ILOG Business Rules Product Management

ILOG JRules
Performance Analysis and Capacity Planning

Version 1.0

Last Modified: 2005-09-31

Introduction

JRules customers and evaluators often ask fundamental questions such as: “How fast is the rule engine?”, “How much memory does it require?”, “How many CPUs will I require to execute 1,000 rules per second?”, “What is the best JVM to use with JRules” etcetera.

For a product like JRules these questions turn out to be very difficult to answer in a genuinely useful way. When you view the rule engine as a general-purpose execution mechanism for an arbitrary set of program statements, it becomes clear that the most influential factor in answering all of these questions is the contents of the rule engine (rules and working memory) while under test. This is of course no different to many other software components, such as the Java Virtual Machine itself. However this rather flippant answer is obviously not helpful to architects performing capacity planning or trying to understand the impact of JRules on the complex architectural trade-offs they have to make.

Capacity planning is not an exact science. Every application is different. This document is intended as a guide for developing capacity planning numbers and encourages you to err on the side of caution. In particularly the rules used for this capacity-planning document will not be representative of the rules in your application.

Any and all recommendations generated by this guide should be adequately verified before a given system is placed into production. There is no substitute for adequately testing a prototype for capacity planning numbers.

1 Academic Benchmarks

It is briefly worth mentioning the academic benchmarks, such as Manners, Waltz or Fibonacci. In most cases these benchmarks test worst-case Rete-algorithm agenda usage, and are not very representative of most eBusiness applications. While they may be interesting from an academic and learning perspective they generally are a fairly poor predictor of real-world application performance.

2 Sample Rule Engine Results

The table below shows a range of benchmark results obtained with code based on shipped JRules code samples. The examples vary in profile from memory intensive, Rete algorithm intensive, sequential algorithm specific, through to XML processing using rules. The table is intended to give you a general sense for the type of out-of-the-box performance you can achieve with JRules. In most cases these results could be improved upon using the optimization techniques described later in the paper.

Configuration	Operation	Working Memory	Rules	Average Evaluation Time (ms)
Dell D600. SUN 1.5.0_01 JVM	Rules are evaluated against all input objects.	1 million objects accessed using a collect	7 rules, with functions and ruleflow	523
Dell D600. SUN 1.5.0_1 JVM	Rete algorithm with complex pattern matching rules. Incremental updates triggered by assert/update/retract with dynamic scheduling using refraction, priorities, and recency.	500 objects	15 rules, two function	310

Configuration	Operation	Working Memory	Rules	Average Evaluation Time (ms)
Dell D600. SUN 1.4.2_03 JVM	Sequential mode used to evaluate simple rules against a single input parameter. The ruleset uses dynamic select to optimize the set of rules to evaluate based on the characteristics of the incoming data.	1 input parameter	6000 rules	6
2x Xeon, RHEL 2.4.9-e.49smp. BEA 1.4.2_04 JVM	ILOG XML binding used to process two incoming XML documents (an order and a customer) using some BAL rules and a 400-row decision table.	Two incoming XML documents (with different schema)	400 row decision table, plus BAL rules.	3

3 Business Rules, Algorithms and Tuning

As mentioned earlier, the most important factors in determining the performance of the rule engine are the business rules and data that you put into the rule engine. There are a number of guidelines and configuration options that can be used to maximize the performance of the rule engine.

3.1 Structure of Rules

The conditions of rules should be as similar as possible. The JRules implementations of the Rete and Sequential algorithms both analyze the structure of the conditions of a rule and can share condition tests. Decision Tables and Decision Trees will have this property automatically, ensuring that repetitive tests shared between many rules are optimized.

3.2 Number of Rules

JRules can execute a very large number of rules. Using the Sequential algorithm in particular, rules can easily number into the hundreds of thousands. As the number of rules increases the rules may need to be examined more closely for potential performance issues and carefully orchestrated using Ruleflow. With large numbers of rules, issues such as parsing time for the ruleset and memory footprint may become factors worthy of specific benchmark studies.

3.3 Executable Object Model

All code called from the conditions of rules should be optimized. The conditions of a rule may be invoked many times depending on the data in working memory.

The actions (the “Then” part) of business rules should execute as quickly as possible. Avoid performing high latency tasks inside the rule engine if it can be avoided. These tasks may be invoked more frequently than expected, particularly as applications evolve and the contents of working memory changes.

3.4 Ruleflows

Using a Ruleflow to segment and control rule execution is a very good technique to not only create more maintainable rule projects, but also to improve performance. The execution algorithm (see below) can be specified at the Rule Task level, allowing you to mix Rete-based inference tasks with Sequential algorithm tasks, while sharing state between them using ruleset parameters.

In addition, using static or dynamic select allows you to create Rule Tasks that contains lists of rules that are specified at authoring or runtime. This allows you to compute the subset of rules to be evaluated; even postponing this decision until the runtime input parameters are assigned and known. These highly focused rule tasks ensure that rules that should never be fired are not present in the task definition. Such an approach should generally be favored over using an agenda filter for example, where rules that have been matched are programmatically prevented from firing using a ruleset function or Java code.

