JCoz: A Causal Java Profiler

By David Vernet and Matt Perron

Example program with 32 threads

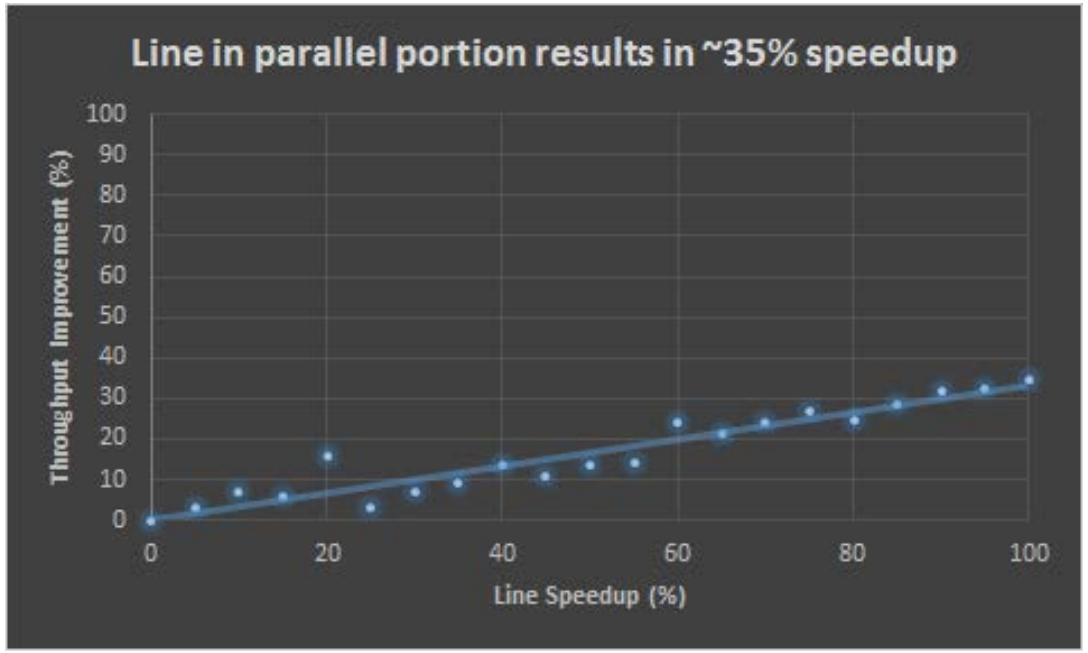
```
public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < numThreads; i++)
                threads.add(new ParallelWorker());

        while (true) {
                doParallel();
                doSerial();
                System.out.println("Iteration done");
        }
}

public static void doSerial() throws InterruptedException {
        long sum = 0;
        for (long i = 0; i < LOOP_ITERS; i++)
                sum += (System.nanoTime() % 9999);
}
```

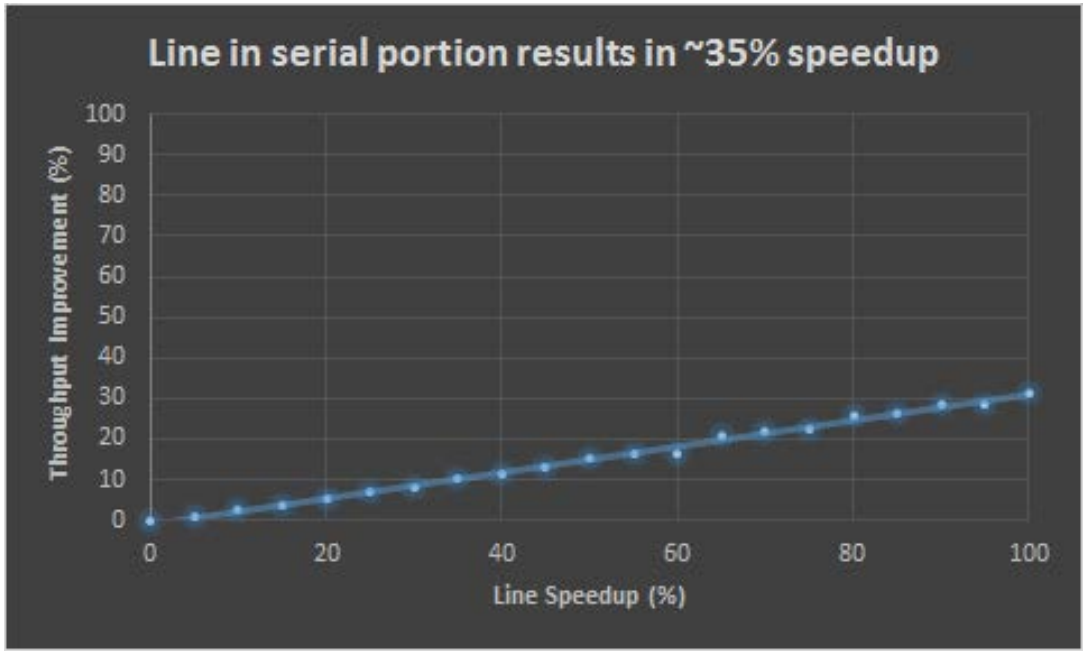
```
public static void doParallel() throws InterruptedException {
        executor.invokeAll(threads);
}

static class ParallelWorker implements Callable<Void> {
        public Void call() {
                long sum = 0;
                for (long i = 0; i < LOOP_ITERS; i++)
                        sum += (System.nanoTime() % 9999);

                return null;
        }
}
```

