HoGent

Faculty Business & Information Management

Research of caching strategies in mobile native applications using external data services

Appendix

Frederik De Smedt

Bachelor's thesis submitted to achieve the degree of
Bachelor in the Applied Computer Science

Supervisor:
Joeri Van Herreweghe
Co-supervisor:
Jens Buysse

Organization: —

Academic year: 2015-2016

Examination period 2

Faculty Business & Information Management

Research of caching strategies in mobile native applications using external data services

Appendix

Frederik De Smedt

Bachelor's thesis submitted to achieve the degree of
Bachelor in the Applied Computer Science

Supervisor:
Joeri Van Herreweghe
Co-supervisor:
Jens Buysse

Organization: —

Academic year: 2015-2016

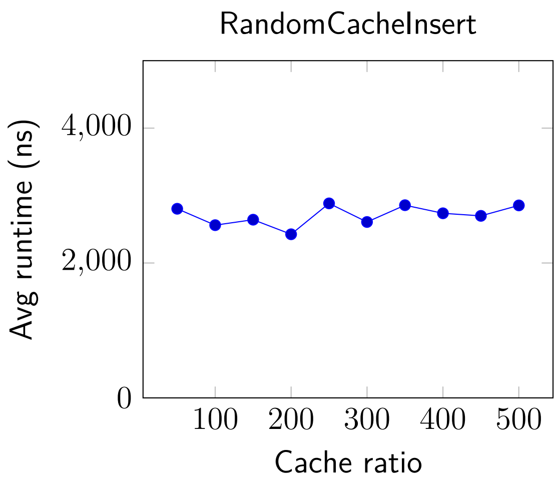
Examination period 2

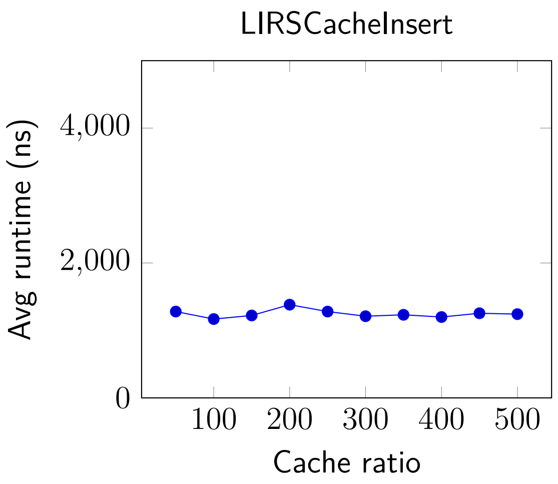
Contents

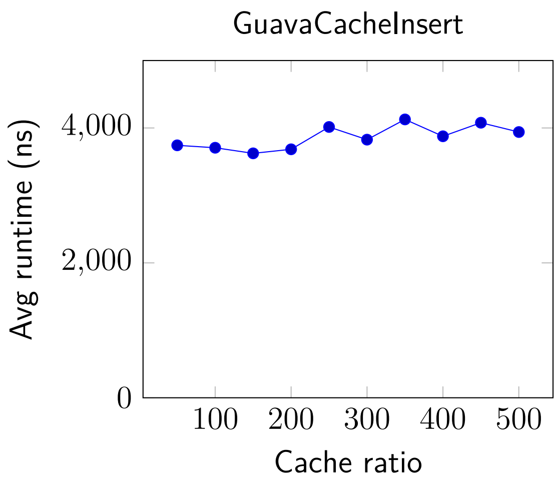
Appendices

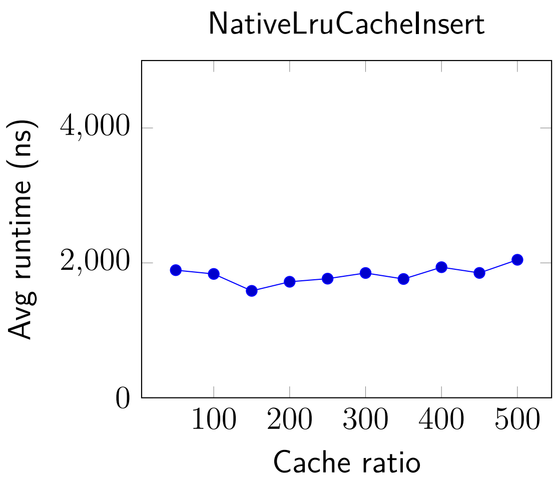
Appendix A

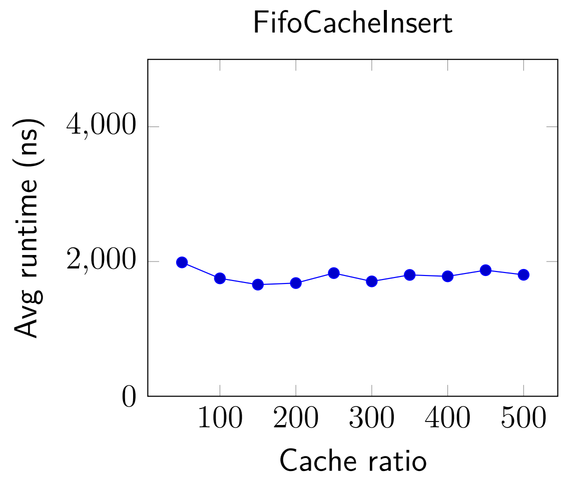
Insert benchmarks



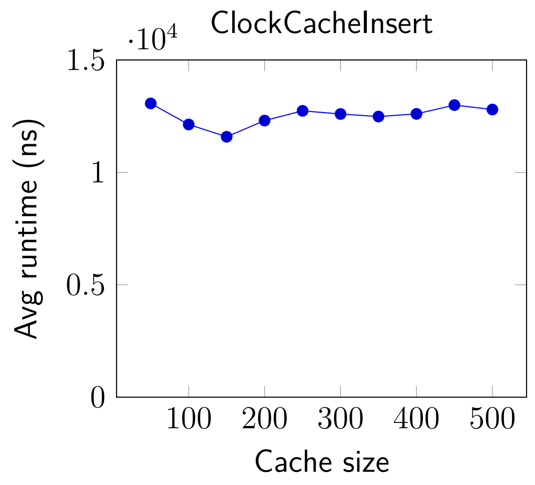


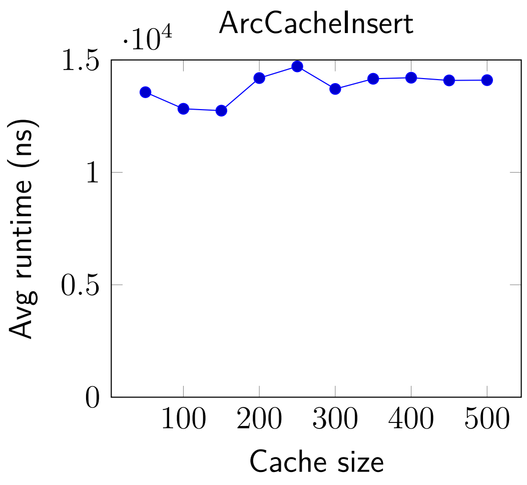






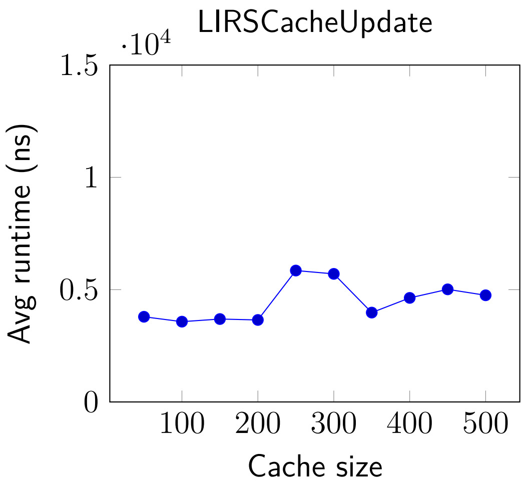
These benchmarks have an average execution time above 5000 ns. Due to the scale difference they are placed separately.

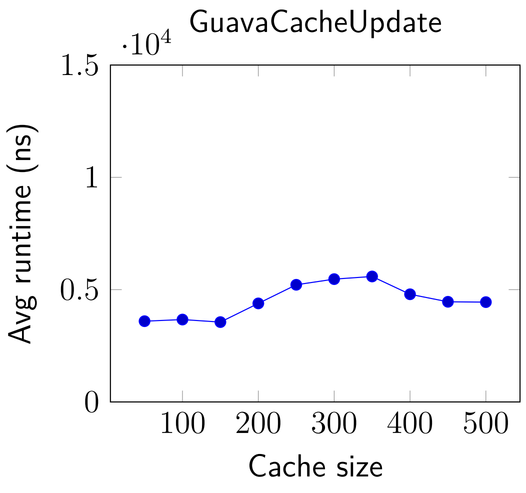


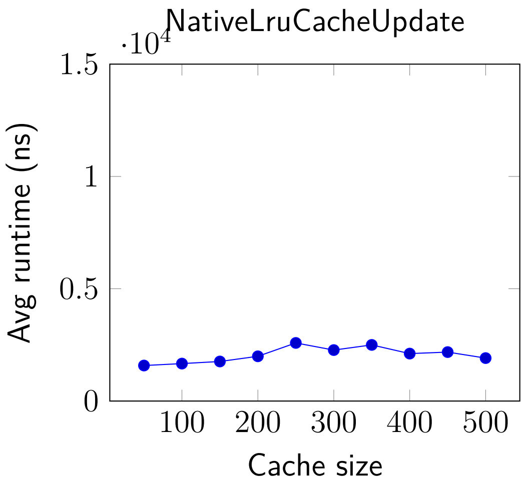


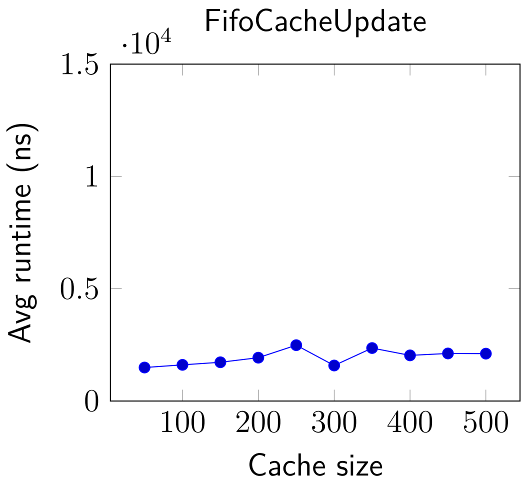
Appendix B

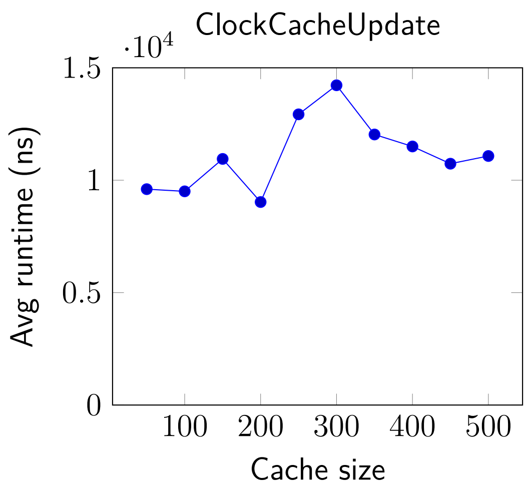
Update benchmarks



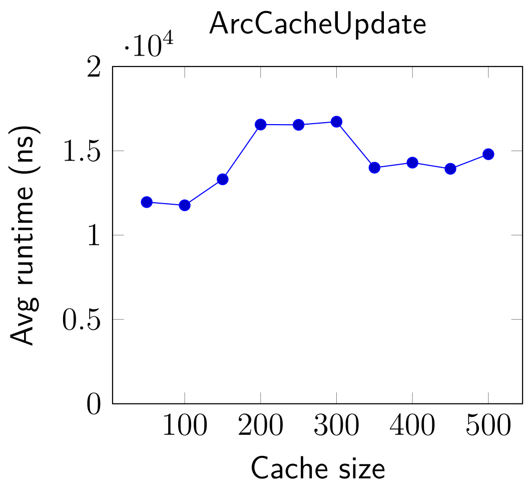


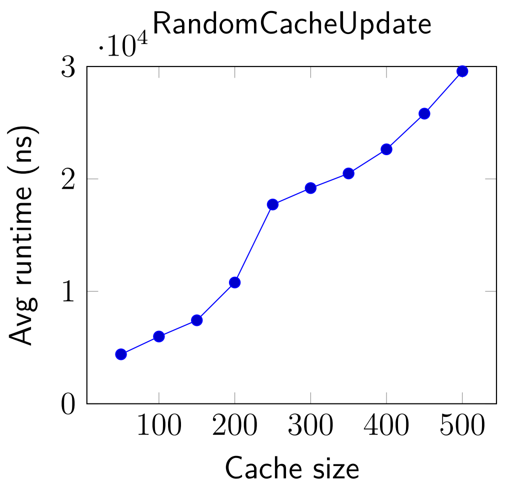






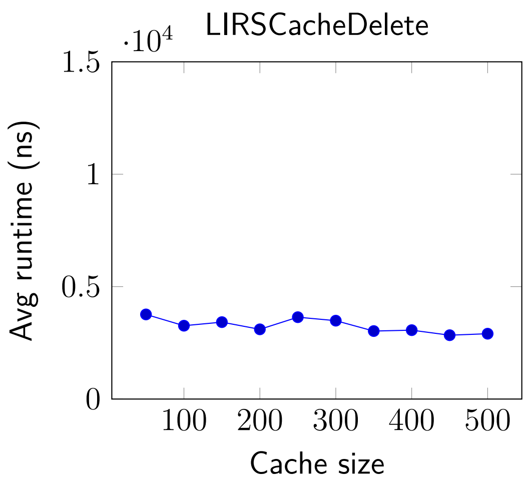
Due to high average runtimes of ARC and random cache, they are placed separately and each have a different scale.