3.5 Execution Mode

JRules implements two rule evaluation algorithms: the Rete algorithm for inference-based problems and the Sequential algorithm for applying a list of rules sequentially to a set of input objects.

The choice of evaluation algorithm is an architectural decision that typically has an impact on how you choose to structure your rules project and develop a rule-based solution. In addition the evaluation algorithm may have an impact on runtime execution performance, parsing time and memory requirements. A detailed description of the differences between the Rete and Sequential algorithms is includes within the JRules documentation under:

- *Application Development > Implementation > Rule Engine > Sequential Processing > Comparing the Rete and the Sequential Algorithms*

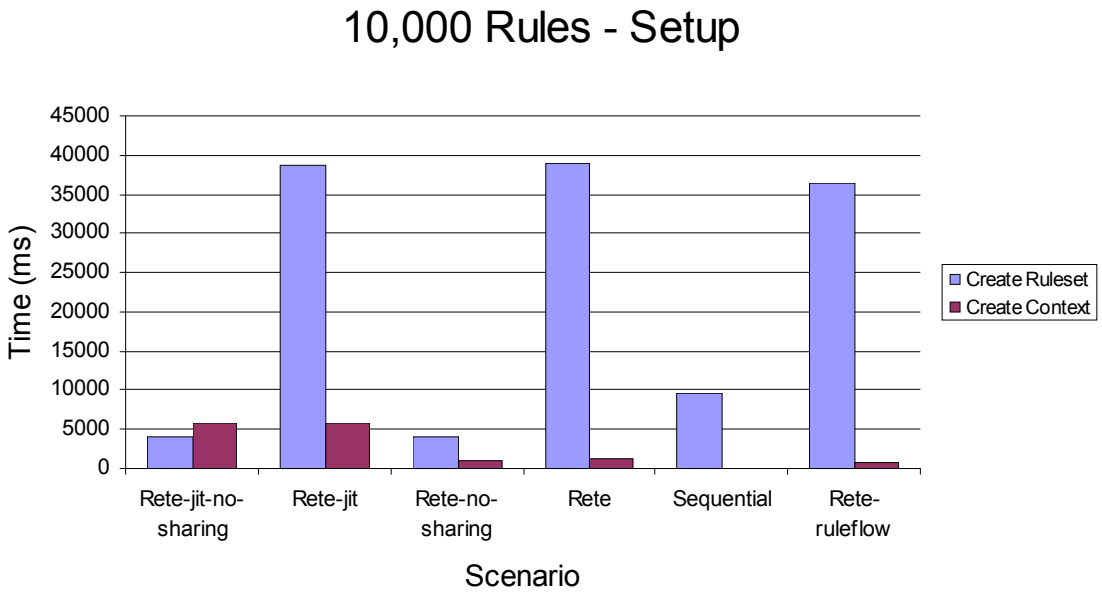

Figure 1 Parsing and IlrContext creation time. Dell D600, Windows XP

Figure 1 shows the time required to parse a ruleset with 10,000 rules and then create an executable IlrContext rule engine instance. The parsing time is dominated by the detection of conditions that can be shared between rules (which is a key benefit of the Rete graph). Therefore disabling Rete sharing detection reduces parsing time for Rete algorithm rulesets, however some of the benefits of Rete are lost. In situations where it is known that little or no sharing is present however this may be a useful optimization. The Sequential algorithm has a simpler parsing and context creation code path; however more work is performed at the first ruleset execution.

Figure 2 First and Average execution time. Dell D600, Windows XP

Figure 2 shows the runtime execution times for a 10,000-rule ruleset in six sample execution modes. The graph illustrates the high cost of the first sequential mode execution, while the Just-In-Time transformation from the ILOG Rule Language to Java byte code takes place. This is typically a one time cost however, if IlrContext pooling, as implemented by the JRules Business Rule Execution Server is used.

