Standard Performance Evaluation Corporation (SPEC)

# SPECjbb2013
# User Guide

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 guide explains how to setup and run SPECjbb2013.

To check for possible updates to the Run and Reporting Rules, please see http://www.spec.org/jbb2013/docs/UserGuide.pdf.

### 1.1. The SPECjbb2013 suite

The SPECjbb2013 benchmark does not require any third party software, with the exception of a JRE (Java Runtime Environment) version JDK 7 Update 2 or higher and a properly configured operating environment.

### 1.2. Benchmark components:

The benchmark consists of following three components:

#### 1.2.1. Controller (Ctr):

Controller directs the execution of the workload. There is always one controller.

#### 1.2.2. Transaction Injector(s) (TxI):

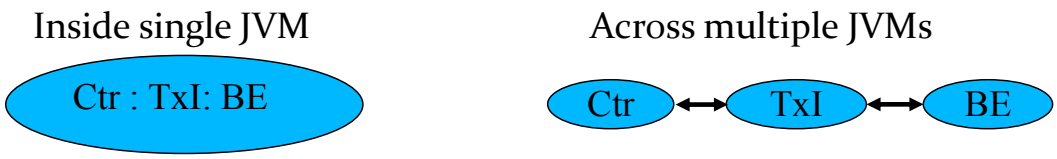
The TxI issues requests and services to Backend(s) as directed by the Controller and measures end-to-end response time for issued requests.

#### 1.2.3. Backend(s) (BE):

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

### 1.3. Benchmark components configuration and deployment:

Benchmark components Controller, TxI(s) and Backend(s) can be configured to run inside a single JVM instance or across multiple JVM instances with each component using its own JVM instance deployed to run across single or multiple hosts.




SPECjbb2013 components can be deployed across multiple-JVM configurations, in which there is always one Controller component and at least one Backend, with each Backend having one or more dedicated Transaction Injector(s). This logical mapping of Backend to TxI(s) is called a "*Group*". A *Group* can have only one Backend, but can have one or more Transaction Injectors if one TxI is not able to fully load the Backend. The topology for a *Group* can be defined using command line options in the run script. The benchmark can be run in single or multiple *Group*(s) configurations described below.

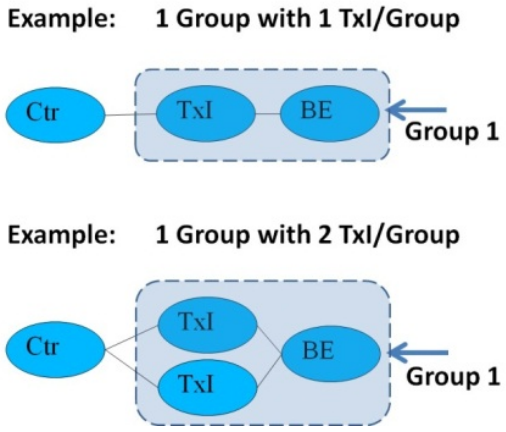
#### 1.3.1. Configuration: All components inside a single JVM instance

This is the simplest case of deployment with intended goal to encourage scaling inside a single JVM instance.

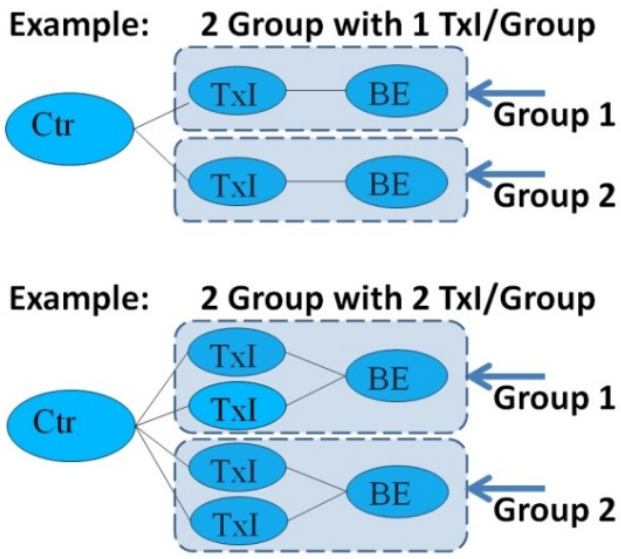
### 1.3.2. Configuration: Components across multiple JVM instances using Single Group

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




### 1.3.3. Configuration: Components across multiple JVM instances using Multiple Groups:

A multiple group configuration consists of one or more group where each group has one Backend and one or more Transaction Injectors (TxI). Since there are multiple Backends, some percentage of requests and transactions require interaction with other Backends initiating Inter-Java process communication amongst Backends.



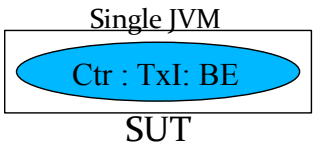

 

### 1.4. Run categories:

Based on most common trends in Java deployments, SPECjbb2013 components Controller, TxI(s) and Backend(s), configurations have been mapped into three different run-categories: SPECjbb2013-Composite, SPECjbb2013-Multi-JVM, SPECjbb2013-Distributed. Since these categories are unique, comparison is disallowed across run-categories.

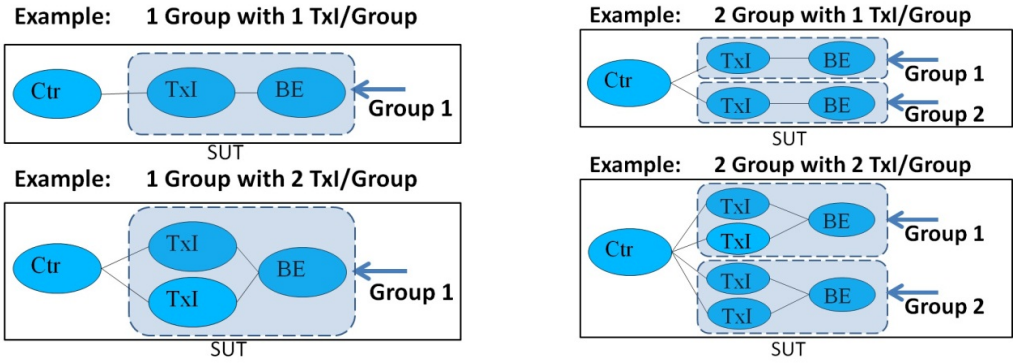
#### 1.4.1. SPECjbb2013-Composite: Single JVM /Single Host:

All the benchmark components (Controller, TxI and Backend) run in a single JVM process. Only a non-virtualized single OS image is allowed. Only one group deployment is allowed for a compliant run. One Group could be configured using one or more TxI(s) if needed to fully load the Backend.



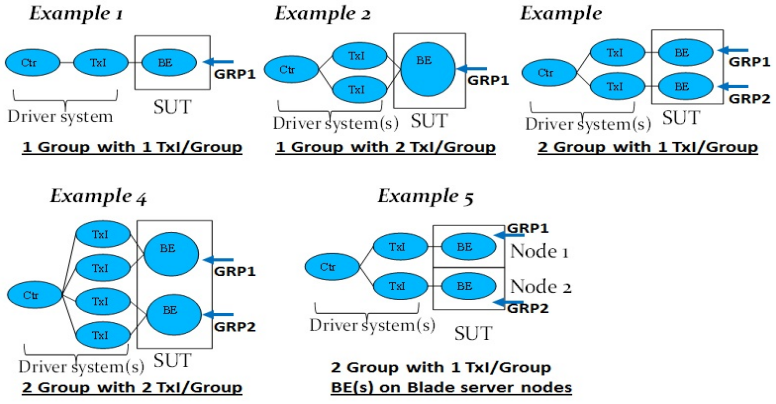

#### 1.4.2. SPECjbb2013-MultiJVM: Multiple JVMs /Single Host:

The Controller, transaction Injector(s) and Backend(s) run in separate JVM processes, but all of them must run on the same system under test (SUT). Only a non-virtualized single OS image is allowed.




#### 1.4.3. SPECjbb2013-Distributed: Distributed JVMs /Single or Multi Hosts:

The Controller, transaction Injector(s) and Backend(s) run in separate JVM processes. In addition to SUT, this category requires a driver system(s). Controller and transaction Injector(s) must run on a driver machine(s) using a non-virtualized OS. The SUT only runs Backend(s), which can be virtualized environments in this category. The SUT can consist of one or more Backends running on one or more Nodes.



# 2. Installation and setup of SPECjbb2013

This section will go over the installation and setup procedures for two types of SPECjbb2013 deployments:

- Configurations running ONLY on SUT (System Under Test)
  - SPECjbb2013-Composite and SPECjbb2013-MultiJVM
- Configurations which require Driver systems(s) in addition to SUT (System Under Test)
  - SPECjbb2013-Distributed

For both configurations, HW and SW installations are similar, with differences in the run scripts executed. If the benchmark user is running either SPECjbb2013-Composite or SPECjbb2013-MultiJVM, feel free to ignore the installation information about Driver system(s) in the description below.

## 2.1. Software installation:

The software components to be installed are the SPECjbb2013 kit and JRE. The benchmark kit and JRE need to be installed in all OS images on SUT as well as on all Driver systems(s).

### 2.1.1. OS (Operating System) requirements

Benchmark could run on most OS environments including virtual OS instances. But for compliant runs, SPECjbb2013 categories have different requirements as describe below:

- SUT (System Under Test):
  - Category SPECjbb2013-Composite and SPECjbb2013-MultiJVM must use non-virtualized single OS instance
  - Category SPECjbb2013-Distributed can use non-virtualized single or multiple OS images as well as virtualized OS environments and deployments across multiple OS instances
- Driver system(s) (only needed for SPECjbb2013-Distributed) must use non-virtualized OS environment

### 2.1.2. Java Runtime Environment (JRE) set-up

Before proceeding it is the responsibility of the user to select a suitable Java Runtime Environment (JRE) (Java SE 7 or higher) is installed on the SUT. There are a variety of JREs available from a number of different vendors. Also note: The JRE is the only auxiliary software that must be installed within the benchmark environment with the SPECjbb2013 benchmark suite. No additional database or web server components are required.

### 2.1.3. Benchmark kit

Unzip the attached SPECjbb2013.tar.gz or SPECjbb2013.zip file into the directory of your choice or follow the install instructions in the benchmark readme.txt file.

