Oracle Complex Event Processing Performance

An Oracle White Paper
Updated November 2008

ORACLE

Oracle Complex Event Processing Performance

INTRODUCTION

Oracle Complex Event Processing is a lightweight, Java-based (non–Java 2 Platform, Enterprise Edition) application server designed specifically to support event-driven applications. These applications, including those in financial services and other markets, are frequently characterized by the need to provide low and deterministic latencies while handling extremely high rates of streaming input data. This presents performance challenges that are quite different from those faced by more-traditional application servers, which tend to focus on obtaining the highest-possible throughput for transactional workloads.

The benchmark study described in this white paper includes a use case that is very typical of financial front-office applications in capital markets to demonstrate Oracle Complex Event Processing's ability to provide low latency at very high data rates. Implementing a signal generation scenario in which the application is monitoring multiple incoming streams of market data, the benchmark application watches for certain conditions that will then trigger some action.

In the benchmark study, Oracle Complex Event Processing was able to sustain an event injection rate of up to 1 million events per second while maintaining low average and peak latencies. At this injection rate, the average event latency for the full processing path within the server was 67.3 microseconds, with 99.4 percent of the events processed in less than 200 microseconds and 99.99 percent processed in less than 5 milliseconds.

The remainder of this white paper includes a discussion of the product features that enable this level of performance and a detailed description of the benchmark and its results.

ORACLE COMPLEX EVENT PROCESSING ARCHITECTURE

Oracle Complex Event Processing provides not only a CEP and EPL, but also an overall platform that enables queries written in EPL to be tightly integrated with custom Java code written to a POJO/Spring programming model. Performance was not sacrificed by providing this rich development platform.

Oracle Complex Event Processing has taken a unique approach among the products that are targeting event processing, by providing not only a complex event processing (CEP) engine and event processing language (EPL), but also an overall platform that enables queries written in EPL to be tightly integrated with custom Java code written to a Plain Old Java Object (POJO)/Spring programming model. Oracle Complex Event Processing is truly an application server that provides container services and enables applications to be deployed and managed. An

important design goal for the solution was that performance not be sacrificed as a consequence of providing this rich development platform.

The Oracle Complex Event Processing architecture is composed of three main layers: runtime, CEP processor, and development environment.

The Oracle Complex Event Processing architecture is composed of three main layers: runtime, CEP processor, and development environment. Figure 1 shows the high-level architecture of the platform. At the lowest level is the Java runtime consisting of the Java virtual machine (JVM) and core Java classes. Above this is Oracle Complex Event Processing, which is implemented using standard Java Platform, Standard Edition 5 APIs and is certified on Oracle JRockit and Oracle JRockit Real Time. The lowest latencies and highest levels of determinism are obtained when using the Oracle JRockit Real Time runtime environment, which is based on Oracle JRockit with extensions for deterministic garbage collection.

