General Data Protection Runtime: Enforcing Transparent GDPR Compliance for Existing Applications

Supplemental

David Klein
Technische Universität Braunschweig
david.klein@tu-braunschweig.de

Benny Rolle
SAP SE
benny.rolle@sap.com

Thomas Barber
SAP Security Research
thomas.barber@sap.com

Manuel Karl
Technische Universität Braunschweig
m.karl@tu-braunschweig.de

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de

1 BENCHMARKING

In addition to the performance measurements in Section 7.5 which quantify the overhead for full applications, we measured the performance impact of different components of Fontus. To this end we used two standard benchmark suites, the DaCapo benchmarking suite [2] and the "Yahoo! Cloud Serving Benchmark" [3] (YCSB) framework. Both benchmark suites were ran on a Thinkpad T14s with AMD Ryzen 7 PRO 4750U 8 core CPU and 32GB RAM.

1.1 DaCapo

The DaCapo benchmarking suite consists of several Java applications covering a diverse set of application domains and behavior. We used DaCapo version f4800648.[1] To account for JIT compilation, we set up the benchmark execution in the same way as Bell and Kaiser [1]. That is, we ran the same benchmark repeatedly in the same JVM instance until the results converge, such that they do not deviate by over 3% for the 3 last executions. Once the result converged, we then took the next run as the final measurement. This process was repeated 5 times, and we report the average over those runs. This is in line with academic recommendations, e.g., by Georges et al. [4].

Table 1: Runtime Overhead induced by Fontus

Benchmark	Regular Runtime		Tainted Runtime		Overhead
avrora	10,772.4 ms	±601.0	11,501.2 ms	±383.8	6.8%
batik	1,544.0 ms	±10.8	1,717.0 ms	±17.4	11.2%
biojava	9,041.2 ms	±183.6	18,482.6 ms	±350.7	104.4%
graphchi	6,937.4 ms	±587.8	6,774.4 ms	±601.9	−2.3%
luindex	4,765.0 ms	±92.1	5,107.6 ms	±256.1	7.2%
sunflow	8,109.8 ms	±557.8	8,013.4 ms	±512.4	−1.2%
zxing	2,187.6 ms	±52.4	2,310.0 ms	±21.2	5.6%
fop	671.8 ms	±67.5	898.6 ms	±18.6	33.8%
h2	4,991.4 ms	±56.8	10,541.6 ms	±148.3	111.2%
jme	6,921.2 ms	±11.5	7,000.4 ms	±15.9	1.1%
Average					27.8%

[1]This is a development version of the benchmark suite, as official releases are not compatible with Java 11.

Table 1 shows the overhead when running different benchmarks from the DaCapo suite running with Fontus compared to running it directly. The table shows that for different applications the overhead can vary by a significant degree. This is caused by the different kind of workloads present in the applications. The *sunflow* benchmark, for example, has negligible overhead, which is due to it being a rendering engine, so almost all the executed code is related to, e.g., matrix multiplications, where almost no additional code is injected by Fontus. *Biojava*, on the other hand, has an overhead of 104.4%. This application performs genome sequencing over string values, so effectively all the code involves the creation and garbage collection of strings, causing a large amount of Fontus code to be executed.

Interestingly, for *graphchi* and *sunflow* the benchmarks run faster with Fontus. This is an artifact of the JVM optimization process and the benchmarks code structure. During the initial instrumentation by Fontus certain methods are invoked frequently enough to cause the JIT compiler to heavily optimize them, while they are never optimized in a regular run. When setting the threshold of invocations before the JIT optimizes them to 10, running the benchmark natively is faster again. This highlights the difficulties of benchmarking complex systems such as the JVM.

1.2 Yahoo! Cloud Serving Benchmark

The YCSB framework is a widely used benchmark to compare how different data storage systems such as relational databases perform for varying workloads. While we do not want to compare different database backends, it is a convenient way to run workloads of different sizes and operations against a database. To measure the impact of the taint persistence component detailed in Section 6.2, we set up YCSB as follows: We installed a MariaDB server locally and set up two databases with the YCSB table layout. For the second database schema, we preprocessed the database to add the taint columns. We then ran YCSB with the JDBC backend, once without any modifications and once with Fontus and its JDBC driver. We report the numbers for *Workload A* which consists of a roughly equal amount of read and write operations, varying the number of total operations.

As the number of operations goes up, the throughput difference goes down. This can be explained by the fact, that our JDBC driver caches tainted SQL queries. As parsing SQL is fairly expensive,

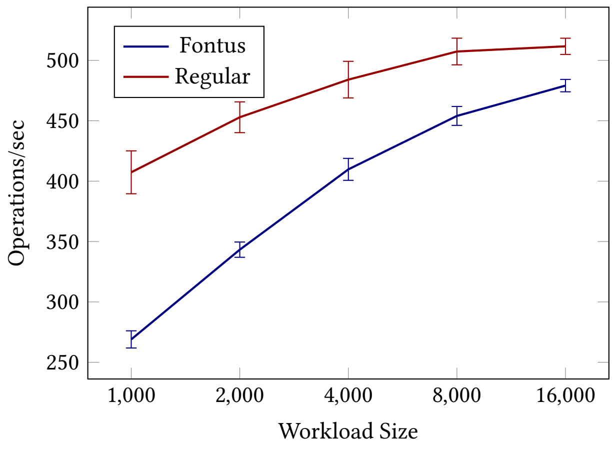

Figure 1: Fontus Overhead for YCSB

Table 2: Latency difference for YCSB and Fontus

App	Regular				Fontus			
	Avg		Max		Avg		Max	
Read	0.8 ms	±0.0	12.3 ms	±5.3	1.0 ms	±0.1	104.3 ms	±61.7
Write	3.6 ms	±0.1	17.0 ms	±5.2	3.7 ms	±0.2	66.8 ms	±60.2

caching frequently used queries has a huge performance impact in all applications we evaluated. How throughput and number of operations are related is pictured in Figure 1. This difference is directly visible if we look into the detailed latency numbers. For example, when running *Workload A* with 1,000 operations, the results look as shown in Table 2. The average latencies are similar, while the max latencies for Fontus are noticeably higher. This is caused by the query rewriting overhead, as for cached queries the runtime is very close.

2 PERFORMANCE EVALUATION: APPLICATIONS

For the different applications, our performance test consists of the following interactions:
OpenOlat: A user logs in and inspects two of the courses she is enrolled in.
Broadleaf: A user signs up, logs into the shop, purchases an item and logs off.
OpenMRS: A clerk registers a patient and logs out. Then a doctor admits the patient to a visit, registers a condition and then ends the visit.

REFERENCES

[1] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages & Applications.*
[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. of the ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications.*
[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the ACM Symposium on Cloud Computing.*
[4] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications.*