Standard sampling profiler output gives misleading results

Hot Spots - Method	Self Time [%]	Self Time	**Self Time (CPU)** ▼	Total Time	Total Time (CPU)
test.TestThreadSerial$ThreadTest.**call** ()		1,532,364 ms (97.5%)	1,532,364 ms	1,532,364 ms	1,532,364 ms
test.TestThreadSerial.**doSerial** ()		39,445 ms (2.5%)	39,445 ms	39,445 ms	39,445 ms
test.TestThreadSerial.**doParallel** ()		57,550 ms (0%)	99.2 ms	57,550 ms	99.2 ms
test.TestThreadSerial.**main** ()		0.000 ms (0%)	0.000 ms	96,996 ms	39,544 ms

JCoz gives accurate profile results





How does causal profiling work?

Virtual Speedup

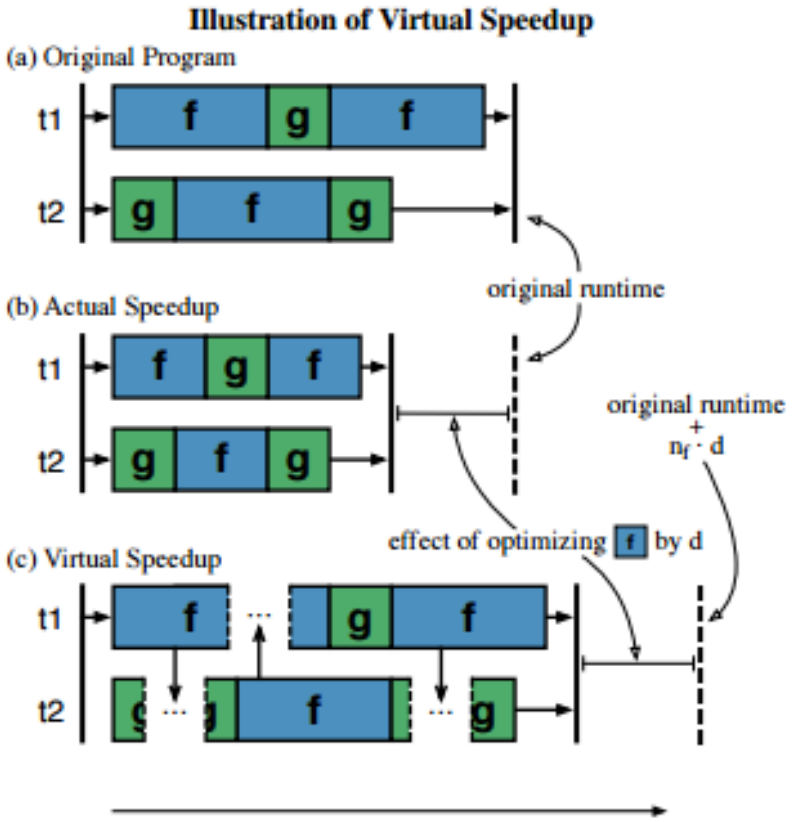


Figure 3: An illustration of virtual speedup: (a) shows the original execution of two threads running functions `f` and `g`; (b) shows the effect of a *actually* speeding up `f` by 40%; (c) shows the effect of *virtually* speeding up `f` by 40%. Each time `f` runs in one thread, all other threads pause for 40% of `f`'s original execution time (shown as ellipsis). The difference between the runtime in (c) and the original runtime plus $n_f \cdot d$—the number of times `f` ran times the delay size—is the same as the effect of actually optimizing `f`.