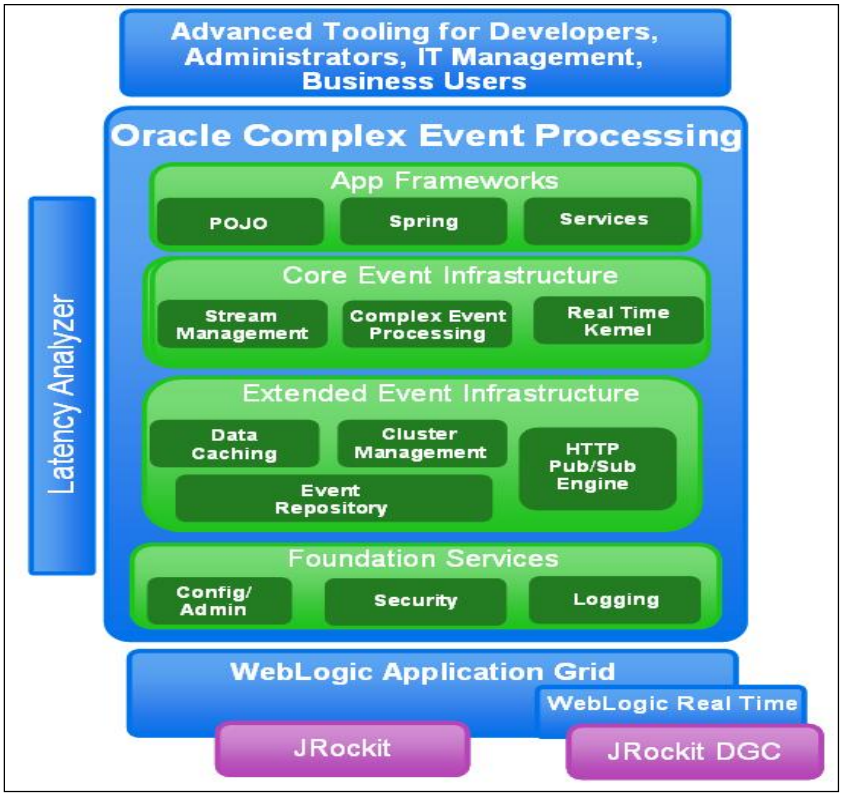

Figure 1: This figure illustrates Oracle Complex Event Processing's high-level platform architecture.

Oracle Complex Event Processing is a Java container implemented with a lightweight, modular architecture based on the Open Services Gateway initiative framework.

Oracle Complex Event Processing is a Java container implemented with a lightweight, modular architecture based on the Open Services Gateway initiative framework. The container services include logging, security, and management as well as services more specific to event-driven applications, including stream management and a CEP engine. Oracle Complex Event Processing's core also includes a real-time kernel providing thread scheduling and synchronization support tuned for low latency and determinism. Applications consisting of a mix of POJOs and EPL queries can be dynamically deployed to the server and access the various services via Spring-based dependency injection. Rounding out the overall architecture is an integrated monitoring framework for the precise monitoring of

event latencies and a development environment based on the Eclipse integrated development environment (IDE).

There is no requirement to include a stream component in the event processing path, but it can be very useful in increasing concurrency for applications that might otherwise have limited concurrency.

Figure 2 illustrates the typical data flow through an Oracle Complex Event Processing application. On the inbound (left) side are event data streams from one or more event sources. The incoming data is received, unmarshalled, and converted into an internal event representation within an adapter module. The internal event representation can be an application-defined Java object or a Java map. As the adapter creates event objects, it sends them downstream to any components that are registered to "listen" on the adapter. In Figure 2, the listening components are so-called stream components: essentially a queue with an associated thread pool that enables the upstream and downstream components to operate asynchronously from each other. There is no requirement to include a stream component in the event processing path, but it can be very useful in increasing concurrency for applications that might otherwise have limited concurrency, for example, a data feed coming in over a single connection.

The next component in the Figure 2 data flow is the processor component. A processor represents an instance of the CEP engine and hosts a set of queries written in EPL. Such queries support filtering, aggregation, pattern matching, and joining of event streams. The output of the configured EPL queries is sent to any downstream listeners. In this example, a POJO is configured to listen to the processor output. The POJO can perform additional processing on the events output from the queries and trigger actions or send the output data to external systems via standard or proprietary messaging protocols.

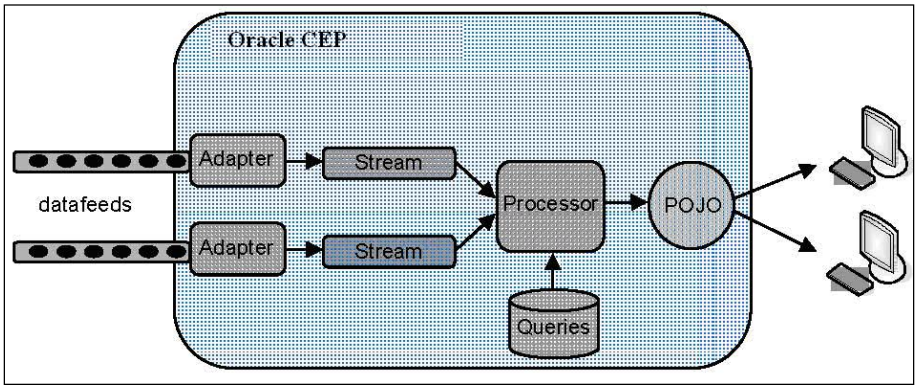

Figure 2: This illustrates the typical data flow through an Oracle Complex Event Processing application.

The collection of interconnected adapter, stream, processor, and POJO components is collectively referred to as the event processing network (EPN). Although the example in Figure 2 shows a common topology, arbitrary EPN graphs can be wired together consisting of any number of components of each type in any order.

