Standard Performance Evaluation Corporation (SPEC)

SPECjbb2013
Design Document

7001 Heritage Village Plaza, Suite 225
Gainesville, VA 20155,
USA

SPEC OSG JAVA Committee

Table of Contents

1. Introduction

SPECjbb2013 replaces SPECjbb2005 as the next generation Java server business benchmark. This document describes the design of SPECjbb2013.

For updates, please see: http://www.spec.org/jbb2013/docs/designdocument.pdf.

1.1. About SPEC

The Standard Performance Evaluation Corporation (SPEC) was formed by the industry in 1988 to establish industry standards for measuring compute performance. SPEC has since become the largest and most influential benchmark consortium world-wide. Its mission is to ensure that the marketplace has a fair and useful set of metrics to analyze the newest generation of IT equipment.

The SPEC community has developed more than 30 industry-standard benchmarks for system performance evaluation in a variety of application areas and provided thousands of benchmark licenses to companies, resource centers, and educational institutions globally. Organizations using these benchmarks have published more than 20,000 peer-reviewed performance reports on SPEC's website (http://www.spec.org/results.html).

SPEC has a long history of designing, developing, and releasing industry-standard computer system performance benchmarks in a range of industry segments, and peer-reviewing the results of benchmark runs. Performance benchmarking and the necessary work to develop and release new benchmarks can lead to disagreements among participants. Therefore, SPEC has developed an operating philosophy and range of normative behaviors that encourage cooperation and fairness amongst diverse and competitive organizations.

As Java Business Benchmark, SPECjbb2000 was the first benchmark in this series emulating a 3-tier workload inside a single JVM (Java Virtual Machine) instance. It was updated with SPECjbb2005 which featured more complex transactions as well as multiple JVM instances within single OS image exercising some JDK 5 functionalities. Due to its simplicity and accuracy at measuring software and hardware capabilities, SPECjbb2005 has been very successful and among the most published benchmarks. The latest update in this series is SPECjbb2013. SPECjbb2013 is addressing many recent trends including distributed deployments and environments such as the Cloud and Virtualized environments.. SPECjbb2013 has been architected from the ground up with no code borrowed from earlier SPECjbb versions. The benchmark measures throughput as well as response time with graduated load levels.

1.1.1. SPEC Membership

SPEC membership is open to any interested company or entity. OSG members and associates are entitled to licensed copies of all released OSG benchmarks and unlimited publication of results on SPEC's public website. An initiation fee and annual fees are due for members. Nonprofit organizations and educational institutions have a reduced annual fee structure. Further details on membership information can be found on http://www.spec.org/osg/joining.html or requested at info@spec.org. Also a current list of SPEC members can be found here: http://www.spec.org/spec/membership.html.

1.1.2. SPEC's General Development Guidelines

SPEC's philosophy and standards of participation are the basis for the development of SPECjbb2013. The benchmark is being developed cooperatively by a committee representing diverse and competitive companies. The following guides the committee in the development of a benchmark that will be useful and widely adopted by the industry:

- Decisions are reached by consensus. Motions require a qualified majority to carry.
- Decisions are based on reality. Experimental results carry more weight than opinions. Data and demonstration overrule assertion.
- Fair benchmarks allow competition among all industry participants in a transparent market.
- Tools and benchmarks should be architecture-neutral and portable.

- All who are willing to contribute may participate. Wide availability of results on the range of available solutions allows the end user to determine the appropriate IT equipment.

Similar guidelines have resulted in the success and wide use of SPEC benchmarks in the performance and power/performance industry and are essential to the success of SPECjbb2013.

2. Scope and Goals

The goal of the SPECjbb2013 benchmark is to evaluate the performance and scalability of environments for Java business applications. In addition, the benchmark provides a flexible framework for measuring system performance, including JRE performance, operating system performance and underlying hardware / system performance. The following goals were put in place to meet these requirements:

- Simulate a relevant application model.
- Exercise the components of the Java runtime and its OS and hardware environment.
- Include a response time metric because response time is a major factor in service level agreements.
- Scale to use all available processors and memory resources.

2.1. Sockets and Systems

The System Under Test (SUT) can be a single stand-alone server, or a multi-node set of servers (limited to a set of homogenous servers or blade servers) running single or multiple OS images. A multi-node SUT could consist of server nodes that cannot run independent of a shared infrastructure such as a backplane, power-supplies, fans or other elements. These shared infrastructure systems are commonly known as "blade servers" or "multi-node servers". Only identical servers are allowed in a multi-node SUT configuration. However, an SUT can consist of multiple stand-alone server systems if those systems are being marketed as single solution. For more details, please refer to the SPECjbb2013 Run and Reporting Rules.

2.2. Scaling

One of the drawbacks of SPECjbb2005 is the lack of interaction among JVM instances which causes the workload to exhibit linear scaling. In SPECjbb2013, transactions interact across JVM processes and this allows the workload to exhibit more realistic sub-linear scaling.

2.3. IO Component

SPECjbb2013 exercises the CPU, memory and network I/O, but not disk I/O.

2.4. Redundancy

Many servers have redundancy built in for power supplies and cooling fans. Some servers include different levels of redundancy for memory, disk, and even processors. SPECjbb2013 does not test redundant components.

2.5. Run Time

The average run time is 2 hours. Benchmark has the following phases:

- Search HBIR: ~15-20 minutes for typical system. It can take much longer for larger systems.
- RT curve building: ~90 minutes
- Validation: ~5 minutes
- Profile: ~2 minutes
- Report: ~2 minutes for 'level 0' while 30 minutes or more for 'level 3' with large Groups config

2.6. Platforms

SPECjbb2013 supports the following platform/OS/JVM combinations.

HW Platform	x86 (AMD)	x86_64 (AMD)	x86_64 (AMD)	x86_64 (Intel)	x86_64 (Intel)	x86_64 (Intel)	Itanium (Intel)	POWER (IBM)	POWER (IBM)	SPARC (Oracle)	SPARC (Fujitsu)
OS	Windows Server 2008 R2	LINUX	Solaris	Windows Server 2008 R2	LINUX	Solaris	HP-UX 11i	AIX	LINUX	Solaris	Solaris

Note: OS refers to versions (service packs and patch levels) that are current at the SPECjbb2013 release.