3.5.1 Rete Algorithm

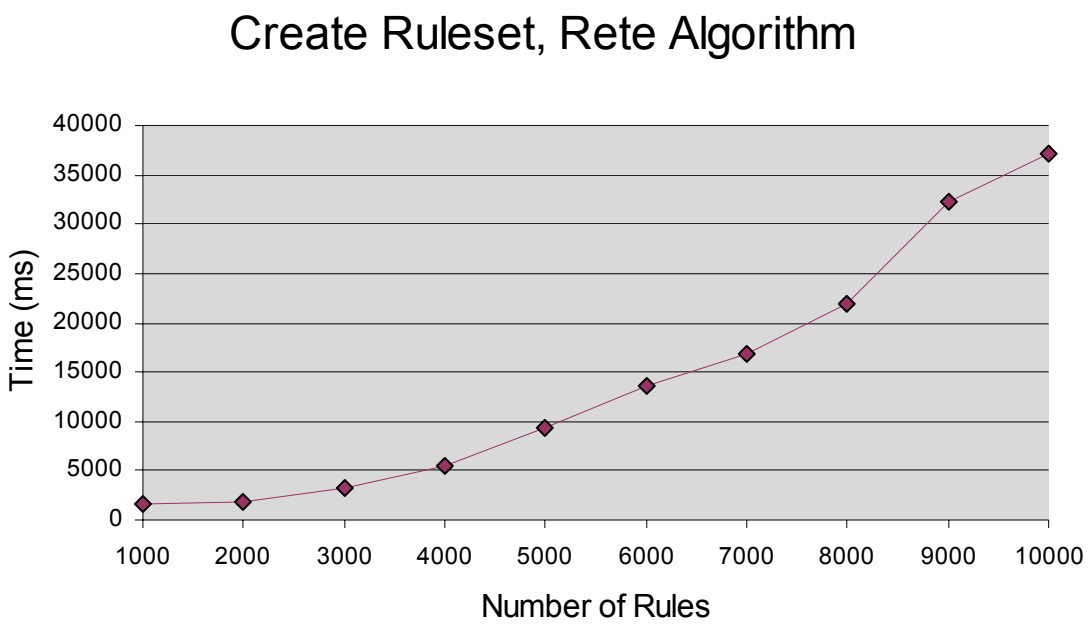

Figure 3 Parsing scalability for Rete ruleset. Dell D600, Windows XP

Figure 3 shows the scalability of the parser for Rete mode rulesets. Parse time can become a significant consideration (37.5 seconds for 10,000 rules) reinforcing the importance of a reliable caching strategy for parsed rulesets, such as offered by the Business Rule Execution Server (see below).

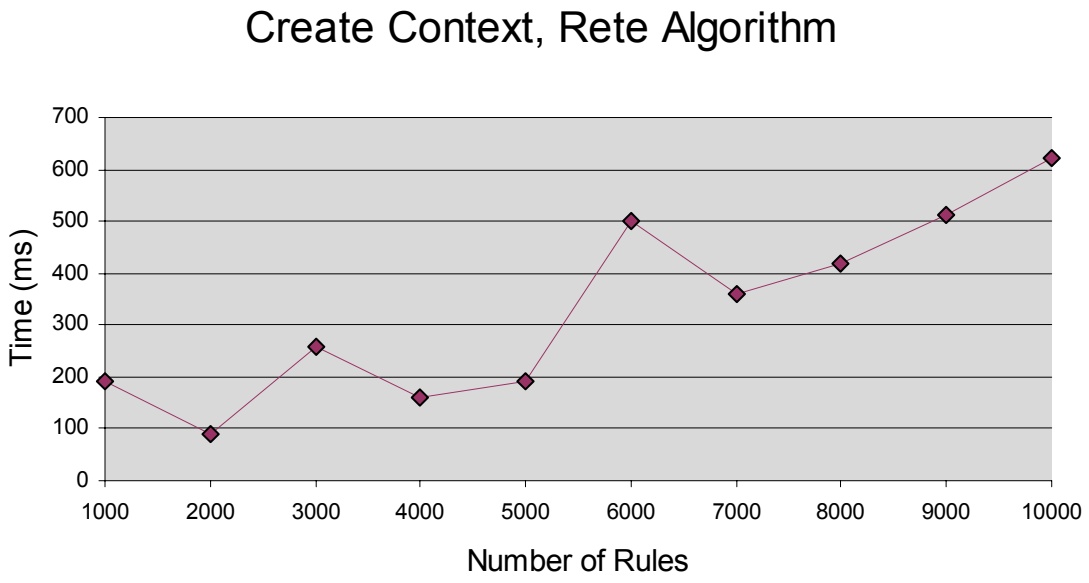

Figure 4 IlrContext creation time for Rete ruleset. Dell D600, Windows XP

As Figure 4 shows, creating an IlrContext from a parsed Rete ruleset is rarely a performance issue, showing approximately linear scalability and a creation time of less than 650ms for a ruleset with 10,000 rules.