Photo credit: Charlie Curtsinger and Emery Berger

C. Curtsinger, E. Berger. COZ: Finding Code that Counts with Causal Profiling. *SOSP '15 ACM SIGOPS*

Virtual Speedup implementation

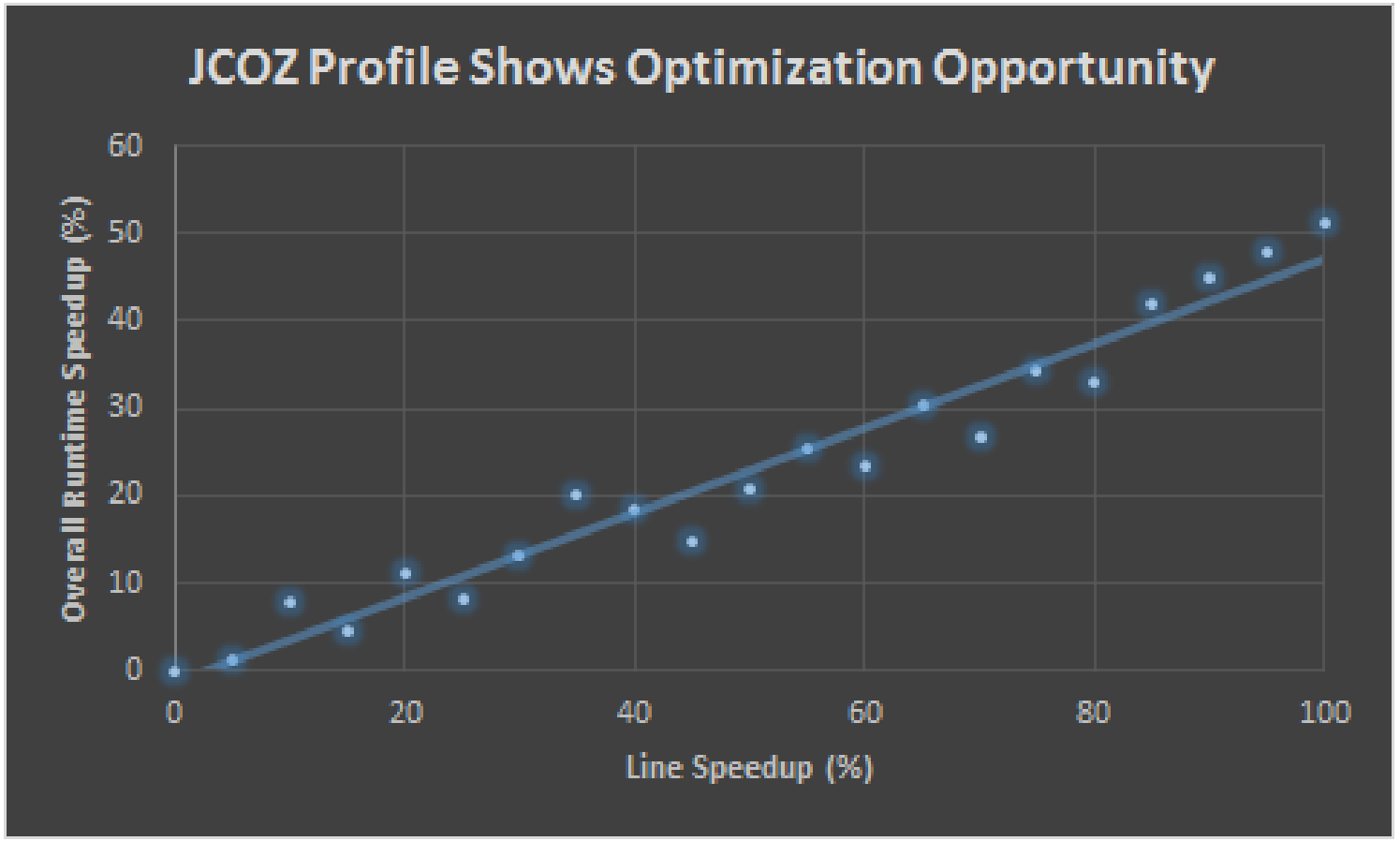
- Throughout runtime of program, run “experiments”
  - Randomly choose a recently executed line
  - Randomly choose a speedup between 0-100%
  - Every time a thread hits that line in a sample, freeze other threads for the speedup threshold chosen above
  - Measure how much throughput is achieved for the given experiment line and speedup
- At the end of the program (when enough results have been collected), experiments that indicate biggest throughput increases relative to the experiment runtime are highlighted as lines for optimization