2.7. Implementation Languages

The benchmark code is written in Java using Java SE 7 APIs.

3. SPECjbb2013 Architecture

The SPECjbb2013 benchmark has been architected, designed and implemented from scratch to represent the latest Java application features. It also exercises the latest data formats (XML), communication using compression, messaging with security as well as the latest JDK 7 features. Overall, it features several improvements over SPECjbb2005, including but not limited to:

- A sustainable full system capacity throughput metric (max-jOPS) and a throughput under response time constraint metric (critical-jOPS).
- Multiple supported run configurations (Composite/single host, MultiJVMs/ single host, Distributed/single or multi hosts) that allow diverse users to analyze and overcome bottlenecks at multiple layers of the system stack (e.g., hardware, OS, JVM, application).
- The exercising of features that are new in Java 7 (e.g., the fork-Join framework)
- Support for virtualization and cloud environments that have well-defined hardware dedicated to the benchmark while the benchmark is running. This includes all private cloud environments and public cloud environments that meet the above requirement.

3.1. Workload modeling

The application scenario chosen for SPECjbb2013 is a world-wide supermarket company IT infrastructure being exercised by

- POS(Point Of Sales) in local Supermarkets(SM) as well as online purchases,
- Issuing and managing coupons/discounts and customer payments
- Managing receipts, invoices and user database in the company Headquarters (HQ)
- Interaction with Suppliers (SP) for replenishing the inventory
- Data mining(DM) operations in the company Headquarters to identify sale patterns and generating quarterly business reports

Customers purchase products from Supermarkets and Supermarkets replenish their inventories from Suppliers as needed. The Headquarters manages the information about the other entities and may mine this information to determine or influence "hot" purchases. Even though purchases are predominant, other Java APIs are also being exercised to achieve a balanced design.

The benchmark can be deployed in a way that it simulates customer data being spread across several Java processes, resulting in inter-process communication. This scenario offers a natural way to scale the workload and still have a compliant run. Refer to the User's Guide for more details.

3.2. Workload components

The benchmark consists of following three components:

3.2.1. Controller (Ctr):

The Controller directs the execution of the workload. There is always one controller.
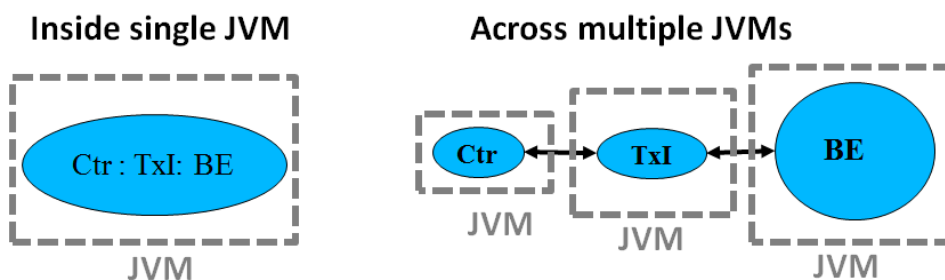
3.2.2. Transaction Injector(s) (TxI):

The TxI issues requests and services to Backend(s) as directed by the Controller and measures end-to-end response time for issued requests.

3.2.3. Backend(s) (BE):

The Backend contains business logic code that processes requests and services from the TxI, and notifies the TxI that a request has been processed.

4. Benchmark components configuration

The benchmark components (Controller, TxI(s) and Backend(s)) can be configured to run inside a single JVM instance or across multiple JVM instances.

There is always exactly one Controller component and at least one Backend. Each Backend has one or more dedicated Transaction Injector(s) mapped to it. This logical mapping is called a "*Group*". The topology for a *Group* can be defined using command line options in the run script.
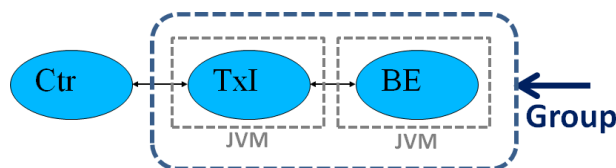
4.1. Single JVM instance using a single group

This is the simplest case of a deployment with the intended goal to encourage scaling inside a single JVM instance. In this scenario, a group consists of only one Backend and one Transaction Injector mapped to that Backend.
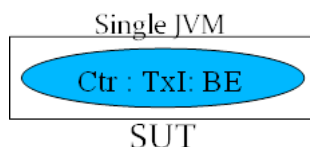
4.2. Multiple JVM instances using a Single Group

In this scenario, a group consists of one Backend and one or more Transaction Injectors (TxI) mapped to this Backend. However, all requests and transactions are confined to a single Backend.

Example: 1 Group with 1 TxI/Group
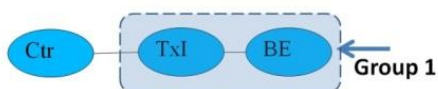
Example: 1 Group with 2 TxI/Group
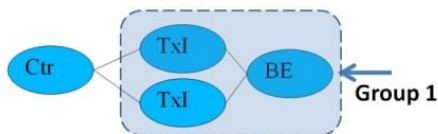
4.3. Multiple JVM instances using Multiple Groups

In a multiple group configuration each group has one Backend and one or more Transaction Injectors (TxI). By design, some percentage of requests and transactions involve inter-process communication between Backends.
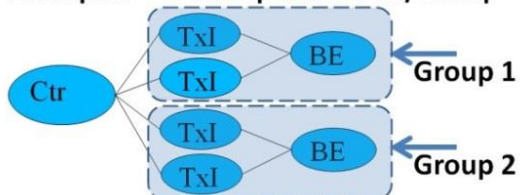
5.   Controller details

The Controller directs the execution of all the other benchmark components.

At the beginning of a run, the Controller waits for handshake messages from all the other benchmark components. Once these benchmark components are launched, they send handshake messages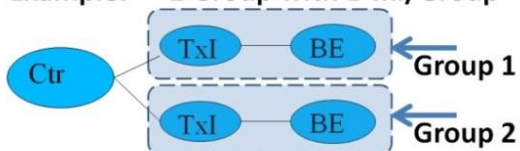 to the Controller. The Controller starts recording the progress of the run in the controller.log and controller.out files. The Controller also records detailed measurement information into a binary log file used to generate the reports later. Simultaneously, other agents start individual logs to record progress.

