SPECjbb2012: Updated Metrics for a Business Benchmark

Aleksey Shipilev
Oracle Corporation
Saint-Petersburg, Russian Fed.
aleksey.shipilev@oracle.com

David Keenan
Oracle Corporation
Albany, NY, USA
david.keenan@oracle.com

ABSTRACT

SPEC [1] benchmarks have an excellent track record as useful tools for performance engineers. Many hardware vendors, software developers, and researchers continue to use SPEC benchmarks as reference workloads to characterize systems, test compiler optimizations, track performance regression tracking, and software quality.

SPECjbb2005 is the industry standard benchmark for evaluating the performance of servers running typical Java business applications. Modern customer requirements and use cases have shifted the focus of performance assessments from pure throughput to include throughput/response time and throughput/power considerations.

SPECjbb2012 is the new incarnation of SPECjbb2005, targeted to assess these new demands. This paper gives the highlights for new metrics in SPECjbb2012, the rationale behind them, and technical challenges faced in its implementation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Measurement, Performance

Keywords

SPECjbb2005, SPECjbb2012, response time, max injection rate, critical injection rate

1. DESIGN CONSIDERATIONS

The design goal for SPECjbb2012 was to keep the simplicity of SPECjbb2005 [2], while addressing more requirements and use cases for the benchmark.

The classic metric in a business benchmark is *raw throughput*, which tells a lot about system capacity at its peak. However, the emerging importance of power and response time requirements has pushed benchmark vendors to adopt new metrics.

The usual design for server-centric client/server benchmark is having one or multiple clients (*drivers*) to issue requests for one or multiple *Systems Under Test* (*SUT*) with some characteristic requested *injection rate (IR)*, and realistic think times delays and distributions.

The largest complication in this scheme is that clients do not have instant feedback on server utilization and capacity. Speficifically, queueing on the server side, communication latencies, delays in processing, etc. are causing delays in feedback to the client. Intelligent schemes for tuning IR, i.e. increasing IR when the server can process more, or decrease IR when the server can not, should take these considerations into account.

In the grand scheme of things, introducing an adaptive scheme into workload implies including a feedback loop, bringing the work into scope of control theory. In fact, we had experienced the predictions of control theory, when naive schemes for auto-tuning experienced semi-harmonic oscillations, or long transitional periods.

Because of that, most workloads are falling back to running at a stable, or *preset*, IR, and letting the user decide at which IR level to run. While SPECjbb2012 supports running at a preset IR to facilitate performance analysis and workload development, we considered getting maximum workload scores in this mode a tedious task for the user.

1.1 Settling Criteria

The key observation for automatic tuning of IR is that the Driver has quite a limited opportunity to infer the state of the SUT. The mere fact that the Driver's request was accepted does not imply it will get processed in a reasonable time, nor does the rejection of the Driver's request imply the SUT is unable to process more in the next time slot.

One of the naive feedback solutions to this problem is having a bounded acceptance queue on the SUT side, which will deny Driver requests from being accepted when the SUT is over-saturated. However, we observed the performance of the workload to be extremely sensitive to queue size across different JVMs, and different architectures, etc. Hence, we saw the need for a more vendor-neutral solution.

To address this, we instrumented the SUT to provide us with the actual *processing rate (PR)*, which tells us how many requests were actually processed. By comparing this with *requested IR (rIR)* we can deduce whether the SUT is capable of handling the IR we are injecting or not. Additionally, the Driver is instrumented to provide *actual IR* *(aIR)*, which tells how many actual requests were submitted to the system.

This enables us to change the IR dynamically, and observe whether the system had settled on new a IR by cross-comparing rIR, aIR, and PR. If those three match, then the system is running steadily on the requested IR. Discrepancies in either highlights a capacity problem with the Driver, or the SUT.

1.2 Saturating the SUT

One of the goals for SPECjbb2012 is to keep the Driver light enough so not to require large machines to feed the SUT with requests. The key observation is that once the Driver had to maintain the state for each outstanding request to the SUT, the overhead of maintaining this state grows with number of outstanding requests. Then, by Little's Law, the number of requests is growing as the product of throughput and response time.

From the implementation standpoint, this either calls for an asynchronous processing scheme with a small number of threads dealing with larger amount of clients, or a synchronous processing scheme with a huge number of threads, one per client. While asynchronous processing can solve resource problems, measuring response time requires precise timings, and waiting for someone to process asynchronous response will skew the response time measurement.

