Presentation

Benchmarking Concurrent Data Structures

Elias Papavasileiou

University of Crete, Department of Computer Science

eliaspap@csd.uoc.gr

May 11, 2019

Definition & Motivation

What is a benchmark ?

The act of executing carefully designed *test runs* of an algorithm, to evaluate its performance.

Definition & Motivation

What is a benchmark ?

The act of executing carefully designed *test runs* of an algorithm, to evaluate its performance.

Why benchmark a concurrent data structure (CDS) ?

- To evaluate its **absolute performance**. For example, measure how many Find operations it can execute in a given amount of time.

Definition & Motivation

What is a benchmark ?

The act of executing carefully designed *test runs* of an algorithm, to evaluate its performance.

Why benchmark a concurrent data structure (CDS) ?

- To evaluate its **absolute performance**. For example, measure how many Find operations it can execute in a given amount of time.
- To evaluate its **relevant performance**, i.e. compare its absolute performance to that of other CDSs, and determine: Which CDS performs better, how much better it performs, and for which test runs that happens.

Throughput

- CDSes usually support at least the three fundamental operations: Insert, Delete and Find.

Throughput

- CDSes usually support at least the three fundamental operations: Insert, Delete and Find.
- To measure the performance of a CDS operation (i.e. determine how "fast" a CDS operation is) we calculate the **Throughput** of that operation:

$$\text{Throughput} = \frac{\text{Number of completed operation calls}}{\text{Time interval in which they were completed}}$$

Throughput

- CDSes usually support at least the three fundamental operations: Insert, Delete and Find.
- To measure the performance of a CDS operation (i.e. determine how "fast" a CDS operation is) we calculate the **Throughput** of that operation:

$$\text{Throughput} = \frac{\text{Number of completed operation calls}}{\text{Time interval in which they were completed}}$$

- Completed operation calls include both successful and unsuccessful calls.

Throughput

- For example, to measure the performance of Finds, we calculate the throughput of Finds:

$$\text{Find throughput} = \frac{\text{Number of completed Find operation calls}}{\text{Time interval in which they were completed}}$$

Throughput

- For example, to measure the performance of Finds, we calculate the throughput of Finds:

$$\text{Find throughput} = \frac{\text{Number of completed Find operation calls}}{\text{Time interval in which they were completed}}$$

- To measure the performance of the CDS as a whole, we calculate the **total** throughput:

$$\text{Total throughput} = \frac{\text{Number of completed Find, Insert and Delete operation calls}}{\text{Time interval in which they were completed}}$$

Methodology

How should we run the benchmark ?

Methodology

How should we run the benchmark ?

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

Methodology

How should we run the benchmark ?

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- Fix either Operation calls or Time, and measure the other

Methodology

How should we run the benchmark ?

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- Fix either Operation calls or Time, and measure the other
- Let's say we fix Operation calls, and measure time

Methodology

How should we run the benchmark ?

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- Fix either Operation calls or Time, and measure the other
- Let's say we fix Operation calls, and measure time

Do you see any problem ?

Methodology

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- To ensure that all threads run **in parallel all the time**, we fix Time and measure Operation calls.

Methodology

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- To ensure that all threads run **in parallel all the time**, we fix Time and measure Operation calls.
- Suppose that we want to measure the throughput of Insert and Delete operations.

Methodology

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- To ensure that all threads run **in parallel all the time**, we fix Time and measure Operation calls.
- Suppose that we want to measure the throughput of Insert and Delete operations.
- 1st approach: Half of the threads execute Insert operations and the other half Delete operations.

Methodology

$$\text{Throughput} = \frac{\text{Operation calls}}{\text{Time}}$$

- To ensure that all threads run **in parallel all the time**, we fix Time and measure Operation calls.
- Suppose that we want to measure the throughput of Insert and Delete operations.
- 1st approach: Half of the threads execute Insert operations and the other half Delete operations.

Do you see any problem ?

Operation Mixes

- Due to the **producer-consumer problem**, the CDS might end up being empty or full, all the time.