If the controller receives handshake messages from all the agents, it takes the benchmark through various phases. Otherwise, it shuts down the benchmark.

The Controller.log file stores performance information (updated every second) whereas the controller.out file stores summary progress information.

The Controller also operates a mechanism called *heartbeat*. Each agent sends heartbeats at a configurable frequency to the Controller. If the timing threshold for receiving heartbeats is exceeded, the Controller shuts down the benchmark.

By default the Controller will launch the Reporter at the end of a run. The Reporter produces result files in a result directory and moves log files to an appropriate directory. The Reporter may also be invoked manually as described in the User Guide.

The Controller by itself can run using a 512MB heap. The reporter requires 2GB heap for most run configurations and may require larger heap for larger systems.

6. Transaction Injector details

Transaction Injectors (TxI) are the load generators and trackers.

The Controller passes configuration parameters to each TxI after successful handshakes. The Controller divides the total aggregated injection rate uniformly across all TxI. The heterogeneous loading of Transaction Injectors is not supported and was not a design goal.

The TxI issues three types of requests: *probe requests*, *saturate requests*, and *service requests*. All these requests drive the load, but the probe requests also collect profiling data. Probe requests are issued individually and saturate requests are issued as variable sized batches. Service requests are issued at a fixed time interval to simulate monthly installment payments and daily shipping of orders, for example.

The TxI has a distinct thread pool for each type of request. In the case of saturate requests and service requests, a thread issues the request and doesn't wait for the response. In the case of a probe request, a thread waits for the response so that it can record the response time.

Once a TxI receives the load to be issued, it issues the load to the mapped Backend and records response time using probes. The TxI compiles an overall summary of response time information for each iteration and sends this summary to the Controller. The Controller aggregates the summaries from all TxI(s) to produce an overall summary.

7. Backend details

The Backend is the component that executes business logic and has roles such as processing requests coming from TxI(s), storing data, and performing background tasks.

7.1. Backend entities

In each Backend, there are three main entities *Supermarkets*(SM), *Suppliers*(SP), and *Headquarters*(HQ). There is exactly one HQ in a Backend. The number of SM(s) and SP(s) are configurable though only two SMs and two SPs are allowed for a compliant run. All these entities communicate using the Interconnect described later in this document.
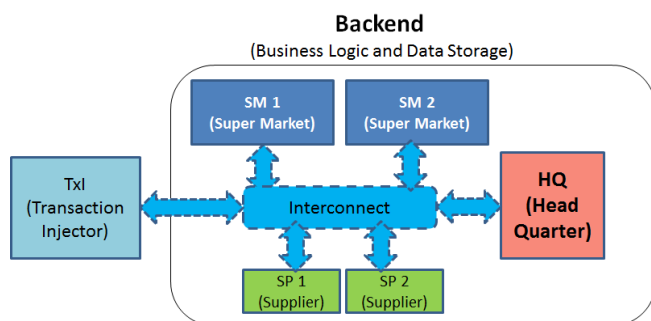
7.1.1. Supermarket (SM)

For a compliant run, there are two SMs per Backend. The SM maintains inventory as well as some locally cached customer information. A Customer purchases groceries from a Supermarket; this purchase transaction is exercised heavily during a benchmark run. The Supermarket will complete the transaction and send the receipt to the HQ. Over time, this will result in depleting the Supermarket's shelves. This triggers the Supermarket to generate orders to the Supplier to replenish those shelves.
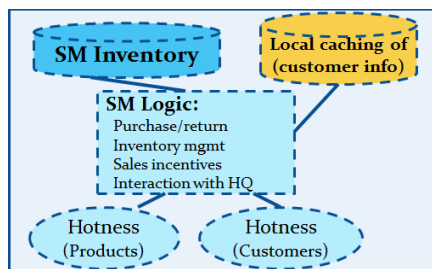

7.1.2. Suppliers (SP)

The Suppliers provide Supermarkets with goods to re-stock their shelves. Every Supplier is responsible for a set of Supermarkets in a given area. The Suppliers are involved in taking orders from Supermarkets and delivering goods to Supermarkets. If a Supermarket replenishes its inventory using a Supplier from a remote Backend, it is called a remote replenish transaction. Each Supplier has an endless supply of products.
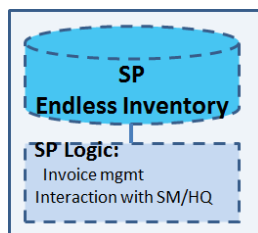
7.1.3. Headquarters (HQ)

There is one HQ per Backend. Each HQ manages the Supermarkets, Suppliers, and Customers for its associated Backend. This role involves managing information about the goods and products offered in the Supermarket(s), managing selling prices and monitoring the flow of goods and money between Suppliers and Supermarkets. The HQ also maintains customer information such as financial records, receipts sent by SM as well as invoices. The HQ also maintains a mirror inventory in case any SM goes down. The HQ maintains receipts, which are used by the "Data Mining" transactions which analyze the buying and selling behavior observed in the Supermarkets. Overall, HQ functionality is the second most heavily exercised transaction after SM.
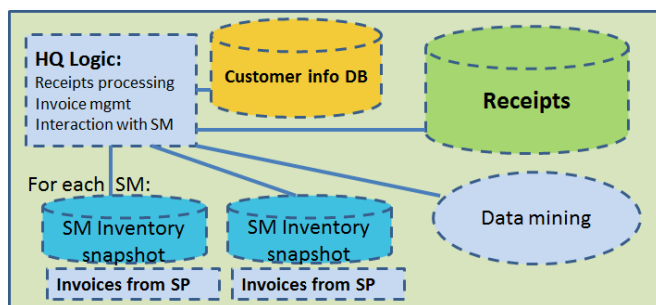
7.2. Backend thread pools

All entities within a Backend share a common thread pool to enable load balancing across the entities. Multiple thread pools are required to make progress and avoid circular dead locks.
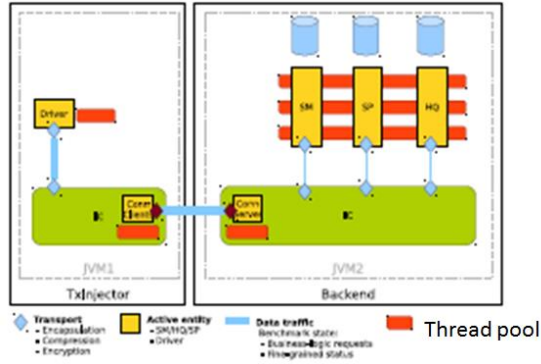
7.3. Backend thread executors