The top-level directory contains the jar executable and a config/ directory that stores the prop files as well as HW and SW configuration files. They are named *template-C.raw* for SPECjbb2013-Composite, *template-M.raw* for SPECjbb2013-MultiJVM and *template-D.raw* for SPECjbb2013-Distributed. By default the HW and SW details will be taken from these files. If a different file name and/or directory needs to be used, the user can give the path to the new file on the command line by adding "-raw <raw file>" to the launch command of the Controller component.

The exact location within the OS directory structure into which the SPECjbb2013 suite is to be installed is subject to the user's discretion. The user must either add the path of the Java executable to the PATH environment variable (consult the documentation of the OS for instructions), or invoke the JRE with the fully qualified path and filename based on the installation directory of JRE and benchmark suite.

### 2.2. Network setup

Network requirements are very different among three run categories.

SPECjbb2013-Composite does not require any network as all components run inside a single JVM instance.

SPECjbb2013-MultiJVM uses 'localhost' as all components deployed across multiple JVM instances still run inside a single OS image. But, an 'IP address' could be used if the user wants to limit network traffic between Backend←→TxI(s) through specific network cards.

SPECjbb2013-Distributed requires network setup as it involves Driver system(s) and a SUT, and consist of blade servers with multiple nodes.

For a valid SPECjbb2013 benchmark implementation, there is a set of physical environment criteria that must be satisfied. Please consult the SPECjbb2013 Run and Reporting Rules. It is required to connect the network interface of the SUT so that it will establish a TCP/IP connection with the Driver system(s). Configure the operating system's network configuration such that the SUT is on the same subnet as the controller. Also disable the firewall for all systems to allow them to communicate through TCP/IP. Consult the operating system's documentation for specific instructions regarding this procedure.

### 2.3. Minimum hardware requirements

For categories SPECjbb2013-Composite and SPECjbb2013-MultiJVM all components of the benchmark run on the SUT. For category SPECjbb2013-Distributed, in addition to SUT, a minimum of one Driver system is needed.

- SUT (System Under Test) requirements are:
  - Minimum configuration is single Group and it needs at least 4G of RAM
  - Larger memory configurations (> 24G) may be needed for performance runs
  - Minimum configuration is single Group
- Driver system(s) (only needed for SPECjbb2013-Distributed)
  - Minimum configuration is Controller and single TxI /Group requiring at least 4G of RAM
  - Need 2GB RAM for each additional TxI.
  - Approximate total RAM needed = 2GB Controller + 2GB * Number of all TxI(s)

## 3. Quick tips

Dedicated Transaction Injectors (TxI) ←→ Backend(BE) together are called a Group.

- A Backend requires at least one TxI but if needed can have more than one TxI mapped to it.
- A topology can be defined for this mapping.
- One TxI cannot go across two Backends.
- Run category depends on type of Controller invoked (-m [*composite / multicontroller/ distcontroller*] )

The following heap size and GC tips are for guidance purpose only:

- Usually for each Backend, a minimum heap setting of 2GB is needed. Larger heaps may be needed for generational garbage collectors.
- For each TxI, heap size of 2GB is suggested.
- For Controller, heap size of 2GB is suggested. For configuration running large number of groups (>64), even larger size heap for controller is needed to avoid OOM (Out Of Memory) error during report generation phase. If OOM is experienced, just re-run the reporter with larger heap using the *run_reporter.** scripts.
- Benchmark stresses response time and as a result GC policies can have significant impact. Collecting GC pause data can also be helpful (-verbosegc on JVM command line).

Capturing the Controller standard output can be helpful in debugging many issues.

# 4. SPECjbb2013 trial run

A test run for category SPECjbb2013-Composite is the easiest since all components are inside a single JVM instance. Once you've installed the benchmark kit and appropriate JRE, update modified the run_composite.* script with correct path to java and just run the script. Depending on the capabilities of your test system, you may need to increase the Java heap size and other JVM tuning in the run_composite.* ascript. The benchmark run will last around 2 hours, and the results will be written to the result directory.

Running SPECjbb2013-MultiJVM is next easiest since all components are launched inside a single OS instance. This category requires launching three components: the controller, at least one Backend, and at least 1 TxI. The Backend and TxI combined are known as a Group. It is described later as how to define a Group when launching these three components. After a successful run, the result is also stored in the result directory.

Running SPECjbb2013-Distributed is the most complex but may more closely map to many deployments where user requests comes from an outside source. In the simplest form, the user needs a SUT system and a Driver system. It is described later as how to modify the scripts to launch Controller and TxI on the Driver system and another script to launch Backend on the SUT. The final result is generated and user can find it on the Driver system in result directory of Controller component.

# 5. Running the benchmark

The benchmark user needs to populate the template file with HW/SW details, the property file for run configuration, and launch script to specify intended run category. Benchmark kit has templates for each of these.

### 5.1. HW and SW configuration: *config/template-[C/M/D].raw* file

In the config directory, there are three *template-[C/M/D].raw* files (*template-C.raw* for SPECjbb2013-Composite, *template-M.raw* for SPECjbb2013-MultiJVM and *template-D.raw* for SPECjbb2013-Distributed). User needs to populate the appropriate configuration file based on SPECjbb2013 category user is planning to run.

This file is not used during the benchmark run. At the end of run the reporter takes *template-[C/M/D].raw* and <binary log> of the run as input to produce default 'level 0' HTML report and submission file *.raw. If default name is used for *template-[C/M/D].raw*, reporter will automatically pick correct template file based on run category else use "-raw <raw_file>" command line option for the Controller launch command to give specific name raw file. Since Controller invokes the reporter, this command line option only needed for Controller. Note: If needed, the reporter can be manually invoked after the benchmark run.

### 5.2. Edit benchmark configuration: *config/SPECjbb2013.prop* file

Many settings are defined by properties that can be set either in SPECjbb2013.props file or on the command line at launch. Note that command line property setting overrides the same setting in the properties file.

To make it easy, most of the properties are passed from Controller to all other components like TxI(s) and Backend(s) after the handshake with these components. User only needs to modify property file being used by Controller. Note: The property passed by Controller overrides the same being set locally.

There are few properties that TxI(s) and Backend(s) need before handshake during initialization. These properties need to be set for each component either at launch or in property file.

All components of SPECjbb2013-Composite and SPECjbb2013-MultiJVM run in the same directory. As a result all properties can be defined in one file SPECjbb2013.prop for these two categories.

SPECjbb2013-Distributed is launched on SUT and Driver system(s). Most properties need to be defined in directory that is being used to launch Controller. TxI(s) and Backend(s) only use a small set of the remaining properties. They are needed for initialization of these components; host IP of Controller, thread pools etc., must be defined at launch command or property file of these components.

### 5.3. Edit benchmark run scripts

Basic scripts to launch different SPECjbb2013 categories in Unix (Linux, Mac OSX) and Windows environments are included with the kit. The benchmark user needs to modify JDK path and command line options for the desired configuration and deployment.

1. **SPECjbb2013-Composite: Single JVM /Single Host**

   Script to use: run_composite.sh or .bat

2. **SPECjbb2013-MultiJVM: Multiple JVMs /Single Host**

   Script to use: run_multi.sh or .bat

3. **SPECjbb2013-Distributed: Distributed JVMs /Single or Multi Hosts**

   Two scripts to use:

   run_distributed_ctrl_txI on Driver machine outside SUT

   run_distributed_sut for running Backends on SUT

The examples below are to launch the benchmark components are the simplest command line. To add JVM command line options and other benchmark properties please refer to scripts and properties files for further explanation.

### 5.4. Running SPECjbb2013-Composite: Single JVM /Single Host

This category is the simplest of all the three categories as all components run within single JVM instance.

```
java -jar SPECjbb2013.jar –m composite
```

### 5.5. Running SPECjbb2013-MultiJVM: Multiple JVMs /Single Host

All benchmark components like Controller, TxI(s) and BE(s) run on SUT inside non-virtualized single OS image. Launch the following separate JVM processes.

```
java -jar SPECjbb2013.jar -m multicontroller

java -jar SPECjbb2013.jar -m txinjector -G <groupid> -J <jvmid>

java -jar SPECjbb2013.jar -m backend -G <groupid> -J <jvmid>
```

Where groupid and jvmid are alphanumeric strings.

#### 5.5.1. Setting multiple Groups

This configuration allows for multiple Groups, each with its own set of transaction Injectors and Backends. The *groupid* uniquely identifies each Group (TxI(s) ←→ Backend) whereas the *jvmid* distinguishes among JVM instances launched for multiple TxI(s) and Backend that are part of the same Group. The *jvmid* needs to be unique only within a group and same the *jvmid* could be reused across groups. See examples later in this section.

The property *specjbb.group.count* sets the number of Groups. This property needs to be set when you are running with more than one Group.

#### 5.5.2. Setting more than one "TxI(s) / Backend" (or TxI(s) / Group):

It is possible to use more than one TxI for each Backend. The property can be set either on the command line via –Dspecjbb.txi.pergroup.count=<value2> or in the config/SPECjbb2013.props file.

Try increasing *specjbb.txi.pergroup.count* value if you observe "IR is under limit" failures, which indicate that you may need more transaction Injectors to produce the requested injection rate. See examples later in this section.

