its original size.

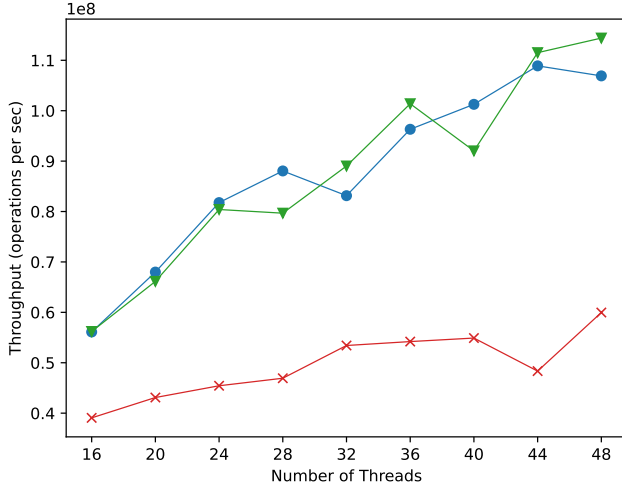


Figure 3: Uniform Workload with range 10^4 and update ratio 5%

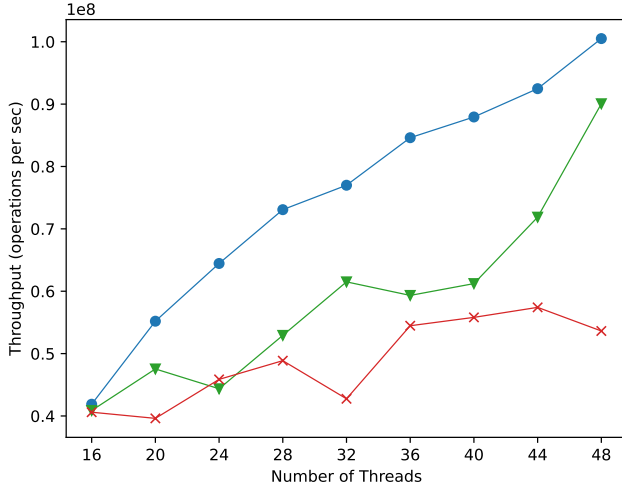


Figure 4: Zipfian Workload with $\alpha = 1$, range 10^5 , and update ratio 5%.

Such a workload emulates the generation of fork-join tasks: when the task finishes it creates two new ones working on the close identifiers, then, the tasks are in “progress” while the read requests check their status, and, finally, they finish. Also, such a workload can be seen as the graph exploration algorithm.

The first experiment (Figure 5) is run with parameters: range is 10^5 , temp-oper-count is 3; $ot_0 = ot_2 = 10000$, $ot_1 = 5000$; $ui_0 = ue_2 = 60\%$; $ui_2 = ue_0 = 40\%$; $ui_1 = ue_1 = 0\%$; get and remove distributions are uniform distributions; insert distribution is a Zipfian distribution with $\alpha = 2$. In this configuration, the order of CF and CO switches.

In Figure 6, we use different parameters: range is 10^7 ; temp-oper-count is 2; $ot_0 = ot_1 = 20000$; $ui_0 = ue_1 = 90\%$; $ui_1 = ue_0 = 10\%$; get and remove distributions are Uniform; and insert distribution is Zipfian with $\alpha = 0.99$. In this case, BCCO performs best, and CF worst.

In a third configuration (Figure 7) we use these parameters: range is 10^8 ; temp-oper-count is 3; $ot_0 = ot_1 = ot_2 = 100000$; $ui_0 = ue_2 =$

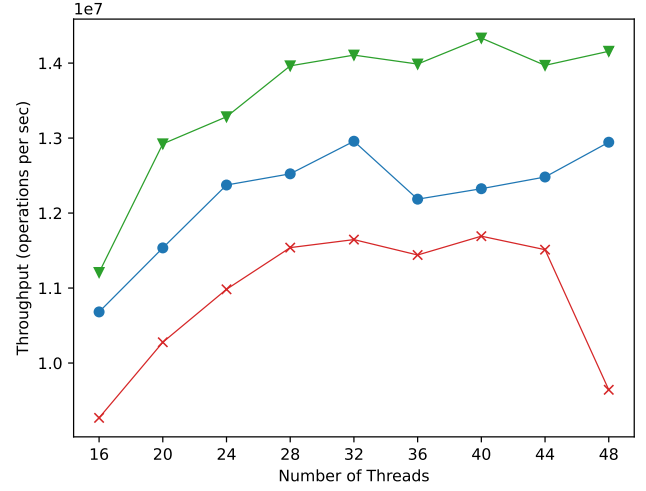


Figure 5: Infinite Leafs Handshake Workload with range 10^5 .