Requests to the Backend may be processed in batches. Batch processing has been implemented using the Java SE 7 Fork/Join framework which exploits parallelism on the modern multi-core processors. The implementation also serves as a best-practices example of the Fork/Join framework.
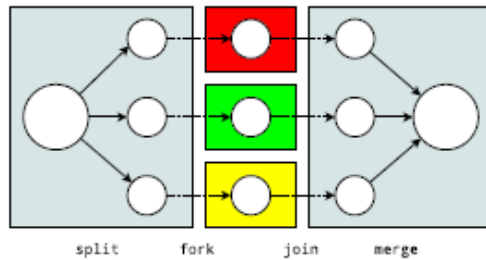
8. Traffic flow

There are two types of traffic: *control traffic* and *business data traffic*. The Controller directs the benchmark phases using *control traffic* with agents inside the other components. The Transaction Injector initiates the *business data traffic* to drive each Backend.
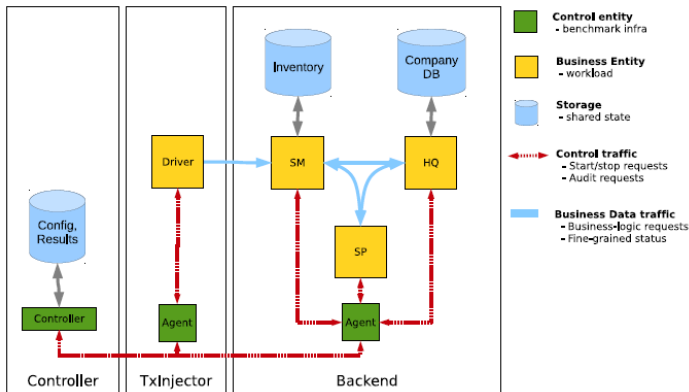
8.1. Communication

SPECjbb2013 exercises two types of communication, *intra-JVM communication* and *inter-JVM communication*. The entities inside a Backend exhibit intra-JVM communication, but sometimes these entities also communicate with the entities inside remote Backend(s).

The communication mechanism consists of the following components:

8.1.1. Interconnect

The Interconnect (IC) is the central communication fabric. Each JVM instance has its own IC. The IC provides inter-JVM communication, including the name space for all Backend entities and automatic routing.

8.1.2. Transport

The Transport mechanism provides business data marshalling among the entities of a Backend. In SPECjbb2013 the configuration of this marshaling mechanism for different transaction types cannot be changed for compliant runs.
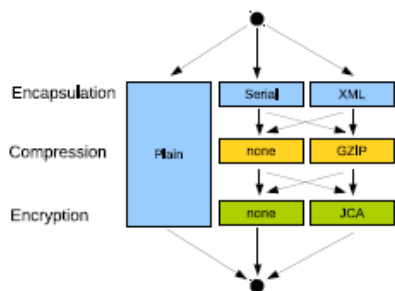
8.1.3. Connectivity

The SPECjbb2013 connectivity mechanism uses the client-server protocol. The mechanism is configurable to use either of the Grizzly/NIO, Grizzly/HTTP or Jetty/HTTP protocols. For compliant runs, the mechanism must be configured to use Grizzly/NIO. The Grizzly/NIO protocol is tunable via properties described in the sections 16.1.2 and 16.1.3 of the User's Guide.

8.2. Single Group Communication

A single group consists of one Backend and one or more Transaction Injectors (TxI) mapped to this Backend. All TxI(s) and Backend share a common IC for the communication layer. In a single group, all requests and transactions are confined to a single Backend.
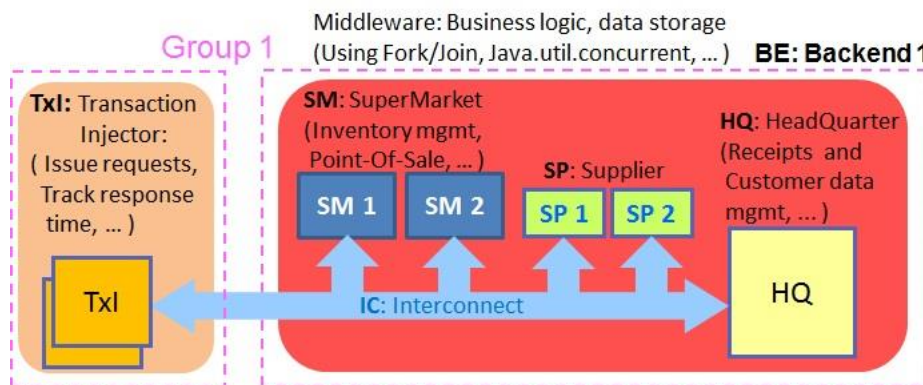
8.3. Multi-Group Communication

In a benchmark run involving multiple groups, a percentage of requests and transactions involve inter-process communication between different Backends. As the number of groups increases, this remote traffic also increases.

Remote traffic among Backend(s) consists of messages and requests. Messages are sent without the sender thread blocking for the response, whereas requests are sent with the sender thread blocking to receive a response. To avoid a circular deadlock, the thread pools inside the Backend are tiered.

There are two types of remote transactions: remote product replenishes and remote customer information queries. The remote traffic percentage for a replenish remains fixed whereas the percentage for a customer information query increases based on the formula, Round (Ln (n^3), 0), where n is number of groups.
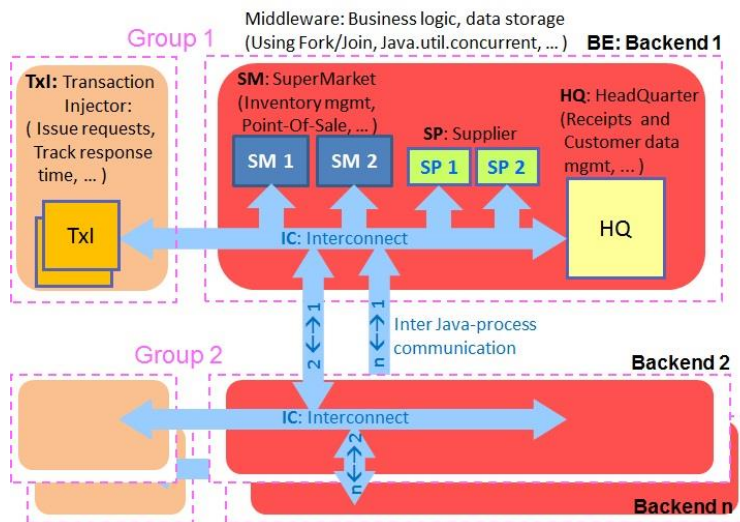
9.   Workload Requests and Transactions

The Transaction Injector issues two types of requests:

- *Requests issued at a negative exponential rate*: The requests in this category include probe requests and saturate requests
- *Requests issued at a fixed interval*: These requests simulate daily or monthly repetitive activities such as monthly installment payments, and the weekly issuing of coupons and advertisements.