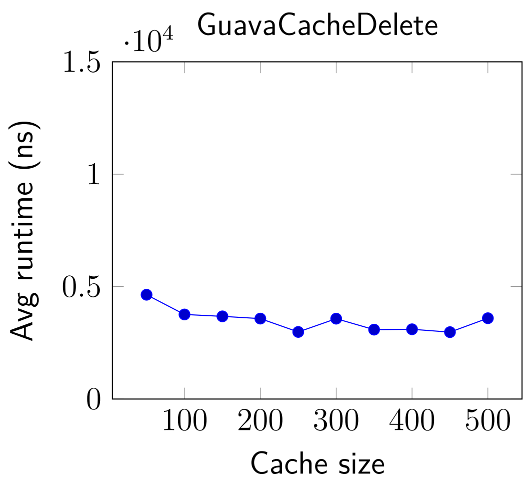


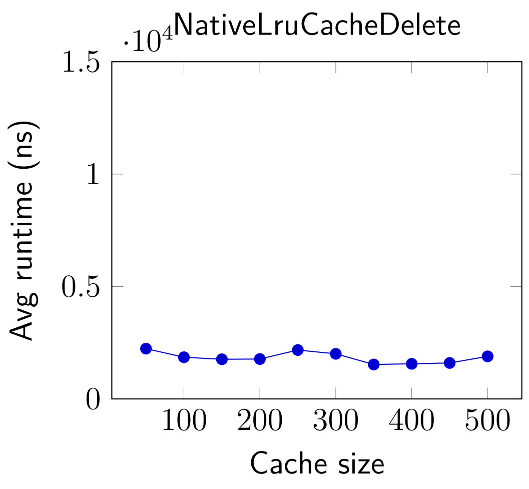


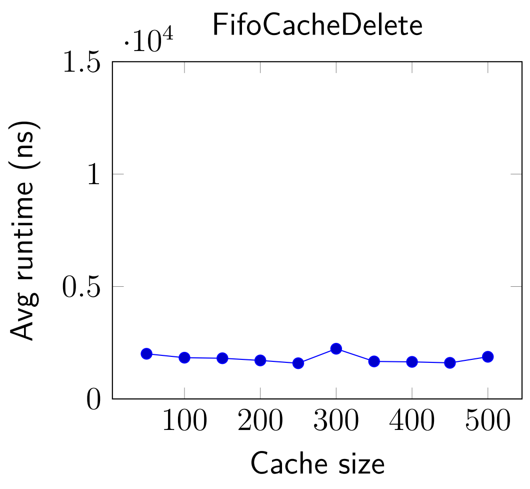
Appendix C

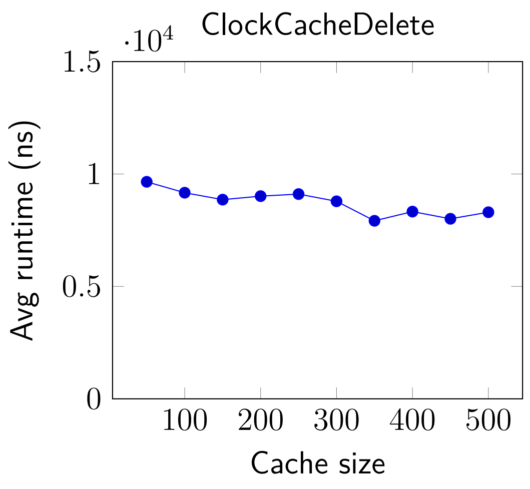
Delete benchmarks

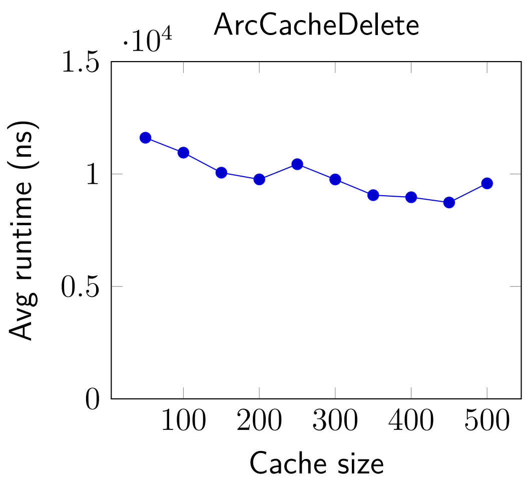


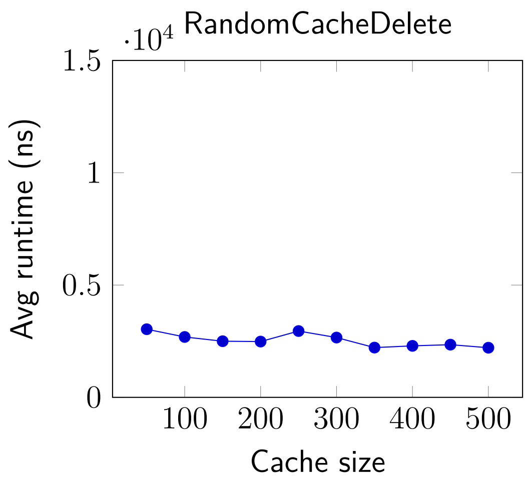






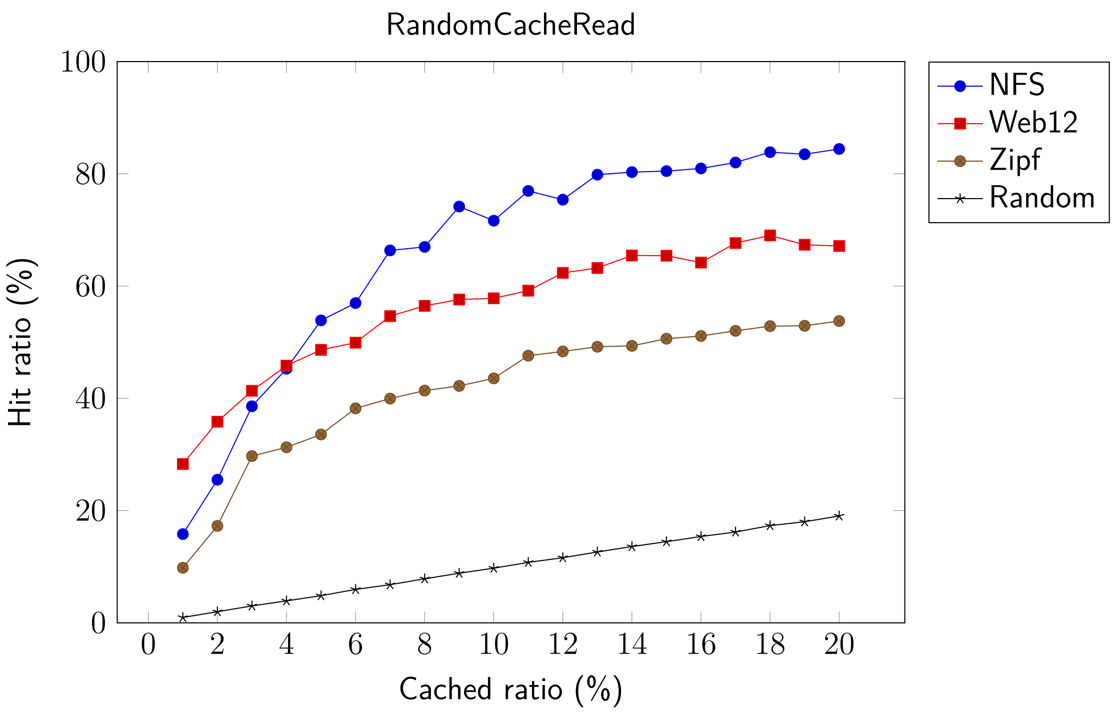

ClockCacheDelete
$\cdot 10^4$
1.5
1
0.5
0
100
200
300
400
500
Avg runtime (ns)
Cache size

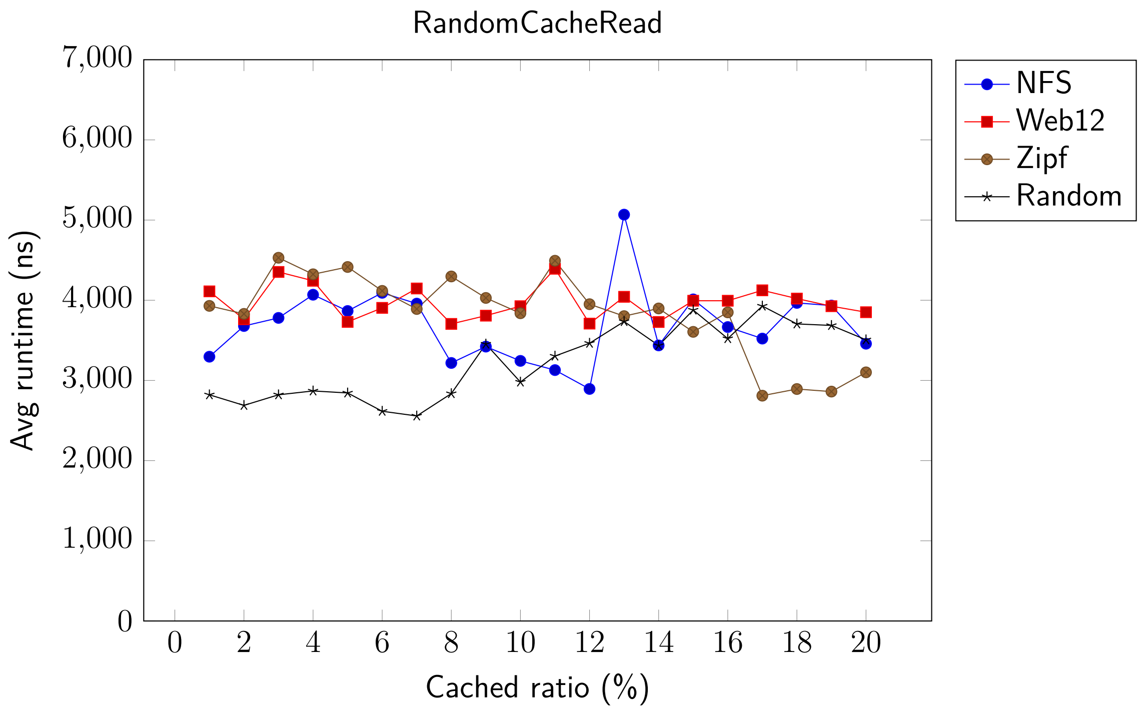

ArcCacheDelete
$\cdot 10^4$
1.5
1
0.5
0
100
200
300
400
500
Avg runtime (ns)
Cache size

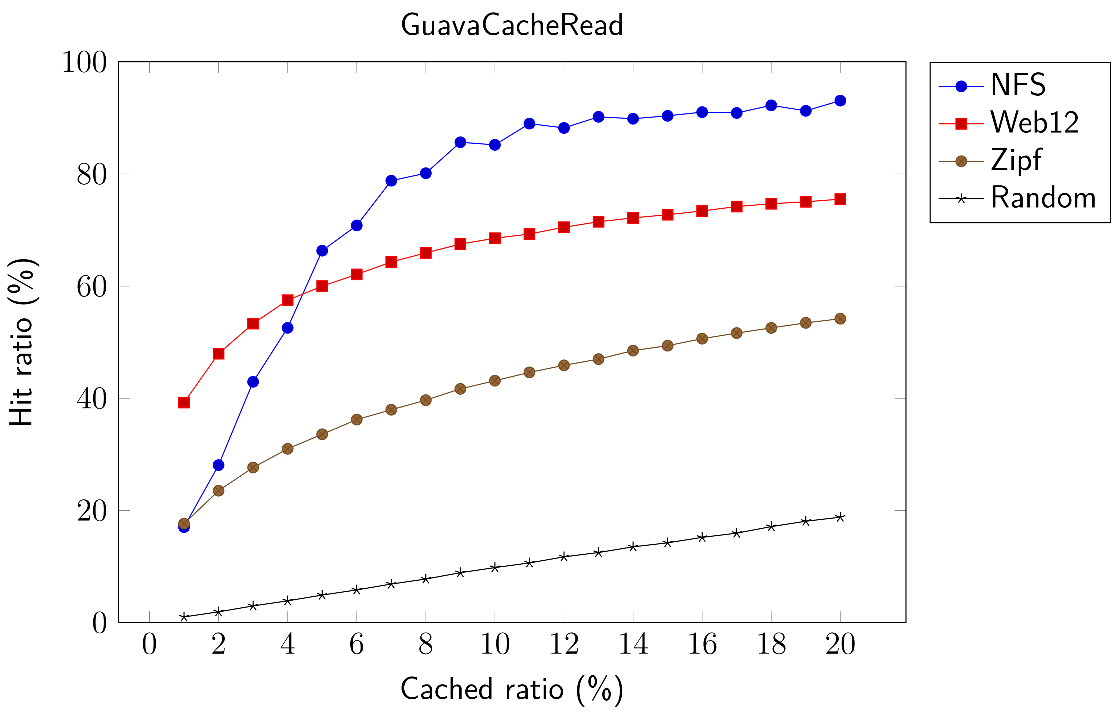

RandomCacheDelete
$\cdot 10^4$
1.5
1
0.5
0
100
200
300
400
500
Avg runtime (ns)
Cache size