REAL-TIME ORACLE COMPLEX EVENT PROCESSING KERNEL

Meeting the stringent latency requirements of typical event-driven applications requires specialized support in the areas of thread scheduling, synchronization, and input/output (I/O). Techniques used by the Oracle Complex Event Processing kernel to support low-latency processing include the following:

- Thread scheduling attempts to minimize blocking and context switching in the latency critical path. Whenever possible, a given event will be carried through its full execution path on the same thread with no context switch. This approach is optimal for latency and also ensures in-order processing of events for applications that require this. However, in some cases, handoff of an event between threads might be desirable; for example, an application might want to handle data from a single incoming network connection concurrently in multiple threads. The kernel provides flexible thread pooling and handoff mechanisms that enable concurrency to be introduced wherever it is desired in the processing path, with minimal impact on overall latency.
- Synchronization strategies minimize lock contention that could otherwise be a major contributor to latency.
- Careful management of memory including object reuse, use of memory efficient data structures, and optimized management of retain windows within the CEP engine support low-latency processing. The memory optimizations benefit latency by reducing both the allocation rate and the degree of heap fragmentation—both of which help the garbage collector achieve minimal pause times.
- A pluggable adapter framework enables high-performance adapters to be created for a variety of network protocols, with support for multiple threading and I/O handler dispatch policies.
- Use of Oracle JRockit Real Time (described in the following section) offers the lowest latencies and highest level of determinism.

ORACLE JROCKIT REAL TIME

Although Oracle Complex Event Processing is certified on the standard version of Oracle JRockit Real Time, the lowest latencies and highest level of determinism are obtained when running on the Oracle JRockit Real Time solution. Oracle JRockit Real Time consists of the Oracle JRockit JVM with enhancements for low latency and deterministic garbage collection.

Although Oracle Complex Event Processing is certified on the standard version of the Oracle JRockit JVM, the lowest latencies and highest level of determinism are obtained when running on the Oracle JRockit Real Time solution. Oracle JRockit Real Time consists of the Oracle JRockit JVM with enhancements for low latency and deterministic garbage collection. Typical garbage collection algorithms stop all the threads of an application when they perform a collection. The resulting garbage collection pause time can be very long (several seconds or longer) in some environments and is a major contributor to latency spikes and jitter. Oracle JRockit Real Time's deterministic garbage collector uses a different approach designed to make garbage collection pause times both shorter and more predictable. The deterministic collector handles much of the collection while the application is running, and pauses only briefly during critical phases of the garbage collection. In

addition, the Oracle JRockit Real Time collector will monitor the duration of individual pauses to ensure that the amount of time spent in a given garbage collection pause doesn't exceed a user-specified pause target. For example, with a 10-millisecond user-specified pause target, the deterministic collector would limit the duration of individual garbage collection pauses to no more than 10 milliseconds, providing a high degree of predictability compared to traditional garbage collection algorithms.

In addition to the deterministic garbage collection feature, the Oracle JRockit Real Time solution also includes a latency analyzer tool, integrated with the Oracle JRockit JVM runtime analyzer tool.

In addition to the deterministic garbage collection feature, the Oracle JRockit Real Time solution also includes a latency analyzer tool, integrated with the Oracle JRockit JVM runtime analyzer tool. The latency analyzer tool is a unique performance analysis tool that identifies and analyzes sources of latency within a Java application. Whereas typical profiling tools focus only on where central processing unit time is spent while the application is running, the latency analyzer tool provides detailed information about where, when, and how long the various threads of the application block or wait. The latency analyzer tool can identify the cause and duration of a thread wait, assessing whether the cause is due to garbage collection, I/O, synchronization, or an explicit sleep or wait requested by the application. The ability to locate and analyze these sources of latency is indispensable in tuning a latency-sensitive application; therefore, the latency analyzer tool was used extensively in tuning the benchmark described in this white paper.

BENCHMARK APPLICATION

The application used for this benchmark study implements a signal generation scenario in which the application is monitoring multiple incoming streams of market data, watching for the occurrence of certain conditions that will then trigger some action. This is a very common scenario in front-office trading environments. Figure 3 shows the overall structure of the benchmark test.