The transactions generated by the above requests are of two types: *primary transactions* and *secondary* aka *background* transactions. Primary transactions are initiated as a result of request(s) issued by the Transaction Injector. Secondary (background) transactions are initiated when a threshold is crossed. An example of a secondary transaction would be the *product replenish transaction*, which is initiated when a Supermarket's inventory falls below a threshold.

9.1. Requests issued at a negative exponential rate:

These requests fall into two categories: customer-initiated requests and data-mining requests. As noted below, there are different types of customer-initiated and data-mining requests, and the benchmark maintains the mix ratio, or percentage of each of these request types.

Customer-initiated requests include:

- In Store Purchase – a customer purchases items from a supermarket in-store.
- Online Purchase – a customer purchases goods online from one or more supermarkets
- Installment Purchase – A customer pays for an item in installments. Examples would be a car purchase or a high-end electronics purchase.
- Product return – A customer returns an item and receives a refund.

Data-mining requests include:

- Associativity of Category – Identify the categories of products that are commonly purchased together

- Associativity of Products – Identify the products that are commonly purchased together
- Business Report – Simulate quarterly business summary
- Customer Buying Behavior – Analyze the customer buying behavior

The TxI maintains the following ratios that specify what percentage of the total requests are made up of requests of a specific type. These ratios cannot be changed for a compliant run but users have the ability to alter them for research purposes.

- In Store Purchase – 50%
- Online Purchase –35%
- Installment Purchase – 10%
- Product Return – 2.65%
- Associativity Of Category – 0.1%
- Associativity Of Product – 1%
- Business Report - 0.25%
- Customer Buying Behavior – 1%

Below is a more detailed description of the execution flow for each of the above requests.

9.1.1. In-store purchase:
1) The Transaction Injector sends a Supermarket an InstorePurchaseRequest.
2) The Supermarket receives the request and initiates the execution of an InStorePurchaseTransaction, This transaction executes the below steps.
3) Select a random customer who will execute the store purchase.
4) Retrieve a collection of products to purchase based on a "hotness" algorithm that considers what products the customer has frequently purchased in the past.
5) Reserve a specific quantity of each product and calculate the total price taking into account available discounts and coupons. Then add all the reserved products to the customer's shopping cart. In the process, it issue advertisements for each product purchased.
6) If most of the products are available, proceed to checkout. Otherwise, if too many products need to be replenished throw an exception.
7) Generate a receipt with the total price of items purchased.
8) Check that the customer has sufficient credit to purchase the items.
9) If the customer has sufficient credit, then move the desired quantities of each item out of the store inventory and debit the cost of each item from the customer's credit. If the store runs out of any item in the process, send the Suppliers a request to replenish that item.
10) Send a receipt back to the Headquarters indicating that the purchase has completed.

9.1.2. Online Purchase
The steps are similar to those of an InStorePurchase. However, each online purchase has a target supermarket for reserving the goods. If the target supermarket does not have enough goods to service the purchase, the transaction will send requests to one or more supermarkets on an "Alternate Supermarket List" to reserve the remaining goods. Also during the checkout phase, the goods will either be picked up in store or shipped to the customer.

9.1.3. Installment Purchase
The steps are similar to those of an InStorePurchase. However, the payment is in installments. In addition, the receipt is placed in an *installment entity* which debits a customer's bank account every time a payment is due until the full payment has been made.

9.1.4. Product Return
1) The transaction injector sends a supermarket a ProductReturnRequest.

2) The supermarket receives the request and initiates the execution of an ProductReturnTransaction.
3) The Supermarket sends its associated Headquarters a request to pick a random receipt containing products to be returned.
4) The Headquarters responds with a receipt containing a list of products, their quantities, prices and the supermarket from which each item was purchased.
5) The Supermarket scans each item on the receipt. If the item was purchased locally, it moves the specified quantities of that item into its own inventory. Otherwise, it messages the owning supermarket to restock its shelves with that item.
6) The Supermarket messages the Headquarters to refund the customer. This involves crediting the customer's account with the total prices of the returned items.

9.1.5. Associativity of Category (AOC)

Each product belongs to one or more categories. The AOC transaction randomly picks a product category and identifies the purchase receipts containing at least one product purchase from that category. It then compiles a list of the barcodes appearing on each of these receipts which are local to that receipt's SM. Based on this list it identifies the product categories for other frequently appearing barcodes. This information is used when issuing advertisements.

9.1.6. Associativity of Product (AOP)

The AOP transaction randomly picks a receipt and a barcode appearing on that receipt. It then compiles a list of the barcodes appearing on any receipt that also contains the specified barcode. Based on this list, it identifies other frequently appearing barcodes. This information is used to issue advertisements.

9.1.7. Business Report (BR)

The BR transaction analyzes the collected receipts and invoices. It summarizes the number of receipts, total paid money, total quantity of items, number of used coupons, quantity of items per barcode, number of unique items in receipts, number of times a coupon was used and the money saved by a customer from coupon usage. For invoices it also summarizes the quantity of items replenished per barcode.

After collecting the data above, the BR transaction generates a Business Report for the HQ containing the following information: total unique customers, average spending per customer, average visits per customer, worst selling products, top replenished products, total replenished products, and total money paid in all SMs. The resulting HQ Business Report is sent as a response to the BR request.

9.1.8. Customer Buying Behavior

The CBB transaction picks a customer from a randomly chosen receipt. It then lists the barcodes for products which this customer has purchased. From this list it identifies other frequently appearing barcodes. This information is used to drive the issuing of advertisements.

