Benchmark Framework with Skewed Workloads

Abstract

Concurrent data structures are essential building blocks for applications in almost every domain. Differences among domains make it difficult to share results, however. 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 can outperform the others.

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 data structure design can translate to significant differences in performance.

Consequently, it is difficult to translate research into practice, even when researchers share their implementations as open-source code. Data structures are often tightly integrated into their applications. Thus the undertaking to replace one data structure with another in a production-quality program is not trivial, even when the application's uses of the data structure exactly match its API, and the application satisfies the data structure's other requirements (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 data structure is not favorable for the workload, or the integration introduced unanticipated overheads. Often, the conclusion is that the effort was for naught.

Concurrent data structures matter to the Operating Systems, Programming Languages, Architecture, Database, Cloud, HPC, and Theory communities. These communities have fundamentally different workloads. In order to share their data structures, it is essential that the structures can be evaluated in a synthetic environment that can emulate the sorts of workloads of interest to the respective communities.

Our work is a first step toward this goal. 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, while our declarative design ensures reproducability and enables communities to distribute workloads that are representative of their applications' needs.

To emphasize the significance of the problem, we present a set of experiments that distinguish three concurrent search trees' performance on six synthetic workloads inspired by the real-world behaviours. 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. We start with two overview sections. In Section 2 we discuss the existing benchmarks and their issues and in Section 3 we pose the requirements on a desired benchmarking suite. In Section 4, we present a software design of our suite and provide an example on how one can add a new workload. In Section 5, we describe the predefined entities in our suite which one can already use to build more sophisticated workloads. In Section 6, we show that our framework is powerful and can induce different behaviours of three well-known binary search trees. Finally, we conclude in Section 7.

2. Existing Benchmarks for Concurrent Data Structures

In this work, we consider a key-value concurrent data structure as our target.

2.1. Microbenchmarks

The most common approach for evaluating concurrent key-value data structures is stress-test microbenchmarks. Two of the most popular are Synchrobench [9] and Setbench [6]. Figure 1 depicts the general structure of these microbenchmarks: Each thread executes the same loop (ThreadLoop) on each thread, which employs a pseudo-random 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 also supports a Zipfian [11] distribution, to simulate some keys being "hot". In both cases, operands are word-sized.

The motivation for such a workload is that it serves as a "stress test," highlighting the scalability of the data structure when it is under heavy load. If a data structure has lower latency at one thread, and higher throughput at high thread counts, then it is clearly "best." While stress tests are an important tool for data structure evaluation,

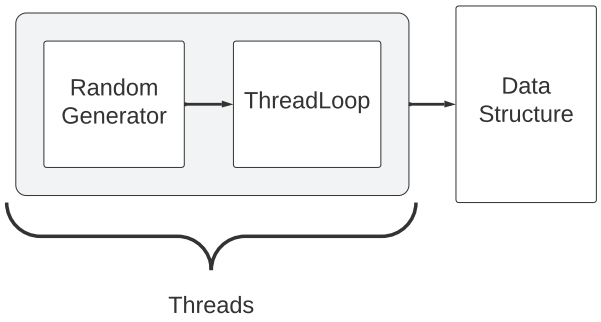

Figure 1: Structure of Syncrobench and Setbench

they are not without flaws. First, for large key ranges, the uniform distribution means that operations rarely contend, even at high thread counts. Second, the lack of non-data structure work means that essential components of the data structure may never be evicted from the cache. Third, the data structure is typically "warmed up" so that its size remains roughly constant throughout the execution; this can suppress essential behaviors, like tree rebalancing.

Still, stress-test microbenchmarks are surprisingly complex. They require many parameters for defining a data structure's configuration (e.g., hash table size and rehash threshold; size of B-tree nodes, split and merge thresholds, and sorting; a number of skip list index layers). They also require many parameters for describing the experiment itself (how to warm up the data structure, whether threads run for a fixed time or a fixed number of operations), and for describing the stress workload (key range, key distribution, operation distribution). In addition, there are external configuration dependencies, most notably on the allocator and memory reclamation techniques.

2.2. Benchmark Suites

In contrast to stress-test 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. Note that Setbench also includes one workload each from YCSB and TPC (specifically, from TPC-C).