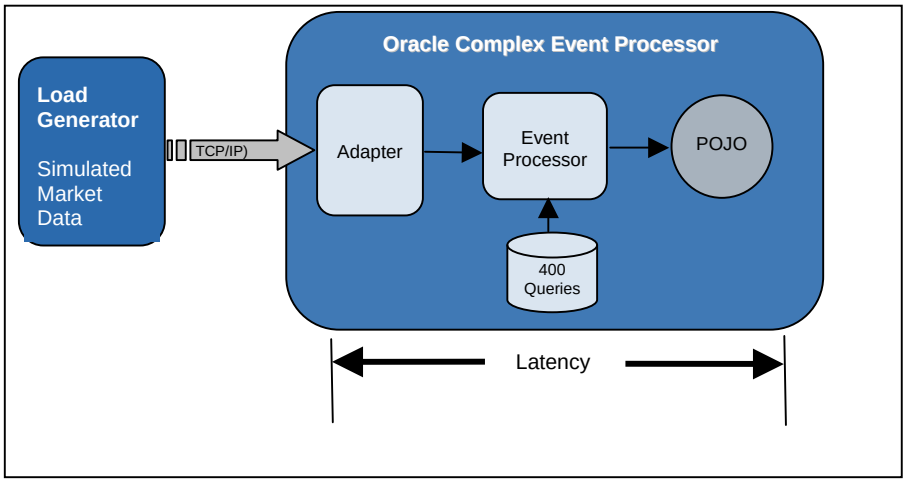

Figure 3: In the benchmark study, simulated data is generated and sent to the event processor over a TCP/IP connection. The event processor monitors the data and searches for two specified conditions.

The incoming data is generated by a load generator, which creates simulated stock market data and sends it to the server over one or more Transmission Control Protocol connections at a configured, metered rate. The format of the data on the wire is specific to the implementation of the load generator and adapter and is designed for compactness. Within the event processor, the adapter reads the incoming data from the socket, unmarshalls it, creates an event instance (a Java object conforming to certain conventions) for each incoming stock tick, and forwards the events to the event processor.

In the benchmark study, the event processor is configured to monitor the incoming data for any one of 200 different stock, or ticker, symbols. Each of these stock symbols is monitored for two price-related conditions.

The event processor is configured to monitor the incoming data for any one of 200 different stock, or ticker, symbols. Each of these stock symbols is monitored for the following two conditions:

- The stock price increases or decreases by more than 2 percent from the immediately previous price.
- The stock price has three or more consecutive upticks without an intervening downtick.

The EPL syntax that is used to implement these rules for the stock symbol WSC is shown below:

```
SELECT symbol, lastPrice, perc(lastPrice),
clientTimestamp, timestamp
      FROM (select * from StockTick where symbol='WSC')
      RETAIN 2 EVENTS
      HAVING PERC(lastPrice) > 2.0 OR PERC(lastPrice) <
      -2.0

SELECT symbol, lastPrice, trend(lastPrice),
clientTimestamp, timestamp
      FROM (select * from StockTick where symbol='WSC')
      RETAIN 3 EVENTS
      HAVING TREND(lastPrice) > 2
```

These two queries are replicated for each of the 200 symbols being monitored, resulting in a total of 400 queries that the event processor must execute against each incoming event. When an incoming event matches one of the rules, an output event is generated with the fields specified in the SELECT clause and sent to any downstream listeners. In this case, the downstream listener is a Java POJO, which computes aggregate statistics and latency data for the benchmark based on the output events it receives.

Latency data for the benchmark is computed based on time stamps taken in the adapter and POJO. The adapter takes the initial time stamp after reading the data from the socket and prior to unmarshalling. This initial time stamp is inserted into each event created by the adapter, passed through the event processor, and inserted

into any output events generated by a matching rule. When the POJO receives an output event, it takes an end time stamp and subtracts the time stamp generated by the adapter to compute the processing latency for that event. These latencies are aggregated to produce overall latency data for the duration of the benchmark run.

BENCHMARK CONFIGURATION AND METHODOLOGY

Load Injection

The load generator can be configured to specify the number of connections it should open to the CEP processor server and the rate at which it should send data over each connection. We will refer to the aggregate send rate across all connections as the aggregate injection rate. For this benchmark, the data sent by the load generator for each event consists of a stock symbol, simulated price, and time stamp data. The average size of the data on the wire is 20 bytes per event not including Transmission Control Protocol/Internet Protocol header overhead. The stock symbols are generated by repeatedly cycling through a list of 1,470 distinct stock symbols. If the load generator is configured to open multiple connections to the server, the symbol list is partitioned evenly across the set of connections. The price data is generated dynamically based on a geometric Brownian motion algorithm, and the price for a given symbol is updated each time the symbol is sent.

Oracle Complex Event Processing Configuration

The event processing network configuration within the CEP processor server consists of a single adapter instance, single processor instance, and single POJO, as described in the previous section.

