MERCURY™

OPTIMIZING THE BUSINESS VALUE OF INFORMATION TECHNOLOGY

Performance Impact Analysis of Mercury Diagnostics Probe for J2EE

This report examines the impact that the Mercury Diagnostics Probe for J2EE has on the performance of the applications that it monitors.

Table of Contents

1. Abstract

The Mercury Diagnostics Probe for J2EE (Probe) gathers the metrics that are used to analyze the performance of J2EE enterprise applications. The Probe is installed on the application server instances to collect the performance measurements for servlets, JSPs, EJBs, JNDI, JDBC, JMS, and Struts method calls, and user-defined custom classes. These measurements are stored and used by Mercury Diagnostics for analysis and reporting on the performance of these applications.

As the Probe is installed on the application server, it is important to understand and quantify the performance impact of the Probe. This paper reports on the results of a study performed to assess how much impact the Diagnostics Probe has on the performance of the applications that it is monitoring.

The actual impact that the Probe has varies depending on the application that is being monitored and the load that is being applied to the application. For the study, in order to understand the performance impact of the Probe on a typical application, we applied varying load levels to the industry-standard J2EE benchmark, SPECjAppServer2004™ [1].

The study consisted of sets of long duration SPECjAppServer2004 runs, with and without the Probe, and at varying load levels. For each run, CPU usage, HTTP response latency, transaction throughput, and network bandwidth were measured. The results of each run were analyzed, and the statistical confidence for each metric was calculated.

As a result of the study, when the Probe was running on a system below maximum capacity, the performance impact showed the following results:

- Transaction latency increased by 5%.
- Transaction throughput (rate) decreased by 3%
- CPU utilization increased by 7%
- Network utilization increased by 1%

At loads beyond system capacity, the Probe had the following effects:

- Transaction latency increased by 20%.
- Transaction throughput (rate) decreased by 7%
- CPU utilization increased by 7%
- Network utilization increased by 6%

Enabling data sampling on the Probe reduced the overhead to one-fourth the levels.

---

[1] SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 results or findings in this publication have not been reviewed or approved by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2004 is located at http://www.spec.org/osg/jAppServer2004.

For a system running within its capacity, the latency impact of the Mercury Diagnostics Probe for J2EE is generally minor, rarely more than 1%-2%, and is almost certainly not noticeable to end users.

The methodology used in the study, along with performance results, is discussed in subsequent sections of this paper.

2. Methodology

2.1. Application Under Test

The overhead of the Probe as it monitors an application is highly dependent upon the actual application being tested. The SPECjAppServer2004 application was selected as the target application that the Probe would monitor in order to ensure that the test results would be representative of the overhead in a real-world enterprise. The following information from http://www.specbench.org/jAppServer2004 describes the SPECjAppServer2004 application:

SPECjAppServer2004 (Java Application Server) *is a multi-tier benchmark for measuring the performance of Java 2 Enterprise Edition (J2EE) technology-based application servers. SPECjAppServer2004 is an end-to-end application which exercises all major J2EE technologies implemented by compliant application servers as follows:*

- *The web container, including servlets and JSPs*
- *The EJB container*
- *EJB 2.0 Container Managed Persistence*
- *JMS and Message Driven Beans*
- *Transaction management*
- *Database connectivity*

Moreover, SPECjAppServer2004 also heavily exercises all parts of the underlying infrastructure that make up the application environment, including hardware, JVM software, database software, JDBC drivers, and the system network.

2.2. System Specifications

The test environment consisted of a set of rack-mounted Windows 2000 servers on their dedicated (built-in) 1 gigabit network.

The host for the **system under-test** (SUT) was a dual-processor 1.4 GHZ Pentium III with 1GB RAM. The SPECjAppServer2004 application was configured to use a 700 MB JVM heap. [2]

The **load driver** was a single-processor 1.4 GHZ Pentium III with 1GB RAM. This machine was also the host for the SPECjAppServer2004 emulator servlet.

In order to test high CPU utilizations on the application server, the **database** was distributed across three systems: two single-processor 1.4 GHZ Pentium IIIs and one dual-processor 1.4 GHZ Pentium III. All three systems had 1GB RAM and were using XA JDBC drivers to handle the cross-machine transactions.