YCSB is targeted primarily at cloud-scale data storage systems, but its workloads can be extracted and applied directly to data structures. It supports 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 online 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.

3. Requirements for a Concurrent Data Structure Benchmark

Stemming from experience with the above microbenchmarks and suites, we contend that there are five reasonable requirements for a benchmarking suite:

1) Evolution: 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]; it should allow different threads to perform different operation mixes;
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.

In each dimension, we aim to exceed the state of the art, as discussed below.

Evolution: We emphasize modularity, so that users can add new benchmarks in a declarative fashion. In particular, we ensure that users do not need to re-write logic for launching threads, reading configuration, calculating statistics, etc.

Duration: Some of the past benchmarking suites tend to support timed runs or fixed numbers of operations, not both. They also may tightly couple the data structure warm-up with the operation count and distribution, i.e., to ensure that a target fraction of `remove` operations will succeed. A key facet of our design is to support robust declaration of warm-up, and to enable the user to declare the type of execution (duration, total operation count, per-thread operation count, infinite).

Heterogeneity: While Synchrobench is able to track a single "maintenance" thread that executes a different workload

than the others, we find that otherwise there is limited support for heterogeneity: in essence, there is a many-to-one relationship between threads and workloads to execute. By introducing a JSON-based declarative description of a workload, we enable many-to-many mappings.

Realism: Every benchmark tries to model the real world to some degree, so the greatest challenge is when hard-coded decisions limit the ability to produce experiments that model a new facet of the real world. For example, TPC-C only supports two distributions, Uniform and NURandom (non-uniform random). YCSB can require that *all bytes* of a value are accessed by an operation, and also allows configuring the number of bytes. TPC-C includes work between data structure operations, which can increase realism by causing nontrivial changes to the cache. Through modularity and a declarative interface, we aim to support even greater flexibility (e.g., introducing new distributions, key types, and non-data-structure access patterns).

Exploration: Modern data structures have tuning parameters, and during design, it is important that developers can modify these parameters without re-compiling their programs. Similarly, workload designers should be able to iterate on a workload's configuration until its behavior models a real-world system. Through our declarative JSON interface, we aim to address these needs.

4. Software Design

One of the biggest challenges when designing a flexible benchmark system is to manage complexity. Figure 1 depicts the design of a stress-test microbenchmark, such as Synchrobench or Setbench: (1) Each thread has its own PRNG (Random Generator) and executes the same ThreadLoop code; (2) The ThreadLoop hard-codes how the thread interacts with the Data Structure; (3) Command-line arguments allow a small degree of customization of the ThreadLoop behavior. When considering the six workloads of YCSB, or the behavior of TPC-C, the structure is mostly the same: the benchmark behavior is encoded in a single ThreadLoop that is lightly parameterized.

In Figure 2, we manage the complexity of our more flexible benchmark through a top-down design. Each thread (gray box) is assigned its own ThreadLoop. Each ThreadLoop, in turn, is assigned a set of configurations, which correspond to the operations it will run (light blue box). Each operation generates its arguments via a set of PRNGs, distributions over those PRNGs, and mapping functions for converting the output of a distribution into a key or value. Note that for simplicity, we depict a tree, but it is possible for a ThreadLoop to share a PRNG, DataMap, or distribution among its blue boxes, and even for a read-only DataMap to be shared among ThreadLoops.

To recap, the key entites are:

- `Distribution` – a distribution of a random variable
- `DataMap` – for converting a distribution's output into a key
- `ArgsGenerator` – creates operands for an operation

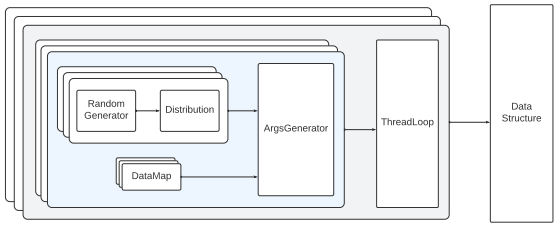

Figure 2: Structure of our benchmark

- `ThreadLoop` – the logic for interacting with a data structure.

In addition, a read-only `GlobalParameters` object stores additional configuration.