9.2. Service requests being issued at a fixed interval:

Service requests are issued at a fixed interval to simulate daily, weekly or quarterly activities. These requests represent background tasks resulting from the negative exponential rate requests described in section 9.1 . For example, an online purchase may result in an online shipping order. As a result, a periodic service request will be generated to process each online shipping order.

There are the following types of service requests:

9.2.1. Discount issue/remove

The Discount issue and remove transactions are used by the SM to adjust the hotness of products and impact purchasing decisions as well as the price of a product. These requests are usually sent in pairs, with a discount being issued and later removed.

9.2.2. Advertisement issue/remove

The Advertisement issue/remove request manages advertisements for a specific SM. If a customer chooses a product which has an associated advertisement, then this advertisement is copied to the customer's profile and influences the customer's future purchasing decisions.

9.2.3. Process invoice

This request triggers the HQ to process invoices and pay its Suppliers.

9.2.4. Installment purchase payment

The Installment Purchase request triggers the HQ to process an installment payment for an installment purchase in flight.

9.2.5. Installment purchase payment void

This request triggers the HQ to process the return of a product that was purchased in installments. This involves refunding the customer for all installment payments made.

9.2.6. Online In Store Pick Up

This request results in online orders being picked up from an SM.

9.2.7. Online Shipping Order

This request results in online orders being shipped.

10. Workload inter-JVM process interaction

Many applications deployments leverage inter-JVM process communication. To encourage optimizations in this space, this workload has two types of transactions that result in remote communication between JVM processes.

10.1. Remote customer

All purchase requests require a customer association to execute a purchase. For most purchases, a local customer associated to the local HQ is used. In a certain percentage of purchase requests, the customer information is requested from a remote HQ within a remote Backend. The remote customer percentage increases with the number of groups, according to the following formula: *(round(ln (n^3), 0))* , where *n* is number of groups. This results in several interactions between the JVM processes to acquire customer information for the customer ID, product checkout and purchase receipt.

10.2. Remote replenish

A replenish transaction is a secondary or background transaction initiated by an SM when the inventory for an item falls below a threshold. Most replenish transactions are processed by local Suppliers, but a specific percentage of them are processed by suppliers in a remote Backend. This results in several interactions with other JVM processes to acquire information about inventories, purchase confirmations, and invoices.

11. Workload Execution Phases

Once all the benchmark components are launched, the TxI(s) and Backend(s) send handshake messages to the Controller. The Controller starts recording progress information in the controller.log and the controller.out files and detailed measurement information in the binary log file in the current benchmark directory. Other agents start their own logs to record progress. After a successful handshake, the controller directs the execution of the benchmark through the following stages:
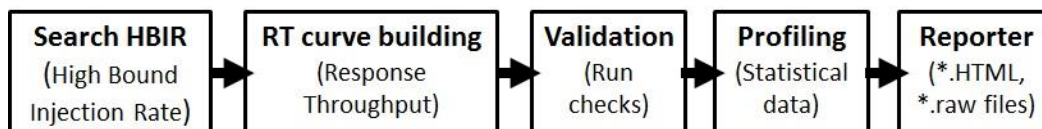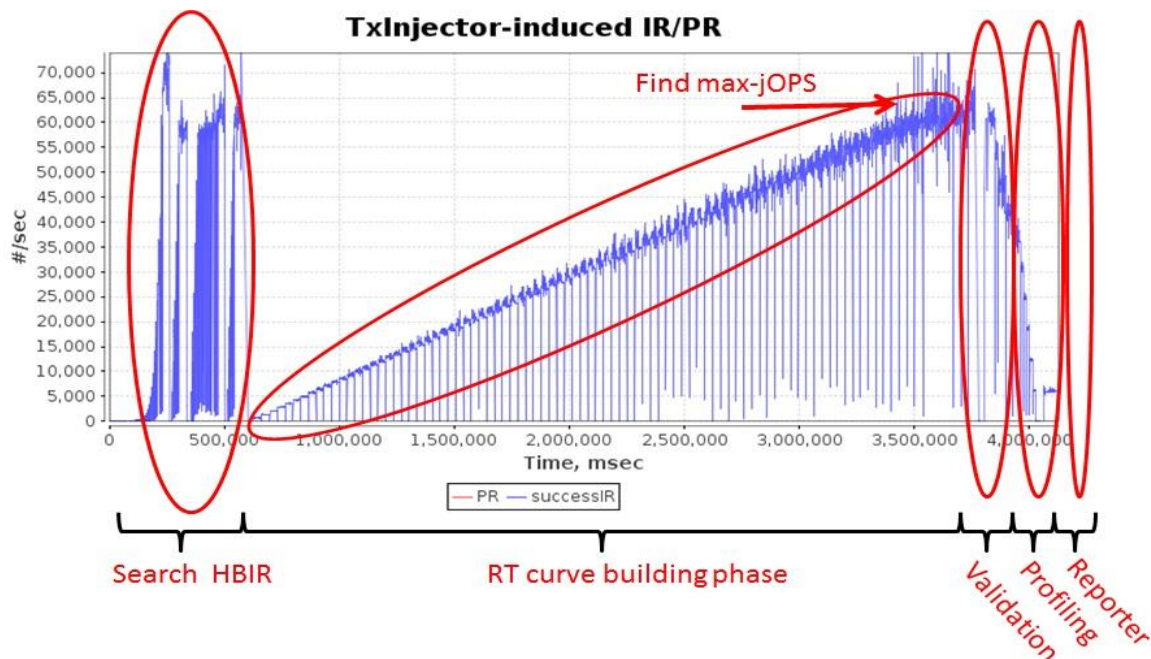
- The *High-Bound IR (HBIR Search)* is used to determine an upper bound on the system capacity for throughput, the maximum load a system can handle without response time constraints.
- The *Response-Throughput curve (RT-curve) building* phase iterates over successively increasing IR levels to determine the final metrics of the benchmark, namely the *max-jOPS* which is a pure throughput metric, and the *critical-jOPS* which is a throughput under response time constraint metric
- The *validation* phase checks that data structures were in their correct state during the run.
- The *profiling* phase gathers additional statistics that will be displayed in the advanced report.
- The Reporter is launched automatically by the Controller at the end of the run to generate the default 'level 0' report

11.1. The High-Bound IR (HBIR Search)

The HBIR phase estimates the maximum Injection Rate (HBIR) the system can handle. In the later phase known as *RT curve building*, the RT step levels are incremented by 1% of this computed HBIR. For testing and research, the HBIR value can be manually set using a property but for compliant runs, it must be determined automatically by this phase.