The remaining Mercury Diagnostics components were also run on 1.4 GHZ Pentium IIIs.

2.3. Measurement Tools

The system performance was measured using the SPECjAppServer2004 latency and throughput results. System metrics, such as CPU, network, and disk utilization were gathered using Mercury's Deep Diagnostics for J2EE and .NET.

2.4. Run Duration

Run duration was selected to comply with the SPECjAppServer run rules[3] that require a 10 minute warm-up period and a 60 minute measurement interval for recording results.

2.5. Statistical Measurement

At least three runs of each type (3 baselines, 3 for each set of Probe settings) were performed. Reported statistics were based on the mean calculated from the measurements for those runs.

Statistical confidences were calculated using a one-tailed Student's t-Test.[4] Additional runs were performed when it was necessary to improve statistical confidences.

Each of the SPECjAppServer runs was audited by the load driver to ensure correctness, as per section 2.13 of the SPECjAppServer run rules.

---

[2] For legal reasons, we are not able to disclose the actual Application Server, JVM or Database used.

[3] http://www.specbench.org/jAppServer2004/docs/RunRules.html

[4] The t-Test is a statistical test comparing the distribution of two means for the purpose of determining whether they are significantly different.

2.6. Types of Overhead

Several types of overhead were measured during the runs. These were:

- **Latency Overhead**: Latency overhead is the additional amount of time taken for HTTP responses after the Probe was installed. By default, the Probe is configured so that it collects more diagnostics data for “slower” operations than for “faster” operations.
- **Network Overhead**: Additional network bandwidth is needed to send diagnostics information from the Probe to the Diagnostics Server in Mediator mode.
- **CPU Overhead**: Additional background CPU on the System Under Test (SUT) is needed by the Probe to process diagnostics data.
- **Synchronization Overhead: The Probe works by adding byte codes into your classes that perform** small amounts of work before and after method invocations. If the application is holding a lock when the byte code is performing its work, then the duration for which the lock is held will be increased slightly. This could potentially lead to additional lock contention within the application.
- **Disk Overhead**: The Probe does not make significant use of the disk. Measurements were taken but are not reported as they all show statistical insignificance.
- **Memory Overhead**: The Probe was designed to use a fixed amount of additional memory. By default, this is about 25 megabytes. However, the amount of memory that the Probe uses can be configured.

3. Overhead Measurements

Over 130 hour-long tests were executed at varying levels of load using the Mercury Diagnostics Probe for J2EE.

Overhead measurements are reported as percentages that are calculated relative to the baseline performance when there is no probe installed. If the operation has a baseline latency overhead of 1 second and shows latency of 1.1 seconds when the Probe is running, an overhead of 10% is reported.

These tests were executed using the Probe’s default configuration, as described below.

Note: In the next section it will be shown that significant overhead reductions can be achieved by changing these parameters:

- **51ms Latency Trim**: Events faster than 51 milliseconds are not recorded. The rationale being that events below 51ms are generally not interesting from a diagnostics perspective. Events include methods, SQL queries, etc.
- **25 Depth Trim**: Up to 25 calls in a call chain (method A calls method B calls method C, and so on) are recorded. This limit is not encountered frequently, and is intended to prevent worst-case performance with large instrumentation plans.

- **100% sampling**: Diagnostics data is collected about every HTTP request processed by the server.
  Note: Reducing sampling reduces probe overhead and granularity of performance information gathered.
- **SQL parsing enabled**: SQL statements are converted to their general form before being placed in the database, enabling aggregation and trending.
  For example,

  "SELECT * FROM Customers WHERE cust_id = 5"

  is converted into

  "SELECT * FROM Customers WHERE cust_id = ?".
- **Standard J2EE instrumentation is used**: This will instrument J2EE methods that could potentially cause problems, such as JNDI lookups, JMS actions, JSP and Servlet calls, Web services, RMI connections, all JDBC calls, and all EJB methods.

3.1. Performance Results

The following figures and tables (1 and 2) show the impact on average latency, transaction throughput, average CPU, and network bandwidth consumption at varying load factors. The load, as defined by SPECjAppServer, is the IR (injection rate) at which business transaction requests are injected into the SUT (system under test).