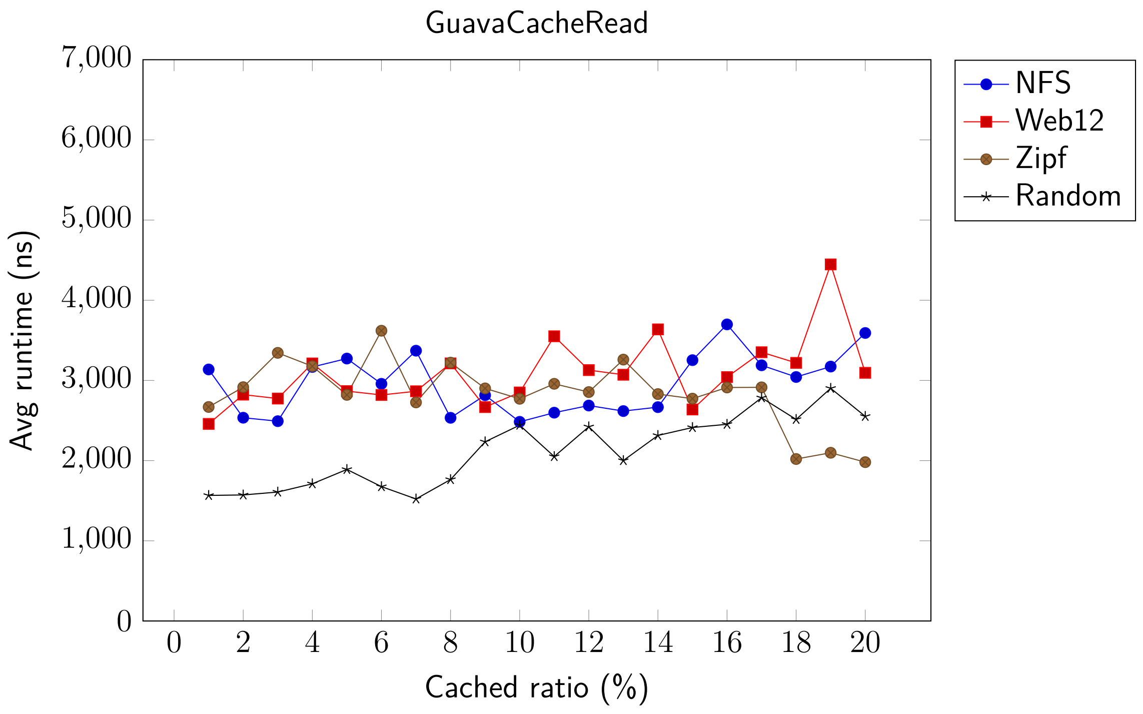
Appendix D

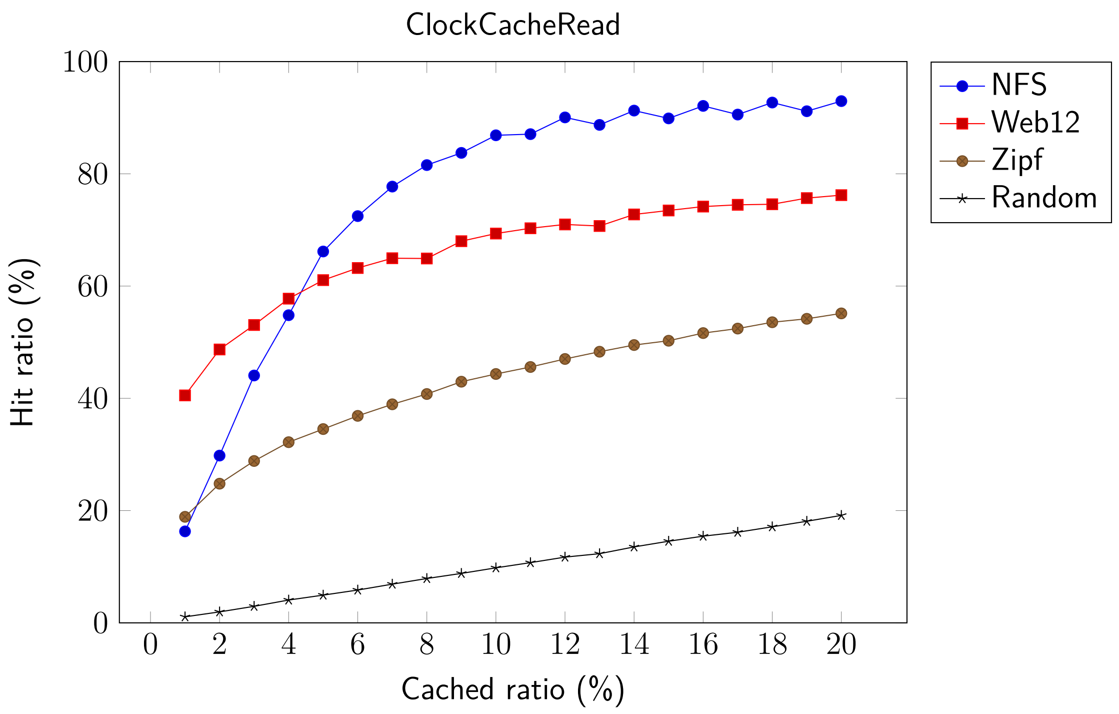
Read benchmarks

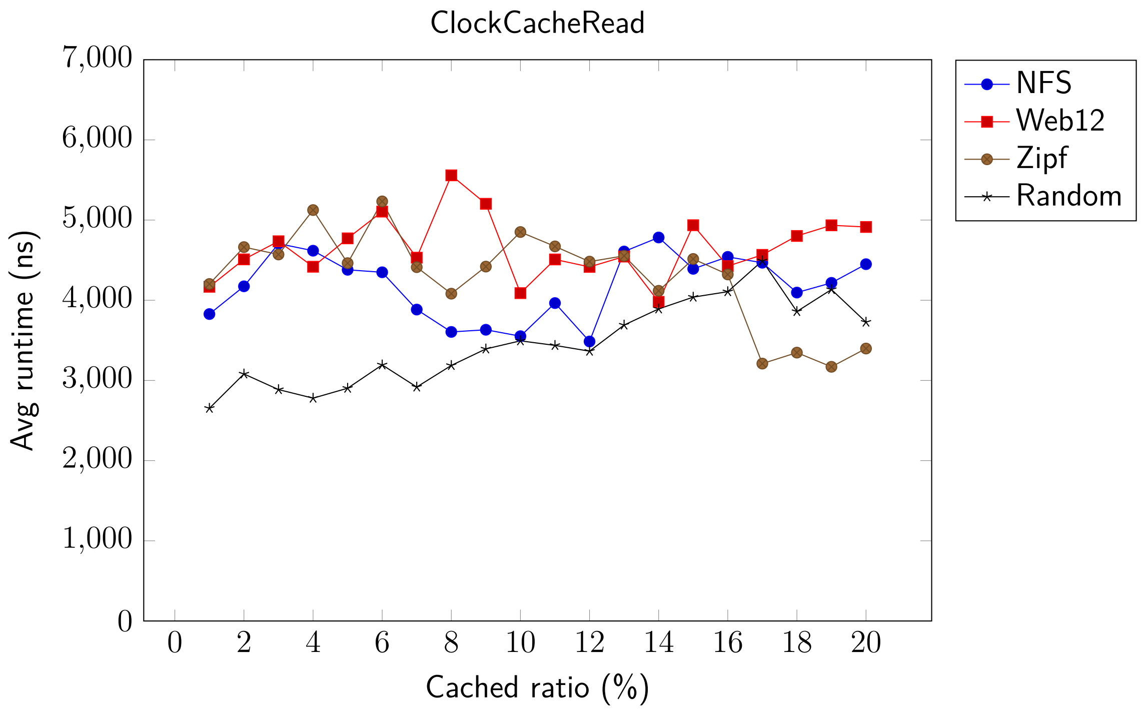

RandomCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

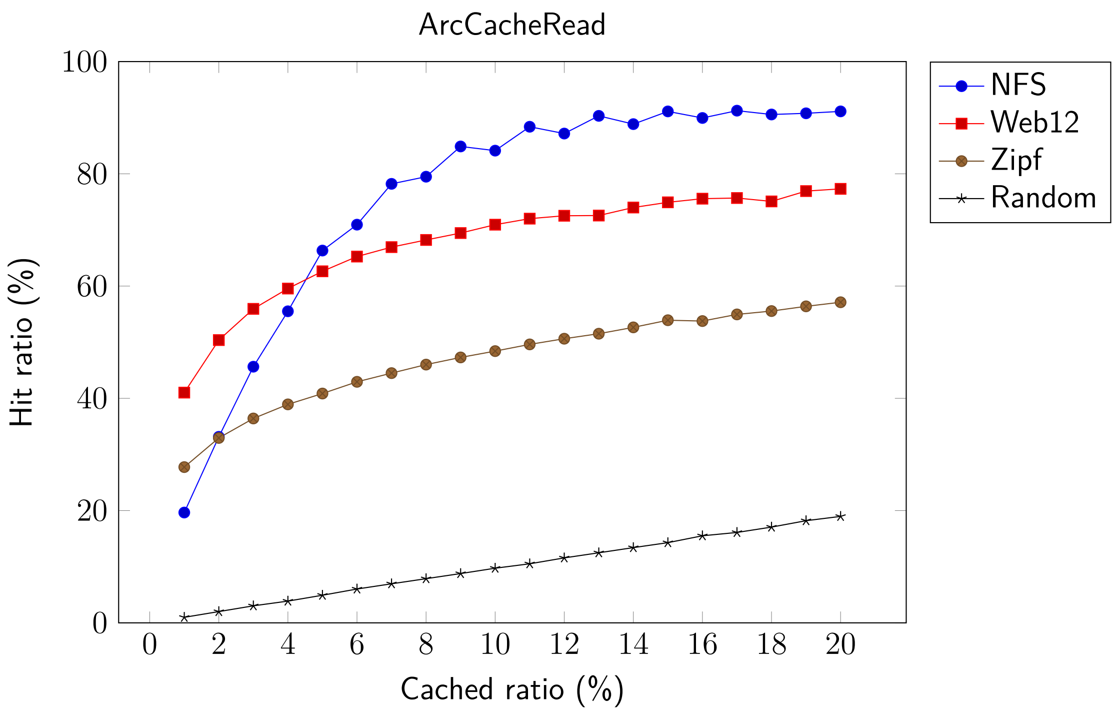

RandomCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

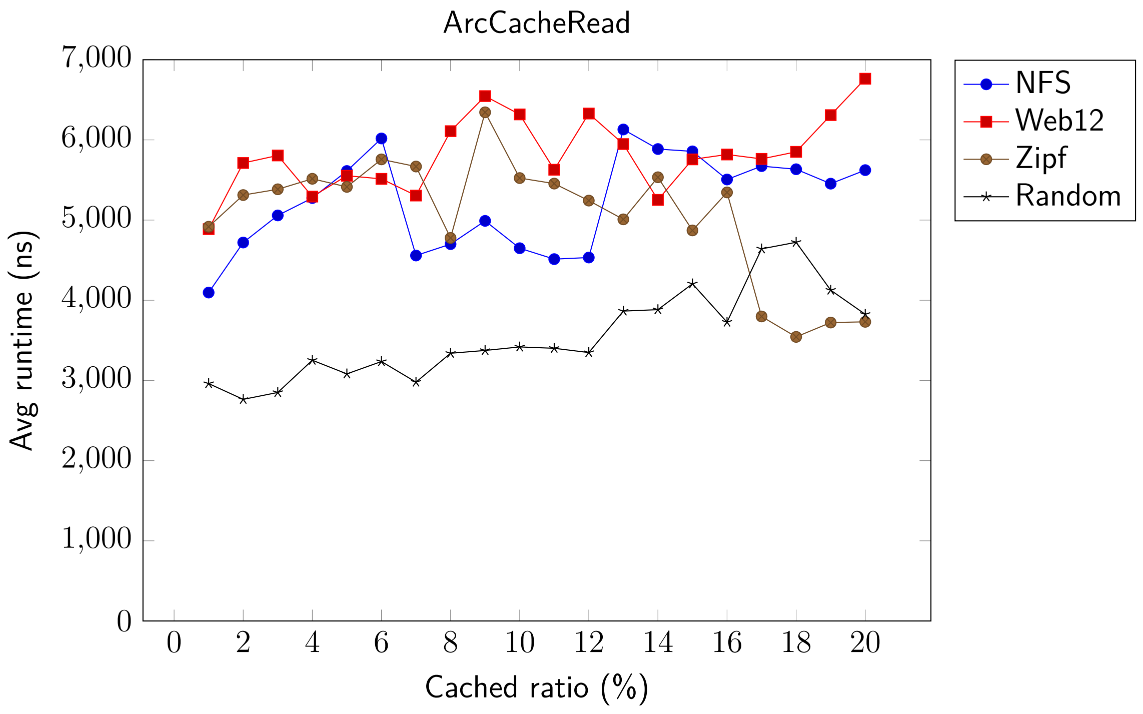

GuavaCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

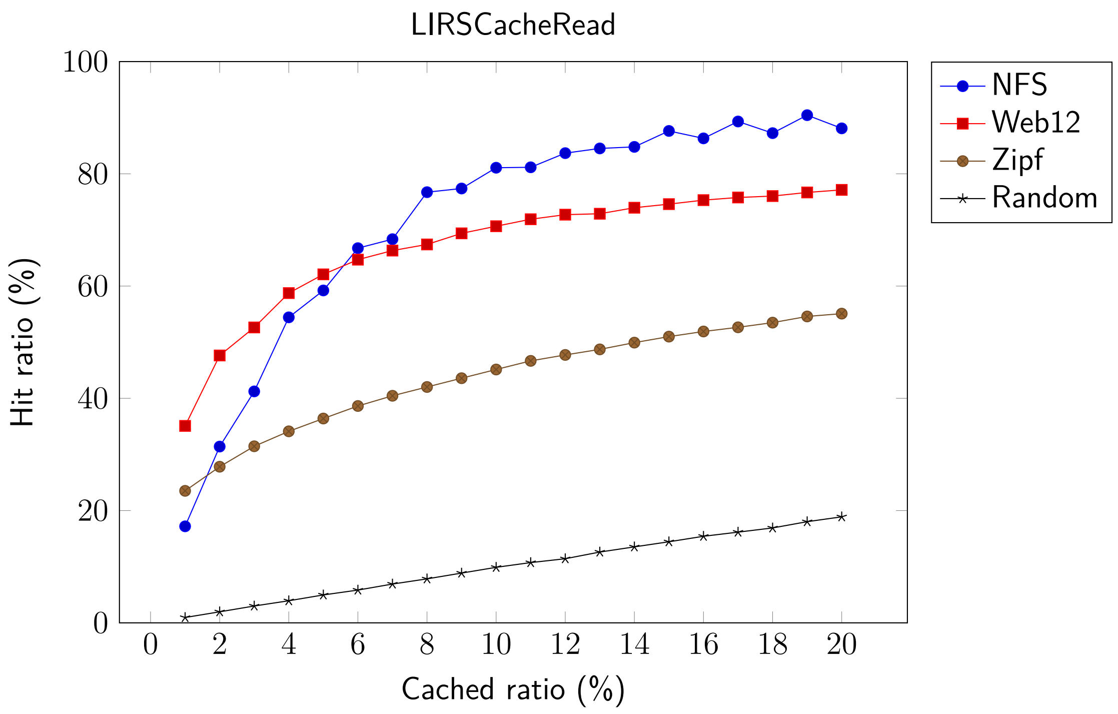

GuavaCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

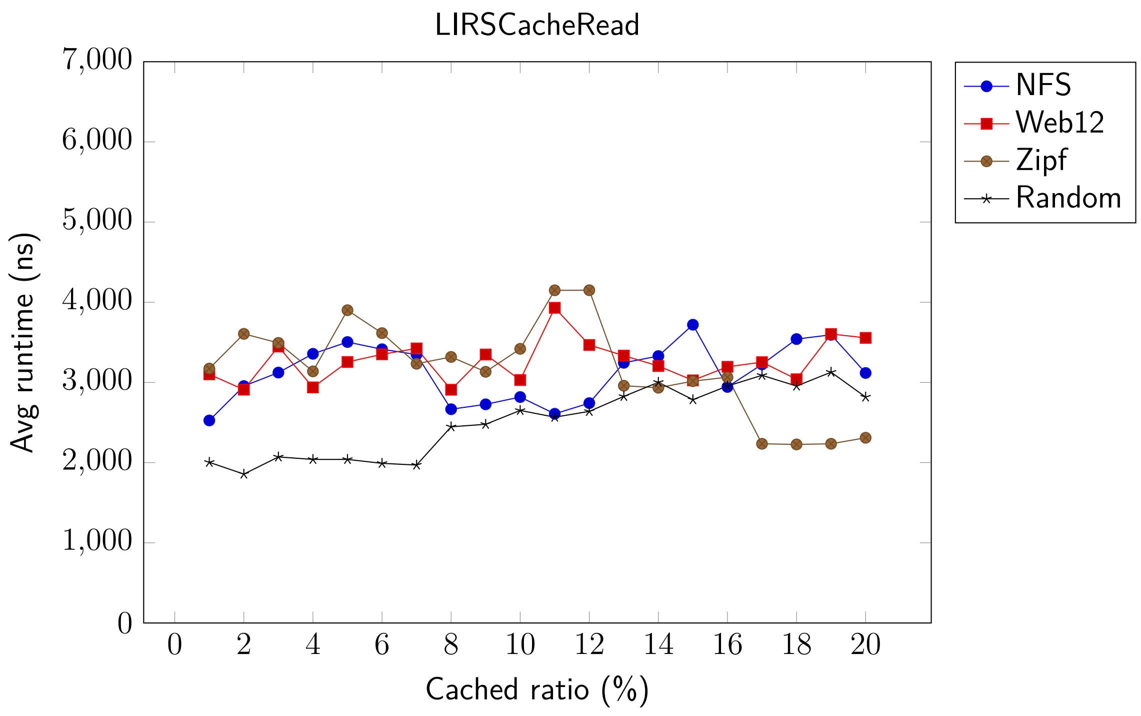

ClockCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

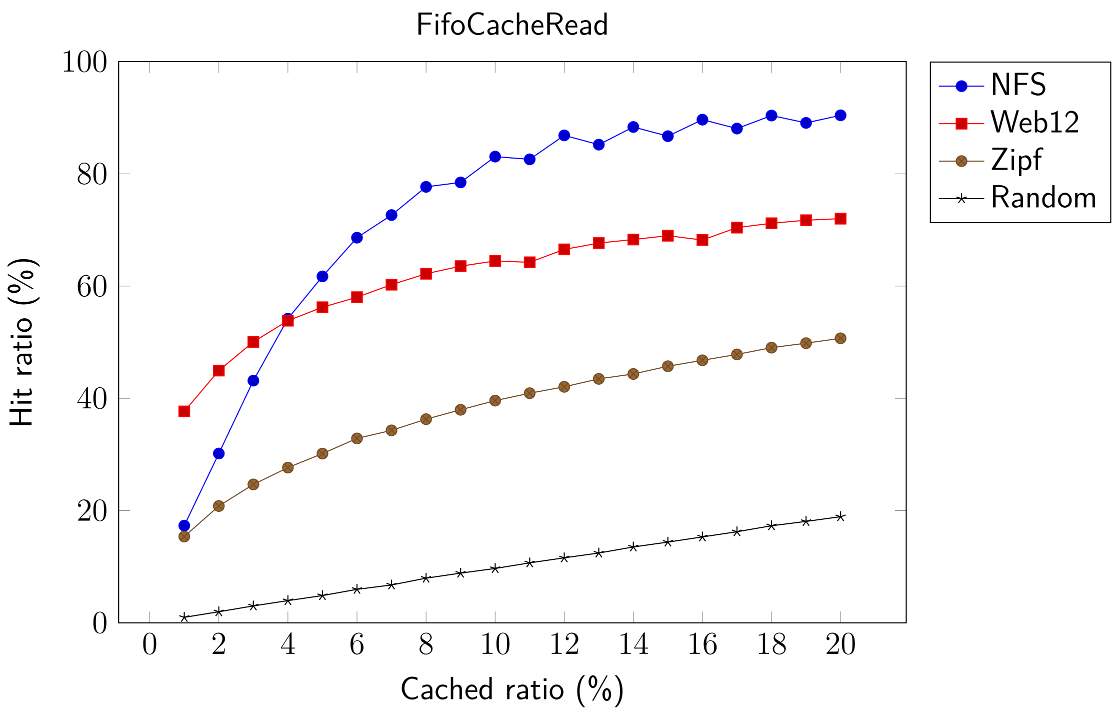

ClockCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