In more detail, the Controller increases the injection rate in fixed increments and runs for a fixed duration at each injection rate level to determine whether the system is able to successfully execute at that level. If the system successfully executes at that level, the controller increases the injection rate. Otherwise, it backtracks to the previously successful injection rate (after draining all the queues and ramping up again from 0 to that level) and ramps up from there in smaller increments.

At the end of this process, the High Bound IR (HBIR) is identified and used during the RT-curve building phase.

11.2. The Response-Throughput curve (RT-curve)

Before this phase, all the data structures are re-initialized. The benchmark then enters a fixed-length warm-up period where the injection rate is by default 10% of the High Bound IR computed during the previous phase. At the conclusion of the warm-up period the RT-curve phase begins. The Controller increases the injection rate from 0 (in

steps that are 1% of the HBIR) until it reaches an injection rate level where the benchmark is no longer able to execute successfully.

At the conclusion of this phase, the benchmark calculates the two primary metrics of *max-jOPS* and *critical-jOPS*.

Note that the Controller continues to drive the system beyond the max-iOPS to demonstrate that the system has indeed reached full capacity. The result report contains a detailed graph showing Response-Throughput details.

11.3. The Validation phase

Upon the completion of the RT curve building phase, the benchmark data structures are re-initialized because system was tested beyond maximum capacity and it is possible that system queues may have grown to be too large. After a successful re-initialization, the validation phase is initiated to validate the business logic. During this phase, a given IR is executed and then validation checks are performed against data structures corresponding to the business logic to ensure a correct state and accurate execution.

11.4. The profiling phase

The profiling phase is used to collect an intrusive profile of the benchmark run. The gathered information can be found in the advanced report which the user can generate by invoking the reporter manually or by enabling the option *–l <report_level 0/1/2/3>* at the Controller launch command line. For the manual invocation of reporter, refer to section 11 in SPECjbb2013 User Guide.

11.5. Reporter

At the end of the benchmark run, the Controller automatically invokes the reporter unless the command line option "-skipReport" is enabled.

Note: Until the reporter is invoked, no result directory is created and all logs including the binary log are in the current directory.

When the reporter is invoked, it takes two input files, the binary log file and the template-[C/M/D].raw file. The reporter creates a directory in SPECjbb2013/result/<binary log name dir>/report-<NUM>/ and generates all the output files for a default 'level 0' HTML report as well as the *.raw files into this directory. (The "-t <result_dir>" option may be used to override this default directory setting.) The user can override the default reporting level using the option "-l <report level 0/1/2/3>" is used. For a compliant run, the report level must be 0.

12. Workload RT curve building details

The RT curve building phase is the phase that determines both the benchmark metrics max-jOPS and critical-jOPS. It consists of three sub-phases: warm-up, RT curve building till max-jOPS, and stressing the system beyond max-jOPS. These sub-phases are described below:

12.1. Warm-up phase:

Once HBIR is determined, the benchmark re-initializes all the data structures at the beginning of the RT curve phase. It then executes a warm-up phase to exercise these re-initialized data strictures. This warm-up phase runs for 180 seconds at 10% of the HBIR. This default can be set up to 90 % of the HBIR for a compliant run. There is no passing criterion applied to the warm-up phase.

12.2. RT step levels

After the warm-up phase, the RT curve building phase begins. The benchmark increments the injection rate at steps that are 1% of the previously determined HBIR and determines whether or not the benchmark successfully executes at each injection rate level. This process continues until the benchmark stresses the full system capacity, at which point the metric max-jOPS is determined.

In more detail, each RT step level has a settle period followed by a steady period. When transitioning from the (n-1)th RT step level to n th step level, the IR is increased from (n-1)% to n% of the HBIR and the starting time for nth RT step level is recorded.
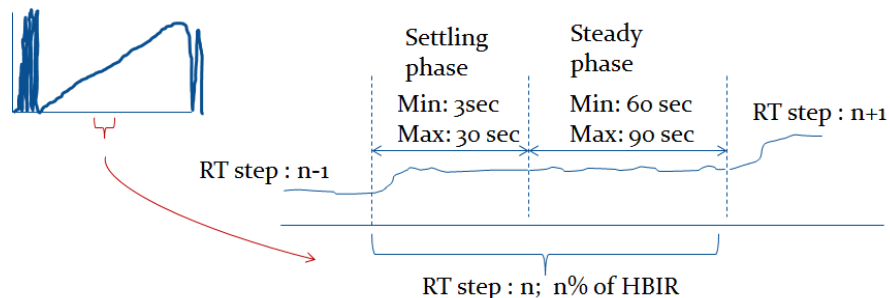
The criteria used to decide whether the benchmark is able to successfully execute at a step level is the following:

The Actual Injection Rate (aIR) and the rate at which the backend processes requests (PR) must be within ±3% of the target injection rate IR

For the settle period, if a system meets the above criteria any time after 3 seconds from the start of the RT step level, the benchmark passes execution at the *settle phase*, and we move to the *steady phase*. If system is unable to pass the above criteria to a maximum of 30 seconds, system fails to settle. A retry criterion is applied as described later.

The steady phase has a minimum duration of 60 seconds. After this duration has elapsed, the pass/fail criteria described above are applied every second. If the system passes the criteria before its 90 seconds into the steady phase, the current RT step level is declared successful and the Controller initiates the transition to the (n+1)th RT step level. If the passing critera fails, a retry criterion is applied as described below.

12.2.1. Retry criterion:

For an RT step, if either of the settle or steady phases fails to meet the passing criterion, a retry is allowed. The rule is that an RT step level can have exactly 1 retry and that there can be a maximum of 5 total retries over the entire RT curve building phase.

During a retry attempt, the settle period can be up to 180 seconds while the steady period has the same 60 sec minimum and 90 sec maximum time window. If an RT step level is able to pass after a retry, the number of retries left is decremented by one and the Controller takes the benchmark to the next RT step level. However, if the RT step level is not able to pass even after the retry, that RT step level is declared a failure. If there are no retries left, an RT step level will be declared a failure if it does not pass at the first attempt.