Note: At low loads, where the load factor ranges from 1 to 8, the performance impact of the Probe is negligible, averaging an increase of within 5% for average latency, a decrease of 3% for transaction rate, and increase of 7% for CPU utilization, and an increase of 1% for network utilization.

However, as the application system approaches its saturation or capacity based on transaction throughput, at load factor of 9 and above, the Probe is no longer able to insulate the SUT from the impacts of data collection, its performance impact averaging an increase of 20% for average latency, a decrease of 7% for transaction rate, an increase of 7% for CPU utilization, and a decrease of 6% for network bandwidth.

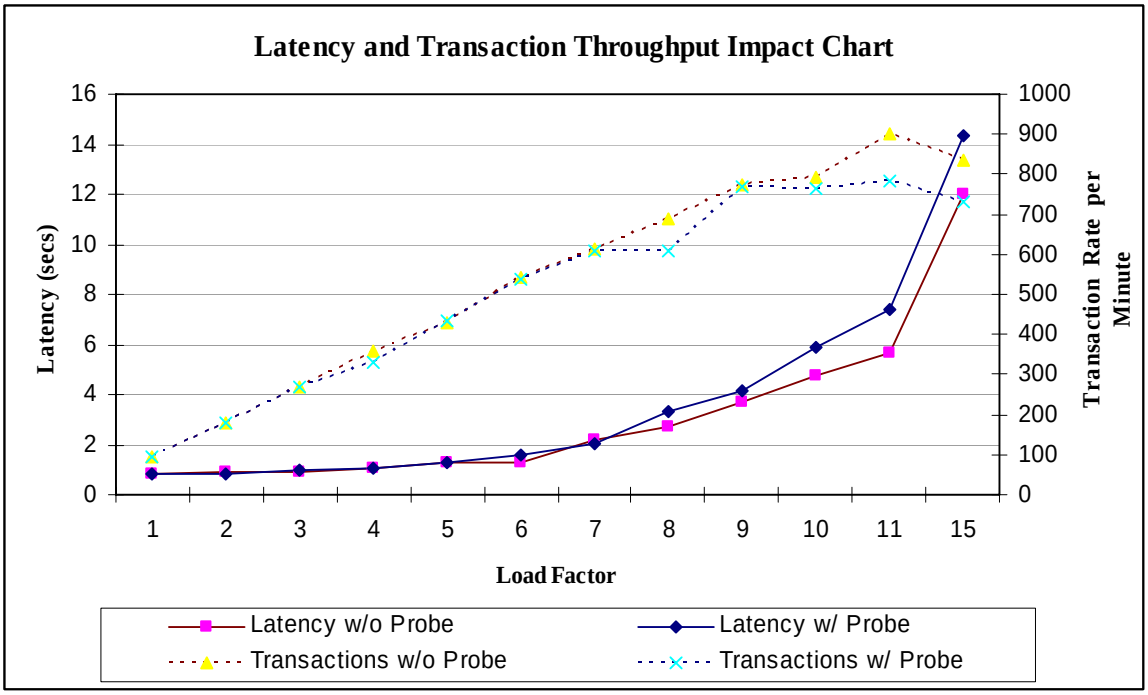

Figure 1: Average Latency and Transaction Rate Impact

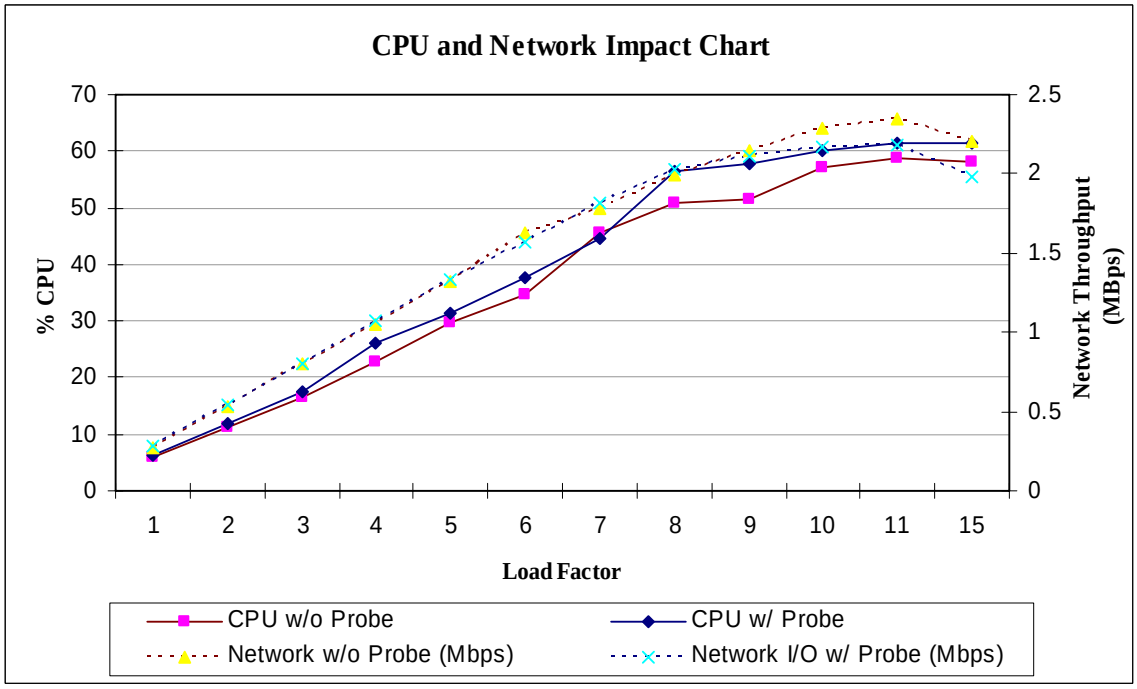

Figure 2: Average CPU and Network Bandwidth Impact

Load Factor	Average Latency (secs)			Average Transaction Rate (per minute)		
	w/o Probe	w/ Probe	Difference	w/o Probe	w/ Probe	Difference
1	0.83	0.82	-1.20%	92.98	91.93	-1%
2	0.89	0.85	-4.49%	177.78	177.5	0%
3	0.91	0.97	6.59%	269.81	269.39	0%
4	1.03	1.05	1.94%	360.1	330.36	-8%
5	1.28	1.29	0.78%	427.56	436.4	2%
6	1.32	1.56	18.18%	540.56	538.84	0%
7	2.17	2.03	-6.45%	613.82	607.72	-1%
8	2.68	3.29	22.76%	687.15	606.63	-12%
9	3.71	4.15	11.86%	773.03	766.57	-1%
10	4.75	5.9	24.21%	793.19	764.48	-4%
11	5.63	7.36	30.73%	900.48	781.52	-13%