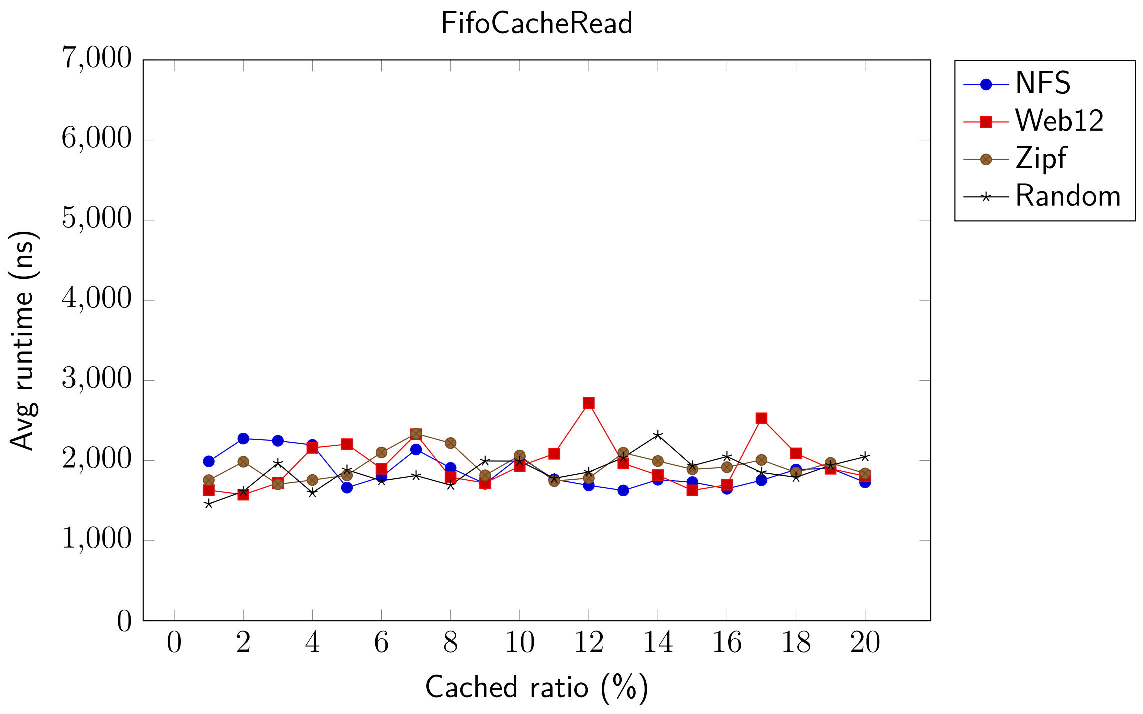

ArcCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

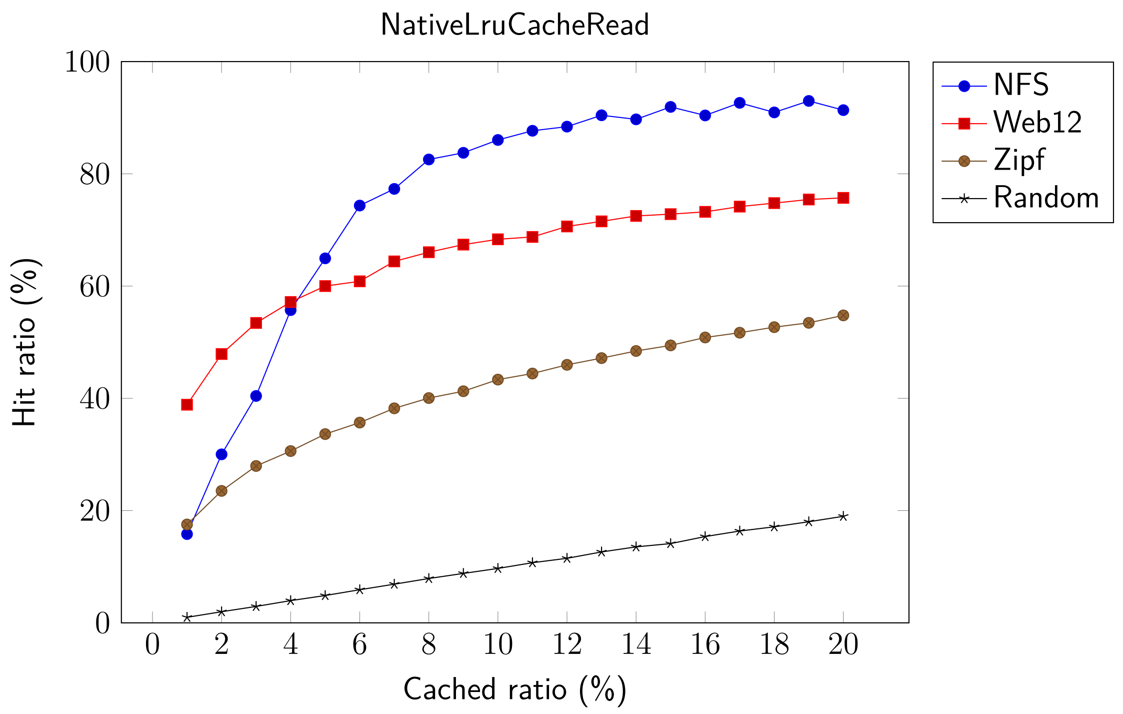

ArcCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

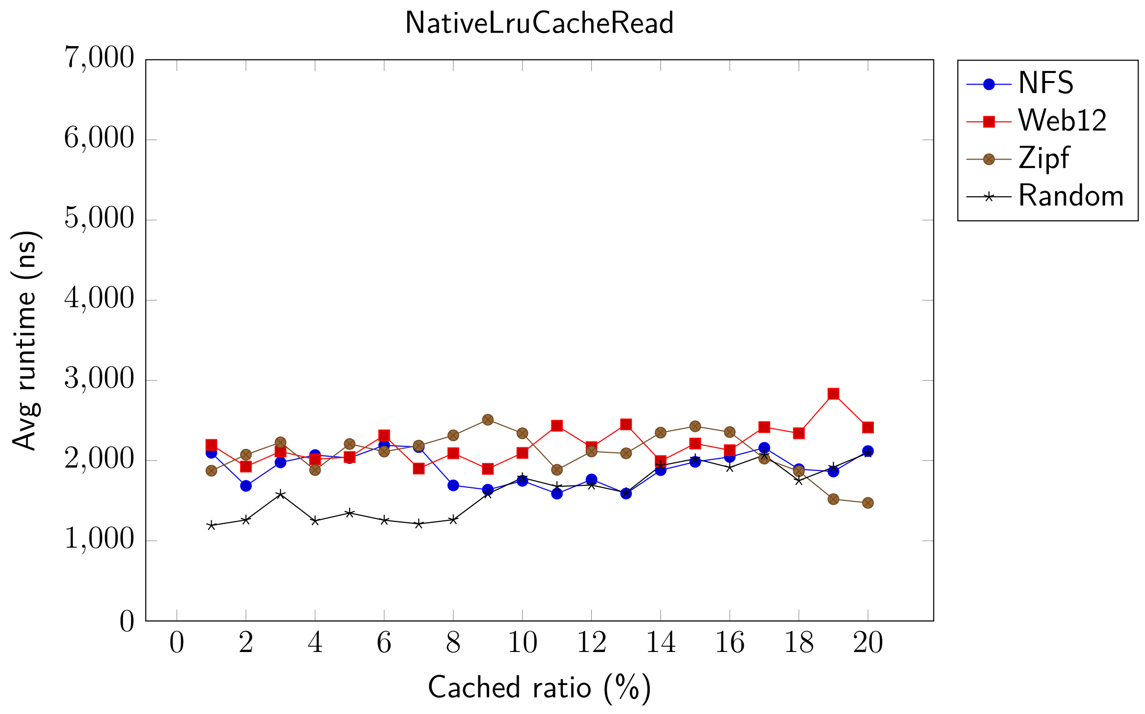

LIRSCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

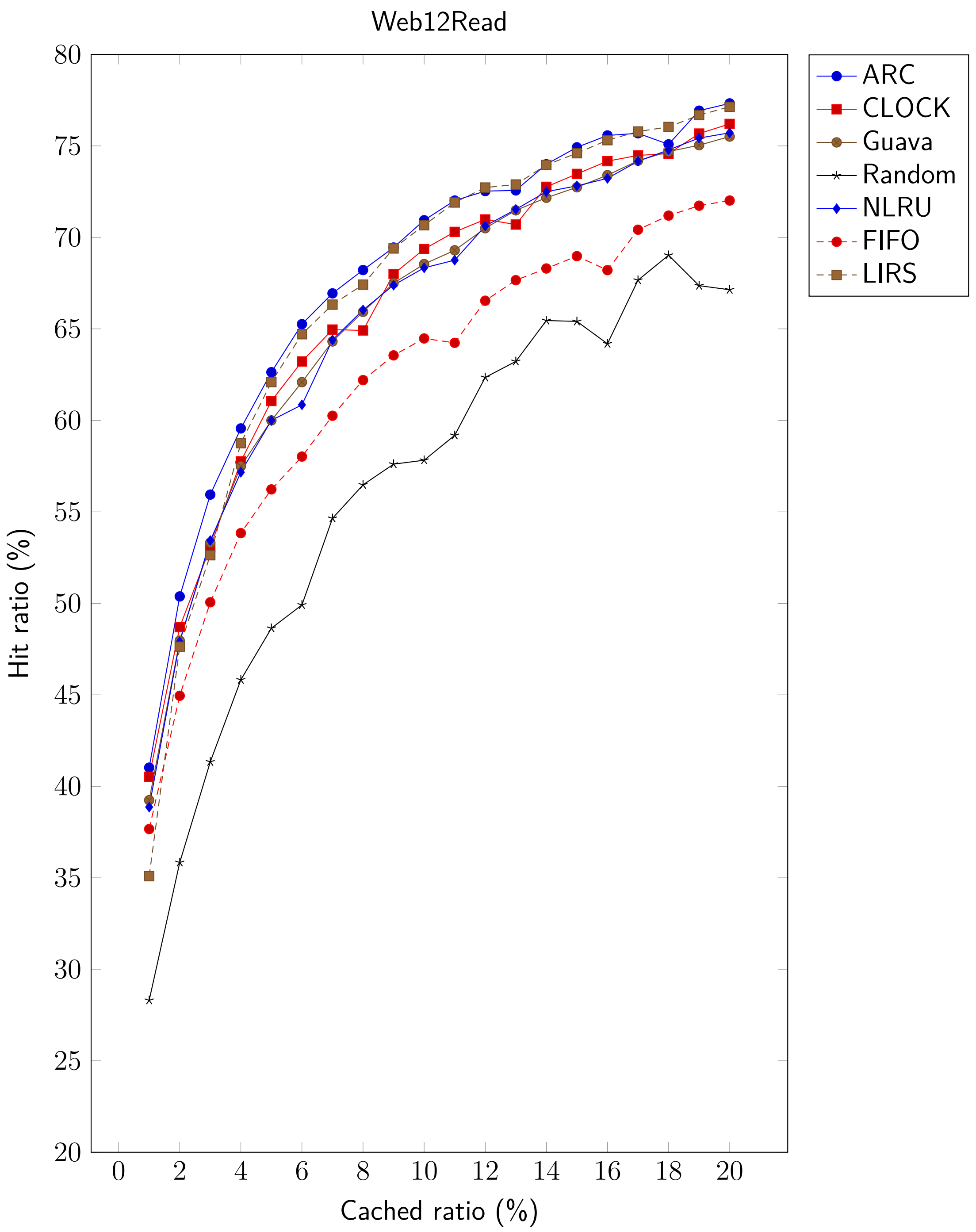

LIRSCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

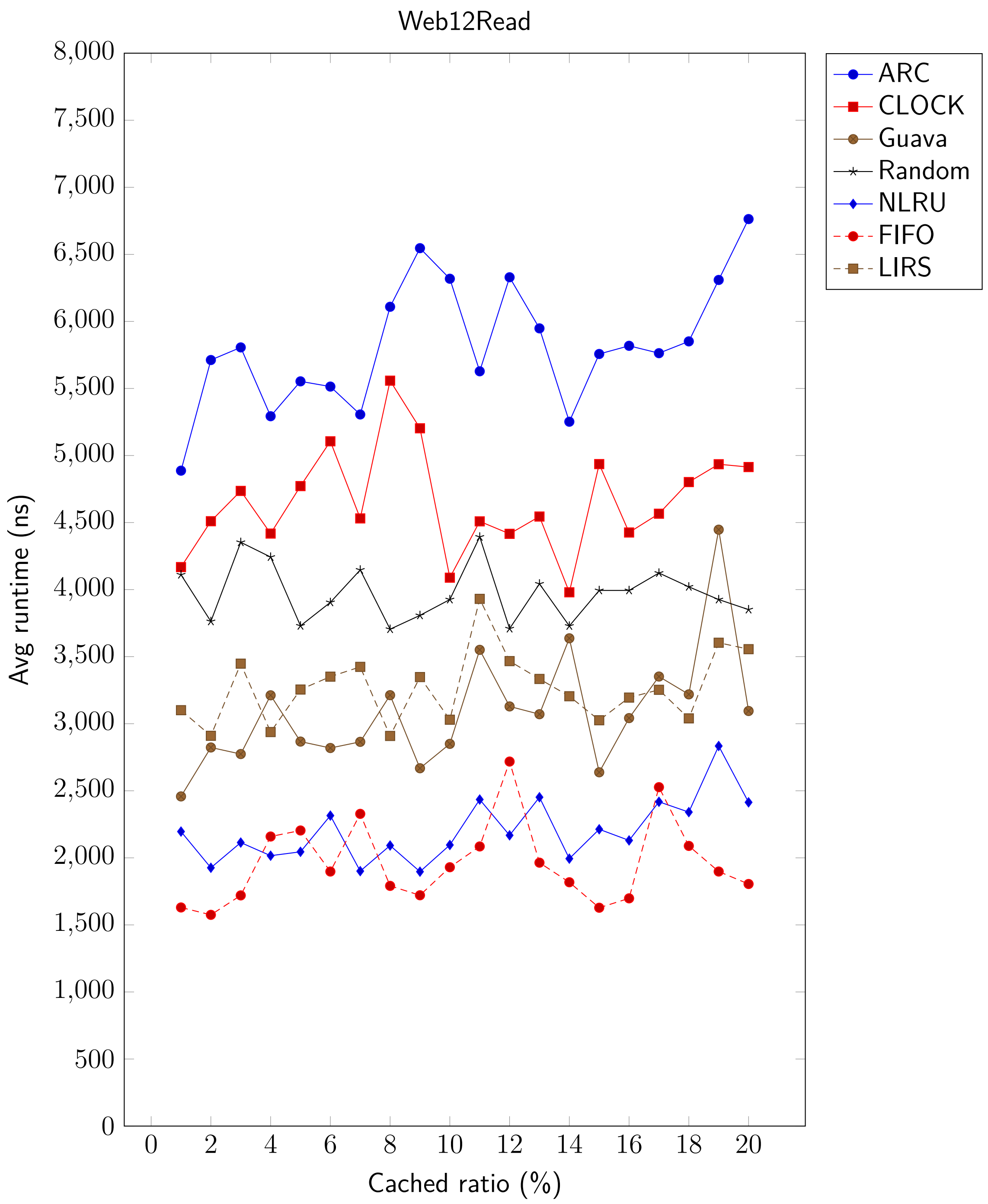

FifoCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

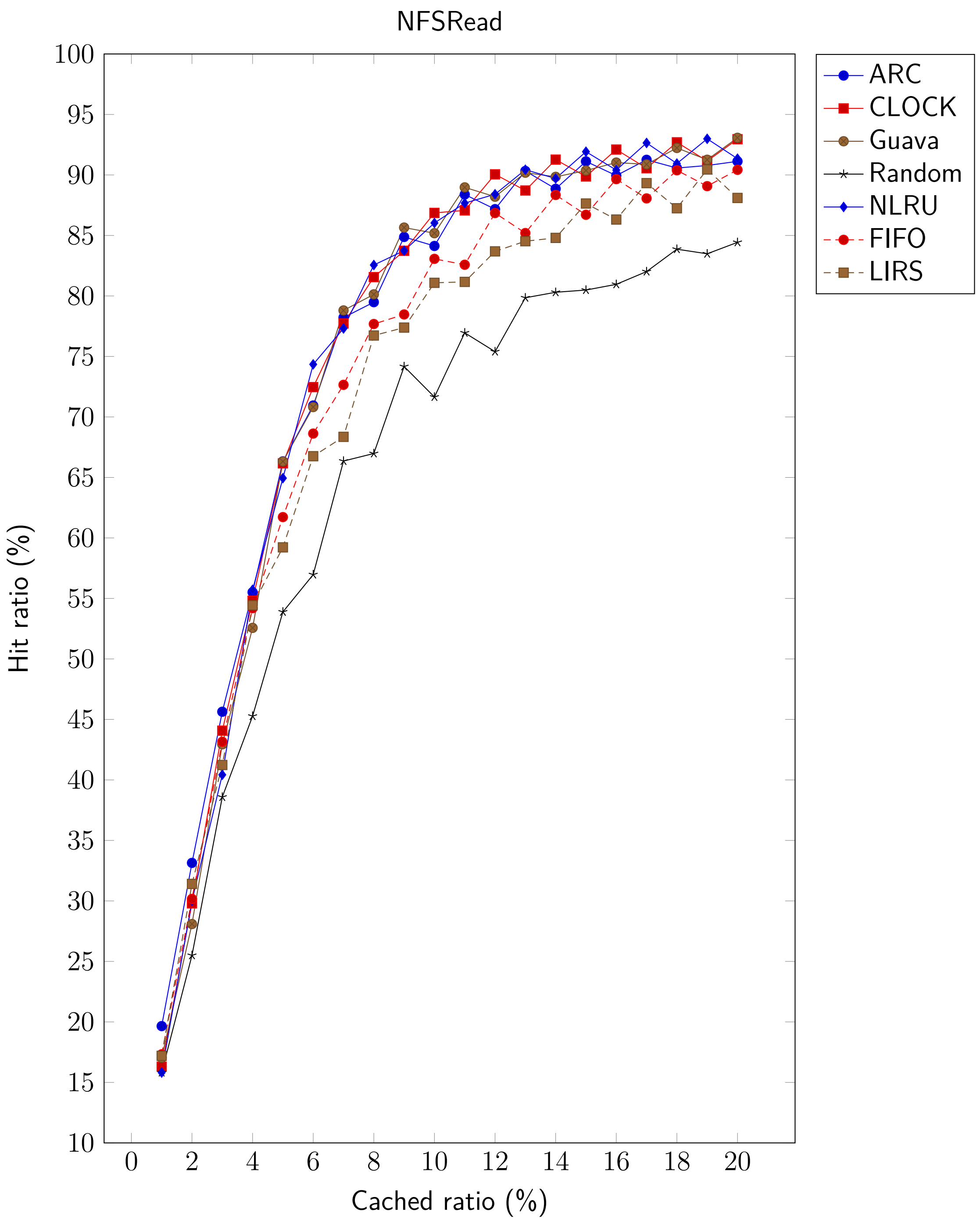

FifoCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

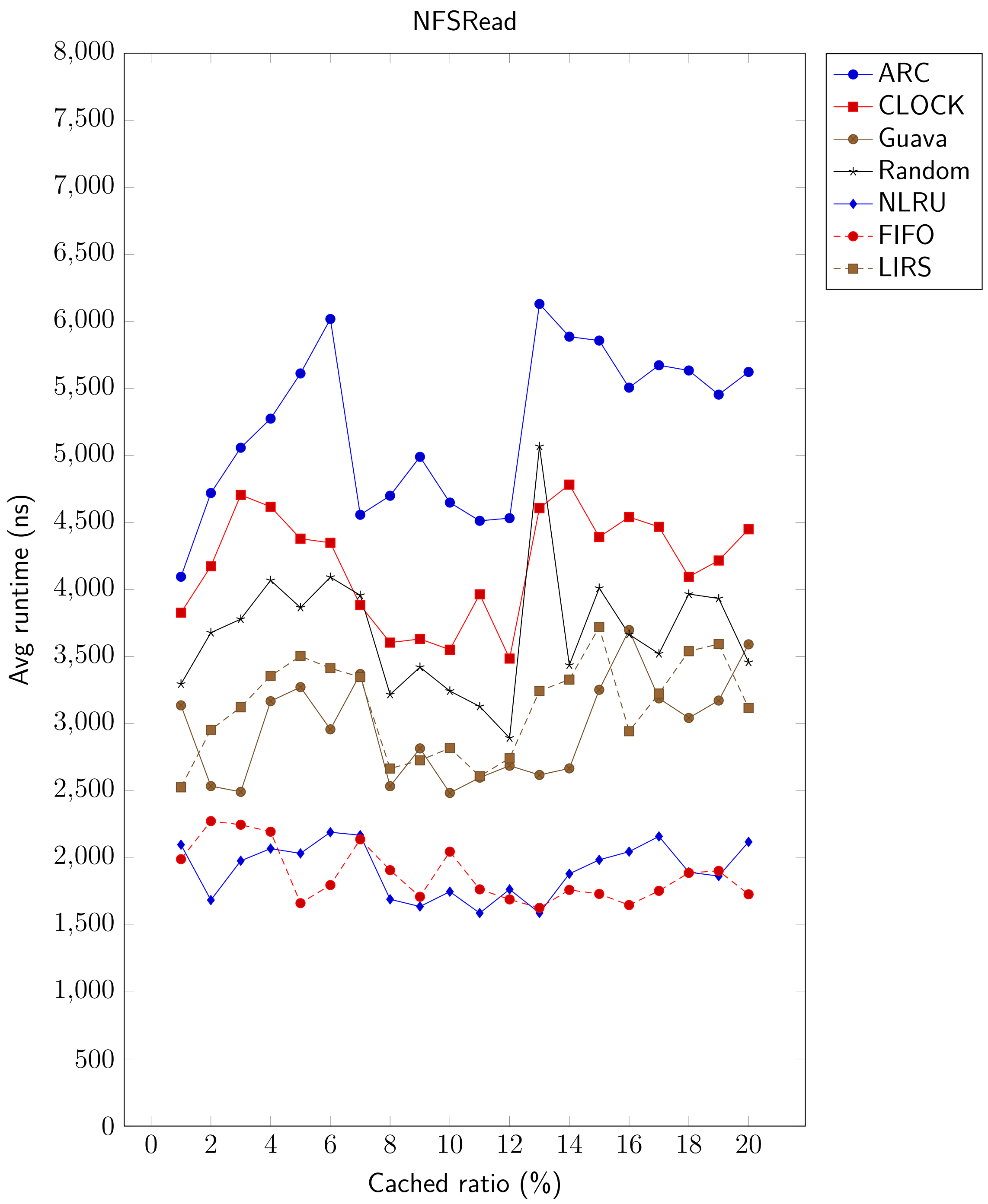

NativeLruCacheRead
Hit ratio (%)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