We solved this dilemma by clearly separating *probe* and *saturate* requests. Probe requests are synchronously waiting for a response, thus providing the means for measuring response time accurately. There's a bounded number of threads servicing probe requests. Saturate requests are submitted without waiting for a response; hence, the Driver threads are freed up once a saturate request is accepted, and can be recycled to submit another saturate request. Since there's no response time measurement, we are able to submit *batches* of saturate requests to further unload the Driver. There's a tradeoff between the accuracy of think time distributions maintained by Driver, and the impact of the Driver itself. Both probe and saturate requests share the same submission budget, which ensures that the Driver submission rate does not out-pace the requested IR. We have verified that this approach is able to inject several orders of magnitude more requests than the SUT had to process.

2. METRICS

2.1 Raw throughput metric: max IR

Given the mechanics above, we can gradually increase the IR within one run, and see if the system settles on it. There are some complications with warmup and steady state. The measurement is done in two phases: searching for a high-bound of maxIR (hbIR), and then searching for maxIR itself.

The schematics for hbIR searching is to grow IR exponentially, starting from some base IR:

$$IR(n) = IR_{base} + IR_{step} * \alpha^n, \alpha > 1, n \in N$$

Once we hit over-saturation, we set the base IR to the previous successful IR, and restart the search. Obviously, since base IR is always growing, and there exists the system limit on possible IR, this search converges. The mere fact we can't step upwards anymore means we had also completed warmup. Each step during this search takes a few seconds, hence hbIR is not measured in steady state.

The next phase is to measure maxIR during steady state. We have to try multiple IRs, each of those should be done in steady state, which will take some time to achieve. However, there's also the reasonable limit on run time. To fit these contradicting requirements, the workload slowly grows IR in a linear fashion, hoping for a smooth transition between steady states on consequent IRs:

$$IR(n) = IR_{hb} * \beta, \beta \in [0; 1]$$

In default mode, we are doing 1% steps with 30 sec measurement each. At some point the system will fail to settle, and the IR preceding that point will be counted as the actual maxIR. There are also retry mechanisms in place to tolerate inadvertent hiccups. We had found maxIR is within [75; 95]% of hbIR on most systems.

2.2 Response time metric: critical IR

We can reuse the data from maxIR measurement to infer response time metrics. Since probe requests are happening during search for maxIR, we are effectively gathering the response time samples on different levels of maxIR. In fact, we can build the *throughput – response time (TRT) curves*, which is the ultimate characteristic of the workload running on the system.

Point measures are still more useful as metrics, so we have to infer something from the TRT curve. In SPECjbb2012 we had settled on throughput-related metric named *critical IR*, which is defined as maximum IR, at which some service-level agreement for response time is still achieved.

By default, critical IR is measured at 100 ms response time target in the 99^{th} percentile. However, since all response time samples are saved in the logs for the run, there are ways to post-mortem compute critical IR for different response time targets and percentile levels.

Additional methods for conditioning response time samples, like bootstrapping, are possible to provide more robust approximations for critical IR at high percentile levels.

2.3 Power metric: W/ops

Once we determine a precise maxIR, we can go for power measurement. Best practices for power measurement mandate measurement on different IR levels, going from peak IR down to essentially zero IR (so called *"active-idle state"*) [3]. This requirement does not allow us to do power measurement during TRT curve, which goes in an upward direction. However, we can dedicate another phase of the workload specifically for power measurement.

3. IMPLEMENTATION CONSIDERATIONS

3.1 Re-initialization

The major issue with changing the IR during the run is the risk of over-saturation, when the system is not able to handle the IR we are injecting. In the best case, this will grow the occupancy of the submission queues in the SUT. In the worst case, this over-saturation might impede normal operation of the SUT, knocking it off of steady state, or even drive it to an inconsistent state.

To clean up after over-saturation, the workload can invoke *re-initialization*, which will shutdown all parts of the workload except for a minimal infrastructure, ask for aggressive GCs, and then initialize the workload again. This effectively resets the system to a pristine state.