### 5.5.3. Example Multi-JVM run for 1 Group with 1 TxI / Backend (or 1 TxI / Group)

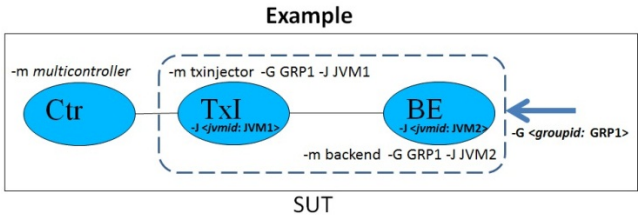
Ensure specjbb.group.count=1 is set in the props file and launch:

```
java -jar SPECjbb2013.jar -m multicontroller
```

Then launch the transaction Injectors and Backend with the following commands:

```
java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM2
```




### 5.5.4. Example Multi-JVM run for 1 Group with 2 TxI / Backend (or 2 TxI / Group)

Set the property specjbb.txi.pergroup.count=2 and launch:

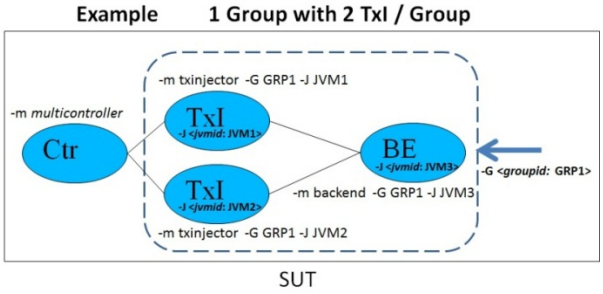
```
java -jar SPECjbb2013.jar -m multicontroller
```

Then launch the transaction Injectors and Backend with the following commands:

```
java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM2

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM3
```




### 5.5.5. Example Multi-JVM run for 2 Groups with 1 TxI / Backend (or 1 TxI / Group)

Set specjbb.group.count=2 in the props file and launch:

```
java -jar SPECjbb2013.jar -m multicontroller
```

Or use:

```
java -Dspecjbb.group.count=2 -jar SPECjbb2013.jar -m multicontroller
```

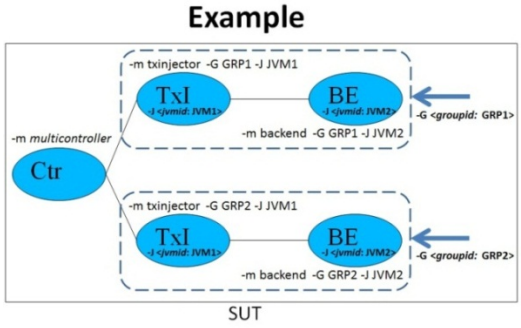
Then launch the transaction Injectors and Backends with the following commands:

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM2

java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM1

java -jar SPECjbb2013.jar -m backend -G GRP2 -J JVM2




The above approach can be used for more than 2 groups as well.

### 5.5.6. Example Multi-JVM run for 2 Groups with 2 TxI / Backend (or 2 TxI / Group)

Set specjbb.group.count=2 and specjbb.txi.pergroup.count=2 in the props file and launch:

java -jar SPECjbb2013.jar -m multicontroller

Or use:

java -Dspecjbb.group.count=2 -Dspecjbb.txi.pergroup.count=2 -jar SPECjbb2013.jar -m multicontroller

Then launch the transaction Injectors and Backends with the following commands:

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM2

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM3

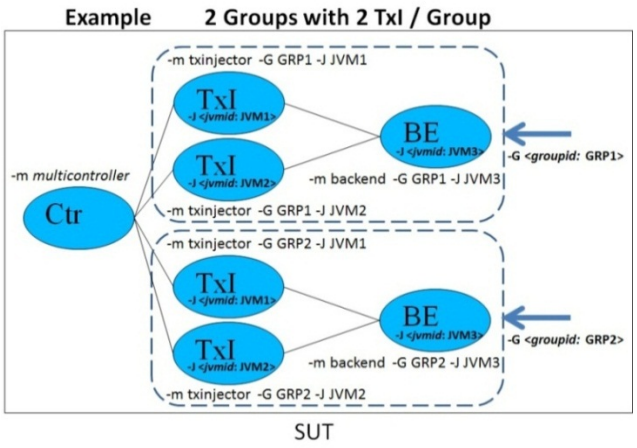
java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM1

java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM2

java -jar SPECjbb2013.jar -m backend -G GRP2 -J JVM3

Example configuration of 2 Groups using 2 TxI / Group can use *groupid* GRP1 can have *jvmid* (JVM1 for TxI 1, JVM2 for TxI 2 and JVM3 for Backend) and *groupid* GRP2 can have same *jvmid* (JVM1, JVM2 and JVM3). But, within a group, all *jvmid*(s) must be unique.




The above approach can be used for more than 2 groups as well.

### 5.6. Running SPECjbb2013-Distributed: Distributed JVMs /Single or Multi Hosts:

This category requires a driver system(s) to run Controller and TxI(s). Only Backend(s) run on SUT.

User should read all configurations of SPECjbb2013-Multi-JVM category to understand this category better. The main difference between SPECjbb2013-Distributed vs. SPECjbb2013-Multi-JVM is that Controller and TxI(s) are deployed on separate Driver system(s) instead of residing on the SUT. All nomenclature for defining the *groupid*, *jvmid* and properties *specjbb.group.count* and *specjbb.txi.pergroup.count* are similar between these categories.

Unpack binaries on Driver machine and on the SUT (for multi hosts, unpack on all Hosts.) Many configurations for single and multi-host are possible. Please see examples below:

#### 5.6.1. Example distributed run for 1 Group with 1 TxI / Backend (or 1 TxI/Group)

Unpack binaries and set in the props file on Driver machine and on the SUT (all hosts):

```
specjbb.group.count=1
specjbb.controller.host=<ControllerIP >
```

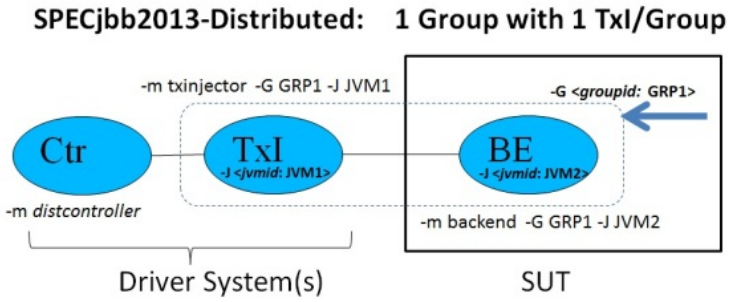
Launch Controller and TxInjector on the Driver machine with:

ControllerIP: java -jar SPECjbb2013.jar -m distcontroller

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

Launch Backend on SUT:

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM2



### 5.6.2. Example distributed run for 2 Groups using 1 TxI / Backend

Unpack binaries and set in the props file on Driver machine and on the SUT (all hosts):

specjbb.group.count=2
specjbb.controller.host=<ControllerIP >

Launch Controller and TxInjectors on the Driver machine with

ControllerIP: java -jar SPECjbb2013.jar -m distcontroller

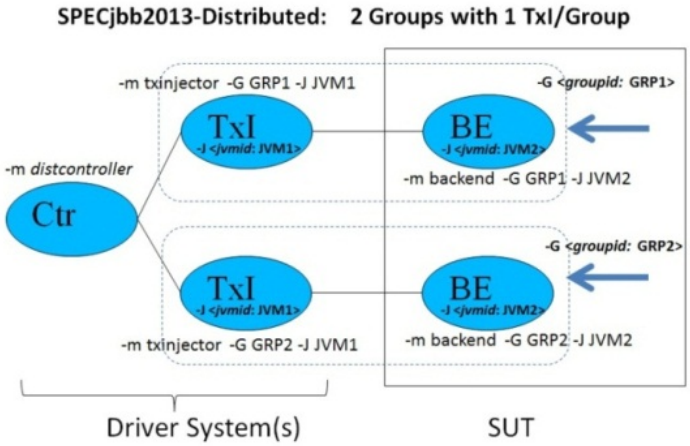
java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM1

Launch Backend on SUT (each host OS can have one or more Backends):

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM2

java -jar SPECjbb2013.jar -m backend -G GRP2 -J JVM2




Above example can be used for groups 2 or larger.

### 5.6.3. Example distributed run for 2 Groups using 2 TxI / Backend

Unpack binaries and set in the props file on Driver machine and on the SUT (all hosts):

specjbb.group.count=2

specjbb.txi.pergroup.count=2

specjbb.controller.host=<Controller IP address>

Launch Controller and TxInjectors on Driver machine with

ControllerIP: java -jar SPECjbb2013.jar -m distcontroller

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM1

java -jar SPECjbb2013.jar -m txinjector -G GRP1 -J JVM2

java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM1

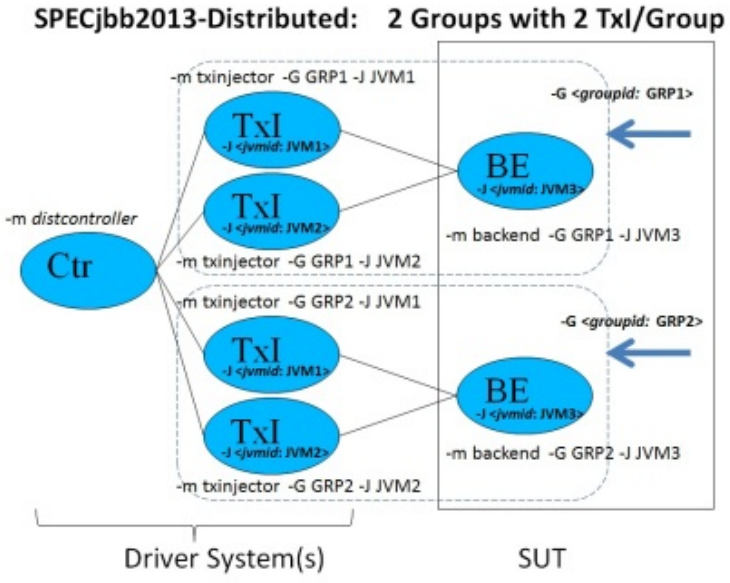
 

java -jar SPECjbb2013.jar -m txinjector -G GRP2 -J JVM2

Launch Backend on SUT (each host OS can have one or more Backends):

java -jar SPECjbb2013.jar -m backend -G GRP1 -J JVM3

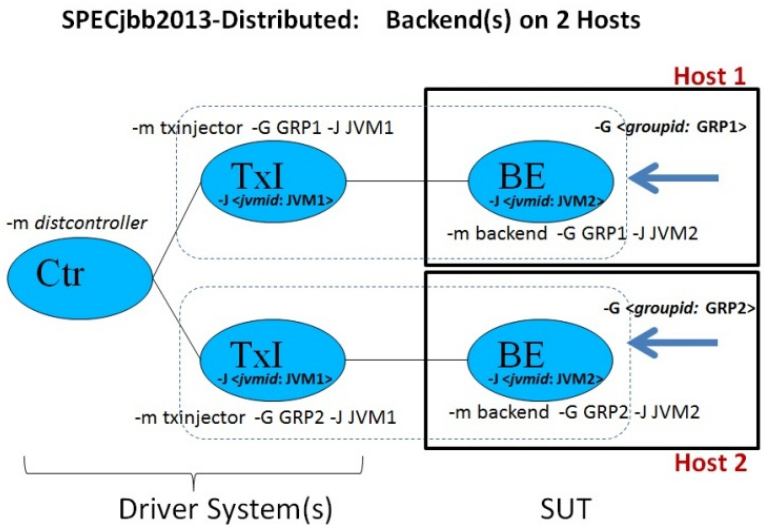
java -jar SPECjbb2013.jar -m backend -G GRP2 -J JVM3




Above example can be used for groups 2 or larger using 2 or more TxI / Backend.