15	12	14.31	19.25%	836.84	729.03	-13%

Table 1: Average Latency and Transaction Rate Impact

Load Factor	CPU Utilization (%)			Network Bandwidth Consumption (Bps)		
	w/o Probe	w/ Probe	Difference	w/o Probe	w/ Probe	Difference
1	5.8	6.12	6%	271426.65	277992.72	2%
2	11.17	12.05	8%	533343.62	540592	1%
3	16.48	17.62	7%	799873.62	804216.16	1%
4	22.92	26.19	14%	1046415.97	1072281.53	2%
5	29.76	31.32	5%	1323163.23	1335005.58	1%
6	34.65	37.49	8%	1625105.67	1573099.59	-3%
7	45.63	44.49	-2%	1775713.18	1810445.6	2%
8	50.96	56.54	11%	1991520.22	2032207.34	2%
9	51.45	57.73	12%	2142690.24	2107307.65	-2%
10	57.08	60.01	5%	2283011.05	2173984	-5%
11	58.79	61.27	4%	2348723.18	2184315.93	-7%
15	58.21	61.43	6%	2207738.04	1976842.75	-10%

Table 2: Average CPU and Network Bandwidth Impact

It is interesting to note that for load factors above 9, Probe CPU overhead remains constant at around 5% - the system has saturated another resource and even though high throughput is achieved, there is no additional overhead.

In order to reduce the performance impact at high load levels, the strategy of data sampling may be utilized.

4. Overhead Measurements with 25% Sampling

If the application system typically runs near capacity and you wish to reduce diagnostics overhead, the simplest way to do so is to enable sampling. Sampling is a feature that causes the Probe to record diagnostics data only for a specified percentage of the HTTP calls (as opposed to all of them).