4.1. Overview of an Execution

The benchmark initializes a ThreadLoop for each thread. Each ThreadLoop runs for a duration or number of operations according to the `GlobalParameters`. In each iteration, it chooses a query to execute next by choosing an operation, getting a key and value from some instances of ArgsGenerator, and performing the operation with that argument on the data structure. Each instance of ArgsGenerator uses several Distrubutions and DataMaps to generate keys for 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.

Prior to the execution, we support pre-filling the data structure. We do so via the same technique: we initialize several threads with appropriate ThreadLoops, and then run the resulting workload until some condition is satisfied (e.g., the data structure reaches a target size). Note that both sequential and multithreaded pre-filling is possible. This is important, since NUMA systems expect objects/nodes to be distributed evenly over different sockets, which is only achieved by threads on those sockets creating/inserting those objects [10].

4.2. Entities

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

Distributions: The lowest-level entity is the Distribution, which is used to simulate some 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. We also provide a MutableDistribution, which can change the random variable in run-time by modifying the range of keys: `setRange(int range)` and `next(int range)`. Default distributions in our benchmark include uniform, zipfian, and gaussian.

DataMaps: The DataMap is used by an ArgsGenerator to translate an index into a key or value. We have several implementations of a DataMap: a shuffled array, a hash, and the identity map.

ArgsGenerators: ArgsGenerators are used to generate keys and values 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: a thread-private ArgsGenerator can use a distribution to occasionally select recently accessed keys without extra synchronization.

ThreadLoops: The ThreadLoop decides which operation a thread should execute next. It is initialized for each thread separately and uses the described ArgsGenerators. 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 phase. The method `step()` is called in an conditional loop by the `run()` method. A programmer can override `run()` when more complicated logic is needed.

The default ThreadLoop randomly chooses the next operation using a uniform distribution and the `GlobalParameters`, receives the key for the selected operation from the simplest ArgsGenerator with the uniform distribution and the identity datamap, and executes it. Note that 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.

4.3. Entities in code

We currently provide Java and C++ versions of our benchmark tool. 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 begins, it processes the JSON configuration and initializes each ThreadLoop. This step makes use of the builder pattern to reduce complexity. 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).

4.3.1. Implementing a new entity. Each entity consists of three parts: its logic, its Builder, and a Parameters object. To give a sense for the effort involved in creating a new, reusable entity, we present a generic ExampleEntity class.

```
class ExampleEntity implements Entity:
  ExampleEntity(entityParameters,
                globalParameters, ...):
    {...}

  *implementation*
```

Then, an automated, language-specific step makes the entity visible, e.g.,:

```
enum EntityType:
  {...}, EXAMPLE_ENTITY
```

The programmer must provide a corresponding Parameters class with an `init` function that uses the global parameters. This function provides all the necessary parameters for associated Builder classes.

```
class ExampleEntityParameters
      implements EntityParameters:
  *parameters*

  void init(globalParameters):
    ...
```

Lastly, the programmer produces a Builder class with a `build` function. Please note that `EntityBuilder` also provides `init` function that initializes all the parameters.

```
class EntityBuilder:
  EntityType entityType
  EntityParameters entityParameters
  GlobalParameters globalParameters

  *fields setter methods*

  void init(globalParameters):
    this.globalParameters = globalParameters
    this.entityParameters.init(globalParameters)

  void init(globalParameters,
            entityParameters):
    this.globalParameters = globalParameters
    this.entityParameters = entityParameters
    this.entityParameters.init(globalParameters)

  Entity build(*special parameters*):
    switch (entityType):
      case EXAMPLE_ENTITY:
        return new ExampleEntity({
                    entityParameters,
                    globalParameters,
                    *special parameters*,
                    ...})
      case {...}
```

4.4. Example

We now present a complete example, by adding the skewed read-update workload from [3] in our suite. This workload is used for testing key-value data structures supporting three operations `insert`, `remove`, and `get`.

This workload is specified by five parameters:

- n, the size of the working set of keys;
- $w\%$, the amount of updates in the total number of operations;
- $x\%$ of get operations choose a key uniformly at random from a random subset of keys of proportion $y\%$, while other get operations choose a random key from the rest of the set;
- `insert` and `remove` operations choose a key uniformly at random from a random subset of keys of proportion $s\%$.