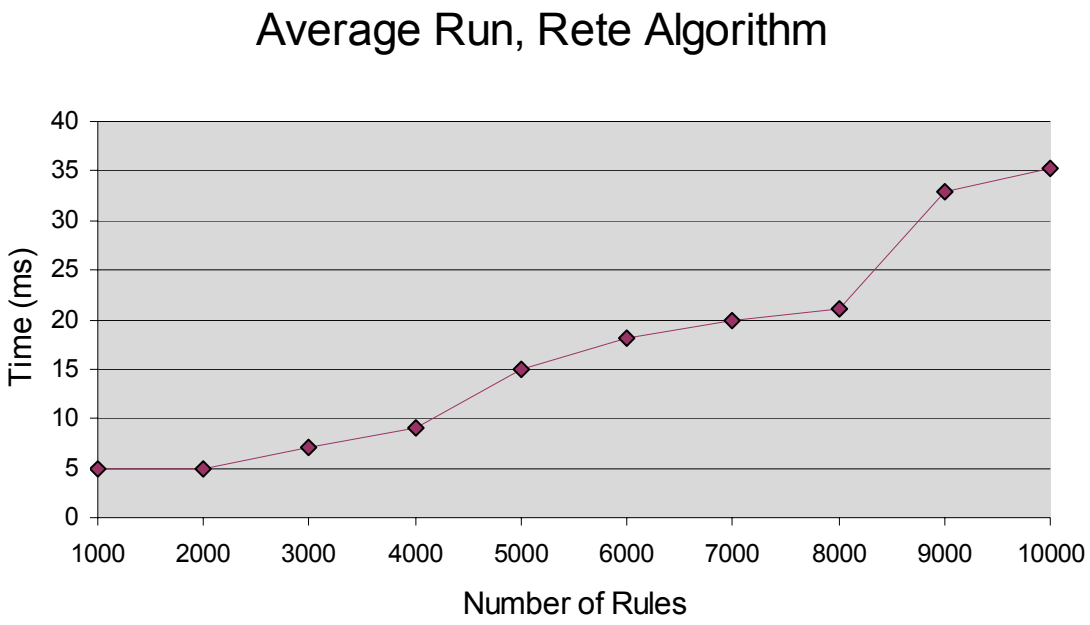

Figure 5 Average Execution Time, simple Rete rules. Dell D600, Windows XP

Figure 5 shows the high performance of the JRules Rete engine, capable of evaluating 10,000 simple Rete rules in just over 35 ms. The rete-ruleflow scenario was used, and the test run on a Dell D600 with Sun Microsystems JDK 1.5.

3.5.2 Sequential Algorithm

The JRules Sequential algorithm is a very fast mechanism to apply a list of rules to a list of input objects. The Sequential algorithm always executes as Java byte code (generated using ILOG's Just-In-Time compiler) offering the best execution on modern optimizing Java Virtual Machines. The JRules Sequential algorithm is particularly fast on the Sun Microsystems Hotspot JVM.

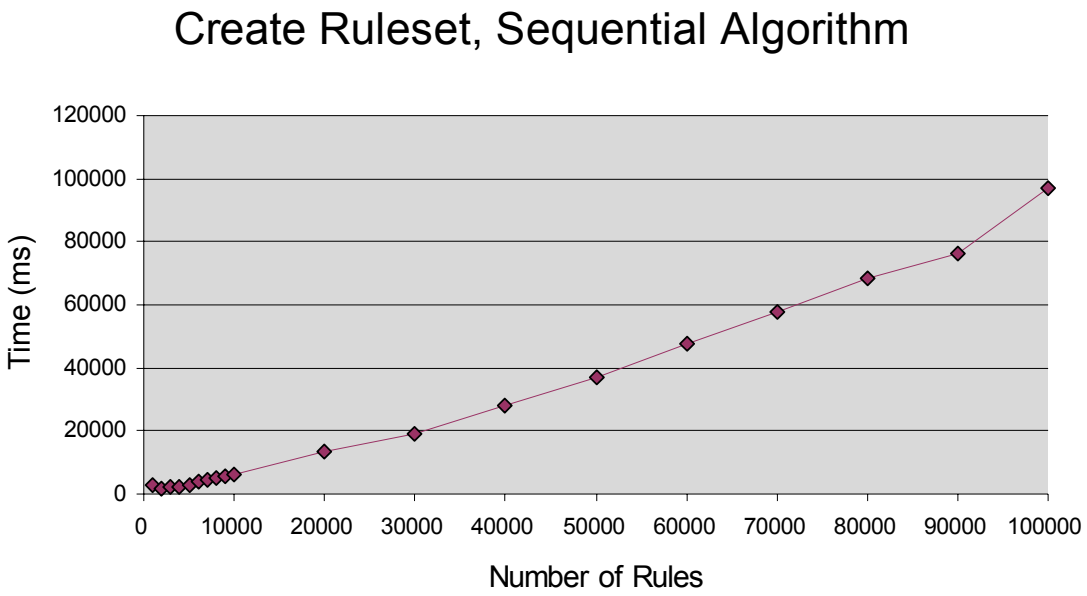

Figure 6 Parsing time for sequential mode rulesets. Dell D600, Windows XP

While parsing Sequential algorithm rules the rule engine does not have to build the complex Rete network. Sequential mode therefore also benefits from improved parsing times when compared to Rete (with condition sharing detection enabled). Figure 6 illustrates that 100,000 sequential mode rules can be parsed in less than 2 minutes and that the parsing time varies linearly with the number of rules.