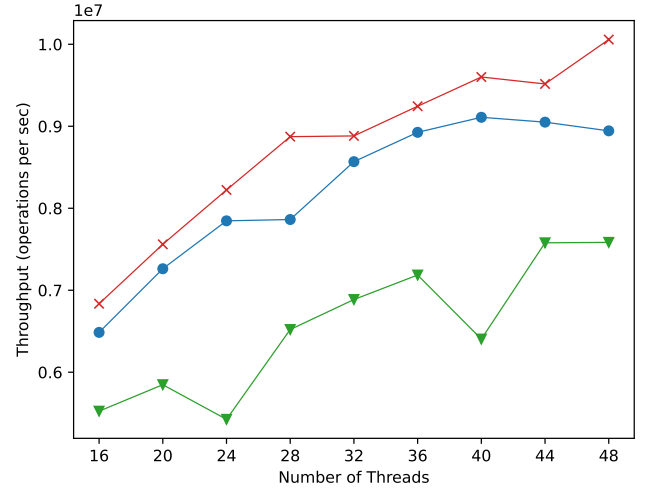
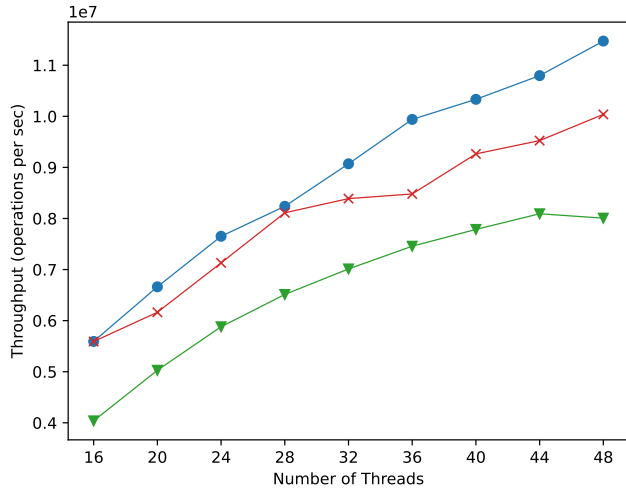
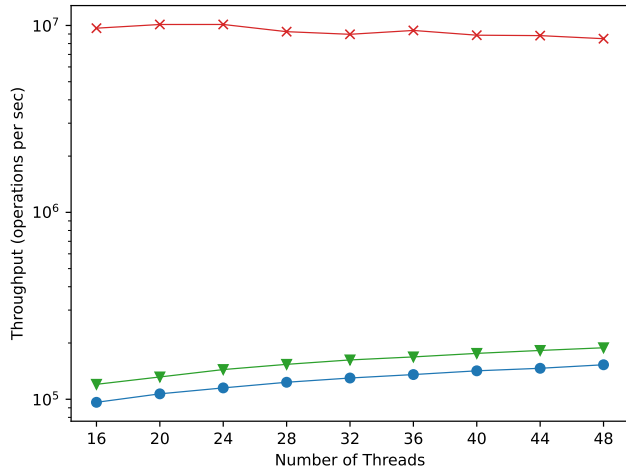


Figure 6: Infinite Leafs Handshake Workload with range 10^7 .

80%; $ui_2 = ue_0 = 20\%$; $ui_1 = ue_1 = 0\%$; get and remove distributions are Uniform; insert distribution is a Zipfian distribution with $\alpha = 0.99$. We see another change in order, with CO best and CF worst.

From the results, it can be seen that CF-BST lags far behind at large ranges (Figures 6 and 7), while at small ranges it behaves about the same or better than others (Figure 5). This happens because the daemon thread does not catch up with remove operations and the tree has a longer traversal length. Also, CO-BST and BCCO-BST behave differently at large ranges. The lower the skew between insert and remove operations, the better CO-BST handles the workload. This is because with more skew there are more chances that the tree will grow non-uniformly making the CO-BST perform poorly.