### 5.6.4. Backend(s) deployed across multiple Hosts (Blade servers)

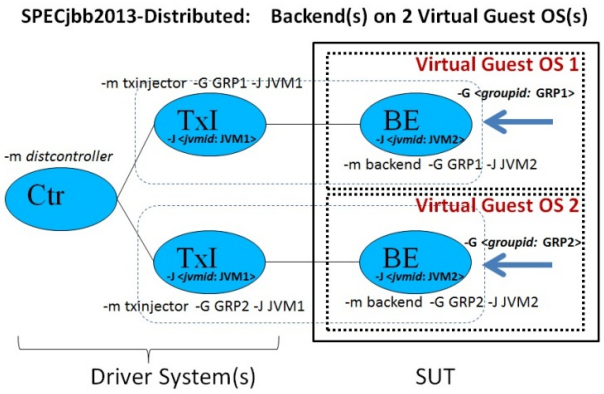
For SPECjbb2013-Distributed category, Backend(s) can be deployed across multiple hosts.



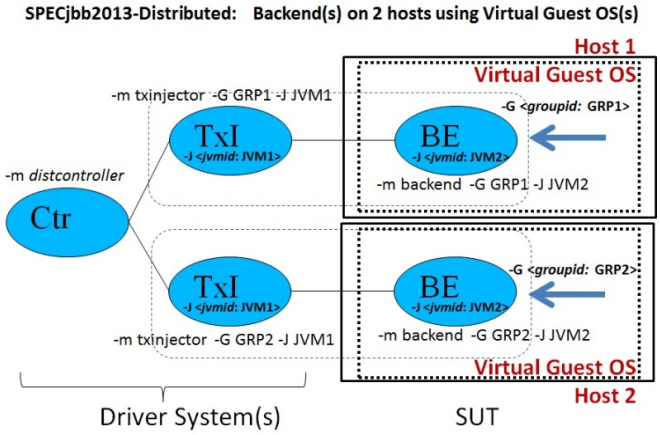
### 5.6.5. Backend(s) deployed across virtualized OS images

For SPECjbb2013-Distributed category, Backend(s) can be deployed across multiple Virtual Guest OS(s).



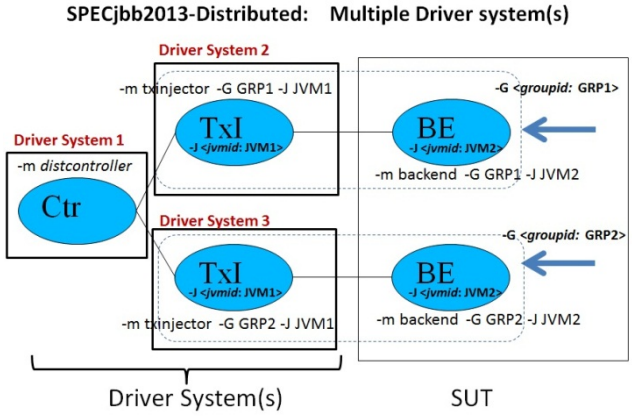

### 5.6.6. Backend(s) deployed across multiple Hosts using virtualized OS images

For SPECjbb2013-Distributed category, Backend(s) can be deployed across multiple hosts using virtualized OS(s).




### 5.6.7. Multiple Driver systems

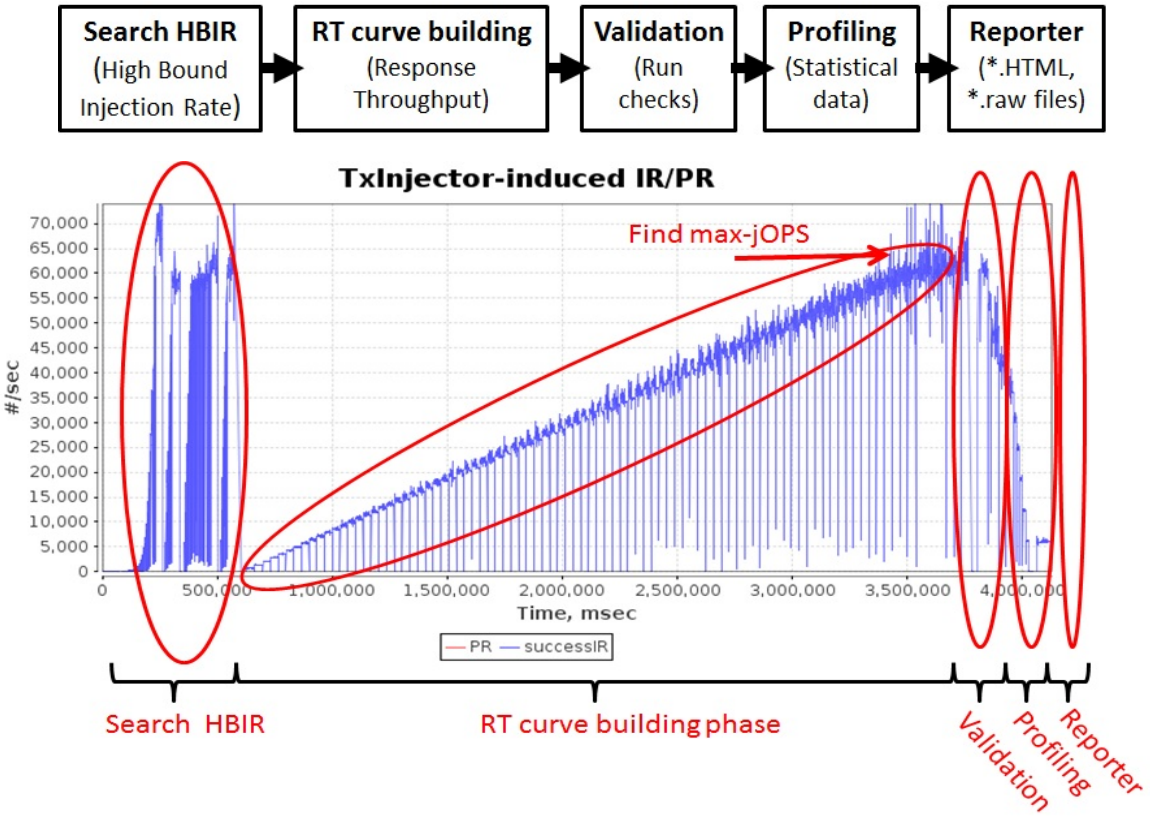
For SPECjbb2013-Distributed category, if needed, multiple driver systems can be configured. For compliant runs, all driver systems must be identically configured using non-virtualized OS for deployment of controller and TxI(s) components across driver systems.



# 6. Run progress

Once all the benchmark components are launched, all TxI(s) and Backend(s) components send handshake messages to the Controller. The controller starts recording progress information in controller.log and controller.out and detailed measurement information in the <binary log> in the current benchmark directory. Other agents start individual logs to record progress. After successful handshake, the controller takes the benchmark through following stages:




## 6.1. Search HBIR

Search HBIR, or the search for the High Bound Injection Rate, approximates the maximum Injection Rate (HBIR) the system can handle. Later, during RT curve building phase, RT step levels are incremented by 1% of HBIR. For testing and research, HBIR value can be manually set using property but for compliant runs, it must be determined automatically by the "Search HBIR" phase.

## 6.2. RT curve building

Response-Throughput (RT) curve building phase is used to determine the overall maximum throughput capacity of the system, both with and without response time requirements. This is done by evaluating at each RT step level starting 0% of HBIR, then incrementing the step level by 1% of HBIR until maximum capacity is reached. This phase produces the data that is used to determine both metrics max-jOPS and critical-jOPS. After finding max-jOPS the benchmark runs for several more step levels to show as how system handles throughput levels that are higher than max-jOPS.

For most systems, max-jOPS should be between 70% and 90% of HBIR. In rare cases, max-jOPS > 100% of HBIR is possible and benchmark will continue to test RT step levels >100% of HBIR to determine max-jOPS.

Systems using JVM configurations which result in large pauses from GC (Garbage Collection) may find that max-jOPS sometime can be much lower than HBIR and many RT step levels are continue to pass even beyond max-jOPS.

This happens because severe pause(s) are occurring during the RT curve building phase that results in a RT step level failure, while beyond this RT step levels will pass, as pauses are occasional. Since max-jOPS is last successful RT step level before first failure, the max-jOPS metric will be lower. User should resolve the cause of severe pauses to remedy this issue.

### 6.3. Validation

The validation phase is used for validation of business logic. During this phase a given IR is executed and then validation checks are performed against data structures corresponding to business logic to ensure a correct state and accurate execution.

### 6.4. Profiling

Profiling phase is used to do an intrusive profile of the benchmark run. The gathered information can be found in the advanced report.

### 6.5. Reporter

At the end of benchmark run, Controller invokes the reporter unless command line option "-skipReport" was used.

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

On invocation of reporter, it takes the <binary log> and template-[C/M/D].raw as input files. A directory in SPECjbb2013/result/<binary log name dir>/report-<NUM>/ is created all files generated for default 'level 0' HTML report and *.raw files are placed here unless command "-t <result_dir>" is used. Output report is for 'level 0' HTML unless "-l <report level 0/1/2/3>" is used. For compliant report 'level 0' HTML must be generated.

## 7. Approximate run length

A typical SPECjbb2013 benchmark run takes approximately 2 hours. Approximate time for different 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 many Groups

## 8. Operational validity

In order to create compliant results, a run must pass several runtime validity checks. These checks are described in the SPECjbb2013 Run and Reporting Rules.

## 9. Benchmark metrics

The benchmark has two metrics. Comparison can be done on any one metric, but both metrics are required to be reported in proximity of each other.