Operation Mixes

- Due to the **producer-consumer problem**, the CDS might end up being empty or full, all the time.
- How can we make the CDS maintain a constant size during the whole benchmark process ?

Operation Mixes

- Due to the **producer-consumer problem**, the CDS might end up being empty or full, all the time.
- How can we make the CDS maintain a constant size during the whole benchmark process ?
- This can be achieved by creating *operation mixes*.

Operation Mixes

- An **operation mix** is the group of variables i, d, f and r

Operation Mixes

- An **operation mix** is the group of variables i, d, f and r
- i, d and f are the probabilities for a thread to execute an Insert, Delete or Find operation, respectively.

Operation Mixes

- An **operation mix** is the group of variables i, d, f and r
- i, d and f are the probabilities for a thread to execute an Insert, Delete or Find operation, respectively.
- Keys are integers drawn uniformly at random from the range [1,r]

Operation Mixes

- An **operation mix** is the group of variables i, d, f and r
- i, d and f are the probabilities for a thread to execute an Insert, Delete or Find operation, respectively.
- Keys are integers drawn uniformly at random from the range [1,r]
- This way, the CDS maintains an **approximately constant size** of $\frac{ri}{i+d}$ keys throughout the benchmark.

Experiments

- An **experiment** is an instance of an operation mix.

Experiments

- An **experiment** is an instance of an operation mix.
- Suppose that we want to measure the throughput of Insert and Delete operations, as before.

Experiments

- An **experiment** is an instance of an operation mix.
- Suppose that we want to measure the throughput of Insert and Delete operations, as before.
- The operation mix will be: i=0.5, d=0.5, f=0

Experiments

- An **experiment** is an instance of an operation mix.
- Suppose that we want to measure the throughput of Insert and Delete operations, as before.
- The operation mix will be: i=0.5, d=0.5, f=0
- We know that the CDS size will be: $\frac{ri}{i+d} = 0.5r$

Experiments

- An **experiment** is an instance of an operation mix.
- Suppose that we want to measure the throughput of Insert and Delete operations, as before.
- The operation mix will be: i=0.5, d=0.5, f=0
- We know that the CDS size will be: $\frac{ri}{i+d} = 0.5r$
- By selecting a value for r, we can control the CDS size

Experiments

- An **experiment** is an instance of an operation mix.
- Suppose that we want to measure the throughput of Insert and Delete operations, as before.
- The operation mix will be: i=0.5, d=0.5, f=0
- We know that the CDS size will be: $\frac{ri}{i+d} = 0.5r$
- By selecting a value for r, we can control the CDS size
- For example, for $r = 10^6$ we get a quite big CDS size of $5 \cdot 10^5$ keys.

Experiments

To make the experiments more reliable:

Experiments

To make the experiments more reliable:

1) The CDS is prefilled with random keys until it reaches its expected constant size.

Experiments

To make the experiments more reliable:

1) The CDS is prefilled with random keys until it reaches its expected constant size.
2) The experiment is repeated many times, and the average throughput is calculated.

Experiments

To make the experiments more reliable:

1) The CDS is prefilled with random keys until it reaches its expected constant size.
2) The experiment is repeated many times, and the average throughput is calculated.
3) Garbage collection should be deactivated !

Correctness tests

The size test:

Correctness tests

The size test:

1) Every thread maintains a thread-local integer variable, the *node_sum* counter

Correctness tests

The size test:

1) Every thread maintains a thread-local integer variable, the *node_sum* counter
2) Every time it performs a successful Insert (or Delete) call, it increments (or decrements) the counter

Correctness tests

The size test:

1) Every thread maintains a thread-local integer variable, the *node_sum* counter
2) Every time it performs a successful Insert (or Delete) call, it increments (or decrements) the counter
3) After the end of the experiment, the main thread sums up the values of all counters

Correctness tests

The size test:

1) Every thread maintains a thread-local integer variable, the *node_sum* counter
2) Every time it performs a successful Insert (or Delete) call, it increments (or decrements) the counter
3) After the end of the experiment, the main thread sums up the values of all counters
4) The result should be equal to the current CDS size (which is found by a simple traversal of the CDS by the main thread)

Correctness tests

The size test:

1) Every thread maintains a thread-local integer variable, the *node_sum* counter
2) Every time it performs a successful Insert (or Delete) call, it increments (or decrements) the counter
3) After the end of the experiment, the main thread sums up the values of all counters
4) The result should be equal to the current CDS size (which is found by a simple traversal of the CDS by the main thread)

Note that this is true assuming that every Insert (or Delete) adds (or removes) 1 node, and the CDS is initially empty ! Otherwise, this test has to be modified accordingly.