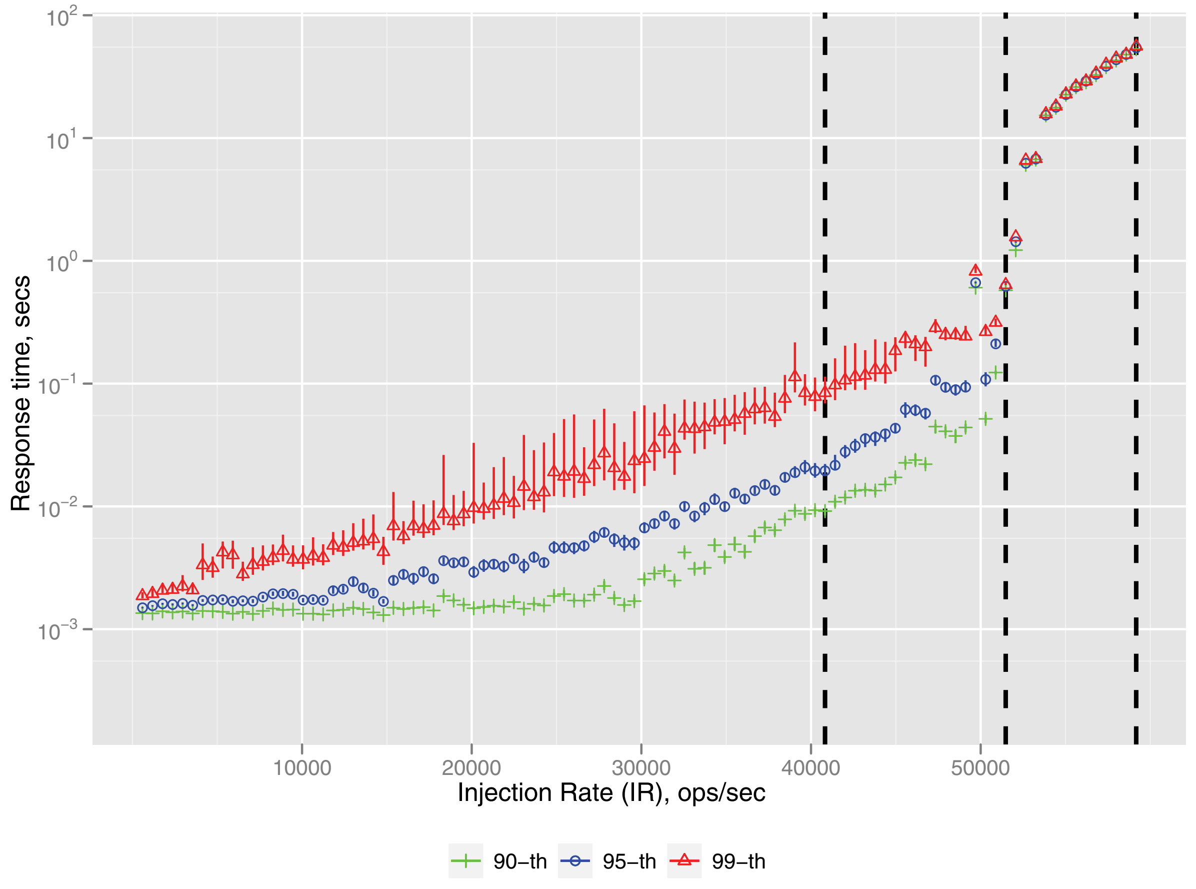

Figure 1: Typical throughput - response time curve

3.2 Snapshots

The problem with re-initialization is that it discards all the business data collected during the run, and starts over. Sometimes this is undesirable, because the data state is a part of the steady state. To account for that, the workload can *snapshot* its state at known good points during the workload lifetime, essentially serializing the state of all data structures to a binary blob, and storing it.

Given the concurrent nature of the workload, snapshots require quiescence; hence, acquiring a snapshot breaks the steady state. Combined with the cost of serializing, compressing, and writing out the snapshots, we can only afford that in several designated places in the workload.

Re-initialization uses the last available snapshot to restore the system state. Each subsequent snapshot is a better starting point should re-initialization be required.

3.3 Heartbeats

Sometimes over-saturation causes the system to go completely haywire. In these conditions, it may happen that remote communication is sometimes stalled, and the Controller's request to terminate the run might not be delivered to the JVM in question.

To address this, the *heartbeats (HB)* infrastructure is used. There are multiple HB watchdogs running in the JVMs, periodically polling each other. Once a HB watchdog fails to receive its pending heartbeats, it assumes the system has lost control, and terminates the activity in the current JVM. It also shuts itself down, so other HB watchdogs can detect the failure. After HB failure occurs, the system can only recover with re-initialization.

While there are other ways to control the system, e.g. let the Controller reinforce the intent to run at a specific IR every once in a while, we had found the HB infrastructure to be useful in other failure modes, e.g. when one of the JVMs participating in the run crashes or suddenly dies.