### 9.1. SPECjbb2013-<run category> max-jOPS

A throughput oriented metric called SPECjbb2013-<run category> max-jOPS is calculated by the following:

- Last success IR of RT step level before the First Failure of an RT step level

During RT curve building phase, RT step levels are incremented by 1% of HBIR and each RT step level is evaluated for PASS / FAIL criterion. First Failure of an RT step level is determined as follows: if IR for that RT step level fails either settle or steady state criterion (see Design Document on benchmark website for details), the IR level is

retried for longer period of time. If the second attempt also fails then this IR is determined as a failure. A total of 5 retries are allowed during RT curve building phase while at any RT step level only one retry is allowed.

### 9.2. SPECjbb2013-<run category> critical-jOPS

A throughput under response time constraint metric called SPECjbb2013-<run category> critical-jOPS calculated:

- Geo-mean of (*critical-jOPS@ 10ms, 50ms, 100ms, 200ms and 500ms response time SLAs*)

During RT curve building phase, for each RT step level, detailed response times are measured by Transaction Injector for various types of requests from issue to receive. For critical-jOPS, p99 (99$^{th}$ percentile) of response times of three types of Purchase requests (refer to SPECjbb2013 Design Document posted on the benchmark website) from RT step level 1% to max-jOPS are considered. To represent response time of diverse Java environments, five response-time SLAs (Service Level Agreement) of 10ms, 50ms, 100ms, 200ms and 500ms are chosen. For each SLA threshold, the individual critical-jOPS is determined by the 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.

Results file contain information about individual critical-jOPS as well as other response time details.

## 10. Results reports

One of the major functions of the Controller is to collect the benchmark data in real time, and output this data into logs that can be retrieved and viewed by the user. When Controller is first initiated, one of the first things that it does is create text log file for recording run progress and binary log file to record complete data of the run (except HW and SW configuration details). Upon successful completion of the benchmark run, Controller will stop writing to the binary log file and launch the reporter.

During the run, untill the reporter is invoked, controller.log, controller.out and <binary log> are in current directory. When invoked, reporter takes following files as input:

- binary log file containing run data
- template.[C/M/D].raw file containing hw/sw configuration details
    - Input raw file could be specified at the Controller launch command using option "-raw <file>".

Once reporter invocation completes, both above input files are left in its original places and reporter produces following files:

- Directory SPECjbb2013/result/SPECjbb2013-<C/M/D>-<Date>-<runNum>/report-<num>/
    - SPECjbb2013-<date>-<run number>.raw file
    - SPECjbb2013-<date>-<run number>.HTML file
- Directory SPECjbb2013/result/SPECjbb2013-<C/M/D>-<Date>-<runNum>/report-<num>/Images/
    - Images for html file
- Directory SPECjbb2013/result/SPECjbb2013-<C/M/D>-<Date>-<runNum>/report-<num>/Data/
    - Data of html graphs
- Directory SPECjbb2013/result/SPECjbb2013-<C/M/D>-<Date>-<runNum>/report-<num>/logs/

The user using "-t <FILE/DIR> on the Controller launch command line can also specify where these results files are stored. User is expected to ensure that the JVM instance for Controller will have the necessary credentials to write to this path.

By default HTML 'level 0' report is produced which contains the benchmark user viewable report and data. Newly generated file SPECjbb2013-<date>-<run number>.raw has all fields from template.[C/M/D].raw, data for 'level 0' HTML report along with fields for SPEC office to edit and manage publications. If a publication is desired, user needs to submit to SPEC this raw file along with FDR package (refer to SPECjbb2013 Run and Reporting Rules section 4).

For various reasons, run could be invalid or show warnings. The HTML result report should indicate the reasons. Most warnings and invalidations can be mitigated by changing parameters and/or by using different JVM tuning parameters.

# 11. Results using manual reporter invocation

For various reasons, user may need to run the reporter manually. The reporter can be run manually with the following command:

```
java -Xms2g -Xmx2g -jar SPECjbb2013.jar -m reporter -s <binary_log_file>
```

**Important note**: Reporter needs three input files and since only one file is being defined on command line, other two are assumed to be in default [dir/file_name] with details below:

1. <binary_log_file> is SPECjbb2013-<run_mode_mark>-<timestamp>.data.gz file produced during the benchmark run
2. By default, reporter takes config/template-[C/M/D].raw file as input for HW/SW details. If this is not the file with proper HW/SW details, refer to section 11.1 to manually specify the template file on the command line.
3. For consistency reasons, reporter searches for SPECjbb2013/config/SPECjbb2013.props file. It does not need to be the exact file that was used for the run. File supplied with the kit in default directory is sufficient. To manually specify, command line option "-p <prop file>" can be used.

The above command will create a (result/SPECjbb2013-<C/M/D>-<timestamp>/report-<num> directory, if it does not exist, and then create a sub-directory named report-<num>. It then generates all results files for 'level 0' HTML report inside that directory. Each time reporter is invoked on same binary log file, it produces a new directory, incrementing 'num' in "report-<num>" under same result/SPECjbb2013-<C/M/D>-<timestamp>/ directory.

### 11.1. HW/SW details input file with user defined name

If user wants to declare a file for HW and SW details, the following command can be used:

```
java -Xms2g -Xmx2g -jar SPECjbb2013.jar -m reporter -raw <hw/sw details file.raw> -s <binary_log_file>
```

### 11.2. To produce higher level HTML reports

To produce various levels of report, following command can be used:

```
java -Xms2g -Xmx2g -jar SPECjbb2013.jar -m reporter –raw <file> -s <binary_log_file> -l <report_level>
```

Where report 0 <= level <= 3 (0 - minimum report, 3 - most detailed report).

### 11.3. Regenerate submission raw file and HTML report

The file SPECjbb2013-<run_mode_mark>-<timestamp>.raw inside the result directory is the raw submission file containing the actual test measurement and result. Once this file is generated and user needs to edit HW/SW details, user can re-generate this submission file with updated HW/SW configuration using following two methods.

#### 11.3.1. Using edited original template file and binary log

In this case user needs both binary log and edited HW/SW details template file and can re-generate submission file and HTML report using following command:

java –Xms2g –Xmx2g -jar SPECjbb2013.jar -m reporter -s <binary_log_file> -raw <raw_template_file>

#### 11.3.2. Edit submission raw file and re-generate HTML 'level 0' report without binary log

User can directly edit submission raw file SPECjbb2013-<run_mode_mark>-<timestamp>.raw, modify the HW/SW details and re-generate HTML report with updated HW/SW configuration using the following command:

java –Xms2g –Xmx2g -jar SPECjbb2013.jar -m reporter -raw <raw_submission_file>

## 12. Results file details

While all of the results files contain valuable data about the benchmark run, many users will likely be more interested in the HTML files to review result and submission raw file to edit any HW/SW information. This section details the format of these two files.

### 12.1. Report summary output for 'level 0' HTML

User can click on a field in HTML report and it will display the definition of the field. HTML 'level 0' report is divided into several sections detailing different aspect of the benchmark.

#### 12.1.1. Top section

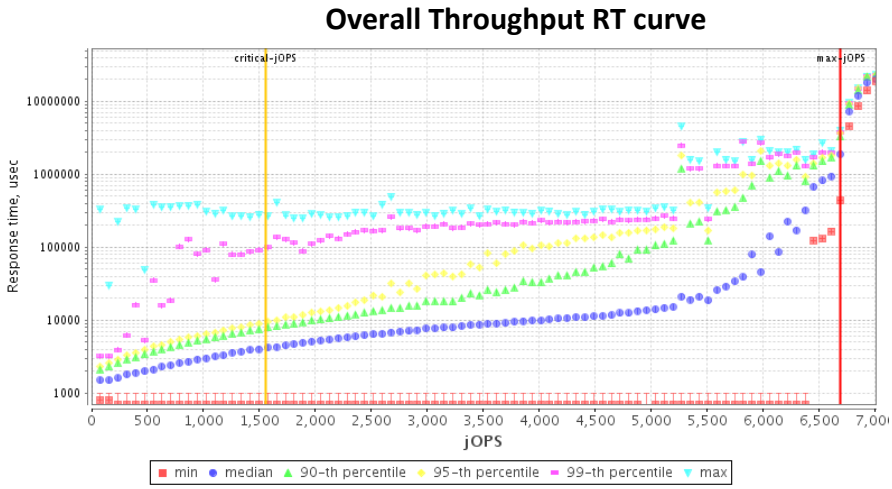
It lists the benchmark metrics max-jOPS and critical-jOPS as well as high-level tester information.

**Note:** Any inconsistencies with the run and reporting rules causing a failure of one of the validity checks implemented in the report generation software will be reported here and all pages of the report file will be stamped with an "Invalid" water mark in case this happens. The printed text will show more details about which of the run rules wasn't met and the reason why. More detailed explanation may also be at the end of report in sections "run properties" or "validation Details". If there are any special waivers or other comments from SPEC editor, those will also be listed there.

#### 12.1.2. Benchmark results summary

The raw data from this graph can be found by clicking on the graph. This graph shows the Response-Throughput (RT) phase of the benchmark. Initial phase of finding High Bound Injection Rate (HBIR) (Approximate High Bound of throughput) and later validation at the end of the run are not part of this graph. X-axis is showing jOPS (Injection Rate: IR) as system is being tested for gradually increasing RT step levels in increments of 1% of HBIR. Y-axis is showing response time (min, various percentiles, max) where 99th percentile determines the critical-jOPS metric as shown by a yellow vertical line. The last successful RT step level before the "First Failure" of an RT step level is marked as red vertical line reflecting the max-jOPS metric of the benchmark. Benchmark continues to test few RT step levels beyond the "First Failure" RT step level. Often, there should be very few RT step levels passing beyond "First Failure" RT step level else it indicates that with more tuning system should be able to pass higher max-jOPS. A user needs to view either controller.out or level-1 report output to view details about levels beyond "First Failure" RT step level. Red line marks the max-jOPS and yellow line marks the critical-jOPS.




### 12.1.3. Overall SUT description

This section shows same information as entered in the template-[C/M/D].raw file.

### 12.1.4. SUT description

This section also shows same information as entered in the template-[C/M/D].raw file for HW and SW products.

### 12.1.5. max-jOPS and critical-jOPS details

These sections show details about max-jOPS and critical-jOPS. For max-jOPS it shows RT step levels just before and after the max-jOPS RT step level.