As part of the study, runs were executed with sampling set to 25%. These runs were compared to identical runs when sampling was turned off in order to demonstrate the impact that sampling can have on the diagnostics overhead of the Probe.

In the following tables you can see that the overhead of the Probe is essentially a quarter of what it was when sampling was at 100%.

Load Factor	Average Latency (secs)				
	w/o Probe	w/ Probe @ 25% sampling	Difference @ 25% sampling	w/ Probe @ 100% sampling	Difference @ 100% sampling
15	12.08	12.59	6%	14.31	19.25%
20	19.21	19.65	8%	-	-

Table 3: Average Latency Comparison between Sampling Configuration

Load Factor	Average Transaction Rate (per minute)				
	w/o Probe	w/ Probe @ 25% sampling	Difference @ 25% sampling	w/ Probe @ 100% sampling	Difference @ 100% sampling
15	836.84	810.38	-3.16%	729.03	-13%
20	761.40	741.14	-2.66%	-	-

Table 4: Average Transaction Rate Comparison between Sampling Configuration

Load Factor	CPU Utilization (%)				
	w/o Probe	w/ Probe @ 25% sampling	Difference @ 25% sampling	w/ Probe @ 100% sampling	Difference @ 100% sampling
15	58.21	58.49	0.36%	61.43	6%
20	59.62	61.94	3.90%	-	-

Table 5: CPU Comparison between Sampling Configuration

While sampling is an effective strategy for reducing the overhead of the Probe, it is important to note that there is a trade-off when you implement sampling. The trade-off may increase the chances that you might miss a single, particularly slow HTTP response. If this is a concern, there are other Probe tuning options that can be used to reduce the amount of processing that the Probe must do. These options include: latency trimming, depth trimming, disabling SQL parsing, and reducing instrumentation.

5. Performance "Cost" for Instrumenting an Application

Another way to look at the impact that the Probe has on your application is to look at the average number of methods for which the Probe recorded diagnostics data for each HTTP response. This indicates how much information was gathered by the Probe and transmitted over the network to the Diagnostics Server in Mediator mode. The methods that are considered "interesting" vary depending upon the content of the different .JSP pages and upon the level of load that is being used in the run.

Based upon the results of the runs in the study, it was determined that, over all loads for every HTTP response, the Probe recorded diagnostics data for an average of 5.56 methods. When we changed the Probe configuration to halve the number of diagnostics points hit, we also ended up reducing the amount of Probe overhead by about half.

This number cannot be extrapolated for all individual cases. For instance, because of the latency trimming feature of the Probe, the longer a method takes to execute, the more likely it is that the Probe will capture the diagnostics data. Thus, when the performance on your application begins to bog down because of processing problems or complicated transactions, the methods will take longer to run and more of the methods will pass the latency trimming thresholds so that the Probe will capture the method's performance metrics.

6. Conclusions

For a system running within its capacity, the latency impact of the Mercury Diagnostics Probe for J2EE is generally minor—rarely more than 1%-2%—and is almost certainly not noticeable to end users.

The primary resource used by the Probe is CPU. However, almost all processing is performed in a background thread to avoid unnecessary impact on system latency and throughput. At high enough loads, if the CPU becomes a bottleneck, latency and throughput do deteriorate. Empiral data suggests that throughput degradation caps at about 13%; latency degradation seems to settle at a maximum of 19%.

As the Probe overhead is quite high without spare CPU on the SUT, some performance tuning of the Probe is recommended if you often run above 50% utilization. During the study, tests were performed with a sampling rate of 25% to demonstrate that Probe overhead is roughly proportional to the sampling rate. The results of a study of the overhead impact from other Probe tuning options are not yet available.

Perhaps the most interesting observation from the results of this study is the variability of the overhead measurements, even with the same test application. A less thorough analysis could easily select a favorable load (such as 7) and demonstrate practically zero latency and CPU overhead and only 1%-2% throughput and network overhead. Because of this variability, we recommend that even the results discussed in this document should only be used as a guide. The Probe overhead will be different for each application and each workload. However, no matter what application and workload you are using, remember that many options exist for reducing the overhead of the Probe.