The number of retries per RT step level and the maximum number of retries for the whole run are user-configurable, but for compliant runs, the default values are one retry per RT step level and a total of 5 retries for whole RT curve building phase.

12.2.2. First failure and max-jOPS

Once the Controller determines that an RT step level has failed exhausting its retries, the current RT step level is called "First Failure". The IR at the previous RT step level is selected as the benchmark metric *max-jOPS.*

The Controller continues to drive the system a few RT step levels beyond the max-iOPS to demonstrate that the system has indeed reached full capacity. The Controller stops theRT phase once three RT step levels fail in a row. The number of consecutive failures of the RT step levels (impacting where the RT curve building phase stops) can be configured as described in section 2.5 of Run and Reporting Rules.

13. Requests issue

There are three types of requests probes, saturate and service. Only probes and saturate requests are counted towards the injection rate (IR). First, the Controller equally divides the total IR to be issued among the TxI(s). Then each TxI tries to issue its quota of the IR using probe requests during a quantum issue window. If the TxI can issue all of its IR quota using probe requests, no saturate requests are issued. In other words, saturate requests are issued only for the part of the IR budget left over from probe requests. As a result, at a lower IR, most of the IR is issued using probes while at a higher IR, a larger share will be issued using saturate requests.

Probe requests are issued by threads which wait for the response. The default number of issuing threads is 64. As a result, as the response time increases, fewer probe requests can be issued. The total number of probe requests must be a minimum percentage of the total IR. Otherwise, the benchmark issues a warning for the user to increase the number of thread pool workers or modify the TxI/Group configuration parameter as described in section 2.5 of Run and Reporting Rules.

Saturate requests are issued in variable sized batches using non-blocking threads. As a result, the TxI is always able to fully load a system. The batch size can be configured for research purposes, but for a compliant run the default setting must be used.

A user can view the distributions of probe requests and saturate requests in the 'level 0' HTML report graph.

Service requests are issued at a fixed time interval and are independent of probe and saturate requests.

14. Response time measurement to determine critical-jOPS

Response times are computed as the time from when a request is issued to the time when a response is received indicating that the request was successfully processed. Response time is measured using probe requests and the accuracy of this measurement depends on having sufficiently many probe requests issued at each IR level.

For each RT step level, transaction response times are measured and recorded in the binary log file of the run. During the report generation, the Reporter analyzes the response time data and computes percentile response times for each RT step level. The Reporter plots the percentile response time data as a function of the RT step level in the "Overall Throughput RT curve" graph of the 'level 0' HTML report.

15. Metrics Computation

The benchmark has two metrics, a pure throughput metric called *max-jOPS* and a throughput under response time constraint metric called *critical-jOPS*. In the reporter output files, these metrics will be displayed in the format:

SPECjbb2013:*run_category_name* max-jOPS   and
SPECjbb2013:*run_category_name* critical-jOPS

where *run_category_*name is Composite, Multi-JVM or Distributed.

These two metrics are computed during the RT curve building phase discussed in section 11.2.

The max-jOPS is the last successful injection rate before the first failing injection rate where the reattempt also fails. For example, if during the RT-curve phase the injection rate of 80000 passes, but the next injection rate of 90000 fails on two successive attempts, then the max-jOPS would be 80000.

The critical-jOPS computation is more complex. The critical-jOPS is computed based on 5 SLA (Service Level Agreement) points for the response time from the issue of request to receipt of a response that the request was completed. These points are: 10ms, 50ms, 100ms, 200ms and 500ms. These values were chosen to maintain a reasonable spread of response time targets used by different industries (without favoring a specific target), and be "challenging enough" that system designers will be motivated to improve the key areas (e.g., hardware, OS, JVM) that impact Java response times.

The overall critical-jOPS is computed by taking the geomean of the individual critical-jOPS computed at these five SLA points, namely:

- Critical-jOPS$_{overall}$ = Geo-mean of (*critical-jOPS@ 10ms, 50ms, 100ms, 200ms and 500ms response time SLAs*)

During the RT curve building phase the Transaction Injector measures the 99th percentile response times at each step level for all the requests (see section 9) that are considered in the metrics computations. It then computes the Critical-jOPS for each of the above five SLA points using the following formula:

```
(first * nOver + last * nUnder) / (nOver + nUnder)
```
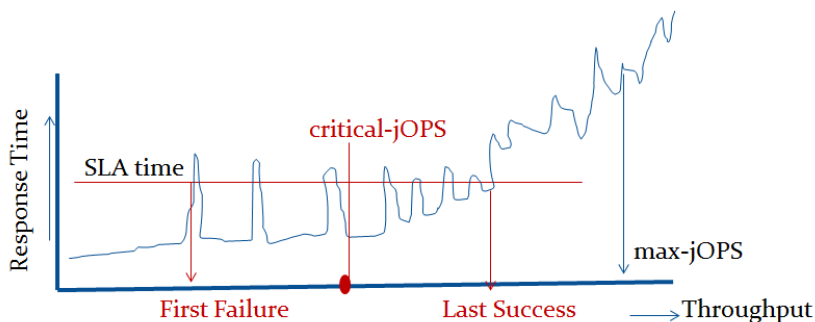
Where:

'*first*' – the first IR of RT step level with p99 response time higher than SLA

'*last*' – the last IR of RT step level with p99 response time less than SLA

nOver – the number of RT step levels between '*first*' to '*last*' where p99 response time > SLA

nUnder – the number of RT step levels between '*first*' to '*last*' where p99 response time < or = SLA.

The benchmark results files contain information about individual critical-jOPS as well as the overall critical-jOPS.

16. Future Enhancements / Stretch goals

The benchmark is highly configurable for research purposes. For details, refer to the *SPECjbb2013 Advanced Options and Research* section on the benchmark website.

17.Trademark

SPEC and the names SPECjbb2013 are registered trademarks of the Standard Performance Evaluation Corporation. Additional product and service names mentioned herein may be the trademarks of their respective owners.

18.Copyright Notice

Copyright © 2007-2014 Standard Performance Evaluation Corporation (SPEC). All rights reserved.

19.Appendix A

20.Index