At a high level, we choose n integer keys and name them as set A. Then we pre-populate the index: we add a key from A with probability 50%. Then, we choose $s \cdot n$ keys uniformly at random to produce a key set U. Also, we choose $y \cdot n$ keys from *inserted* keys to produce a key

set named R. Each process chooses an operation: with probability $100 - w\%$ it chooses `get` and with probabilities $\frac{w}{2}\%$ it chooses `insert` or `remove`. Now, the process has to choose an argument of the operation: `get` it chooses an argument from R with probability $x\%$, otherwise, it chooses an argument from $S \setminus R$; `insert` and `remove` operations choose an argument from U uniformly at random.

Now, we give the full implementation.

4.4.1. Distribution. To implement this workload, we need two distributions: a Uniform Distribution for the arguments of update requests and a new distribution for the arguments of read requests explained above, named as Skewed Uniform Distribution. The Uniform Distribution is simple and is already available in our suite. We now add the Skewed Uniform Distribution. Together with a DataMap, it is used to generate the keys for get operations in the ArgsGenerator.

The distribution type gets added to the corresponding enum:

```
enum DistributionType:
  {...}, SKEWED_UNIFORM
```

Then, we need to provide parameters $x\%$ and $y\%$ to this distribution. Since, parameter y represents only the percentage of "hot" keys, we initialize parameter *hotLength*, which stores the number of "hot" keys, in `init`. For that, we create a Parameters class:

```
class SkewedUniformParameters
        implements DistributionParameters:
  double HOT_SIZE = 0 // y%
  double HOT_PROB = 0 // x%
  int hotLength = 0

  SkewedUniformParameters(HOT_SIZE, HOT_PROB):
    this.HOT_SIZE = HOT_SIZE
    this.HOT_PROB = HOT_PROB

  void init(globalParameters):
    hotLength = HOT_SIZE * globalParameters.range
```

We are ready to implement SkewedUniformDistribution. Since we need to choose a value uniformly from both sets $[0, y \cdot n)$ and $[y \cdot n, n)$, corresponding to $x\%$ and $100 - x\%$, we use two different Uniform Distribution objects.

```
class SkewedUniformDistribution
            implements Distribution:
  int hotLength
  double hotProb
  UniformDistribution hotDist
  UniformDistribution coldDist
  Random random

  int next():
    if (random.nextDouble() < hotProb):
      return hotDist.next()
    else:
      return hotLength + coldDist.next()
```

Distribution returns only a random value, not an operation argument. We will use a DataMap to transform it into a key. To finish the implementation Distribution, we update the `build` function in the DistributionBuilder.

```
Distribution build(int range):
  switch (this.distributionType) {
    case SKEWED_UNIFORM:
      return new SkewedUniformDistribution(
        this.distParameters.hotLength,
        this.distParameters.HOT_PROB,
        new DistributionBuilder(UNIFORM)
          .build(this.distParameters.hotLength),
        new DistributionBuilder(UNIFORM)
          .build(range -
                 this.distParameters.hotLength))
    case {...}
```

4.4.2. DataMap. Since the Distribution returns random variables, the ArgsGenerator needs a DataMap to convert them into keys. In this case, the keys are randomly distributed over sets of size $x\%$ and $100 - x\%$. The ArrayDataMap takes all keys from the entire range and shuffles them. When given a value from a Distribution it maps it to the corresponding element of the array.

To implement the new DataMap we perform the usual steps: 1) add a new type into an enum; 2) implement the DataMap interface; and 3) update a DataMapBuilder class. Note that our new DataMap does not depend on additional parameters, so there is no need for a corresponding Parameters class.

```
enum DataMapType:
  {...}, ARRAY


class ArrayDataMap implements DataMap:
  int[] data

  ArrayDataMap(int range):
    for (i = 0; i < range; i++):
      data[i] = i + 1
    random.shuffle(data)

  int get(int index):
    return data[index]


DataMap build(int range):
  switch (this.dataMapType):
    case ARRAY:
      return new ArrayDataMap(range)
    case {...}
```

4.4.3. ArgsGenerator. Given our Distribution and DataMap, we can create an ArgsGenerator. The steps are: 1) announce it in the enum; 2) write the Parameters class; 3) implement an interface; and 4) update the ArgsGeneratorBuilder class.