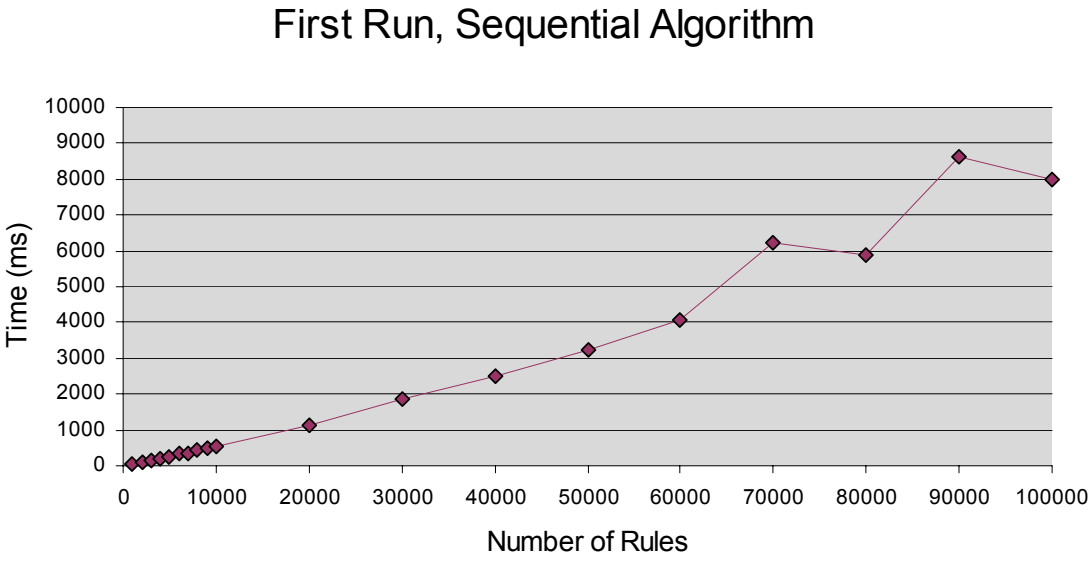

Figure 7 JIT Compilation time for Sequential algorithm rules. Dell D600, Windows XP

The first time a Sequential algorithm ruleset is executed the rules are Just-In-Time compiled to Java byte code. For large numbers of rules this can take several seconds as many classes may be generated, with each class containing many methods. These classes have to be verified and loaded by the JVM. This is typically a one-time operation however, as the generated byte code will be stored within the IlrContext instance, which in the case of deployment using the BRE Server, is itself cached.

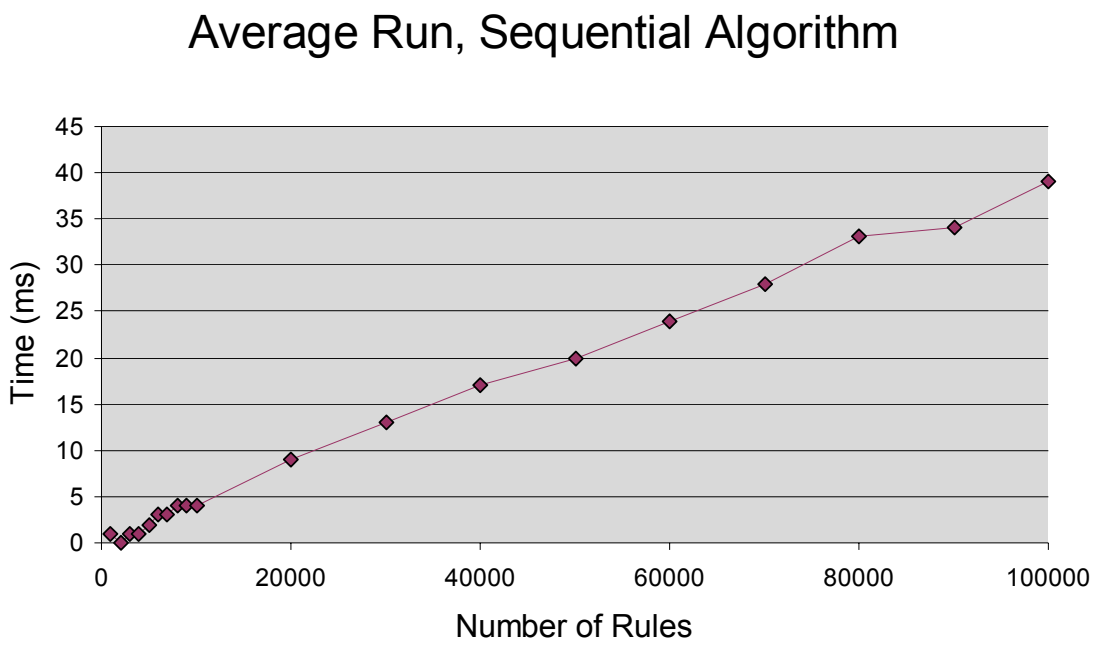

Figure 8 Average execution time for Sequential algorithm rules. Dell D600, Windows XP

Subsequent execution of Sequential algorithm tasks is very fast and varies approximately linearly with the number of rules. As can be see in Illustration 8, 100,000 simple Sequential algorithm rules can be evaluated in less than 40 ms.

3.6 Autohashing, Hashers and Finders

Defining Hashers and Finders for a ruleset data model is a very powerful way to further optimize ruleset execution. Hashers can be used to optimize equality conditions between model elements while Finders are used to optimize navigation through the object model by introducing domain specific knowledge into the rule engine.

These optimization techniques are described in the JRules documentation under:

- Rule Engine > Optimization Techniques

The BR Studio rule engine-tuning example improves the performance of a ruleset by a factor of more than 6, by careful analysis of the structure of the ruleset and the runtime behavior of the ruleset.

3.7 Rete Tuning

The Just-In-Time (JIT) byte code compilation feature for Rete mode can be used to optimize the evaluation of conditions of rules. When JIT is enabled JRules will call methods in the conditions of rules using generated byte code, rather than using the Java reflection APIs. The time to create an IlrContext from an IlrRuleset is increased

when the JIT is enabled, however this is a one time cost for pooled IlrContext instances, and should not be of concern. The JIT can be activated whenever the rule engine has the security permissions to create a custom ClassLoader.

3.8 Sequential Algorithm Tuning

The Sequential algorithm optimizations consists of providing automatic caching for the tests and the values of the condition parts of rules, so that similar expressions will not be computed twice during the sequential application of the rules to a tuple of objects.