Library optimizations with JCoz

Optimized Java H2 Database engine

- Using JCoz, we were able to optimize the widely used, mature, Java H2 Database Engine: http://www.h2database.com/html/main.html
- Found and implemented optimization within 2 hours of running profiler
- No previous knowledge of codebase
- Measured optimization of TPCC using the Dacapo Benchmark Suite
- Used in many projects, including Apache Cayenne, Apache Jackrabbit, Jboss Jopr, and the NIH

JCoz profile output for a line in H2 codebase



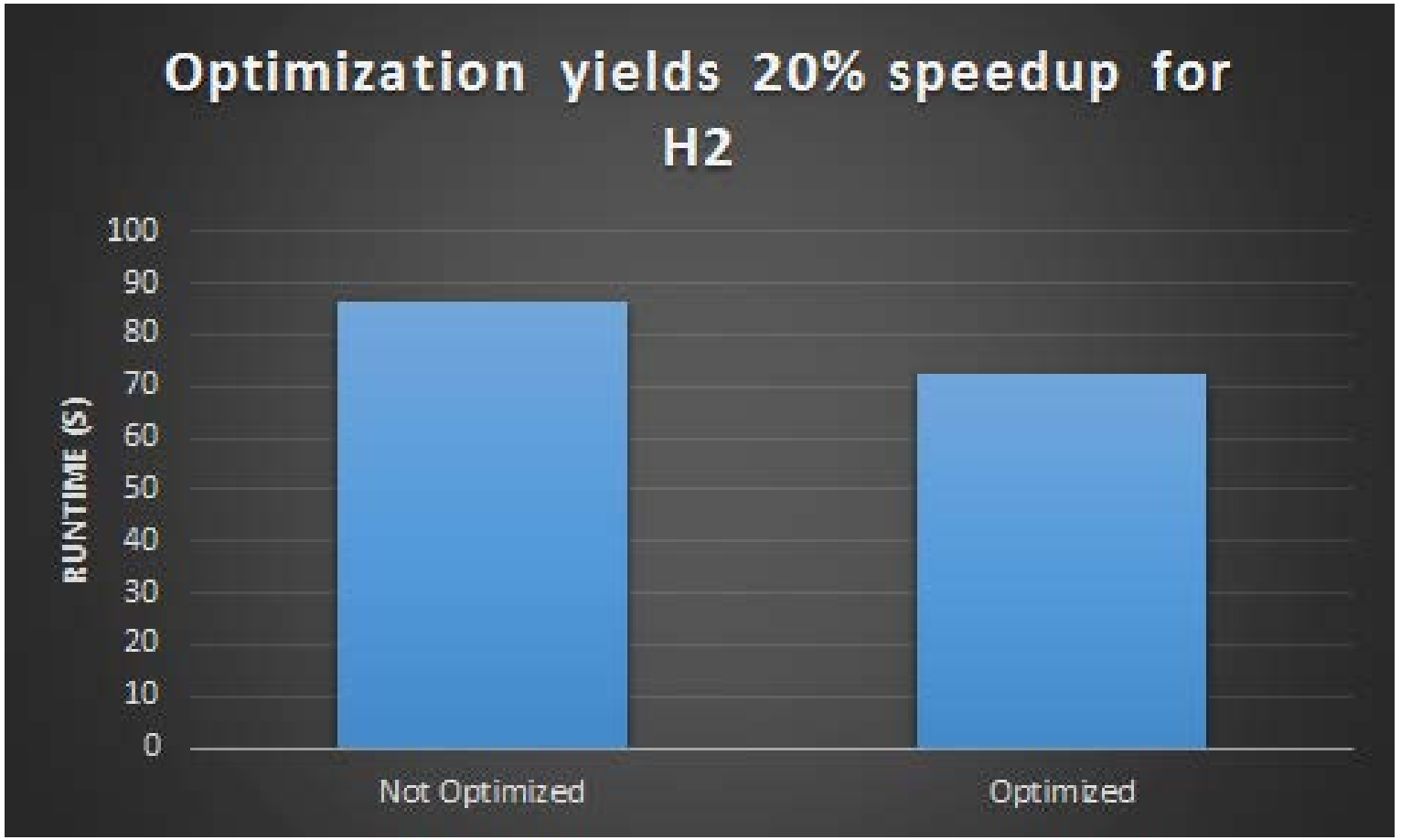
Profiled line was excessive sleep on transaction failure

```
Database database = session.getDatabase();
int sleep = 1 + MathUtils.randomInt(10);
while (true) {
    try {
        if (database.isMultiThreaded()) {
            Thread.sleep(sleep);
        } else {
            database.wait(sleep);
        }
    } catch (InterruptedException e1) {
        // ignore
    }
    long slept = System.nanoTime() / 1000000 - now;
    if (slept >= sleep) {
        break;
    }
}
```