The adapter is configured to use a blocking thread-per-connection model for reading the incoming data and dispatching the events within the server. The adapter feeds all the injected input events to the processor, which is configured with a total of 400 queries (200 distinct symbols with two rules per symbol), as previously discussed. Each of the configured queries is run against each input event, and for each match an output event is sent downstream to the POJO.

Hardware and Software Stack

The hardware consists of one machine for the CEP processor server and one machine for the load generator, connected by a gigabit Ethernet network. The server and load generator machines each have an identical hardware configuration and identical software stack. The exact technical specifications for the components in the benchmark study are supplied in Table 1.

Component	Technical Specifications
Hardware platform	• Quad-Core Intel Xeon 7300–based server • 4 Quad-Core Intel X7350 processors at 2.93GHz (16 cores total) • 8MB L2 cache per processor, shared across the 4 cores • 32GB RAM
Operating system	• Red Hat Enterprise Linux 5.0, 32 bit, kernel 2.6.18-8
JVM	• Oracle JRockit Real Time 2.0 • 1GB heap size, deterministic garbage collection–enabled
Oracle Complex Event Processing server	• Oracle Complex Event Processing 2.0 (with support patch ID XQWK)

Table 1: This table identifies the technical specifications for each component of the benchmark study.

Methodology

The benchmark data was collected as follows:

- An initial 15-minute warm-up run was done with the load generator opening 10 connections to the server and sending data at a rate of 100,000 events per second per connection.
- The warm-up was followed by a series of 10 runs scaling the number of connections from 1 to 10, with the load generator sending 100,000 events per second per connection in all cases (maximum injection rate of 1 million events per second). The duration of each run was 10 minutes.
- An additional series of 10 runs was done holding the number of connections fixed at 10 and scaling the injection rate per connection from 10,000 to 100,000 events (maximum injection rate of 1 million events per second). The duration of each run was 10 minutes.
- The injection rate, output event rate, average latency, absolute maximum latency, and latency distributions were collected for all runs.

BENCHMARK RESULTS

Table 2 and Figures 4 and 5 show the results scaling from 1 to 10 connections at 100,000 events per second per connection.

Connections	Injection Rate Per Connection (events/sec)	Total Injection Rate (events/sec)	Output Event (Match) Rate	Average Latency (microsecs)	99.99% Latency (millisecs)	Absolute Max Latency (millisecs)
1	100,000	100,000	3911	42.6	0.2	8.77
2	100,000	200,000	7811	44	2.1	12.77
3	100,000	300,000	11686	47.2	2.2	12.93
4	100,000	400,000	15595	49.3	2.4	13.45
5	100,000	500,000	19466	51.5	2.5	15.73
6	100,000	600,000	23351	53.6	2.6	16.64
7	100,000	700,000	27234	55.5	2.6	18.6
8	100,000	800,000	31235	58.3	3.1	19.5
9	100,000	900,000	35080	62	3.7	19.21
10	100,000	1,000,000	38890	67.3	4.3	21.52

Table 2: Scaling from 1 to 10 connections at 100,000 events/second per connection.

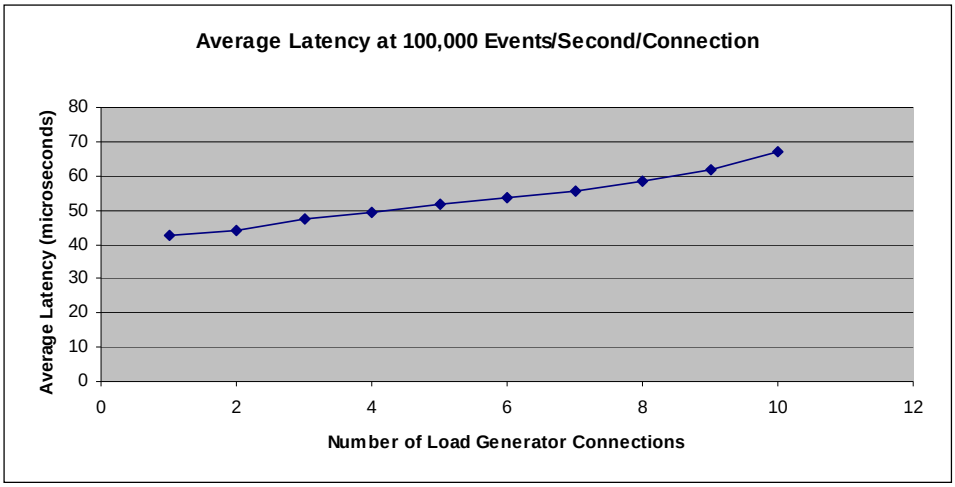

Figure 4: Average latency scaling from 1 to 10 connections (100,000 to 1 million events/second).