First, we make it visible through language-specific tooling:

```
enum ArgsGeneratorType:
  {...}, EXAMPLE_ARGS_GEN
```

Our ArgsGenerator takes three parameters, x, y, and s, from above. x and y are used for the Skewed Uniform Distribution and s is used for the Uniform Distribution. Parameters x, y, and s are loaded either by parsing the command line or by reading a JSON file.

In this case, ArgsGenerator uses separate DataMap objects for read and update operations. However, all ArgsGenerators objects should have the same DataMap instance; this is expressed in the Parameters class.

```
class ExampleParameters
        extends ArgsGeneratorParameters:
  double x, y, s

  DistributionBuilder getDistBuilder =
    new DistributionBuilder(SKEWED_UNIFORM)
  DistributionBuilder updateDistBuilder =
    new DistributionBuilder(UNIFORM)

  DataMap getDataMap
  DataMap updateDataMap

  void init(globalParameters):
    skewParameters =
          new SkewedUniformParameters(x, y)
    getDistBuilder.init(globalParameters,
          skewParameters)
    updateDistBuilder.init(globalParameters)

    getDataMap = new DataMapBuilder(ARRAY)
                  .build(globalParameters)
    updateDataMap = new DataMapBuilder(ARRAY)
                     .build(globalParameters)
```

This workload employs a single, stateful ArgsGenerator, so that operations' operands can depend on each other. ArgsGenerator takes the index generated by the Distribution and passes it to the DataMap to get a key.

```
class ExampleArgsGenerator
      implements ArgsGenerator:
  DataMap getData
  DataMap updateData
  Distribution getDist
  Distribution updateDist

  ExampleArgsGenerator(DataMap getData,
        DataMap updateData,
        Distribution getDist,
        Distribution updateDist)

  int nextGet():
    return getData.get(getDist.next())

  int nextInsert():
    return updateData.get(updateDist.next())

  int nextRemove():
    return updateData.get(updateDist.next())
```

Finally, we need to update the `build()` function in ArgsGeneratorBuilder.

```
ArgsGenerator build():
  switch (this.argsGeneratorType):
    case EXAMPLE_ARGS_GEN:
      range = this.globalParameters.range;
      return new ExampleArgsGenerator(
        this.argsGenParameters.getDataMap,
        this.argsGenParameters.updateDataMap,
        this.argsGenParameters.getDistBuilder
            .build(range),
        this.argsGenParameters.updateDistBuilder
            .build(this.argsGenParameters.s * range))
```

4.4.4. ThreadLoop. The implementation of a new ThreadLoop again consists of four parts: 1) add a new type to the enum; 2) write the Parameters class; 3) implement the `step()` or `run()` methods; and 4) update `build` method in the ThreadLoopBuilder class.

In this case we have only one parameter — $w\%$, which means the probability of an update operation. Also, our new ThreadLoop needs only one ArgsGenerator. w can be loaded from the command line or a JSON file.

```
enum ThreadLoopType:
  {...}, DEFAULT

class DefaultThreadLoopParameters
            extends ThreadLoopParameters:
  double numWrites // w%
  double numInsert
  double numRemove

  ArgsGeneratorBuilder argsGeneratorBuilder

  void init(globalParameters):
    numInsert = numRemove = numWrites / 2
    argsGeneratorBuilder.init(globalParameters)
```

The `step()` method selects a next operation and executes it. The `run()` method calls the `step()` method while the stop flag is false. The ThreadLoopAbstract class implements methods: `executeInsert(key)`, `executeRemove(key)`, and `executeGet(key)`, which execute the corresponding method on the data structure and calculate the statistics.

```
class DefaultThreadLoop
        extends ThreadLoopAbstract:
  DefaultThreadLoopParameters params
  ArgsGenerator argsGen
  Random rand

  DefaultThreadLoop(dataStructure,
        threadLoopParameters):
    super(dataStructure)
    this.params = threadLoopParameters
    this.argsGen = params.argsGeneratorBuilder
                        .build()
    this.rand = new Random()

  void step():
    double coin = rand.nextDouble();
    if (coin < params.numInsert):
      // 1. should we run an insert
      int key = argsGen.nextInsert()
      this.executeInsert(key)
    else if (coin < params.numRemove):
      // 2. should we run a remove
      int key = argsGen.nextRemove()
      this.executeRemove(key)
    else:
      //3. then we should run a get
      int key = argsGen.nextGet();
      this.executeGet(key);
```

Finally, we need to update the ThreadLoopBuilder class.

```
ThreadLoop build(DataStructure<K> dataStructure):
  switch (this.threadLoopType):
    case DEFAULT:
      return new DefaultThreadLoop(
        threadNum,
        dataStructure,
```

```
        this.threadLoopParameters)
    case {...}
```

5. 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).