20
40
60
80
100
0
2
4
6
8
10
12
14
16
18
20

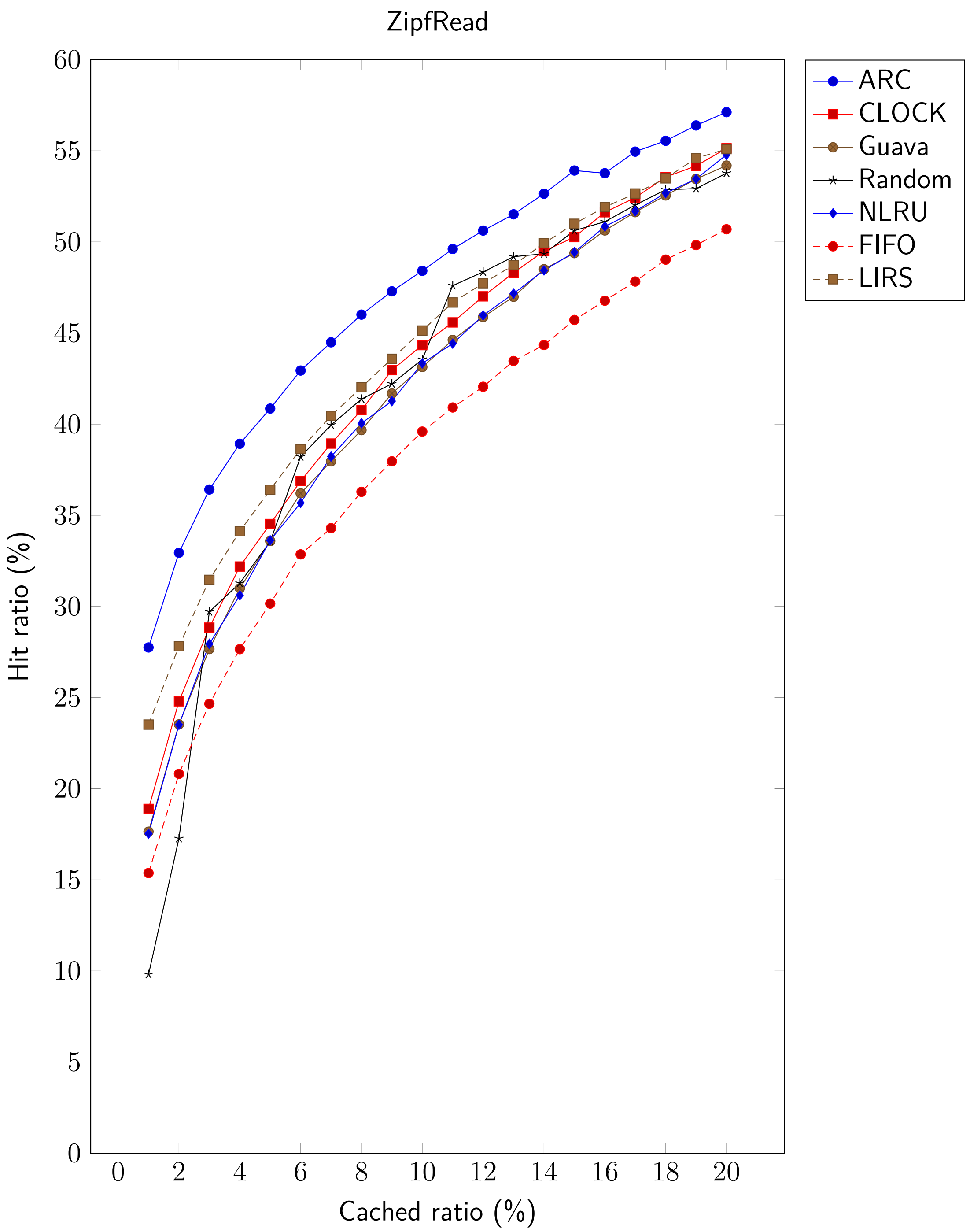

NativeLruCacheRead
Avg runtime (ns)
Cached ratio (%)
NFS
Web12
Zipf
Random
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
2
4
6
8
10
12
14
16
18
20

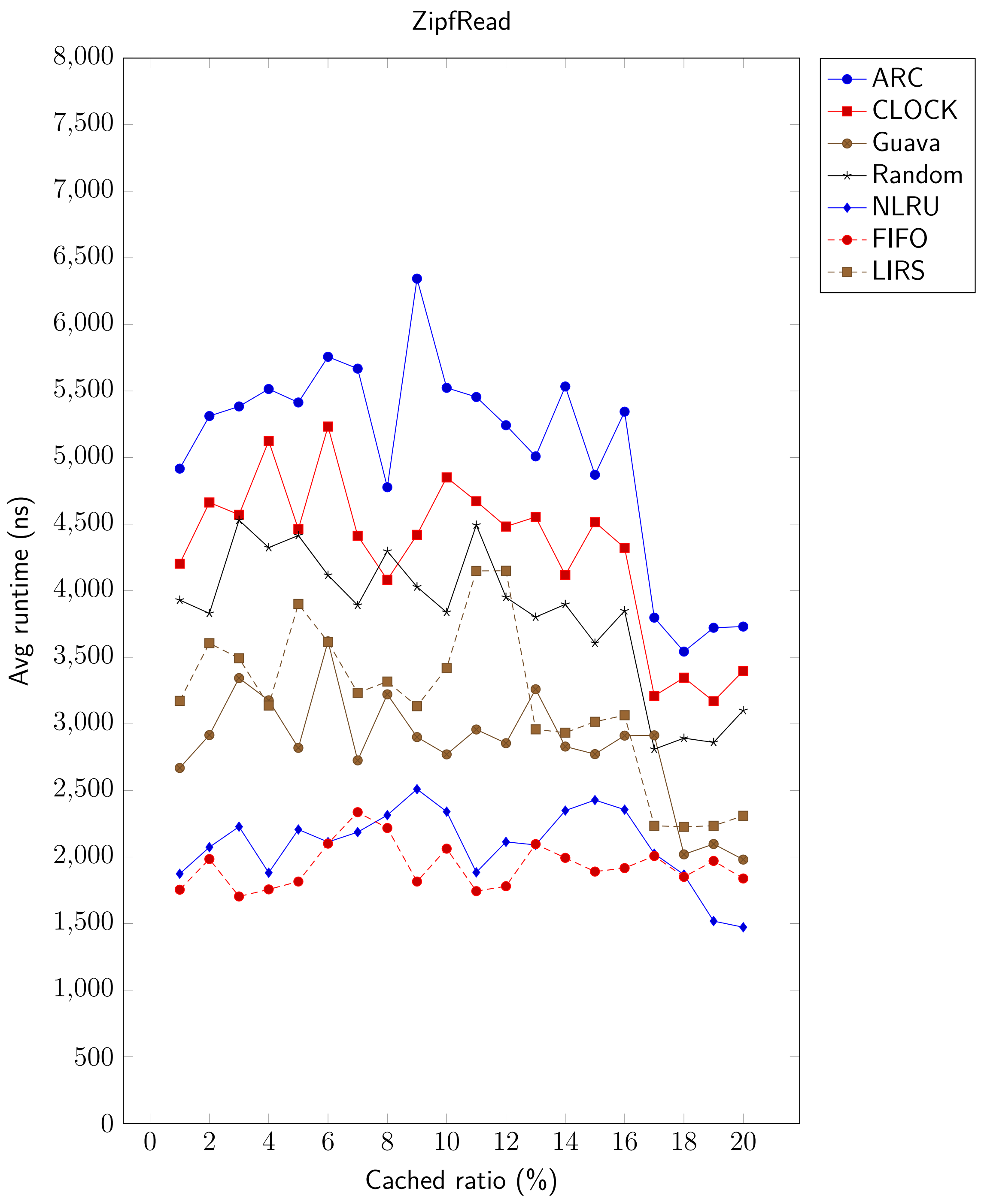

Web12Read
Hit ratio (%)
Cached ratio (%)
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS
80
75
70
65
60
55
50
45
40
35
30
25
20
0
2
4
6
8
10
12
14
16
18
20

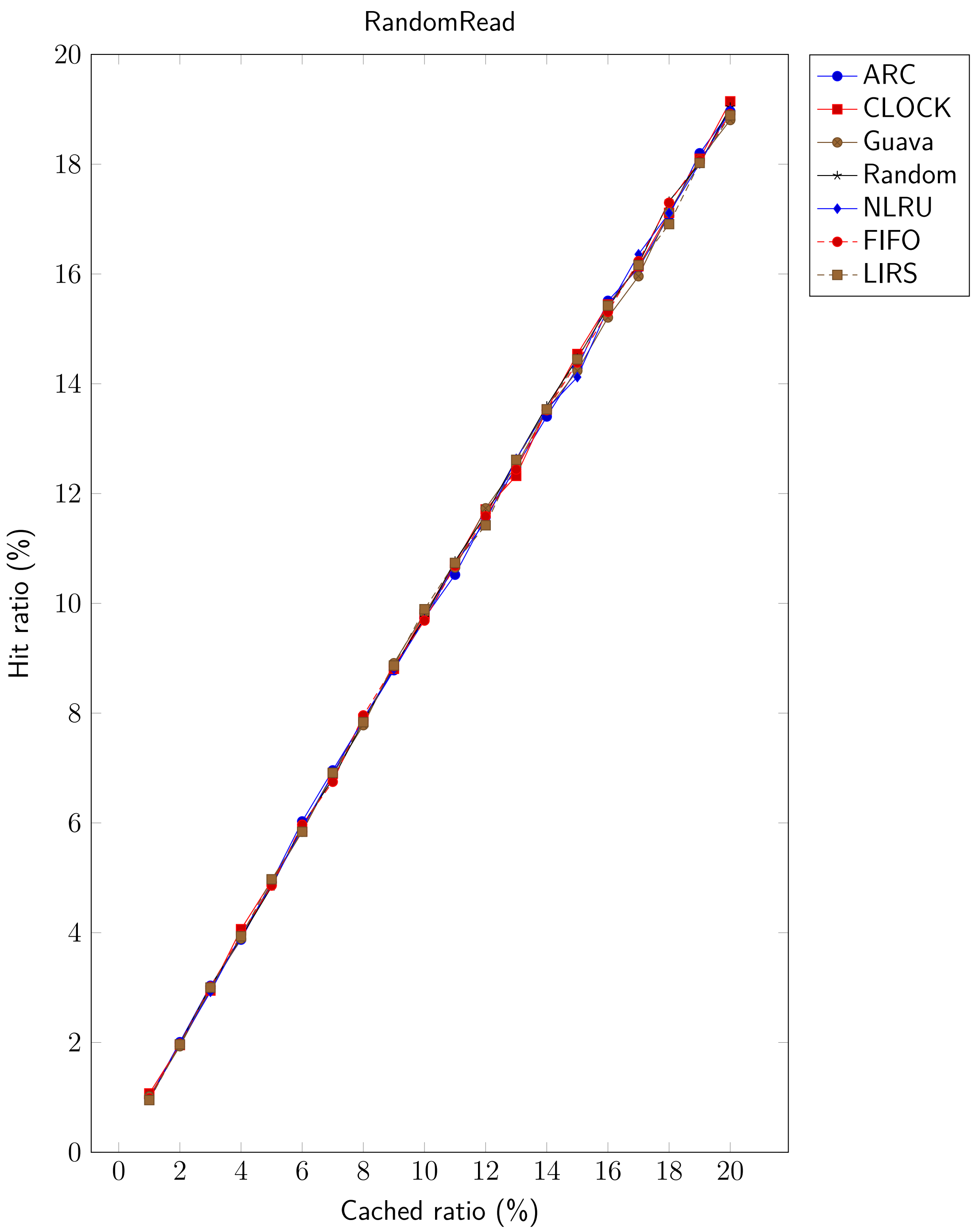

Web12Read
Avg runtime (ns)
Cached ratio (%)
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000
6,500
7,000
7,500
8,000
0
2
4
6
8
10
12
14
16
18
20
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS

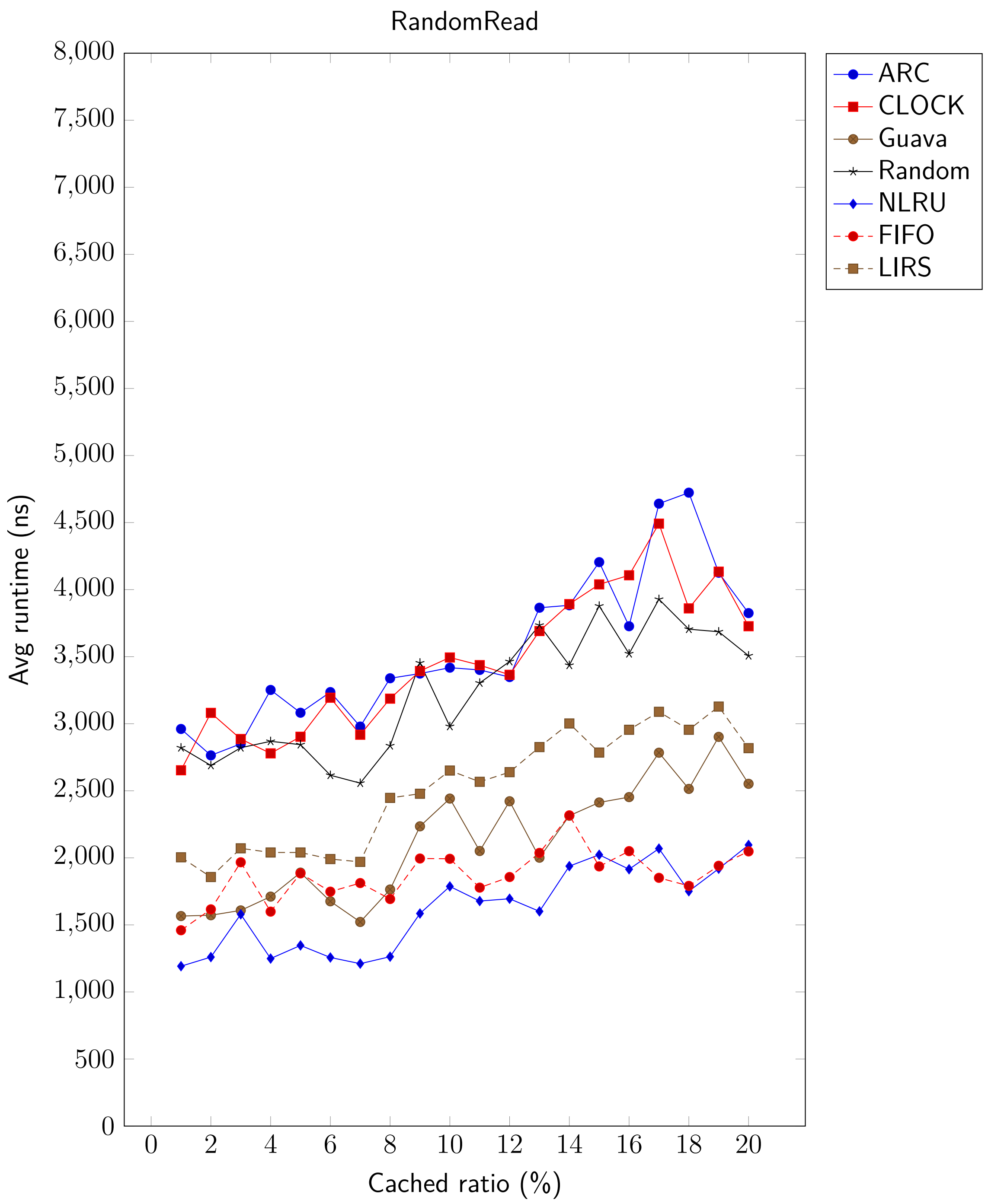

NFSRead
Hit ratio (%)
Cached ratio (%)
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
0
2
4
6
8
10
12
14
16
18
20


NFSRead
Avg runtime (ns)
Cached ratio (%)
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000
6,500
7,000
7,500
8,000
0
2
4
6
8
10
12
14
16
18
20
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS


ZipfRead
Hit ratio (%)
Cached ratio (%)
0
5
10
15
20
25
30
35
40
45
50
55
60
0
2
4
6
8
10
12
14
16
18
20
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS


ZipfRead
Avg runtime (ns)
Cached ratio (%)
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000
6,500
7,000
7,500
8,000
0
2
4
6
8
10
12
14
16
18
20


RandomRead
Hit ratio (%)
Cached ratio (%)
0
2
4
6
8
10
12
14
16
18
20
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS


RandomRead
Avg runtime (ns)
Cached ratio (%)
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000
6,500
7,000
7,500
8,000
0
2
4
6
8
10
12
14
16
18
20
ARC
CLOCK
Guava
Random
NLRU
FIFO
LIRS

Appendix E

Code

E.1 Package cachebenchmarking

Listing E.1: desmedt.frederik.cachebenchmarking.BenchmarkRunner

```
package desmedt.frederik.cachebenchmarking;

import android.util.Log;
import android.util.Pair;

import java.io.IOException;
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;
import java.lang.ref.SoftReference;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Queue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import desmedt.frederik.cachebenchmarking.benchmark.BaseBenchmark;
import desmedt.frederik.cachebenchmarking.benchmark.Cache2KBenchmark;
import desmedt.frederik.cachebenchmarking.benchmark.CustomBenchmark;
import desmedt.frederik.cachebenchmarking.benchmark.GuavaBenchmarks;
import desmedt.frederik.cachebenchmarking.benchmark.JackRabbitLIRSBenchmark;
import desmedt.frederik.cachebenchmarking.benchmark.NativeLruBenchmarks;
import desmedt.frederik.cachebenchmarking.cache.Cache;
import desmedt.frederik.cachebenchmarking.cache.FIFOCache;
import desmedt.frederik.cachebenchmarking.cache.RandomCache;
import desmedt.frederik.cachebenchmarking.generator.Generator;
import desmedt.frederik.cachebenchmarking.generator.NfsGenerator;
import desmedt.frederik.cachebenchmarking.generator.RandomGenerator;
import desmedt.frederik.cachebenchmarking.generator.SearchEngineGenerator;
import desmedt.frederik.cachebenchmarking.generator.Web12Generator;
import desmedt.frederik.cachebenchmarking.generator.ZipfGenerator;

/**
```

```
 * Responsible for running all {@link CacheBenchmarkConfiguration}s.
 */
public class BenchmarkRunner {

    private static final String TAG = BenchmarkRunner.class.getSimpleName();

    private Map<String, List<CacheBenchmarkConfiguration.CacheStats>>
        benchmarkResults = new HashMap<>();

    /**
     * ExecutorService executing every benchmark in a serializable fashion.
         Meaning none of them will
     * run parallel to another benchmark. This is way more reliable than parallel
         execution as in
     * parallel scenario's different benchmarks are both competing for the same
         resources and locks.
     */
    private ExecutorService benchmarkRunnerService = Executors.
        newSingleThreadExecutor();

    private Generator<Cache<Integer, Integer>> generateRandomCache(final int
        cacheSize) {
        return new Generator<Cache<Integer, Integer>>() {
            @Override
            public Cache<Integer, Integer> next() {
                return new RandomCache<>(cacheSize);
            }
        };
    }

    private Generator<Cache<Integer, Integer>> generateFifoCache(final int
        cacheSize) {
        return new Generator<Cache<Integer, Integer>>() {
            @Override
            public Cache<Integer, Integer> next() {
                return new FIFOCache<>(cacheSize);
            }
        };
    }

    public void runBenchmarks() {
        NfsGenerator nfsGenerator = new NfsGenerator();
        for (int i = 1; i <= 20; i++) {
            submitCountedReadBenchmarks(NfsGenerator.getLowerBound(), NfsGenerator
                .getUpperBound(), (double) i / 100, NfsGenerator.TRACE_TAG,
                nfsGenerator, 1000, NfsGenerator.getUpperBound() * 50);
        }
        nfsGenerator = null; // remove strong reference

        SearchEngineGenerator searchEngineGenerator = new SearchEngineGenerator();
        for (int i = 1; i <= 10; i++) {
            submitCountedReadBenchmarks(SearchEngineGenerator.getLowerBound(),
                SearchEngineGenerator.getUpperBound(), (double) i / 1000,
                SearchEngineGenerator.TRACE_TAG, searchEngineGenerator, 1000, 10
                _000);
        }
        searchEngineGenerator = null; // remove strong reference

        Web12Generator web12Generator = new Web12Generator();
        for (int i = 1; i <= 20; i++) {
            submitCountedReadBenchmarks(0, Web12Generator.getUpperBound(), (double
                ) i / 100, Web12Generator.TRACE_TAG, web12Generator, 1000, 100_000
            );
    }
    web12Generator = null;

    ZipfGenerator zipfGenerator = new ZipfGenerator(0, 50000);
    for (int i = 1; i <= 20; i++) {
        submitCountedReadBenchmarks(0, 50000, (double) i / 100, ZipfGenerator.
            TRACE_TAG, zipfGenerator, 1000, 100_000);
    }
    zipfGenerator = null;

    RandomGenerator randomGenerator = new RandomGenerator(0, 50000);
    for (int i = 1; i <= 20; i++) {
        submitCountedReadBenchmarks(0, 50000, (double) i / 100,
            RandomGenerator.TRACE_TAG, randomGenerator, 1000, 100_000);
    }
    randomGenerator = null;

    /* Insert benchmarks */

    for (int i = 1; i <= 10; i++) {
        final int upperBound = 500;
        double cacheRatio = (double) i / 10;
        final int cacheSize = Math.round(upperBound * ((float) i / 10));

        submitCountedBenchmark(new GuavaBenchmarks.Insert(cacheRatio, 0,
            upperBound));
        submitCountedBenchmark(new NativeLruBenchmarks.Insert(cacheRatio, 0,
            upperBound));
        submitCountedBenchmark(new CustomBenchmark.Insert("FifoCache",
            cacheRatio, 0, upperBound, generateFifoCache(cacheSize)));
        submitCountedBenchmark(new CustomBenchmark.Insert("RandomCache",
            cacheRatio, 0, upperBound, generateRandomCache(cacheSize)));
        submitCountedBenchmark(new JackRabbitLIRSBenchmark.Insert(cacheRatio,
            0, upperBound));
        submitCountedBenchmark(new Cache2KBenchmark.Insert(Cache2KBenchmark.
            CLOCK_CACHE, cacheRatio, 0, upperBound));
        submitCountedBenchmark(new Cache2KBenchmark.Insert(Cache2KBenchmark.
            ARC_CACHE, cacheRatio, 0, upperBound));
    }

    /* Update benchmarks */

    for (int i = 1; i <= 10; i++) {
        final int upperBound = 500;
        double cacheRatio = (double) i / 10;
        final int cacheSize = (int) Math.round(upperBound * cacheRatio);

        submitCountedBenchmark(new GuavaBenchmarks.Update(cacheRatio, 0,
            upperBound));
        submitCountedBenchmark(new NativeLruBenchmarks.Update(cacheRatio, 0,
            upperBound));
        submitCountedBenchmark(new CustomBenchmark.Update("FifoCache",
            cacheRatio, 0, upperBound, generateFifoCache(cacheSize)));
        submitCountedBenchmark(new CustomBenchmark.Update("RandomCache",
            cacheRatio, 0, upperBound, generateRandomCache(cacheSize)));
        submitCountedBenchmark(new JackRabbitLIRSBenchmark.Update(cacheRatio,
            0, upperBound));
        submitCountedBenchmark(new Cache2KBenchmark.Update(Cache2KBenchmark.
            CLOCK_CACHE, cacheRatio, 0, upperBound));
        submitCountedBenchmark(new Cache2KBenchmark.Update(Cache2KBenchmark.
            ARC_CACHE, cacheRatio, 0, upperBound));
        }

        /* Delete benchmarks */

        for (int i = 1; i <= 10; i++) {
            final int upperBound = 500;
            double cacheRatio = (double) i / 10;
            final int cacheSize = Math.round(upperBound * ((float) i / 10));

            submitCountedBenchmark(new GuavaBenchmarks.Delete(cacheRatio, 0,
                upperBound));
            submitCountedBenchmark(new NativeLruBenchmarks.Delete(cacheRatio, 0,
                upperBound));
            submitCountedBenchmark(new CustomBenchmark.Delete("FifoCache",
                cacheRatio, 0, upperBound, generateFifoCache(cacheSize)));
            submitCountedBenchmark(new CustomBenchmark.Delete("RandomCache",
                cacheRatio, 0, upperBound, generateRandomCache(cacheSize)));
            submitCountedBenchmark(new JackRabbitLIRSBenchmark.Delete(cacheRatio,
                0, upperBound));
            submitCountedBenchmark(new Cache2KBenchmark.Delete(Cache2KBenchmark.
                CLOCK_CACHE, cacheRatio, 0, upperBound));
            submitCountedBenchmark(new Cache2KBenchmark.Delete(Cache2KBenchmark.
                ARC_CACHE, cacheRatio, 0, upperBound));
        }

        benchmarkRunnerService.submit(new Runnable() {
            @Override
            public void run() {
                logBenchmarkResults();
            }
        });

        benchmarkRunnerService.shutdown();
    }

    public void resetEnvironment() {
        gc();
    }

    public void logBenchmarkResults() {
        for (Map.Entry<String, List<CacheBenchmarkConfiguration.CacheStats>> entry
             : benchmarkResults.entrySet()) {
            Log.i(TAG, TableFormatter.generateHitRatioTable(entry.getKey(), entry.
                getValue()));
            Log.i(TAG, TableFormatter.generateAvgReadRuntimeTable(entry.getKey(),
                entry.getValue()));
            Log.i(TAG, TableFormatter.generateAvgRuntimeTable(
                CacheBenchmarkConfiguration.StatType.INSERT, entry.getKey(), entry
                .getValue()));
            Log.i(TAG, TableFormatter.generateAvgRuntimeTable(
                CacheBenchmarkConfiguration.StatType.UPDATE, entry.getKey(), entry
                .getValue()));
            Log.i(TAG, TableFormatter.generateAvgRuntimeTable(
                CacheBenchmarkConfiguration.StatType.DELETE, entry.getKey(), entry
                .getValue()));
        }

//          TableFormatter hitRatioFormatter = new TableFormatter(String.format("%35
    s", "Benchmark name"), "Min hitrate", "Max hitrate");
//          int i = 0;
//          for (String benchmark : benchmarks) {
```

```
//              hitRatioFormatter.addRow(benchmark, String.format("%.4f",
    minHitrateList.get(i)), String.format("%.4f", maxHitrateList.get(i++)));
//          }
//          Log.i(TAG, hitRatioFormatter.toString());
    }

    public ExecutorService getBenchmarkRunnerService() {
        return benchmarkRunnerService;
    }

    private void submitCountedBenchmark(final CacheBenchmarkConfiguration
        benchmarkConfiguration) {
        submitCountedBenchmark(benchmarkConfiguration, 100, 1_000_000);
    }

    private void submitCountedBenchmark(final CacheBenchmarkConfiguration
        benchmarkConfiguration, final long warmupIterations, final long
        runIterations) {
        benchmarkRunnerService.submit(new Runnable() {

            @Override
            public void run() {
                benchmarkConfiguration.runMany(warmupIterations, runIterations);
                CacheBenchmarkConfiguration.CacheStats stats =
                    benchmarkConfiguration.getStats();

                if (benchmarkResults.containsKey(stats.getPolicyTag())) {
                    benchmarkResults.get(stats.getPolicyTag()).add(stats);
                } else {
                    benchmarkResults.put(stats.getPolicyTag(), new LinkedList<>(
                        Arrays.asList(stats)));
                }

                Log.i(TAG, benchmarkConfiguration.getStats().toString());
                resetEnvironment();
            }
        });
    }

    private void submitCountedReadBenchmarks(int lowerBound, int upperBound,
        double cachedRatio, String traceTag, Generator<Integer> generator, int
        warmupIterations, int runIterations) {
        final int cacheSize = (int) Math.round((upperBound - lowerBound) *
            cachedRatio);
        submitCountedBenchmark(new GuavaBenchmarks.Read(traceTag, generator,
            cachedRatio, lowerBound, upperBound), warmupIterations, runIterations)
            ;
        submitCountedBenchmark(new NativeLruBenchmarks.Read(traceTag, generator,
            cachedRatio, lowerBound, upperBound), warmupIterations, runIterations)
            ;
        submitCountedBenchmark(new CustomBenchmark.Read(FIFOCache.CACHE_TAG,
            traceTag, generator, cachedRatio, lowerBound, upperBound,
            generateFifoCache(cacheSize)), warmupIterations, runIterations);
        submitCountedBenchmark(new Cache2KBenchmark.Read(Cache2KBenchmark.
            RANDOM_CACHE, traceTag, generator, cachedRatio, lowerBound, upperBound
            ), warmupIterations, runIterations);
        submitCountedBenchmark(new JackRabbitLIRSBenchmark.Read(traceTag,
            generator, cachedRatio, lowerBound, upperBound), warmupIterations,
            runIterations);
        submitCountedBenchmark(new Cache2KBenchmark.Read(Cache2KBenchmark.
            CLOCK_CACHE, traceTag, generator, cachedRatio, lowerBound, upperBound)
            , warmupIterations, runIterations);
        submitCountedBenchmark(new Cache2KBenchmark.Read(Cache2KBenchmark.
            ARC_CACHE, traceTag, generator, cachedRatio, lowerBound, upperBound),
            warmupIterations, runIterations);
    }

    /**
     * Force the garbage collection to run, rather than suggesting it. This will
         make sure that every
     * benchmark will be run in a "fresh" memory environment, without the garbage
         collector kicking in
     * clearing objects of previous benchmarks during recording.
     */
    private void gc() {
        Log.v(TAG, "Running␣garbage␣collector");
        Object obj = new Object();
        ReferenceQueue queue = new ReferenceQueue();
        PhantomReference ref = new PhantomReference<>(obj, queue);
        obj = null;

        long end = System.currentTimeMillis() + 2_000;
        while (!ref.isEnqueued()) {
            System.gc();
            if (System.currentTimeMillis() > end) {
                Log.v(TAG, "No␣garbage␣collection␣needed!");
                return;
            }
        }

        Log.v(TAG, "Memory␣is␣garbage␣collected");
    }
}
```