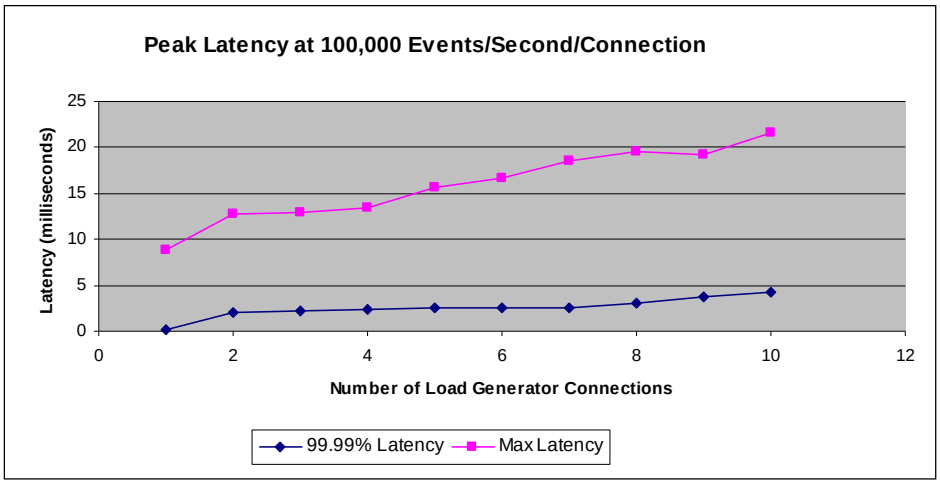

Figure 5: Peak latency scaling from 1 to 10 connections (100,000 to 1 million events/second).

As discussed earlier, the latency values are collected only for those events that are forwarded to the POJO as a result of a match. The values represent the latency from an initial time stamp in the adapter (before the unmarshalling is done and before the internal event object is created) and a time stamp when the event is received by the POJO.

As Table 3 shows, the output event rate was a fixed percentage—3.9 percent—of the injection rate as the load increased. There was a gradual increase in average and maximum latencies as the number of connections and overall injection rate increased. The 99.99 percentile latencies remained fairly flat—between 2.1 and 2.6 milliseconds—with increasing load from 200,000 through 700,000 events per second, and then increased slightly as the injection rate approached 1 million events per second. At the maximum benchmark load of 1 million events per second the average and 99.99 percentile latencies are still quite low and the even the absolute maximum has degraded only slightly with the increased load.

Table 3 and Figures 6 and 7 show the results when holding the number of connections fixed at 10 and scaling the injection rate.

Connections	Injection Rate Per Connection (events/sec)	Total Injection Rate (events/sec)	Output Event (Match) Rate	Average Latency (microsecs)	99.99% Latency (millisecs)	Absolute Max Latency (millisecs)
10	10,000	100,000	3893	45.2	0.4	13.97
10	20,000	200,000	7793	45.7	0.8	13.26
10	30,000	300,000	11676	47.2	1.1	13.81
10	40,000	400,000	15564	49.3	1.3	16.12
10	50,000	500,000	19460	51.6	1.7	19.21
10	60,000	600,000	23348	54.2	2.1	20.96
10	70,000	700,000	27239	56.6	2.6	20.91
10	80,000	800,000	31123	59.4	3.4	20.31
10	90,000	900,000	35001	62.9	3.9	21.83
10	100,000	1,000,000	38890	67.3	4.3	21.52

Table 3: Scaling from 100,000 to 1 million events/second with 10 connections.

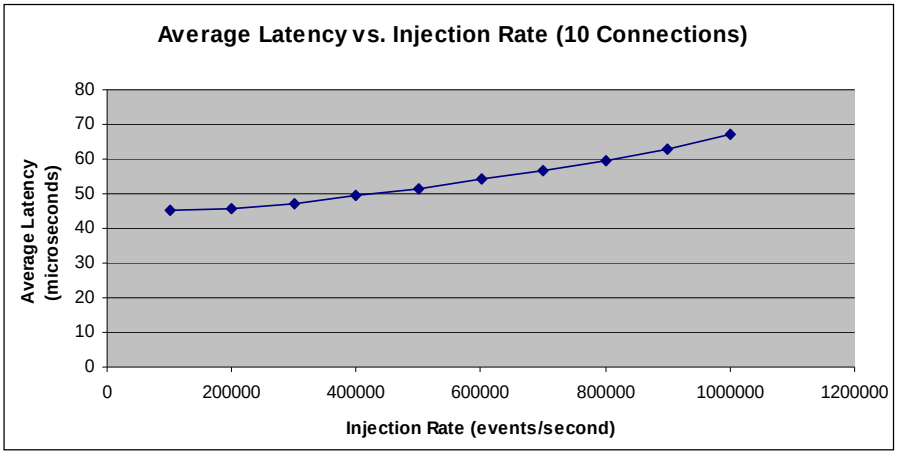

Figure 6: Average latency scaling from 100,000 to 1 million events/second with 10 connections.