5.2.3 Non-shuffle Wave Workload. The Non-shuffle Wave Workload is based on Creakers and Wave ArgsGenerator without the Creakers and with Identity DataMap. This workload adds new keys to the edge of the tree disrupting the balance. That leads to the performance problems of poorly balanced trees.

Figure 7: Infinite Leafs Handshake Workload with range 10^8 Figure 8: Non-shuffle Wave Workload with range 10^6 .

This workload emulates the load on the videos in a social network: a new video has arrived and a lot of users watch them, while old videos are rarely accessed.

The first experiment (Figure 8) is run with parameters: range is 10^6 ; $ws = 20\%$; update ratio is 5%; *w-distribution* – Zipfian distribution with $\alpha = 1$; and $cp = 0\%$. It gives us the relative performance of B-CF-CO.

This result is again related to the fact that BCCO-BST always rotates which is not the case of CF-BST and CO-BST.

The second experiment (Figure 9) is run with parameters: range is $5 \cdot 10^3$; $ws = 10\%$; write ratio is 20%; *w-distribution* – Zipfian distribution with $\alpha = 1$; $cp = 0\%$. It gives us the relative performance of CF-B-CO.

This result is related to the fact that the tree size is quite small, which is why the daemon thread in CF-BST manages to balance the tree quickly.

5.3 Summary

As a result we obtained all six relative performances of three data structures: B-CF-CO on Figure 9, B-CO-CF on Figure 6, CF-B-CO on

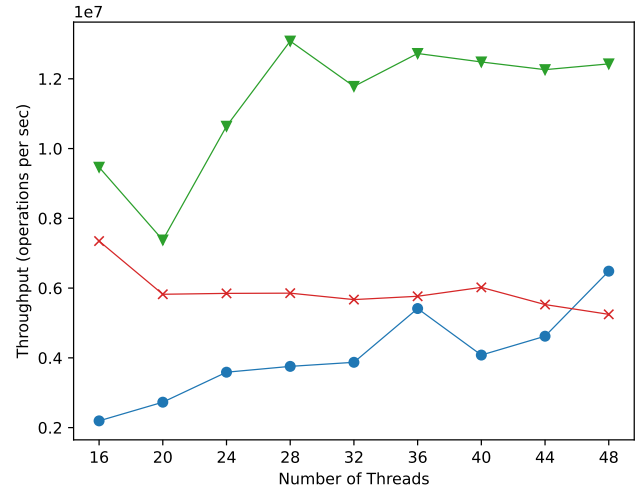
Figure 9: Non-shuffle Wave Workload with range $5 \cdot 10^3$.

Figure 8, CF-CO-B on Figure 5, CO-B-CF on Figure 7, and CO-CF-B on Figures 3 and 4.

6 CONCLUSION

In this work, we presented a new benchmarking suite that aims to make it easier to test concurrent data structures against a wider set of workloads and configurations. Creating new workloads takes little effort, enabling communities to offer more representative workloads than the sorts of stress tests that dominate the literature today. Even just a small number of workloads sufficed to show that each of three popular trees can be “best”, confirming the importance of better benchmarking in this domain.

As future work, we intend to continue refining the tool, particularly with regard to ergonomic issues that make it easier to integrate new workloads. We also intend to release our benchmark as open-source code, and to implement a review process for new contributions, so that we can continue to grow the utility of the tool and increase its value to multiple research communities.

REFERENCES

- [1] [n. d.]. TPC benchmarks. <https://www.tpc.org/>.
- [2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. 2014. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing* 27, 6 (2014), 393–417.
- [3] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. 2023. The splay-list: A distribution-adaptive concurrent skip-list. *Distributed Computing* (2023), 1–24.
- [4] Vitaly Aksenov, Vincent Gramoli, Petr Kuznetsov, Anna Malova, and Srivatsan Ravi. 2017. A concurrency-optimal binary search tree. In *Euro-Par 2017: Parallel Processing*. Springer, 580–593.
- [5] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *ACM Sigplan Notices* 45, 5 (2010), 257–268.
- [6] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th PPoPP*. 276–291.
- [7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [8] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing*. Springer, 229–240.
- [9] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th Symposium on Principles and Practice of Parallel Programming*. 1–10.
- [10] Rosina Kharal and Trevor Brown. 2022. Performance Anomalies in Concurrent Data Structure Microbenchmarks. *arXiv preprint arXiv:2208.08469* (2022).
- [11] David MW Powers. 1998. Applications and explanations of Zipf’s law. In *New methods in language processing and computational natural language learning*.