Simple fix – sleep for less time

```
Database database = session.getDatabase();

// Sleep for 1-10 microseconds instead of 1-10 milliseconds
int sleep = 1000 * (1 + MathUtils.randomInt(10));
while (true) {
    try {
        if (database.isMultiThreaded()) {
            Thread.sleep(0, sleep);
        } else {
            database.wait(0, sleep);
        }
    } catch (InterruptedException e1) {
        // ignore
    }
    long slept = System.nanoTime() - now;
    if (slept >= sleep) {
        break;
    }
}
```

Optimization successful – 20% speedup



JCoz produced expected results

- No prior knowledge of codebase needed
- Pointed to the exact line of bottleneck
- Optimization was not made obvious using other profilers

Conclusion

JCoz works

- JCoz is the first causal Java profiler
- Provides more valuable information than existing profilers for multithreaded Java programs
- Allows profiling of throughput rather than just runtime
- Low runtime overhead

Future work

- Continue optimizing libraries
  - Another optimization opportunity discovered in the Universal Java Matrix Package (UJMP) (poor cache locality on dense matrix multiplication)
- Add more features for tuning profiler
  - Add latency profiling
- Explore ways to extend causal profiling to multi-process and distributed applications

Acknowledgements

- Special thanks to:
  - Charlie Curtsinger and Emery Berger for discovering causal profiling and making the original COZ program available for reference
  - David Capwell and Jeremy Manson for their code that we leveraged to use the infamous undocumented AsyncGetStackTraces JVMTI function
  - Professor Railing, for giving us guidance during this process

Questions?

Appendix

Another simple multithreaded example

```
public static void main(String[] args) throws InterruptedException {
        threads.add(new LongWorker());
        threads.add(new ShortWorker());

        while (true) {
                doParallel();
                System.out.println("Iteration done");
        }
}

public static void doParallel() throws InterruptedException {
        executor.invokeAll(threads);
}
```

```
static class LongWorker implements Callable<Void> {
        public Void call() {
                long sum = 0;
                for (long i = 0; i < 20000000L; i++)
                        sum += (System.nanoTime() % 9999);
                System.out.println("Long done");
                return null;
        }
}

static class ShortWorker implements Callable<Void> {
        public Void call() {
                long sum = 0;
                for (long i = 0; i < 10000000L; i++)
                        sum += (System.nanoTime() % 9999);
                System.out.println("Short done");
                return null;
        }
}
```

Sampling profile again shows misleading results

Hot Spots - Method	**Self Time [%]** ▼	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
test.Test$LongWorker.**call** ()		83,398 ms (63.7%)	83,398 ms	83,398 ms	83,398 ms
test.Test$ShortWorker.**call** ()		42,672 ms (32.6%)	42,672 ms	42,672 ms	42,672 ms
test.Test.**doParallel** ()		4,695 ms (3.6%)	0.000 ms	4,695 ms	0.000 ms
test.Test.**main** ()		96.1 ms (0.1%)	96.1 ms	4,791 ms	96.1 ms

JCoz gives accurate profile results