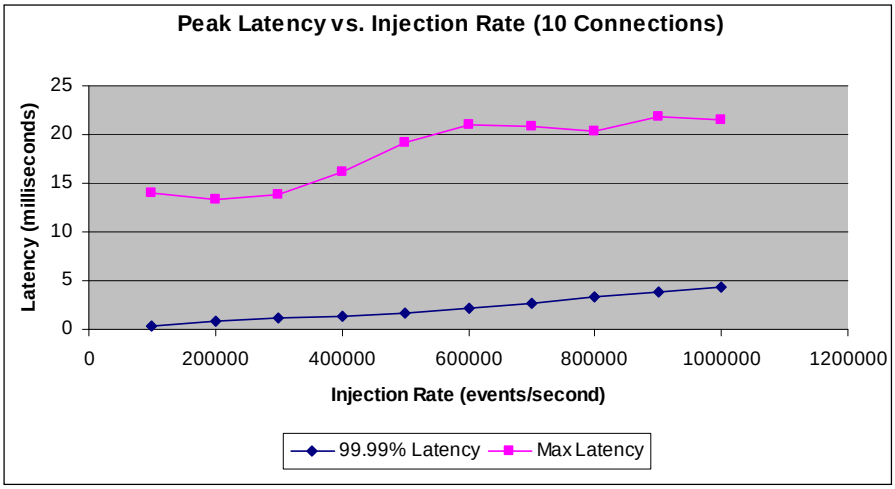

Figure 7: Peak latency scaling from 100,000 to 1 million events/second with 10 connections.

The effect of increasing load on latency when scaling with a fixed number of connections is very similar to the results shown when the load's scale was increased by escalating the number of connections. This similarity in results suggests that the performance of the system at a given input load is mostly independent of the number of connections used to inject the data. A difference is visible in the maximum latency curves in the range of 400,000 to 700,000 events per second, suggesting that in this range maximum latencies could be reduced at a given load by using a smaller number of connections.

The histograms in Figures 8 and 9 show the latency distribution at an injection rate of 1 million events per second. The first histogram uses a linear scale on the *y* axis and consolidates the event count for the range >200 microseconds into a single (barely visible) bar. The second histogram displays the same data using a log scale on the *y* axis to provide additional detail in the >200 microsecond latency range. The histograms illustrate how strongly skewed the distribution is toward the lower end of the latency range, with 86.3 percent of the latency values below 100 microseconds and 99.4 percent of the latency values below 200 microseconds, at an injection rate of 1 million events per second.

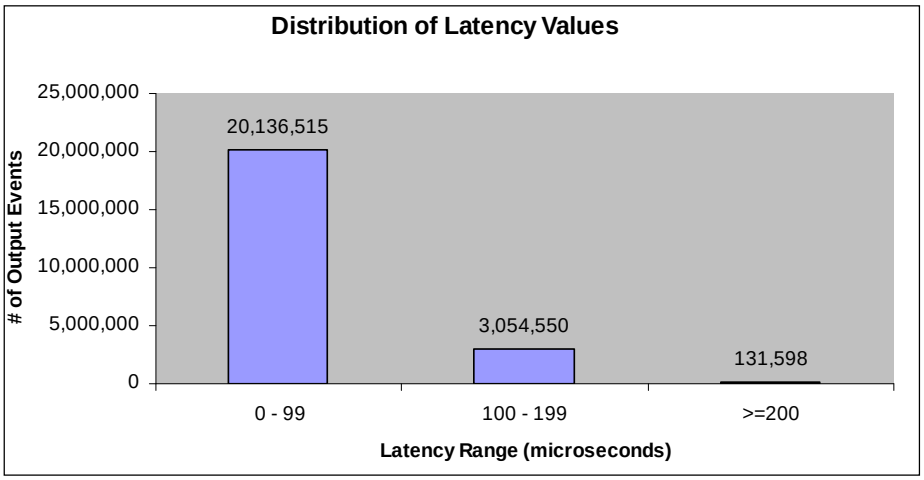

Figure 8: Distribution of output latency values at an injection rate of 1 million events/second using a linear scale for the range >200 microseconds.