With test caching turned on, tests of different conditions that are related, even in different rules, will share a common test value register at runtime that will be computed only once for a given tuple of objects. The level of test analysis for sequential mode can be specified using deployment properties. The most powerful test analysis can go beyond equivalence and is able to identify such things as two tests that are the complement of each other or the fact that a test subsumes another test.

3.9 Minimizing Parsing Time

To minimize the parsing time for large rulesets, use the Sequential algorithm or use the Rete algorithm with the flag *ilog.rules.engine.useReteSharing* set to false. This flag disables the expensive condition analysis that is performed while building the Rete network. It will result in a Rete network with no condition sharing however, so this flag should be used with care as it can have a significant performance impact.

4 Invoking the Rule Engine

In general terms, for high performance systems, architects favor a stateless execution model. This allows for easier load-balancing, horizontal scalability, and in the special case of idempotency, for failed transactions to be retried in the event of failure under load, or for disaster recovery.

A message based invocation pattern (using the Message Driven Bean supplied with the BRE Server for example) can also perform extremely well and also provides easy horizontal scalability and good scalability under peak load conditions. The Java Message Service provider used by the Message Driven Bean can also assure guaranteed delivery of messages and other quality of service contracts for high performance and highly-available systems.

4.1 Ruleflow Task Runners

For some applications, maintaining the state associated with a Ruleflow may be unnecessary. If a given task in a Ruleflow needs to be executed many times, with maximum performance the Ruleflow mechanism can be bypassed and a single Sequential task can be directly executed by API.

This advanced optimization technique is described in the documentation here:

- *Rule Engine > Optimization Techniques > Rule Task Runners*

Although for some use cases the performance advantages may merit such an approach, Task Runners should not be used indiscriminately – as they complicate the deployment, maintenance and orchestration of rules. Typically the control or rule orchestration logic moves from within the ruleset to within Java code, resulting in an overall loss of visibility into how the rules are fired. Task runners cannot be used with the BRE Server.

5 Hardware and Operating Systems

For simple single-threaded, CPU intensive JRules micro-benchmarks, single processor machines with a high clock-speed generally outperform multiple processor machines with slower clock speeds. On real-world applications (which typically perform significant I/O operations for network or data access) clustered machines and multi-CPU machines with server optimized JVMs and Operating Systems are to be preferred. Quality of service, availability, robustness under stress conditions and manageability should also be weighted against raw performance when selecting an execution environment.

6 Java Virtual Machines

Java Virtual Machines evolve very quickly and all have different performance characteristics. In addition JVMs supply a wide array tuning capabilities (particularly for memory management) that may have a considerable impact upon general Java performance. Please refer to the detailed documentation for your JVM for more details.

7 Memory Usage

The table below shows the approximate memory usage for several JRules execution scenarios. The memory requirements are mostly independent of the execution mode and varied between 2.37 Kb and 4.12 Kb per rule, runtime memory commit. This figure will vary considerably however depending on the number and complexity of the rule properties (metadata) and the degree of sharing in the conditions of the rules. The scenarios using JIT show higher JVM memory consumption due to the generated Java classes.

Memory usage was measured using the Sun Microsystems JVM and the new JDK 1.5 JMX MBeans for memory monitoring. The configuration tested was JDK 1.5.0, JRules 5.1, Dell D600, Windows XP.

Scenario	**Total Benchmark Used Memory (1 rule) (MB)**	**Total Benchmark Used Memory (10,000 rules) (MB)**	**Memory Delta (MB)**
simple-rete-jit-no-sharing	10.53	44.9	34.36
simple-rete-jit	10.71	44.87	34.16
simple-rete-no-sharing.	9.3	36.13	26.83
simple-rete-ruleflow	9.21	34.02	24.8
simple-rete	9.29	32.45	23.16
simple-sequential	11.79	51.99	40.21

8 Deployment to a J2EE Application Server

In general the J2EE application server should be tuned for the deployed application and the expected peak or average load. Generally speaking this includes items like:

- Number of request threads
- Number of execute threads
- Size of EJB pools
- Size of JCA pools
- Using native I/O or pure Java I/O
- Pool reclamation policy
- Data replication strategy for clustered deployments

You should refer to the detailed tuning guide for your application server for specific information related to your environment.

9 Using the ILOG Business Rule Execution Server

The BRE Server is a complete execution and management environment for business rule applications. It provides a number of features that are required for high performance, scalable and manageable applications:

- Pooling of rulesets and contexts
- Hot deployment of rulesets
- File and database persistence of deployed rulesets
- Web-based system administration console
- Runtime monitoring of execution using the Java Management Extensions (JMX) API
- Client APIs: stateless Plain Old Java Object (POJO), stateful POJO, stateless EJB, stateful EJB and Message Driven EJB for asynchronous invocation

By deploying the BRE Server (for J2EE or J2SE applications) your business rules applications automatically take advantage of the execution and management services offered by JRules and you avoid having to implement ruleset and context pooling within your application code. Your overall application build and deployment processes can also integrate the Ant tasks ILOG provides for the BRE Server.