5.1. 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 (e.g., it can generate a random value using `random.nextInt(range)`).

Standard Distributions. We include a configurable Gaussian distribution, as well as a distribution based on Zipf's law [11].

Skewed Uniform Distribution. We also provide, by default, the Skewed Uniform Distribution from Section 4.4.1.

5.2. 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 peculiarities in some research data structures, when given `i`, it 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 calling the `get(index)` method, returns the corresponding element from the array.

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

5.3. ArgsGenerators

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 (Section 4.4.3) 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.

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. It has two types of states:

- k-th *excited* state — the keys are selected using k-th SkewedUniformDistribution, i.e., there is always 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. The TemporarySkewedArgsGenerator uses the following parameters:

- *state-count* — Total number of excited states in the workload;
- *ht* — Default duration of an excited state (the duration is specified in the number of operations);
- *rt* — 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 models YCSB workload D. The key feature of the Creakers and Wave ArgsGenerator is the entity Wave. The Wave is a subset of all keys from the range with 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 this tail by one;
- `nextInsert()` — select the key next to the head of the Wave, and make it the new head;
- `nextGet()` — select a key from the Wave according to some specified distribution.

As a default, Wave uses the Zipfian Distribution, 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;
- *c-distribution* — a distribution of keys in the Creakers entity (by default, the Uniform distribution);
- *w-distribution* — a mutable distribution of 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 models workload D from YCSB, but adds the ability to execute an infinite workload instead of a fixed number of operations. Furthermore, unlike YCSB, it achieves this behavior with an $O(1)$ complexity for generating arguments. Similarly, prior benchmarks have tried to produce temporal workloads by associating a "hotness" count with each element; for infinite workloads, removal discards this count, leading to the Creakers ultimately having the largest counts. In contrast, our design makes the two key sets explicit, and thus produces reliable results regardless of the duration of the experiment.

Leafs Handshake ArgsGenerator. The Leafs Handshake ArgsGenerator is stateful and mutable: the key selection for an insert operation is based on the argument of the previous 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 $\alpha = 1$).

5.4. 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;

6. 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.

6.1. 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 a working thread always removes nodes physically and rotates subtrees, if necessary. In the CF-BST a working thread makes only logical removals. However, there is a special daemon thread responsible for physical removals and rotations. In the CO-BST a thread performs physical removal immediately but does not perform rotations at all.

Prior papers using the workloads from Synchrobench, 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.

6.2. 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.

6.2.1. Uniform and Zipfian Workloads. Uniform and Zipfian Workloads are the most common in 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.

6.2.2. Infinite Leafs Handshake Workload. We now consider the Infinite Leafs Handshake workload. It is based on the Leafs Handshake ArgsGenerator and the Temporary Operations ThreadLoop presented in Section 5. It has three time intervals: 1) the filling interval — there are more