Listing E.2: desmedt.frederik.cachebenchmarking.CacheBenchmarkConfiguration

```
package desmedt.frederik.cachebenchmarking;

import android.util.Log;
import android.util.Pair;

/**
 * A benchmark configuration ran by the {@link BenchmarkRunner}.
 * <p/>
 * Note how this class is immutable, this is to enforce a reliable never-changing
     configuration
 * that maintains the integrity of the end results.
 * After the benchmark is run {@link CacheStats} are generated that can be
     retrieved.
 * <p/>
 * Every cache benchmark configuration handles keys and values, where the key is {
     @link Comparable}.
 */
public abstract class CacheBenchmarkConfiguration<K extends Comparable, V> {

    /**
     * How many times should the configuration log updates in a complete
         configuration run
     * ({@link CacheBenchmarkConfiguration#runMany(long, long)} or {@link
         CacheBenchmarkConfiguration#runTimed(long, long)}.
     */
    private static final int CONFIGURATION_RUN_LOG_POINT_COUNT = 5;
    public final String TAG;
    private final String name;
    private final String policyTag;
    private final String traceTag;
    private final double cacheRatio;

    private K lowerKeyBound;
    private K upperKeyBound;

    private CacheStats stats;
    private long totalTimeNanos;

    public CacheBenchmarkConfiguration(String policyTag, String traceTag, double
        cacheRatio, K lowerBound, K upperBound) {
        this.name = policyTag + traceTag + "␣(" + String.format("%.1f%%",
            cacheRatio * 100) + ")";
        this.policyTag = policyTag;
        this.traceTag = traceTag;
        this.cacheRatio = cacheRatio;
        TAG = CacheBenchmarkConfiguration.class.getSimpleName() + "␣-␣" + name;
        this.lowerKeyBound = lowerBound;
        this.upperKeyBound = upperBound;
    }

    public K getLowerKeyBound() {
        return lowerKeyBound;
    }

    public K getUpperKeyBound() {
        return upperKeyBound;
    }

    public String getName() {
        return name;
    }

    public double getCacheRatio() {
        return cacheRatio;
    }

    public String getPolicyTag() {
        return policyTag;
    }

    public String getTraceTag() {
        return traceTag;
    }

    /**
     * Run the operation the benchmark is supposed to evaluate exactly once. The
         behaviour,
     * such as the speed, of this method is recorded and used to generate the
         final result of the
     * configuration. Therefore it is essential that this is as high performance
         as it can be, e.g.
     * you shouldn't log non-essential information or have an entire try-catch
         block unless if this
     * is absolutely necessary.
     *
     * @param key    The key used in the operation, can be null if it is not used
         in some specific
     *               implementation. The key will always be within the lower and
```

```
        upper bounds.
     * @param value The value used in the operation, can be null if it is not used
         in some specific
     *              implementation.
     * @return true if the run succeeded, say by a cache success, false if the run
         did not succeed,
     * say by a cache miss
     */
    protected abstract boolean run(K key, V value);

    /**
     * Generates a possible input for the next run. If the input to {@link
        CacheBenchmarkConfiguration#run(Comparable, Object)}
     * is irrelevant this could return <code>null</code>. Note how this can have
        an arbitrary
     * probability distribution when picking a value from the input space. In
        other words, you could
     * always return the same value from the input space (the obvious one being <
        code>null</code>),
     * or you could have a perfectly random distribution. This method is not
        recorded/timed.
     *
     * @return A key-value pair used as a possible input for a single run
     */
    protected abstract Pair<K, V> generateInput();

    /**
     * Generates statistics regarding the cache that is being benchmarked. Based
        on the type of benchmark
     * some of the properties of the returned stats are allowed to be null.
     * <p/>
     * This method should not be used when trying to get an overview of benchmark
        statistics,
     * {@link CacheBenchmarkConfiguration#getStats()} should be used instead.
     *
     * @return Statistics relating to the cache
     * @see CacheStats
     */
    protected abstract CacheStats generateStats();

    /**
     * Optional step performed after initialization and before the benchmark run.
     * Here the benchmark configuration has the chance of performing operations
        that should be run a
     * single time before the benchmark is started, like initializing the cache.
     * This method is not recorded/timed.
     */
    protected void setup() {
    }

    /**
     * Optional step performed after the complete benchmark run with
     * {@link CacheBenchmarkConfiguration#runMany(long, long)} and
     * {@link CacheBenchmarkConfiguration#runTimed(long, long)}. Here the
        benchmark configuration has
     * the chance of performing operations that should be run a single time after
        the benchmark run,
     * like purging the cache. This method is not recorded/timed.
     */
    protected void tearDown() {
    }
    /**
     * Optional intermediary step performed before each single run. Here the
         benchmark configuration has
     * the chance of performing dependent operations that are required in the run,
          yet should not be
     * recorded. Therefore this method is not recorded/timed.
     */
    protected void prepare() {
    }

    /**
     * Optional intermediary step performed after each single run. Here the
         benchmark configuration has the
     * chance of cleaning everything up to maintain reliable runs. This method is
         not recorded/timed.
     *
     * @param key        The key of the last run
     * @param value      The value of the last run
     * @param succeeded  Whether the last run succeeded or not
     */
    protected void cleanup(K key, V value, boolean succeeded) {
    }

    /**
     * Generates a legal key-value pair to be used as an input for a single run by
          using
     * {@link CacheBenchmarkConfiguration#generateInput()} and then checking the
         lower and upper bounds.
     *
     * @return A legal input key-value pair
     */
    private Pair<K, V> generateLegalInput() {
        final Pair<K, V> input = generateInput();

        if (input != null && (input.first.compareTo(lowerKeyBound) < 0 || input.
             first.compareTo(upperKeyBound) > 0)) {
            throw new IllegalArgumentException("Generated␣an␣input␣" + input.
                 toString() + "␣that␣is␣either␣lower␣or␣higher␣than␣the␣lower␣or␣
                 upper␣bound!");
        }

        return input;
    }

    /**
     * Runs the configuration <code>warmupIterations + runIterations</code> times,
          using
     * {@link CacheBenchmarkConfiguration#run(Comparable, Object)} to execute the
         operation and
     * {@link CacheBenchmarkConfiguration#generateInput()} to generate a random
         input. These results will
     * then be collected and returned.
     *
     * @param warmupIterations How many iterations the configuration should be run
          before recording
     *                         the results
     * @param runIterations    How many iterations the configuration should be run
          and recorded
     * @returns The final result after completing all iterations
     */
    public final void runMany(long warmupIterations, long runIterations) {
        setup();
        final long logPoint = runIterations / CONFIGURATION_RUN_LOG_POINT_COUNT;

        Log.v(TAG, "Starting␣warmup");
        for (int i = 0; i < warmupIterations; i++) {
            runAndRecord();
        }

        Log.v(TAG, "Completed␣warmup,␣starting␣run");

        for (int i = 0; i < runIterations; i++) {
            runAndRecord();
            if (i % logPoint == 0 && i != 0) {
                Log.v(TAG, String.format("Reached␣%d␣iterations␣after␣%d␣millis",
                    i, totalTimeNanos / 1_000_000));
            }
        }

        stats = generateStats();
        stats.benchmarkName = getName();
        stats.policyTag = policyTag;
        stats.traceTag = traceTag;
        stats.cacheRatio = cacheRatio;
        stats.averageRunTime = totalTimeNanos / runIterations;
        tearDown();
        Log.v(TAG, "Completed␣run");
    }

    /**
     * Runs the configuration for <code>millis</code> milliseconds, using
     * {@link CacheBenchmarkConfiguration#run(Comparable, Object)} to execute the
         operation and
     * {@link CacheBenchmarkConfiguration#generateInput()} to generate a random
         input. These results will
     * then be collected and returned.
     *
     * @param warmupMillis How long the warmup run should be in milliseconds
     * @param runMillis    How long the actual recorded run should be in
         milliseconds
     * @returns The final result after completing all iterations fitting in <code>
         runMillis</code> milliseconds
     */
    public final void runTimed(long warmupMillis, long runMillis) {
        setup();
        long nextLogPoint = runMillis / CONFIGURATION_RUN_LOG_POINT_COUNT;

        Log.i(TAG, "Starting␣warmup");
        while (totalTimeNanos < warmupMillis) {
            runAndRecord();
        }

        Log.i(TAG, "Completed␣warmup,␣starting␣run");

        totalTimeNanos = 0;
        int totalIterations = 0;
        while (totalTimeNanos < runMillis) {
            totalIterations++;
            runAndRecord();
            if (totalTimeNanos / 1_000_000 >= runMillis) {
                // Stop the loop as the recording passed the specified run time
                // Repeating the run until there is a recording that fits in the
                    specified run time
                // is both indeterministic and unfair.
                break;
            }

            if (totalTimeNanos >= nextLogPoint) {
                Log.v(TAG, String.format("Reached␣%d␣iterations␣after␣%d␣millis",
                    totalIterations,
                        totalTimeNanos / 1_000_000));
                nextLogPoint = nextLogPoint + runMillis /
                    CONFIGURATION_RUN_LOG_POINT_COUNT;
            }
        }

        stats = generateStats();
        stats.benchmarkName = getName();
        stats.policyTag = policyTag;
        stats.traceTag = traceTag;
        stats.cacheRatio = cacheRatio;
        stats.averageRunTime = totalTimeNanos / totalIterations;
        tearDown();
        Log.i(TAG, "Completed␣run");
    }

    private void runAndRecord() {
        prepare();
        final Pair<K, V> input = generateLegalInput();
        long before = 0;
        long after = 0;
        boolean succeeded;

        if (input == null) {
            before = System.nanoTime();
            succeeded = run(null, null);
            after = System.nanoTime();
        } else {
            before = System.nanoTime();
            succeeded = run(input.first, input.second);
            after = System.nanoTime();
        }

        cleanup(input.first, input.second, succeeded);
        totalTimeNanos += after - before;
    }

    /**
     * Get stats of the benchmark configuration, consisting of a combination of
         statistics generated
     * by the base {@link CacheBenchmarkConfiguration} and statistics generated by
          the cache itself.
     * <p/>
     * When the benchmark configuration is has not been run or is not yet finished
         , this will return
     * null.
     *
     * @return Reliable cache statistics
     */
    public CacheStats getStats() {
        return stats;
    }

    /**
     * Represents statistics of the cache used in the cache benchmark. Several
         statistics might be null
     * based on the use case.
     */
    public static class CacheStats {

        private Integer successCount;
        private Integer failureCount;
        private Integer maxCacheSize;
        private Integer cacheEntryCount;

        private String benchmarkName;
        private double averageRunTime;

        private String policyTag;
        private String traceTag;
        private double cacheRatio;

        private final StatType type;

        private CacheStats(StatType type) {
            this.type = type;
        }

        /**
         * A Simple Factory used for creating {@link CacheStats} of a cache
             benchmark where reading
         * a cache is recorded.
         *
         * @param successCount     The amount of successful reads that have
             occurred in the current benchmark
         *                         configuration
         * @param failureCount     The amount of failed reads that have occurred in
              the current benchmark
         *                         configuration
         * @param cacheSize        The cache size of the cache used in the
             benchmark configuration (in entries), with
         *                         a dynamically sized cache this is the maximum
             amount of entries
         * @param cacheEntryCount The amount of cache entries in the cache used in
              the benchmark configuration
         * @return A {@link CacheStats} object containing the specified data
         */
        public static CacheStats read(int successCount, int failureCount, int
            cacheSize, int cacheEntryCount) {
            CacheStats metrics = new CacheStats(StatType.READ);
            metrics.successCount = successCount;
            metrics.failureCount = failureCount;
            metrics.maxCacheSize = cacheSize;
            metrics.cacheEntryCount = cacheEntryCount;
            return metrics;
        }

        /**
         * A Simple Factory used for creating {@link CacheStats} of a cache
             benchmark where
         * inserting, updating or deleting a cache is recorded.
         *
         * @param maxCacheSize     The cache size of the cache used in the
             benchmark configuration (in entries), with
         *                         a dynamically sized cache this is the maximum
             amount of entries
         * @param cacheEntryCount The amount of cache entries in the cache used in
              the benchmark configuration
     * @return A {@link CacheStats} object containing the specified data
     */
    public static CacheStats nonRead(StatType type, int maxCacheSize, int
        cacheEntryCount) {
        CacheStats metrics = new CacheStats(type);
        metrics.maxCacheSize = maxCacheSize;
        metrics.cacheEntryCount = cacheEntryCount;
        return metrics;
    }

    /**
     * @return The amount of successful reads that have occurred in the
         current benchmark
     * configuration. Null if the benchmark configuration is not a reading
         benchmark.
     */
    public Integer getSuccessCount() {
        return successCount;
    }

    /**
     * @return The amount of failed reads that have occurred in the current
         benchmark
     * configuration. Null if the benchmark configuration is not a reading
         benchmark.
     */
    public Integer getFailureCount() {
        return failureCount;
    }

    /**
     * @return The cache size of the cache used in the benchmark configuration
          (in entries), with
     * a dynamically sized cache this is the maximum amount of entries.
     */
    public Integer getMaxCacheSize() {
        return maxCacheSize;
    }

    /**
     * @return The amount of cache entries in the cache used in the benchmark
         configuration.
     * Null if this does not make sense in the current benchmark configuration
         , e.g. a delete
     * benchmark that should always have 0 cache entries after each run.
     */
    public Integer getCacheEntryCount() {
        return cacheEntryCount;
    }

    private void setBenchmarkName(String benchmarkName) {
        this.benchmarkName = benchmarkName;
    }

    private void setCacheRatio(double cacheRatio) {
        this.cacheRatio = cacheRatio;
    }

    private void setPolicyTag(String policyTag) {
        this.policyTag = policyTag;
    }
    private void setTraceTag(String traceTag) {
        this.traceTag = traceTag;
    }

    /**
     * Sets the average runtime, should only be set by the base {@link
         CacheBenchmarkConfiguration}.
     *
     * @param averageRunTime The average runtime in nanoseconds
     */
    private void setAverageRunTime(double averageRunTime) {
        this.averageRunTime = averageRunTime;
    }

    public double getAverageRunTime() {
        return averageRunTime;
    }

    public String getBenchmarkName() {
        return benchmarkName;
    }

    /**
     * @return The average hitrate of the run, which is a number where {@code
         0 <= val <= 100}
     */
    public double getHitrate() {
        return (double) successCount / (successCount + failureCount) * 100;
    }

    public String getPolicyTag() {
        return policyTag;
    }

    public String getTraceTag() {
        return traceTag;
    }

    public double getCacheRatio() {
        return cacheRatio;
    }

    public String getPolicyWithCacheRatio() {
        return policyTag + "␣(" + String.format("%.3f)", cacheRatio);
    }

    public StatType getStatType() {
        return type;
    }

    @Override
    public String toString() {
        final StringBuilder builder = new StringBuilder(String.format("%-25s␣"
            , benchmarkName));

        builder.append(String.format("Cache␣size:␣%-5d␣", maxCacheSize));

        if (getStatType() == StatType.READ) {
            builder.append(String.format("Hit␣ratio:␣%-5.3f%%␣␣␣␣␣", (double)
                successCount / (successCount + failureCount) * 100));
        }
```

```
            builder.append(String.format("Average␣(ns):␣%-7.1f␣␣␣␣␣",
                averageRunTime));

            if (successCount != null) {
                builder.append(String.format("Successes:␣%-8d␣␣␣␣␣", successCount)
                    );
            }

            if (failureCount != null) {
                builder.append(String.format("Failures:␣%-8d␣␣␣␣␣", failureCount))
                    ;
            }

            if (cacheEntryCount != null) {
                builder.append(String.format("Cache␣entries:␣%-5d",
                    cacheEntryCount));
            }

            return builder.toString();
        }
    }

    public enum StatType {
        READ, INSERT, UPDATE, DELETE
    }
}
```