For critical-jOPS, it shows a table with individual critical-jOPS@10ms, 50ms, 100ms, 200ms and 500ms as well Geomean:

| critical-jOPS = Geomean ( jOPS @ 10000; 50000; 100000; 200000; 500000; SLAs ) | | | | | | |
|---|---|---|---|---|---|---|
| | Response time percentile is 99-th | | | | | |
| SLA (us) | 10000 | 50000 | 100000 | 200000 | 500000 | Geomean |
| jOPS | 432 | 850 | 1447 | 3165 | 5331 | 1551 |

Individual critical-jOPS calculations require knowing the last success and first failure jOPS for p99(99$^{th}$ percentile) for 10ms, 50ms, 100ms, 200ms and 500ms response time SLAs. A table shows jOPS for various threshold of response times using various percentiles.

**Last Success jOPS/First Failure jOPS for SLA points**

| | Percentile | | | | | |
|---|---|---|---|---|---|---|
| | 10-th | 50-th | 90-th | 95-th | 99-th | 100-th |
| 500us | - / 79 | - / 79 | - / 79 | - / 79 | - / 79 | - / 79 |
| 1000us | - / 79 | - / 79 | - / 79 | - / 79 | - / 79 | - / 79 |
| 5000us | 5508 / 5586 | 1888 / 1967 | 865 / 944 | 708 / 787 | 236 / 315 | - / 79 |
| 10000us | 5980 / 6058 | 4013 / 4091 | 1967 / 2046 | 1652 / 1731 | **472 / 393** | - / 79 |
| 20000us | 6137 / 6216 | 5508 / 5272 | 3305 / 3383 | 2439 / 2518 | 708 / 551 | - / 79 |
| 50000us | 6294 / 6373 | 5980 / 5901 | 4406 / 4485 | 3305 / 3383 | **1102 / 787** | 472 / 79 |
| 100000us | 6373 / 6452 | 6137 / 6058 | 4957 / 5036 | 3934 / 3855 | **1888 / 787** | 472 / 79 |
| 200000us | 6373 / 6452 | 6294 / 6216 | 5508 / 5272 | 5508 / 5272 | **3305 / 2675** | 472 / 79 |
| 500000us | 6530 / 6609 | 6373 / 6452 | 5822 / 5272 | 5508 / 5272 | **5508 / 5272** | 5508 / 5272 |
| 1000000us | 6609 / 6688 | 6609 / 6688 | 6373 / 5272 | 6373 / 5272 | 5508 / 5272 | 5508 / 5272 |

### 12.1.6. Number of probes

This validation criterion ensures a proper number of probes for good confidence in measured response time. This graph only shows RT phase step levels. The jOPS for RT step levels are on x-axis and number of probes as % of total jOPS is on y-axis (logarithmic scale). Two horizontal lines are showing limits. To have good confidence in response time, we need to ensure that a good % of total jOPS is being issued as probes. For more details, please refer to validation section 3.4.4 of Run and Reporting Rules document.

### 12.1.7. Request mix accuracy

This validation criterion ensures total issued requests maintained the designed % mix of request. Total requests are issued to maintain a request mix. This graph only shows RT phase step levels. The jOPS for RT step levels are on x-axis and y-axis shows the (Actual % in the mix – Expected % in the mix). For more details, please refer to validation section 3.4.4 of Run and Reporting Rules document.

### 12.1.8. Rate of non-critical failures

This validation criterion ensures that not too many requests and transactions are dropped during the run. If these non-critical failures are 0 during the RT phase, only a message is printed. If non-critical failures during RT phase are >0, then a graph is shown. In case of graph, jOPS for RT step levels are on x-axis and number of non-critical failures for each RT step level is on y-axis. Transaction Injectors (TxI) issue requests to Backend(s) to process. Many times and for various reasons, TxI will timeout after waiting for a threshold. This is counted as non-critical failure. For more details, please refer to validation section 3.4.4 of Run and Reporting Rules document.

### 12.1.9. Delay between performance status pings

This validation criterion ensures that there are not too many very long non-responsive periods during the RT curve-building phase. X-axis is time in milliseconds (msec). Y-axis is showing delay time in msec. Validation criteria applies to whole RT phase and not to individual RT step levels. Also, minimum y-axis value is 5 sec as that is passing criteria and chosen to reduce the size of .raw file for submission. If a user wants to see y-axis data starting with 0, user needs to generate report with level-1 and it will have the detailed graph. For more details, please refer to validation section 3.4.4 of Run and Reporting Rules document.

### 12.1.10. IR/PR accuracy

This graph shows the relationship between IR (Injection rate), aIR (Actual Injection Rate) and Actual PR(Processed Rate). The graph is showing all benchmark phases, starting with HBIR (High Bound Injection Rate) search, RT phase warm-up, RT phase, and finally the validation phase at the end. The X-axis is showing iteration number where iteration means a time period for which IR/aIR/PR being evaluated. IR is targeting Injection rate, actual IR is the IR we could issue for a given iteration and PR is total processed rate for that iteration. To pass the iteration, IR/aIR/PR

must be within certain % of each other. Y-axis shows how much Actual IR and Actual PR differ when compared to IR as base. If those are within low and high bound thresholds the iteration will pass. A user will see many failures during HBIR search. During RT phase and untill max-jOPS is found, there could be some failures as certain number of retries are allowed.

#### 12.1.11. Topology

This section covers the topology for SUT and driver system (Distributed category only). First section shows a summary of the deployment of JVM and OS images across H/W systems. Later sub-sections detail the JVM instances across OS images deployed for each H/W configuration in the SUT and driver system (Distributed category only).

#### 12.1.12. SUT Configuration

This section covers as how JVM instances are deployed inside OS images and those OS images are deployed across HW systems.

#### 12.1.13. Run Properties

This section covers the run properties that are being set by the user.

#### 12.1.14. Validation details

Details about validation are listed in this section.

#### 12.1.15. Other checks

List other checks for compliance as well as High Bound maximum and High Bound settled values during the HBIR (High Bound Injection Rate) search phase.

### 12.2. Submission raw file format

Once a run is complete, reporter by default produces level 0 HTML report and *.raw file for submission. This output *.raw file has two sections:

#### 12.2.1. User editable HW/SW Configuration Details

Data from template-[C/M/D].raw is placed in this section using same exact fields and to correct, user can edit any HW/SW details in this section.

#### 12.2.2. Non-Editable Data and SPEC office ONLY Use Section

This section has all HTML 'level 0' report data in binary format so that 'level 0' HTML file can be produced using just submission raw file. User must not change anything in this section.

Searchable benchmark metrics as well as individual critical-jOPS@10, 50, 100, 200 and 500 ms response time SLAs.

SPEC office section where any changes can only be made by SPEC office.

### 12.3. Controller log file format

The Controller records run progress information. In the beginning it performs a handshake with other TxI(s) and Backend(s). Later it records any messages, including state transition messages, along with every 1 second status updates of overall SUT performance using following format:

```
<Mon Jan 14 00:20:54 NOVT 2013> org.spec.jbb.controller: HIGH_BOUND_IR: settling, (rIR:aIR:PR =
2069:2056:2056) (tPR = 3738) [OK]
```

where:

- `<Mon Jan 14 00:20:54 NOVT 2013>`: Standard time stamp
- `HIGH_BOUND_IR`: Run progress phase (refer to section 6 Run Progress in this document)
    - Other phases can be `TRANSITION`, `WARMUP`, `RT_CURVE`, `VALIDATION` and `PROFILE`.

- `Settling` : Each iteration first need `settling` period before being evaluated in `steady` period
- `rIR:aIR:PR` : Records accumulated values from beginning of iteration. Where:
  - `rIR` ("real IR" target IR) : `aIR` ("actual IR" system could issue) : `PR` (Processed Requests)
- `[OK]`: For current iteration, till this moment in time, accumulated values are meeting the passing criterion.
  - Other message could be `[IR is over limit]`, `[IR is under limit]`, `[PR is over limit]`, `[PR is under limit]`, `[submit error]`

### 12.4. Controller out details

While controller.log keeps track of each 1-second of progress (fine grain status), controller.out logs an overall summary of each iteration during HBIR phase and later a summary of each RT curve step level.

### 12.5. Search HBIR Phase:

When appropriate, benchmark takes snapshots of current benchmark state, which helps ensure faster initialization of datasets for next search state. In the example below, the first field is the time passed from start of the run and 'rampup IR' means gradually ramping to 'trying IR' value.

```
83s: Performing snapshot:(quiescence..) (stop TxI) (stop BE) (saving.. 102819 Kb) (term) (init)..
98s:       rampup IR =      0 ... (rIR:aIR:PR = 0:0:0) (tPR = 0) [OK]
…….          …….              ……….                    ……….
131s:       rampup IR =     90 .... (rIR:aIR:PR = 90:90:90) (tPR = 3642) [OK]
135s:       rampup IR =    100 .... (rIR:aIR:PR = 100:101:101) (tPR = 4160) [OK]
139s:       trying IR =    200 .... (rIR:aIR:PR = 200:194:194) (tPR = 7323) [OK]
143s:       trying IR =    220 .... (rIR:aIR:PR = 220:218:218) (tPR = 7303) [OK]
```

### 12.6. RT Curve Phase:

During RT curve phase, the controller.out stores a summary of each RT step level. Below is an example summary of RT step level at 12% of HBIR, with passing IR at this RT step level (as shown by `[OK]`). Many other details, response time percentiles, probes, and samples for each type of requests are summarized as well.

```
1811s: (12%)   IR =    966 ......|.................(rIR:aIR:PR = 966:966:966) (tPR = 17883) [OK]
```

| 1877s: | Respon | times: | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Request | Success | Partial | Failed | SkipFail | Probes | Samples | min | p50 | p90 | p95 | p99 | max |
| Overall | 55263 | 0 | 0 | 0 | 53716 | 59852 | 500 | 2500 | 4400 | 4900 | 6400 | 198000 |
| InStorePurchase | 29076 | 0 | 0 | 0 | 28458 | 31534 | 500 | 2400 | 4300 | 4800 | 6300 | 197000 |
| OnlinePurchase | 20364 | 0 | 0 | 0 | 19787 | 22004 | 500 | 2300 | 4400 | 4900 | 6400 | 198000 |
| InstPurchase | 5823 | 0 | 0 | 0 | 5471 | 6314 | 600 | 3300 | 4900 | 5400 | 6885 | 198000 |
| AssocOfCat | 60 | 0 | 0 | 0 | 57 | 63 | 35000 | 43000 | 45000 | 47000 | 52000 | 52000 |
| AssocOfProduct | 585 | 0 | 0 | 0 | 503 | 688 | 800 | 2500 | 3800 | 4100 | 5200 | 39000 |
| BusinessReport | 145 | 0 | 0 | 0 | 101 | 147 | 42000 | 48000 | 52000 | 54000 | 58520 | 59000 |
| CustBuyBeh | 582 | 0 | 0 | 0 | 480 | 623 | 200 | 500 | 1300 | 1400 | 2676 | 23000 |
| ProductReturn | 1549 | 0 | 0 | 0 | 1398 | 1667 | 300 | 2500 | 4000 | 4400 | 6432 | 197000 |