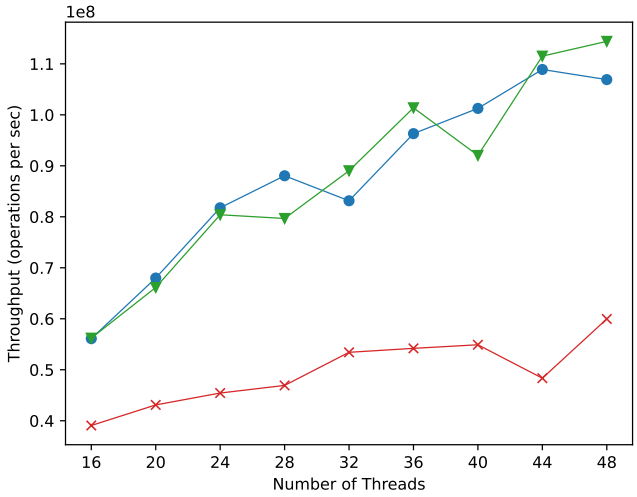

Figure 3: Uniform Workload with range 10^4 and update ratio 5%

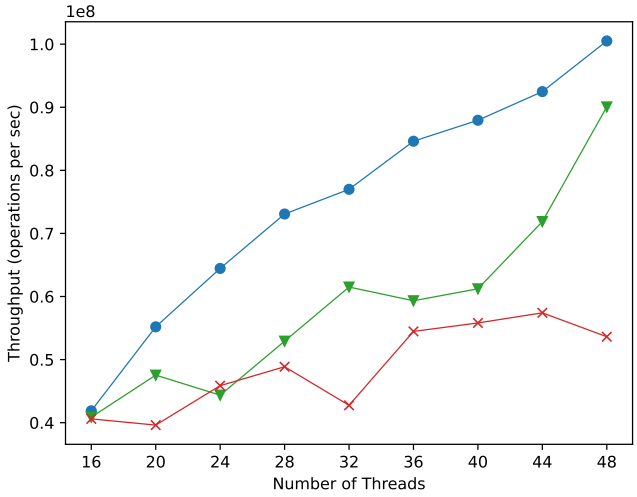

Figure 4: Zipfian Workload with $\alpha = 1$, range 10^5, and update ratio 5%.

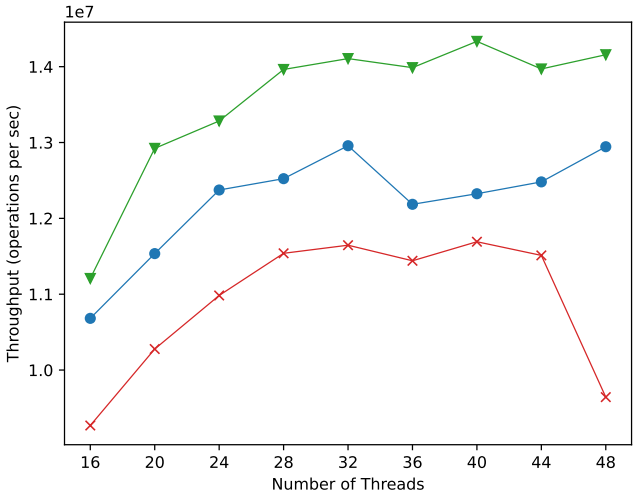

Figure 5: Infinite Leafs Handshake Workload with range 10^5.

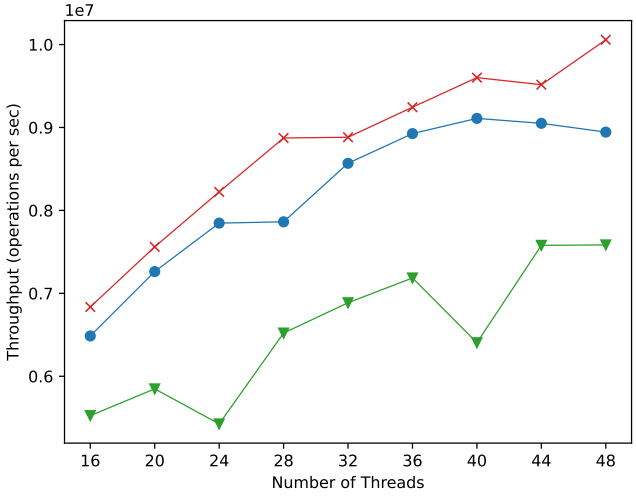

Figure 6: Infinite Leafs Handshake Workload with range 10^7.

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.

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; $ot_0 = ot_1 = ot_2 = 100000$; $ui_0 = ue_2 = 80\%$; $ui_2 = ue_0 = 20\%$; $ui_1 = ue_1 = 0\%$; 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