4. EXPERIMENTAL STUDY

Empirical evaluation of the proposed metrics scheme is the current focus of SPECjbb2012 development. In this section, we highlight the results of one. The experimental setup was as follows:

- Intel Xeon X7560 (Nehalem-EX), 4 sockets, 10 cores per socket, 2 threads per core, running at 2.27 GHz.
- 16x 4Gb PC3-8500F DIMMs, 64 Gb total
- Oracle JDK 7 Update 2, RedHat AS 5 (64-bit)
- SPECjbb2012 beta (EOY 2011), transports disabled
- Java params: -d64 -Xmx8g -Xms8g -XX:+UseNUMA -XX:+UseConcMarkSweepGC -XX:-UseBiasedLocking

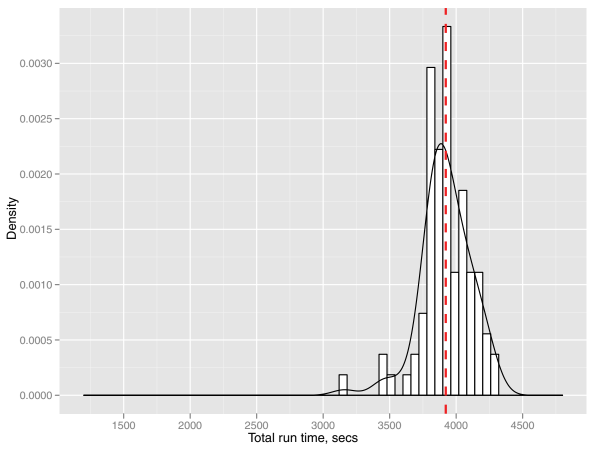

Figure 2: Total run time

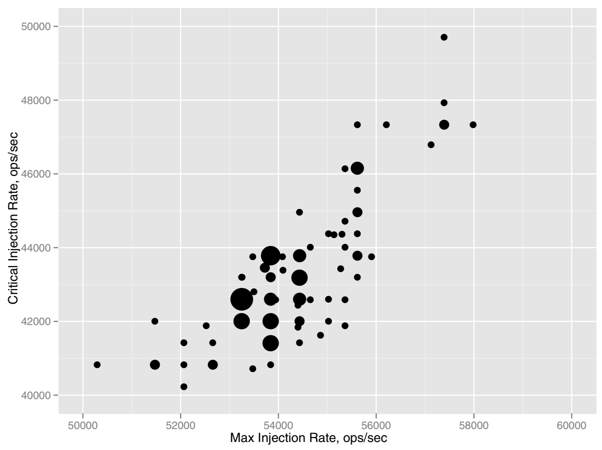

Figure 3: maxIR/criticalIR correlation

4.1 Typical TRT curve

Figure 1 shows typical TRT curve gathered during the run. Vertical dashed lines correspond to critical IR, max IR, and high-bound IR, respectively. The Y axis is log-scale, hence RT quantiles are growing at an exponential rate. Notice that the critical IR line intercepts the 99^{th} percentile curve at 100ms target. The curve itself is rather monotonic to enable measurements on lower RT targets. Additional robust approximation schemes are suggested to remove the high-quantile jitter.

The max IR line clearly demarcates the start of over-saturation, where response times are getting orders of magnitude higher. Over-saturation is detected by failing to settle on requested IR, but the artifacts on TRT curve are visible as well.

4.2 Run time

Obviously, there's tradeoff between workload run time and accuracy. Longer workloads can afford more precise measurements, while shorter workloads provide a better user experience. SPECjbb2012 has sensible targets with regards to run times, to not exceed 2 hours, while aggressively pushing into a 1 hour envelope. Figure 2 gives the impression of usual run times for SPECjbb2012. The mean run time is close to 4.000 seconds, at least 3.600 of which takes building TRT curve.

4.3 Metrics variance

To estimate variance, we had executed 100 complete runs on the target hardware. These measurements include both intrinsic workload variance, as well as general run-to-run variance due to indeterminism in JVM behavior [4].

The results for maxIR and criticalIR are packed into Figure 3. With the default settings, maxIR is quantized in 100 levels, with the highest level equaling hbIR. Since hbIR is also variating, the number of possible values for maxIR is much larger than 100. Critical IR experiences quantization as well for the same reason.

The variance of the workload is much larger than quanta size; hence, finer steps in TRT curve seem unnecessary. We are currently investigating the reasons behind the workload variance, and what can be reasonably done in the workload to shun unwarranted sources of indeterminism.

5. FUTURE WORK

At the present moment, SPECjbb2012 development has reached the stabilization phase. More assessments are in progress, variance is being investigated, and general workload behavior on wide range of platforms is being researched.

6. ACKNOWLEDGMENTS

The authors want to acknowledge the additional members of the SPEC JAVA Subcommittee who have contributed to the design, development, testing, and overall success of SPEC benchmarks. The name SPEC together with its tool and benchmark names are registered trademarks of the Standard Performance Evaluation Corporation (SPEC).

7. REFERENCES

[1] Standard Performance Evaluation Corporation http://www.spec.org/

[2] A. Adamson, D. Dagastine, S. Sarne. SPECjbb2005 - A Year in the Life of a Benchmark.

[3] Standard Performance Evaluation Corporation. SPEC Power and Performance. Benchmark Methodology, V2.1.

[4] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42:57–76, October 2007.