In the execution of the RT step level, each second that the Controller receives a performance status a '.' is printed to standard out. A '?' is printed for each second the Controller did not receive a performance status. Most often a long GC pauses is the cause for Controller to not receive a response. A mark of "|" separates the settle period from steady period. In the example below, the settle period is 4 seconds long and steady period is approximately 65 seconds long where towards the end, the system is not responsive for 10 seconds as marked by "??????????".

```
1797s:(18%)IR = 1581 ....|.....................................................??????????....
```

Below is an example of a system responding smoothly:

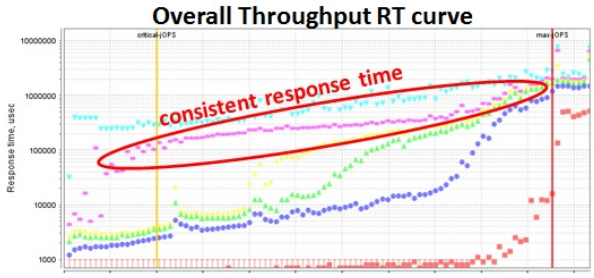
```
1866s:(19%)IR=1058 ........|..................................................... (rIR:aIR:PR =
```

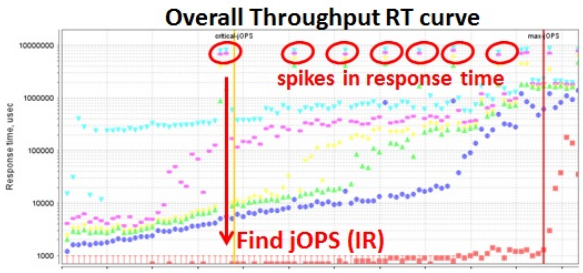
Towards the end of Controller out, details about validation and profile phases are summarized.

## 13. Correlating RT curve data point to Controller.out and Controller.log

A system responding smoothly throughout the run should produce an RT curve where response time values for all RT step levels are following the curve. But, if RT curve has occasional spikes in response time for some RT step levels, often, it is related to "Stop the world" type GC (Garbage Collection) policies that result in very long GC pauses. Such spikes in response time may impact the critical-jOPS benchmark metric. To keep the total benchmark run time to about 2-3 hours, each RT step level is approximately 90 seconds. Its possible that this is not long enough to capture infrequent long GC pauses. As a result, whenever such long GC pauses occur, the p99 (99$^{th}$ percentile) response time for that RT step level may show a big jump in response time Setting each RT step level to a long enough period to capture these infrequent spikes would allow a more accurate response time measurement. However, such a benchmark run would last 12 hours or more, which reduces its usefulness. Considering the many pros and cons, the RT step level duration of 90 sec was selected.




If the user wants to find more details about the data points with high response time, the following procedure can be followed:

- In RT curve, find approximate jOPS (IR: Injection Rate) on x-axis related to the spike in response time and note down the corresponding response time. (RT curve data can be exported to a CSV file by just clicking on the graph)
- In Controller.out file, go to RT curve phase data and find RT step level matching the jOPS value read from the RT curve graph. User can see detailed summary of the RT step level exhibiting the spikes in the response time for p99 response time column of the summary data (as shown in section 12.6 above). Also in the summary, the user may notice "....|.....??????????...." where each "?" represents a second of duration where agents did not respond to Controller. The user can note the absolute time passed from the start of the benchmark run.
- In Controller.log file, find the matching IR in RT_CURVE phase to the corresponding jOPS from RT curve graph. Around this IR level, user can view the system behavior at a 1 second granularity. Based on system time stamps (refer to section 12.3 above), a jump of more than one second indicates the system was not responding and the Controller did not receive the performance status for that time frame. The user should note the system time stamp from Controller.log.

Based on time marking from Controller.log and/or Controller.out, user can investigate the GC verbose log or other system and JVM monitoring parameters to identify possible reasons for poor response time. Resolving such issues should result in improved response time characteristics.

## 14.Exporting HTML report data to CSV or table format

Many graphs and tables have important data for analysis. As a result, data for all graphs in 'level 0' report can be exported to a CSV format by clicking on the graphs. For charts from other levels user should use reporter option "-data4image". When using this option, all charts even in higher level HTML report have links to CSV data.

## 15.Filling HW/SW configuration details in *template-[C/M/D].raw* file

For SPECjbb2013, to describe HW and SW details, user needs to fill appropriate *template-[C/M/D].raw* (SPECjbb2013-Composite *template-C.raw*, SPECjbb2013-MultiJVM *template-M.raw* and SPECjbb2013-Distributed *template-D.raw* file in config/ directory.

Since SPECjbb2013 covers from very simple configuration to very complex distributed deployments, HW/SW configuration file needed flexibility. This section describes below the most complex configuration description, SPECjbb2013-Distributed.

### 15.1. Benchmark and test descriptions

The fields listed are self-explanatory.

### 15.2. Overall SUT (System Under Test) descriptions

Covers overall SUT so that it is easy to understand overall SUT at a glance. The fields are self-explanatory.

### 15.3. SUT or Driver Product descriptions

This section requires a list of all products using the specified format for category SW.JVM, SW.OS, SW.OTHER, HW.SYSTEM HW.OTHER. Each product is given a label.

As example of labels SW.JVM=JVM_1, SW.OS=OS_1, SW.OTHER=OTHER_1, HW.SYSTEM=HW_1 HW.OTHER=Network_1

Note: Information must be included for any software installed. Any irrelevant field can be set to N/A. If a product is not needed, user can comment out or delete those fields. As example if SW.OTHER and HW.OTHER not needed, user could comment all fields of these products.

Also, these product labels work as indexing when describing JVM instance, OS instance and HW host.

### 15.4. Configuration descriptions for unique jvm instances

A JVM Instance is unique if any of the following change: JRE binary, components being launched, JVM parameters and tuning. All unique JVM instances are assigned unique labels like “jvm_Ctr_1”, “jvm_TxI_1”, jvm_BE_1” etc.

Note: Only unique instances need to be defined.

### 15.5. Configuration descriptions for unique OS instances

JVM instances are deployed inside OS images. An OS instance is unique if any of the following is different: unique JVM instances, tuning etc. Each unique OS instance is assigned a label. As example

jbb2013.config.osImages.os_Image_2.jvmInstances = jvm_Ctr_1(1), jvm_TxInjector_1(4)

OS instance label os_Image_2 has: 1 instance of jvm_Ctr_1 and 4 instances of jvm_TxInjector_1

### 15.6. Config descriptions for OS images deployed on a SUT HW or Driver HW

This section describes how unique OS instances are deployed across HW system(s). An example:

jbb2013.config.SUT.system.config_1.osImages = os_Image_1(1)
jbb2013.config.SUT.system.config_1.hw_product = hw_1
jbb2013.config.SUT.system.config_1.nSystems = 1

Config_1 has hardware of 1 system of product hw_1 where only 1 instance of unique OS image “os_image_1” has been deployed.

### 15.7. Modifying HW and SW details after the run

If user needs to change some HW/SW configuration details, user can edit those fields in the template or custom *.raw file and rerun the reporter. User should use –raw option in case of custom *.raw file. User could also edit in submission raw file and rerun the reporter. For details, see earlier section about manual invocation of reporter.

# 16.Benchmark properties

There are many properties that control the operation of SPECjbb2013. The ones that are user-settable are listed in SPECjbb2013 Run and Reporting Rules document section 2.5. This section only covers user settable properties. For details about other advanced options for research and testing purposes refer to SPECjbb2013 Advanced Options and Research section on the benchmark site.

- The SPECjbb2013.prop file within the benchmark kit provides a detailed overview of the properties. Some important points about properties:
    - Benchmark parameters are actually java properties and may also be passed from command line as -Dproperty=value.
    - If a property is set using the launch command line as well as in the property file, launch command has higher priority
    - Most properties are propagated by the Controller to the Agents, overriding property values on the Agent side. A user may update properties files on the Controller host (or update Controller launching command) in case of the Distributed run.
    - Some properties are Controller independent and not propagated by controller. User needs to update such property for every launching component either through property file or command line.

## 16.1. Properties not propagated by Controller

If changing from default, the value should be passed to every launching component either through property file or command line. SPECjbb2013-Composite and SPECjbb2013-MultiJVM are run from the same directory using same property file. In this case, setting them in one property file will be sufficient.

### 16.1.1. Contacting Controller

Properties *specjbb.controller.host=localhost* and *specjbb.controller.port=24000* are given to agents so that all agents can contact this IP address and port for handshaking with Controller.

### 16.1.2. Connection pools for communication among agents:

*specjbb.comm.connect.connpool.size=256,*
*specjbb.comm.connect.pool.min=1*
*specjbb.comm.connect.pool.max=256*

These properties help setup the number of socket connections needed for communications amongst agents. Default numbers should work optimally for most configurations. Larger values or tuning may be needed for larger systems.

### 16.1.3. Timeout for I/O operations

*specjbb.comm.connect.timeouts.connect=60000,*
*specjbb.comm.connect.timeouts.read=60000,*
*specjbb.comm.connect.timeouts.write=60000*

This is the time in milliseconds agents wait for connect or read or write. For large systems or when facing very long pauses, increase these values if observing heartbeat failures or run producing submit errors.