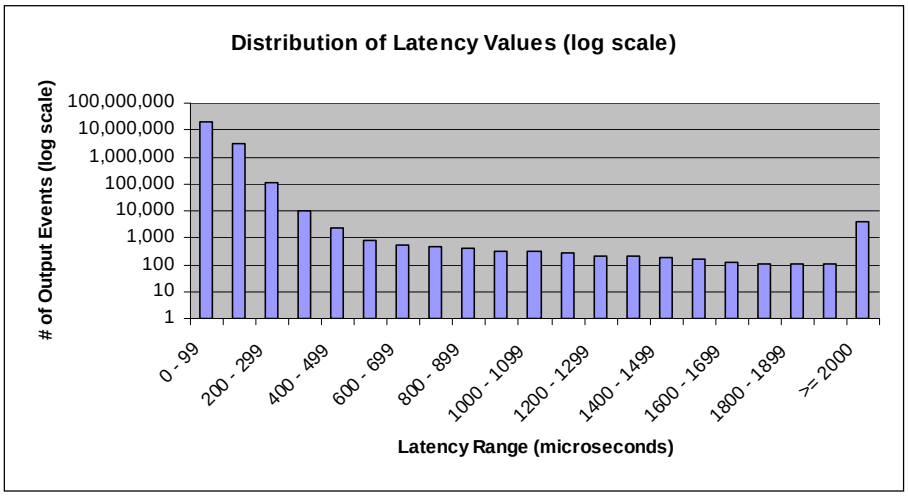

Figure 9: Distribution of output latency values over 10-minute run at 1 million events/second injection rate using a logarithmic scale.

Finally, Figure 10 shows the garbage collection pause times for the duration of the benchmark run at 1 million events per second. There were a total of 477 garbage collections over the course of the 10-minute run. The maximum garbage collection pause during the run was 17 milliseconds, with 97 percent of the pauses at or below 15 milliseconds. The ability of Oracle JRockit Real Time to maintain these short and predictable garbage collection pauses under load was a major factor in limiting the peak application latencies.

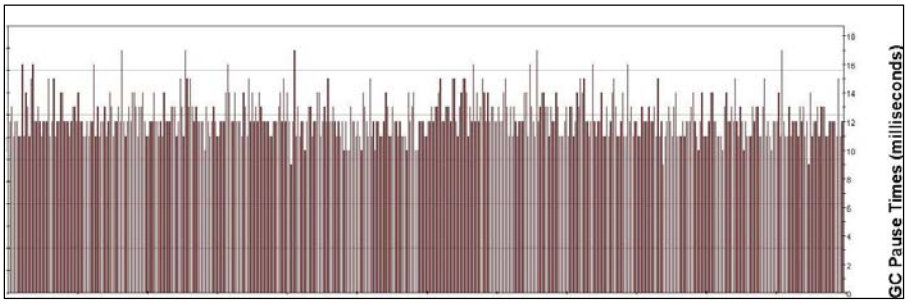

Figure 10: Garbage collection pause times over 10-minute run at an injection rate of 1 million events/second injection.

CONCLUSION

This white paper has reviewed the overall architecture of Oracle Complex Event Processing and some of the specific features and design characteristics that enable it to provide high performance for event-driven applications, including

- Container services such as logging, security, and management, as well as services more specific to event-driven applications, including stream management and a CEP engine
- Specialized support in the areas of thread scheduling, synchronization, and I/O through the Oracle Complex Event Processing kernel
- Oracle JRockit Real Time consisting of the Oracle JRockit JVM, with enhancements for low latency and deterministic garbage collection
- A latency analyzer tool integrated with the Oracle JRockit runtime analyzer tool that identifies and analyzes sources of latency within a Java application
- An integrated monitoring framework for precise monitoring of event latencies and a development environment based on the Eclipse IDE

The results from the benchmark study demonstrate very clearly Oracle Complex Event Processing's ability to achieve low and predictable latency under very high loads.

The performance characteristics of Oracle Complex Event Processing were studied using a very common event processing use case. The results demonstrate very clearly Oracle Complex Event Processing's ability to achieve low and predictable latency under very high loads.

Oracle Complex Event Processing Performance
Updated November 2008

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com