9.1 Tuning the BRE Server

Some simple tuning tips for the BRE Server:

- The log level of the BRE Server Execution Unit should be set to WARNING (the default mode is INFO) by editing the Resource Adapter XML deployment descriptor. If the Message Driven Bean is being used, disable trace messages by editing its deployment descriptor. Typically this will give a performance gain of 25%-50% over the default, more verbose, configuration.
- RuleApps, rulesets or ruleset tasks should not be deployed with debugging enabled unless debugging is being used.
- XU Plugins should only be registered with the BRE Server Execution Unit when necessary. If XU Plugins are used, be very aware of the performance impact of code within the plugin callback method as the callback will be invoked very frequently during execution.
- For J2SE deployments the Execution Unit pool should be sized appropriately: the size should be greater than the number of concurrent requests.

- In J2EE deployments the XU Resource Adapter should be configured (at the Application Server level) to set the number of XU connections appropriately to avoid excessive synchronization of request processing. The recommended settings will depend on the number of CPUs available to the application server and the number of execute-threads configured. In addition, if stateful or stateless EJBs are being used to invoke the BRE Server the maximum size of the pools for the EJBs should also be appropriately sized.

The Message Driven Bean (MDB) is an easy mechanism to achieve high-performance, easily clustered and scalable execution of rulesets. Figure 9 shows the results of invoking a ruleset 100,000 times through the MDB. The ruleset invoked contained 10,000 simple Sequential algorithm rules.

The configuration used was:

- JMS Server: JBoss 4.0.1 running on 2x CPU Xeon, with hyper-threading, 2 GB RAM, Linux configuration running RHEL 2.4.9-e.49smp
- JMS Client: Dell Latitude D600, Windows XP SP2, 1 GB RAM.
- 10,000 simple Sequential algorithm rules
- JRules 5.0

The results obtained were:

- Client processed 10,000 JMS messages in 94,946 ms
- Client side transactions per second: 105
- Average client side processing time per message: 9.5 ms
- Average server side processing time per message: 8.5 ms

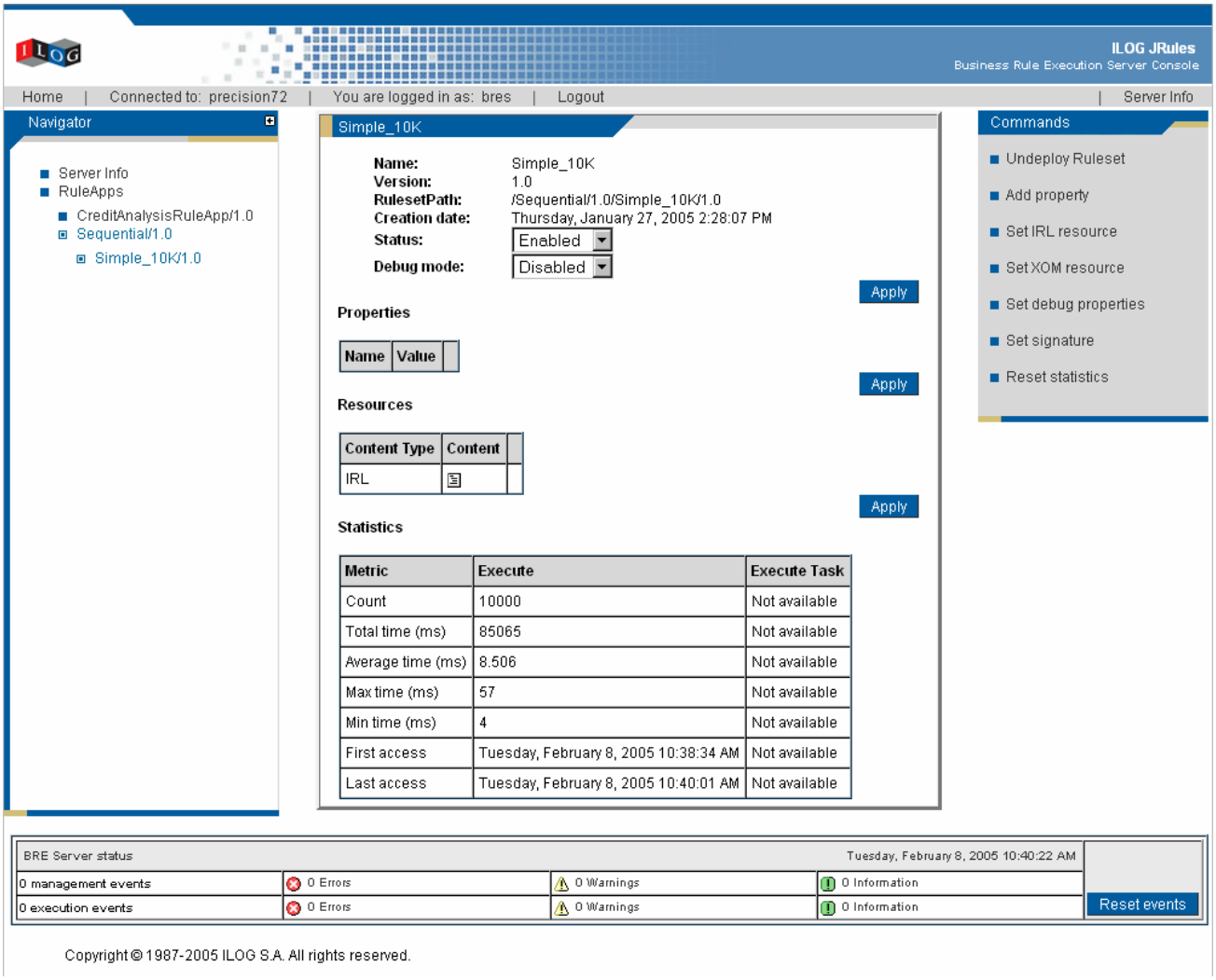

Figure 9 BRE Server console showing the results of 100,000 calls to a ruleset.