Correctness tests

The keysum test:

Correctness tests

The keysum test:

1) Every thread maintains a thread-local integer variable, the *key_sum* counter

Correctness tests

The keysum test:

1) Every thread maintains a thread-local integer variable, the *key_sum* counter
2) Every time it performs a successful Insert(k) (or Delete(k)) call, it adds k to (or subtracts k from) the counter

Correctness tests

The keysum test:

1) Every thread maintains a thread-local integer variable, the *key_sum* counter
2) Every time it performs a successful Insert(k) (or Delete(k)) call, it adds k to (or subtracts k from) the counter
3) After the end of the experiment, the main thread sums up the values of all counters

Correctness tests

The keysum test:

1) Every thread maintains a thread-local integer variable, the *key_sum* counter
2) Every time it performs a successful Insert(k) (or Delete(k)) call, it adds k to (or subtracts k from) the counter
3) After the end of the experiment, the main thread sums up the values of all counters
4) The result should be equal to the sum of the keys that the CDS currently contains (which is found by a simple traversal of the CDS by the main thread)

Graphs

- Among the most common types of graphs is Throughput vs Number of Threads

Graphs

- Among the most common types of graphs is Throughput vs Number of Threads
- It demonstrates how well the CDS **scales** as more and more threads use it concurrently

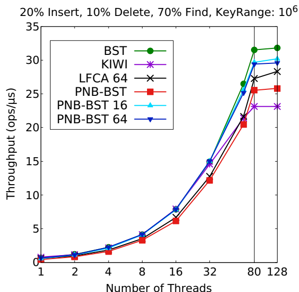
Graphs

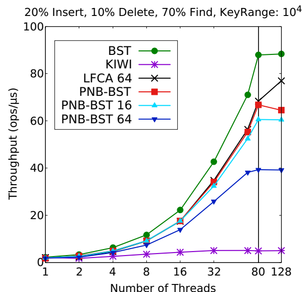
- Among the most common types of graphs is Throughput vs Number of Threads
- It demonstrates how well the CDS **scales** as more and more threads use it concurrently
- All lines are expected to be under the line y=x

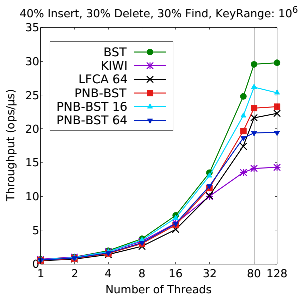
Graphs

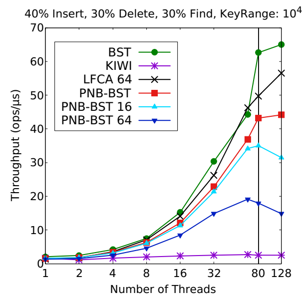
- Among the most common types of graphs is Throughput vs Number of Threads
- It demonstrates how well the CDS **scales** as more and more threads use it concurrently
- All lines are expected to be under the line y=x
- Other graphs are possible as well, depending (not only) on the features of each CDS.

Graphs









Space complexity

- The space complexity can be measured experimentally, by plotting the memory usage over time.