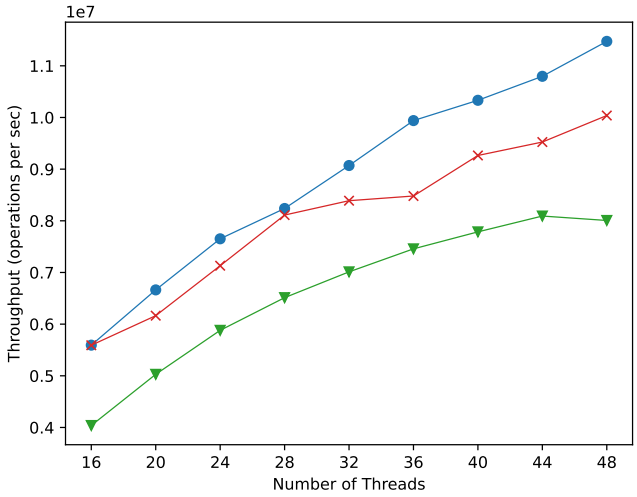

Figure 7: Infinite Leafs Handshake Workload with range 10^8

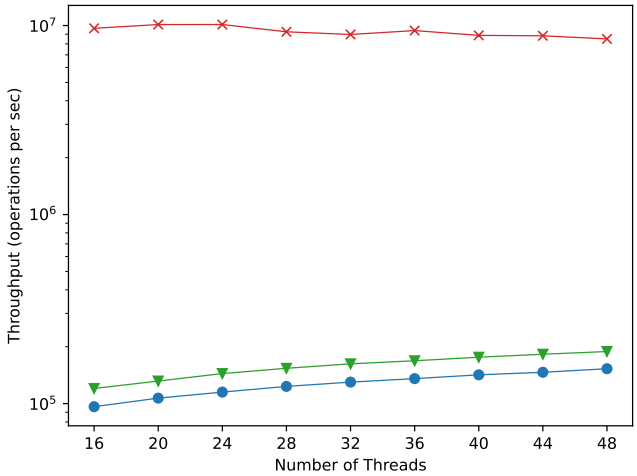

Figure 8: Non-shuffle Wave Workload with range 10^6.

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.

6.2.3. 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.

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 CF-B-CO.

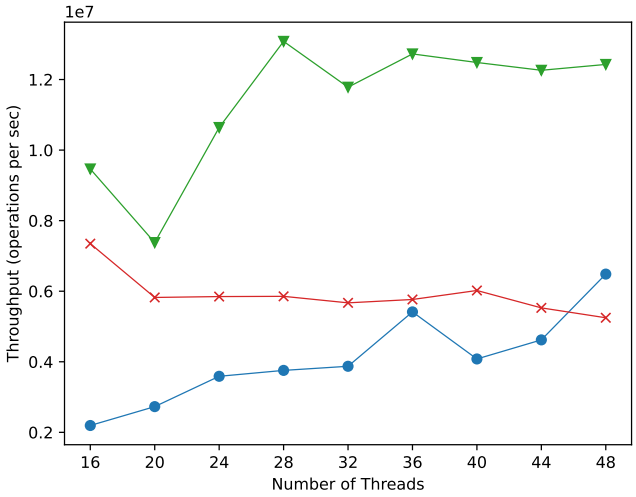

Figure 9: Non-shuffle Wave Workload with range $5 \cdot 10^3$.

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 B-CF-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.

6.3. 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 8, CF-CO-B on Figure 5, CO-B-CF on Figure 7, and CO-CF-B on Figures 3 and 4.

7. 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] "Tpc benchmarks," https://www.tpc.org/.

[2] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, "The cb tree: a practical concurrent self-adjusting search tree," *Distributed computing*, vol. 27, no. 6, pp. 393–417, 2014.

[3] V. Aksenov, D. Alistarh, A. Drozdova, and A. Mohtashami, "The splay-list: A distribution-adaptive concurrent skip-list," *Distributed Computing*, pp. 1–24, 2023.

[4] V. Aksenov, V. Gramoli, P. Kuznetsov, A. Malova, and S. Ravi, "A concurrency-optimal binary search tree," in *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28–September 1, 2017, Proceedings 23*. Springer, 2017, pp. 580–593.

[5] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 257–268, 2010.

[6] T. Brown, A. Prokopec, and D. Alistarh, "Non-blocking interpolation search trees with doubly-logarithmic running time," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 276–291.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[8] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19*. Springer, 2013, pp. 229–240.

[9] V. 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*, 2015, pp. 1–10.

[10] R. Kharal and T. Brown, "Performance anomalies in concurrent data structure microbenchmarks," *arXiv preprint arXiv:2208.08469*, 2022.

[11] D. M. Powers, "Applications and explanations of zipf's law," in *New methods in language processing and computational natural language learning*, 1998.