10 Conclusions

ILOG JRules offers a rich variety of high-performance and scalable services for demanding eBusiness applications. The mature and highly optimized Rete and Sequential rule evaluation algorithms, coupled with the caching and management services of the BRE Server, ensure that solutions incorporating business rules can be deployed quickly and effectively. As can be seen from figure 10, in many cases the time taken to evaluate thousands of business rules may be less than that required for a simple database query.

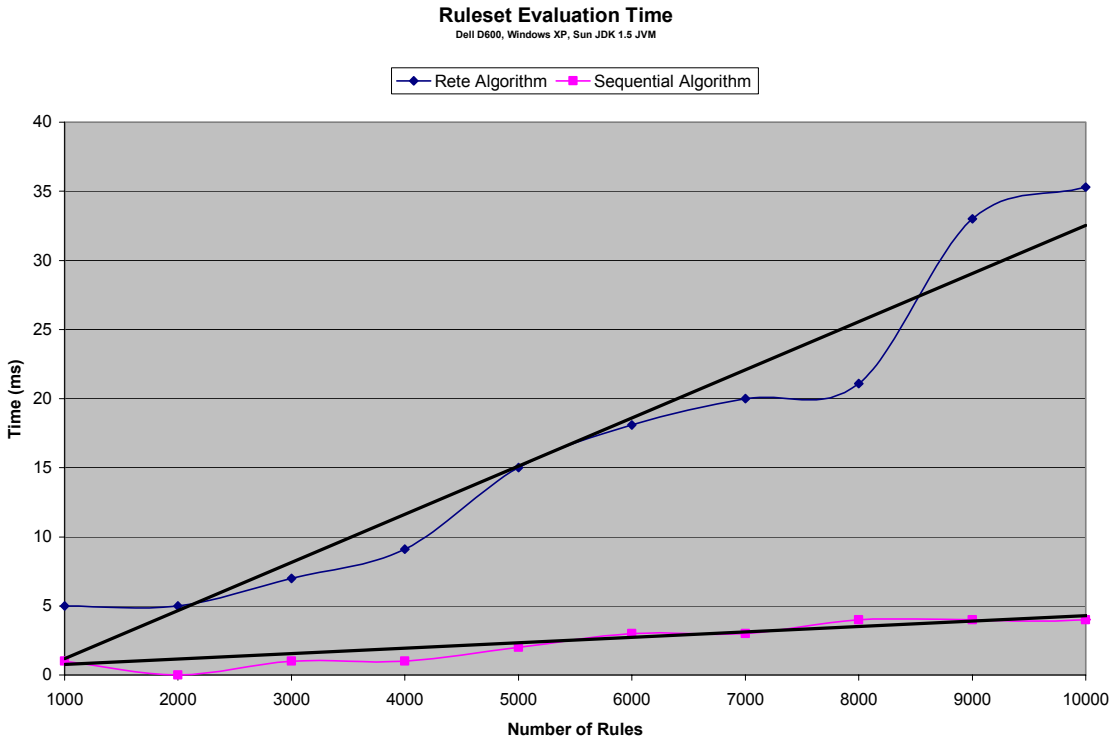

Figure 10 The JRules Rete and Sequential rule evaluation algorithms exhibit excellent performance and scalability

Once solutions are deployed to a staging environment, architects may choose to exploit the powerful JRules optimization features to further tune applications, based on target service level agreements. The high performance coupled with the flexibility of JRules makes it equally suitable for intensive J2EE online transaction processing systems, integration with message-oriented middleware or high-volume J2SE batch processing applications.

If you have additional questions, training requirements, or would like hands-on application design or tuning assistance, please contact your local sales representative or access the ILOG website at **http://www.ilog.com**.

11 References

WebLogic Capacity Planning

http://e-docs.bea.com/wls/docs81/capplan/index.html

MP16: WebSphere MQ for z/OS - Capacity planning & tuning

http://www.ibm.com

IP03: WebSphere MQ Integrator - Capacity planning tool

http://www.ibm.com

IBM Training: WebSphere Application Performance Sizing and Capacity Planning for iSeries servers
http://www.ibm.com

Performance Analysis for Java Web Sites by Joines, Willenborg, and Hygh
ISBN 0201844540
Publisher: Addison-Wesley Professional

J2EE Performance Testing by Peter Zadrozny, Philip Aston, Ted Osborne
ISBN: 159059181X
Publisher: Apress

Anatomy of a flawed micro benchmark by Brian Goetz
http://www-128.ibm.com/developerworks/java/library/j-jtp02225.html?ca=drs-j0805