E.2 Package cachebenchmarking.benchmark

Listing E.3: desmedt.frederik.cachebenchmarking.benchmark.BaseBenchmark

```
package desmedt.frederik.cachebenchmarking.benchmark;

import android.util.Pair;

import java.util.Random;

import desmedt.frederik.cachebenchmarking.CacheBenchmarkConfiguration;
import desmedt.frederik.cachebenchmarking.generator.Generator;
import desmedt.frederik.cachebenchmarking.generator.RandomGenerator;
import desmedt.frederik.cachebenchmarking.generator.ZipfGenerator;

/**
 * A collection of basic {@link desmedt.frederik.cachebenchmarking.
    CacheBenchmarkConfiguration} that
 * implement common code while remaining cache independent.
 */
public class BaseBenchmark {

    public static final String INSERT_TAG = "Insert";
    public static final String DELETE_TAG = "Delete";
    public static final String UPDATE_TAG = "Update";

    /**
     * Base benchmark configuration used by all static classes in {@link
         BaseBenchmark}.
     * It contains all common functionalities of its subclasses as well as
         management of benchmark
     * {@link CacheBenchmarkConfiguration#setup()} and {@link
         CacheBenchmarkConfiguration#tearDown()}.
     * <p/>
     * It expects the cache to be based on {@link Integer} keys.
     *
     * @param <V> The type of values that will be stored in the cache
     */
    public static abstract class BaseBenchmarkConfiguration<V> extends
        CacheBenchmarkConfiguration<Integer, V> {

        private int cacheSize;

        public BaseBenchmarkConfiguration(String policyTag, String traceTag,
            double cachedRatio, Integer lowerBound, Integer upperBound) {
            super(policyTag, traceTag, cachedRatio, lowerBound, upperBound);
            cacheSize = (int) Math.round((upperBound - lowerBound) * cachedRatio);
        }

        @Override
        protected void setup() {
            createCache(cacheSize);
        }

        @Override
        protected void tearDown() {
            clearCache();
        }

        /**
         * Generate a random value to be used as a value in a run.
         *
         * @return A random value
         */
        protected abstract V generateValue();

        /**
         * Create the cache that is used for benchmarking of size {@code cacheSize
             }. This method is
         * called only once before the benchmark run and is not timed.
         *
         * @param cacheSize The maximum amount of entries the cache should have
         */
        protected abstract void createCache(int cacheSize);

        /**
         * Completely clear the cache used for benchmarking. This means cleaning
             as much as possible
         * and possibly even removing the cache reference for the garbage
             collector in case there are
         * lots of strong references that are maintained. This method is called
             only after the complete
         * benchmark run and is not timed.
         */
        protected abstract void clearCache();

        public int getCacheSize() {
            return cacheSize;
        }
    }

    public static abstract class Read<V> extends BaseBenchmarkConfiguration<V> {
        private Generator<Integer> randomGenerator;

        /**
         *
         * @param name The name of the cache policy used
         * @param traceTag The name of trace used
         * @param traceGenerator A generator representing some trace
         * @param cachedRatio How much of the total key space should be available
         *    in the cache, {@code 0 <= cachedRatio <= 1}
         * @param lowerBound The lower bound of the key space
         * @param upperBound The upper bound of the key space
         */
        public Read(String name, String traceTag, Generator<Integer>
            traceGenerator, double cachedRatio, Integer lowerBound, Integer
            upperBound) {
            super(name + "Read", traceTag, cachedRatio, lowerBound, upperBound);
            randomGenerator = traceGenerator;
        }

        protected abstract void addToCache(Integer key, V value);

        @Override
        protected void cleanup(Integer key, V value, boolean succeeded) {
            if (!succeeded) {
                addToCache(key, value);
            }
        }

        @Override
        protected Pair<Integer, V> generateInput() {
            return new Pair<>(randomGenerator.next(), generateValue());
        }
    }

    /**
     * A default benchmark configuration for reading a cache. It simulates random
     *    cache access by
     * continuously generating random values before each individual run.
     * <p/>
     * It expects the cache to be based on {@link Integer} keys.
     *
     * @param <V> The type of the values that will be stored in the cache
     */
    public static abstract class RandomRead<V> extends BaseBenchmark.Read<V> {

        public RandomRead(String name, double cachedRatio, Integer lowerBound,
            Integer upperBound) {
            super(name, RandomGenerator.TRACE_TAG, new RandomGenerator(lowerBound,
                upperBound), cachedRatio, lowerBound, upperBound);
        }
    }

    /**
     * A default benchmark configuration for reading a cache. It simulates GET
     *    HTTP requests that
     * pass by the cache by continuously generating random values according to a
     *    Zipf probability
     * distribution ({@link ZipfGenerator}) before each individual run.
     * <p/>
     * It expects the cache to be based on {@link Integer} keys.
     *
     * @param <V> The type of the values that will be stored in the cache
     */
    public static abstract class ZipfRead<V> extends BaseBenchmark.Read<V> {


        public ZipfRead(String name, double cachedRatio, Integer lowerBound,
            Integer upperBound) {
            super(name, ZipfGenerator.TRACE_TAG, new ZipfGenerator(lowerBound,
                upperBound), cachedRatio, lowerBound, upperBound);
        }
    }

    /**
     * A default insert benchmark, that is used to monitor the performance of
        inserting key-value pairs
     * in a cache. The key passed to the run will always be a key of an entry that
         is not currently
     * stored in the cache. Therefore it will always be a pure insert run and will
         never be updating
     * an existing entry.
     * <p/>
     * It expects the cache to be based on {@link Integer} keys.
     * The keys that are used are completely random.
     *
     * @param <V> The type of the values that will be stored in the cache
     */
    public static abstract class Insert<V> extends BaseBenchmarkConfiguration<V> {

        private Generator<Integer> generator;
        private int nextKey;

        public Insert(String name, double cachedRatio, Integer lowerBound, Integer
             upperBound) {
            super(name + INSERT_TAG, RandomGenerator.TRACE_TAG, cachedRatio,
                lowerBound, upperBound);
            generator = new RandomGenerator(lowerBound, upperBound);
        }

        protected abstract void removeElement(Integer key);

        protected abstract V generateValue();

        @Override
        protected void cleanup(Integer key, V value, boolean succeeded) {
            nextKey = generator.next();
            removeElement(key);
        }

        @Override
        protected Pair<Integer, V> generateInput() {
            return new Pair<>(nextKey, generateValue());
        }
    }

    /**
     * A default benchmark configuration for deleting an entry from a cache.
     * The key passed to the run will always be a key of an existing entry in the
        cache. Therefore
     * the run is never told to try and remove the entry bound to the key that is
        not in the cache.
     * <p/>
     * It expects the cache to be based on {@link Integer} keys.
     * The keys generated are completely random.
```

```
 *
 * @param <V> The type of the values that will be stored in the cache
 */
public static abstract class Delete<V> extends BaseBenchmarkConfiguration<V> {

    private Generator<Integer> generator;
    private int nextKey;

    public Delete(String name, double cachedRatio, Integer lowerBound, Integer
         upperBound) {
        super(name + DELETE_TAG, RandomGenerator.TRACE_TAG, cachedRatio,
            lowerBound, upperBound);
        generator = new RandomGenerator(lowerBound, upperBound);
    }

    public abstract void addToCache(int key, V value);

    protected abstract V generateValue();

    @Override
    protected void setup() {
        super.setup();
        for (int i = 0; i < getCacheSize(); i++) {
            prepare();
        }
    }

    @Override
    protected void prepare() {
        nextKey = generator.next();
        addToCache(nextKey, generateValue());
    }

    @Override
    protected Pair<Integer, V> generateInput() {
        return new Pair<>(nextKey, generateValue());
    }
}

/**
 * A default benchmark configuration for updating entries in a cache.
 * The key passed to the run might be bound to an already existing entry in
     the cache, yet this
 * is not enforced. Considering that a cache update is almost always used with
     a
 * "update or add if non existent" semantics.
 * <p/>
 * It expects the cache to be based on {@link Integer} keys.
 * The keys generated are completely random.
 *
 * @param <V> The type of the values that will be stored in the cache
 */
public static abstract class Update<V> extends BaseBenchmarkConfiguration<V> {

    private Generator<Integer> generator;
    private int nextKey;

    public Update(String name, double cachedRatio, Integer lowerBound, Integer
         upperBound) {
        super(name + UPDATE_TAG, RandomGenerator.TRACE_TAG, cachedRatio,
            lowerBound, upperBound);
        generator = new RandomGenerator(lowerBound, upperBound);
        }

        protected abstract void addToCache(int key, V value);

        protected abstract V generateValue();

        @Override
        protected void prepare() {
            nextKey = generator.next();
            addToCache(nextKey, generateValue());
        }

        @Override
        protected Pair<Integer, V> generateInput() {
            return new Pair<>(nextKey, generateValue());
        }
    }
}
```

E.3 Package cachebenchmarking.cache

Listing E.4: desmedt.frederik.cachebenchmarking.cache.Cache

```
package desmedt.frederik.cachebenchmarking.cache;

/**
 * Interface for every custom cache implementation.
 */
public interface Cache<K extends Comparable<K>, V> {

    /**
     * Get the value of the entry associated with the key or null if there is no
         entry in the cache
     * linked to the key.
     *
     * @param key The key associated with the entry
     * @return The value if the key exists in the cache, false otherwise
     */
    V get(K key);

    /**
     * Put a new entry in the cache with a key and a value.
     *
     * @param key   The key of the entry
     * @param value The value of the entry
     */
    void put(K key, V value);

    /**
     * Removes a single entry from the cache, or does nothing if the key is not
         present.
     *
     * @param key The key of the entry that should be removed
     */
    void remove(K key);

    /**
```

```
     * Removes all elements from the cache.
     */
    void removeAll();

    /**
     * @return The maximum amount of entries present in the cache at any given
          time
     */
    int maxSize();

    /**
     * @return The current amount of entries present in the cache
     */
    int size();

    class Element<K extends Comparable<K>, V> implements Comparable<K> {

        private K key;
        private V value;

        public Element(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public void setKey(K key) {
            this.key = key;
        }

        public V getValue() {
            return value;
        }

        public void setValue(V value) {
            this.value = value;
        }

        @Override
        public boolean equals(Object o) {
            if (o instanceof Element) {
                return key.equals(((Element) o).getKey()) && value.equals(((
                    Element) o).getValue());
            } else {
                return key.equals(o);
            }
        }

        @Override
        public int compareTo(K key) {
            return getKey().compareTo(key);
        }
    }
}
```

Listing E.5: desmedt.frederik.cachebenchmarking.cache.FIFOCache

```
package desmedt.frederik.cachebenchmarking.cache;
```

```
import java.util.LinkedHashMap;
import java.util.ListIterator;
import java.util.Map;

/**
 * A simple FIFO cache replacement policy implementation. Uses the FIFO mode of {
      @link LinkedHashMap}
 * to store, retrieve and remove its elements.
 */
public class FIFOCache<K extends Comparable<K>, V> implements Cache<K, V> {

    public static final String CACHE_TAG = "FifoCache";

    private LinkedHashMap<K, V> heap = new LinkedHashMap<>();
    private int maxSize = 0;

    public FIFOCache(int maxSize) {
        this.maxSize = maxSize;
    }

    @Override
    public V get(K key) {
        return heap.get(key);
    }

    @Override
    public void put(K key, V value) {
        heap.put(key, value);

        if (maxSize < heap.size()) {
            removeElement();
        }
    }

    private void removeElement() {
        K key = heap.keySet().iterator().next();
        heap.remove(key);
    }

    @Override
    public void remove(K key) {
        heap.remove(key);
    }

    @Override
    public void removeAll() {
        heap.clear();
    }

    @Override
    public int maxSize() {
        return maxSize;
    }

    @Override
    public int size() {
        return heap.size();
    }
}
```

Listing E.6: desmedt.frederik.cachebenchmarking.cache.RandomCache

```java
package desmedt.frederik.cachebenchmarking.cache;

import java.util.Arrays;
import java.util.Collections;
import java.util.LinkedList;
import java.util.Random;
import java.util.StringTokenizer;

import desmedt.frederik.cachebenchmarking.cache.Cache;

/**
 * A simple random cache replacement policy implementation.
 */
public class RandomCache<K extends Comparable<K>, V> implements Cache<K, V> {

    public static final String CACHE_TAG = "RandomCache";

    private int maxSize;
    private LinkedList<Element<K, V>> heap = new LinkedList<>();
    private Random random = new Random();

    public RandomCache(int maxSize) {
        this.maxSize = maxSize;
    }

    @Override
    public V get(K key) {
        final int index = Collections.binarySearch(heap, key);
        return index < 0 ? null : heap.get(index).getValue();
    }

    @Override
    public void put(K key, V value) {
        int index = Collections.binarySearch(heap, key);
        if (index < 0) {
            if (maxSize < heap.size() + 1) {
                removeElement();
            }

            int insertIndex = -(index + 1);
            if (insertIndex == heap.size() + 1) {
                heap.add(new Element<K, V>(key, value));
            } else {
                heap.add(insertIndex, new Element<>(key, value));
            }
        } else {
            heap.get(index).setValue(value);
        }
    }

    @Override
    public void remove(K key) {
        int index = Collections.binarySearch(heap, key);
        if (index > 0) {
            heap.remove(index);
        }
    }

    @Override
    public void removeAll() {
        heap.clear();
    }

    private void removeElement() {
        final int index = random.nextInt(maxSize);
        heap.remove(index);
    }

    @Override
    public int maxSize() {
        return maxSize;
    }

    @Override
    public int size() {
        return heap.size();
    }
}
```

E.4 Package cachebenchmarking.generator

Listing E.7: desmedt.frederik.cachebenchmarking.generator.Generator

```java
package desmedt.frederik.cachebenchmarking.generator;

/**
 * Something that can generate values of type {@code E}. This interface is used
     with arbitrary
 * semantics, be it as a random instance generator or a one-time single instance
     generator.
 */
public interface Generator<E> {

    E next();
}
```