Space complexity

- The space complexity can be measured experimentally, by plotting the memory usage over time.
- Tool: **ps**

Space complexity

- The space complexity can be measured experimentally, by plotting the memory usage over time.
- Tool: **ps**
- ps h -p $java_pid -o rssize

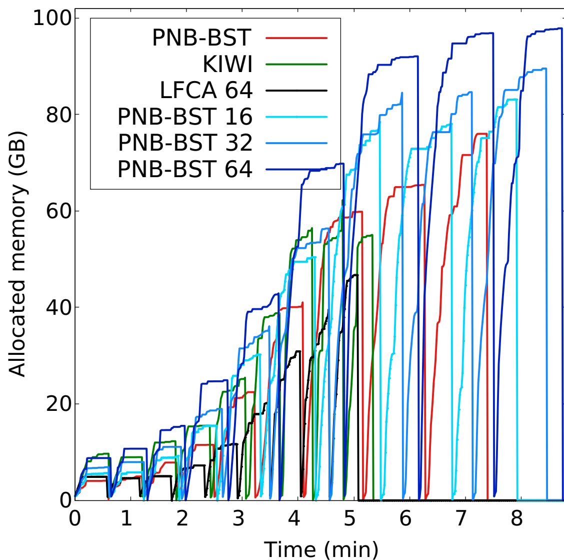
Space complexity

- The space complexity can be measured experimentally, by plotting the memory usage over time.
- Tool: **ps**
- ps h -p $java_pid -o rssize
- We execute the above command once a second, and save the result in a file

Space complexity

- The space complexity can be measured experimentally, by plotting the memory usage over time.
- Tool: **ps**
- ps h -p $java_pid -o rssize
- We execute the above command once a second, and save the result in a file
- After the experiment ends, we can plot the memory usage.

Space complexity



Time complexity

- The performance of a CDS can depend on a lot of factors.

Time complexity

- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.

Time complexity

- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.
- This way, we can estimate how many cache misses every call of an operation produces.

Time complexity

- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.
- This way, we can estimate how many cache misses every call of an operation produces.
- We cannot use a profiler for that ! (It affects the experiment)

Time complexity

- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.
- This way, we can estimate how many cache misses every call of an operation produces.
- We cannot use a profiler for that ! (It affects the experiment)
- Tool: **perf**

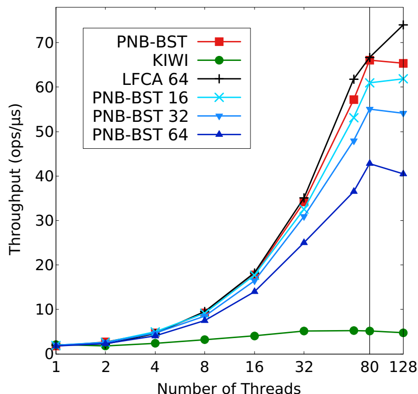
Time complexity

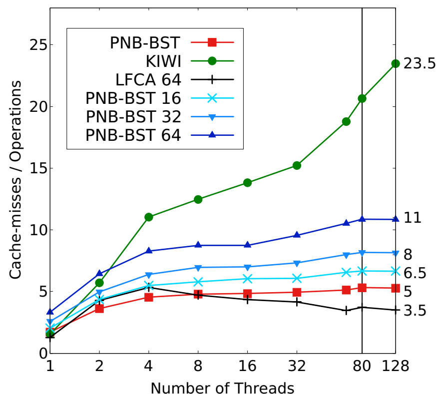
- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.
- This way, we can estimate how many cache misses every call of an operation produces.
- We cannot use a profiler for that ! (It affects the experiment)
- Tool: **perf**
- perf stat -e cache-misses java -jar experiments_instr.jar

Time complexity

- The performance of a CDS can depend on a lot of factors.
- Among the most important ones is the $\frac{\text{cache misses}}{\text{operation calls}}$ rate.
- This way, we can estimate how many cache misses every call of an operation produces.
- We cannot use a profiler for that ! (It affects the experiment)
- Tool: **perf**
- perf stat -e cache-misses java -jar experiments_instr.jar
- Memory footprint can affect the cache miss rate as well !

Time complexity





Finale

The End

Questions ?

Finale

Thank you !

References

Trevor Brown. Benchmark
https://bitbucket.org/trbot86/implementations/src/master/java/src/main/Main.java

Trevor Brown. Experiments
https://bitbucket.org/trbot86/implementations/src/master/java/run-experiments