## 16.2. Properties propagated by Controller to all agents

If changing from default, user should set these properties on Controller only as these will be propagated to all agents (TxIs and Backends) correctly. If user sets these properties on Controller side as well as other agents, property values set on Controller will override values set on agent's side.

#### 16.2.1. specjbb.group.count=1

This sets the number of Groups. Since, each Group can only have one Backend, this indirectly sets number of Backend(s) as well. Often, NUMA systems may benefit by running 1 Group per NUMA node. Also, blade servers may benefit by running at least 1 group per blade server.

Note: Remote traffic resulting from transactions spanning across multiple Groups increases with number of Groups. As a result, setting too many Groups when not needed may result in a lower benchmark score.

#### 16.2.2. specjbb.txi.pergroup.count=1

At least one dedicated TxI per group. A TxI cannot issue requests to more than one Group (or Backend). If one TxI is not sufficient to load a Backend, the user can set more than one TxI per Group.

Note: Setting this to a larger value than needed may result in lower score.

#### 16.2.3. specjbb.forkjoin.workers=2

This property sets the thread pool size of Backend agents. Default value is number of processor threads available. In testing, setting this value to twice the number of processor threads available results in better performance.

Note: When running multiple Groups, it is difficult for the benchmark to correctly assign this value. Since, this value can have significant impact on performance, user is advised to investigate the impact of this property using affinity and as well as how different JVMs account for number of processor threads available when invoked using NUMA and processor affinity.

#### 16.2.4. specjbb.controller.rtcurve.warmup.step=0.1 (10%)

At the beginning RT curve, the benchmark internal data structures are re-initialized. As a result, IR 10% of HBIR is applied for a 3-minute warm-up phase. If a system needs more warm-up time, the user could increase the % of HBIR up to 90% of HBIR but not allowed to increase the 3-minute time window of warm-up.

#### 16.2.5. specjbb.controller.maxir.maxFailedPoints=3

This property should have no impact on performance. This just helps user to be able to observe behavior after the max-jOPS has been determined based on First Failure of RT step level. This setting controls how many consecutive failures of RT step levels are allowed until RT curve phase be stopped.

#### 16.2.6. specjbb.run.datafile.dir=.

Default location of the binary log file is current dir. This property can be set to change the directory location for binary log outputs.

#### 16.2.7. specjbb.customerDriver.threads=64

This property plays an important role for Transaction Injector. By default it sets TxI thread pool for issuing probes, saturate requests, and service. This value should work fine for most systems, but, if there are not enough probes, individual values could be increased using:

*specjbb.customerDriver.threads.probe=* ,
*specjbb.customerDriver.threads.service=,*
*specjbb.customerDriver.threads.saturate=*

#### 16.2.8. specjbb.mapreducer.pool.size=2

Controller ForkJoinPool size supporting parallel work of TxInjector/Backend agents.

#### 16.2.9. Handshake time-out

These properties control as how long Controller will wait for agents to respond back.

*specjbb.controller.handshake.timeout=600000* : Time period (in milliseconds) for logging status of the initial Controller <-> Agent handshaking

*specjbb.controller.handshake.period=5000:* Time period (in milliseconds) for logging status of the initial Controller <-> Agent handshaking

#### 16.2.10. Heartbeat

The heartbeat leader receives heartbeats from all agents. These two properties control the frequency and threshold for heartbeats. This mechanism ensures that all agents are alive and responding.

*specjbb.heartbeat.period=10000*: How often (in milliseconds) Controller sends heartbeat message to an Agent checking if it is alive

*specjbb.heartbeat.threshold=100000*: How much time (in milliseconds) to wait for heartbeat response from an Agent.

Note: If a system is exhibiting long pauses resulting in heartbeat failures and/or submit errors, user should investigate the cause(s) of longer pauses as well as try increasing the heartbeat threshold.

## 17.Invalid run messages

There are several validation checks in this benchmark. The validation criteria are documented in the HTML report. During the benchmark testing, the top causes of invalid runs were messages stating "submit error: an expected error occurred". There can be many causes that result in "submit-error" invalid messages. Some of the possible causes are listed below:

- Agents not responding or taking a long time to respond. In this case, delay longer than heartbeat failure threshold results in shutdown of an agent down which could appear as submit-error since there will be no response for requests from this agent.
- Some very long GC pauses result in agents not responding for long durations. This results in Grizzly NIO timing out and a no response causes submit-error.

Review of Controller.log and Controller.out could identify more exact details about the real cause. Often tuning GC pauses, large enough heap, response time sensitive GC policies, and sufficient number of Fork/Join workers should help in resolving the issue.

## 18.Performance Tuning

All of the techniques and concepts that are involved in this area reach beyond the scope of this document. Therefore, only a few elementary concepts that are related to the SPECjbb2013 benchmark will be briefly discussed in this section. This document is not to be considered, and does not claim to be, a complete reference for SPECjbb2013 performance tuning.

Please note that these techniques may or may not necessarily be beneficial for system performance in a production environment.

### 18.1. JVM Software

This is a critical component affecting the performance of the system and the workload. A good starting point is to review published results of this and other Java server benchmarks. For more information concerning these options please consult your JVM vendor.

### 18.2. Memory

The SPECjbb2013 workload uses at least a heap size of 2GB (suggested 16GB) for each Backend, 1GB (suggested 2GB)for each TxI and 1GB (suggested 2GB) for Controller. Heap size is set via the -ms/-mx (or -Xms/-Xmx) command line arguments for the java executable in many JVMs.

### 18.3. Threads

The number of threads for Fork/Join workers for each Backend is recommended to set to twice the available logical processor threads. This can be set via “specjbb.forkjoin.workers=” a user settable property. .

### 18.4. JVM Locking

With very complex business logic and several tiers of Fork/Join workers working across several entities like Supermarkets, Suppliers and Headquarters, the synchronization of application methods, or the JVM's internal use of locking, may prevent the JVM from using all available CPU cycles. If there is a bottleneck without consuming 100% of the CPU, lock profiling with JVM or operating system tools may be helpful.

### 18.5. Network

Although SPECjbb2013 uses very little network I/O, it is a good idea to make sure that the network fabric being used for the test bed has a very high availability for use by SPECjbb2013. The best practice is to setup the test bed on its own isolated broadcast domain. This will minimize the possibility for network anomalies that may interfere with SPECjbb2013’s network specific components. Also, there is inter-process communication among multiple Groups. When using greater than 16 groups inside single OS image, the number of socket connections settings as well as localhost network traffic optimizations may provide significant boost in performance.

### 18.6. Disk I/O

SPECjbb2013 does not write to disk as part of the measured workload. Classes are read from disk but that should be a negligible part of the workload. If the disk I/O is found to be higher than it should be, then there may be another process accessing the disk during the benchmark run.

### 18.7. Example Oracle JDK tuning flags for a run on a 16 core system:

#### 18.7.1. Single Backend distributed

java **-Dspecjbb.forkjoin.workers=16** –Xms2g –Xmx2g -jar SPECjbb2013.jar –m distcontroller

java –Xms2g –Xmx2g -jar SPECjbb2013.jar -m txinjector -G G1 -J JVM1

java -Xms20g -Xmx20g –Xmn14g -XX:-UseBiasedLocking -XX:+UseParallelOldGC -jar SPECjbb2013.jar -m backend -G G1 -J JVM2

Note: It is suggested that if CPU utilization is <90%, - **Dspecjbb.forkjoin.workers=** could be set 2X that of available processor threads for better performance.

#### 18.7.2. Two Backend (2 Groups) distributed with each Backend run on a 8 cores

java -Dspecjbb.group.count=2 -Dspecjbb.forkjoin.workers=8 –Xms2g –Xmx2g -jar SPECjbb2013.jar -m distcontroller

java –Xms2g –Xmx2g -jar SPECjbb2013.jar -m txinjector -G G1 -J JVM1

java –Xms2g –Xmx2g -jar SPECjbb2013.jar -m txinjector -G G2 -J JVM1

java -Xms10g -Xmx10g –Xmn6g -XX:-UseBiasedLocking -XX:+UseParallelOldGC -jar SPECjbb2013.jar -m backend -G G1 -J JVM2

java -Xms10g -Xmx10g –Xmn6g -XX:-UseBiasedLocking -XX:+UseParallelOldGC -jar SPECjbb2013.jar -m backend -G G2 -J JVM2

Note: It is suggested that if CPU utilization is <90%, - **Dspecjbb.forkjoin.workers=** could be set 2X that of available processor threads for better performance. Also when running >1 groups, affinity of groups to respective NUMA nodes will deliver the most consistent performance.

As with most aspects of the benchmark implementation, there are restrictions concerning the type of performance tuning that is allowed for publishable results. Please see the SPECjbb2013 Run and Reporting Rules for more details.

## 19.Advance level report

SPECjbb2013 reporter when invoked with "-l <0/1/2/3>" produces higher level of report where 'level 0' is the minimum level if detail and 'level 3' is most detailed. Default HTML report is 'level 0'. Refer to earlier section on manual invocation of reporter for details.

## 20. Advanced options and Research

There are many properties that the user cannot change for compliant runs, however they may be useful for testing and research purposes. Some examples are listed below:

- Manual setting of HBIR: Instead of "Search HBIR" phase finding HBIR, for many testing and analysis situations, setting fixed value of HBIR manually will be very useful.
- Increasing steady time of each RT step level: Default steady state time for each RT step level is ~60 sec. A user may want to run each RT step much longer and can set it using a property.
- Preset IR: Some tests may require stressing the system with a fixed load for very long time. Using available benchmark properties a fixed IR running for a given time could be set.
- Running with much larger or smaller data footprint for various entities like Supermarket size, number of users, receipts stored etc.

There are more than 300 hundred properties that could be set to configure and run the benchmark differently than compliant configuration. For such options and details, refer to SPECjbb2013 Advanced Options and Research document at the benchmark website. Known issues.

For known issues, refer to SPECjbb2013 Known issues (html) document. For latest updates, refer to SPECjbb2013 Known Issues.HTML at the benchmark site.

## 21.Submitting results

Upon completion of a compliant run, the results may be submitted to SPEC. Please see the SPECjbb2013 Run and Reporting Rules for details.

## 22.Disclaimer

Product and service names mentioned herein may be the trademarks of their respective owners.

## 23.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.

## 24.Copyright Notice

## 25.Index

