

Movex NextGen and iSeries™: Anatomy of a Successful Java™ Benchmark

In May, 2001 Intenia Research and Development ran official benchmarks of their Movex NextGen ERP on the iSeries platform. The results showed Movex NextGen to scale over 1 million fully processed transactions per hour. Movex NextGen is the first 100% pure Java ERP Application.

Scott A. Moore, Software Engineer, IBM Rochester

1-507-253-0401

samoore@us.ibm.com

January 2, 2002

©2002 International Business Machines Corporation, all rights reserved

Abstract

Intentia and IBM® Rochester have had a long history together. Intentia International, based in Stockholm, Sweden, was founded in 1984. During the last 18 years, their flagship product, Movex, has grown to be a major player in the Enterprise Resource Planning (ERP) arena. Initially, Movex was written entirely in the RPG programming language. Over the last three years, Intentia has ported their RPG version of Movex to a 100% pure Java™ solution. The 100% pure Java solution has demonstrated excellent scalability and performance during official Intentia benchmarks performed at the IBM Rochester Customer Benchmark Center. The Movex mixed load Benchmark crowned years of performance work with the Java application.

Executive Summary

In 1998 Intentia Research and Development underwent a huge development effort to port their iSeries RPG Movex ThisGen ERP application to a multiplatform Java-based Movex NextGen ERP application. Over the next 3 years, Movex NextGen evolved from a pipe dream to a highly scalable ERP solution, running on the IBM eServer iSeries™. This was demonstrated in May, 2001 when a benchmark in the IBM Rochester, Minnesota Customer Benchmark Center topped 1 million full business transactions in one hour.

The goal of this paper is provide chronological information on the life-cycle of the Movex NextGen benchmark in a generic context. The target audience for this paper would be anybody looking to run an iSeries benchmark on their application. Most emphasis of the paper relates to Java problems encountered, or using Java development tools to locate problems.

The reader should have basic knowledge of the Java Runtime Environment (JRE), and have basic knowledge about the iSeries system. For more information on the JRE, please see the Sun Microsystem site at <http://www.java.sun.com>. iSeries information can be found at <http://www.iseries.ibm.com>.

Benchmark Definition

There are many reasons for performing a computer benchmark on an application. A benchmark can be used by any of the following:

- Hardware companies. They need to provide statistics that can be used to compare one platform to another platform.
- Sales force. Both the application and the hardware companies sales force can use a benchmark to determine sizing information to choose a specific system, with the correct memory, disk, and processor requirements. Also, the application sales force can use the benchmark to compare themselves to other competitor products.
- The application provider. The company producing the application can find potential scaling problems when their application is under load. Alternatively, they can test and validate fixes in performance sensitive areas.

A common definition for a computer performance benchmark is ‘a repeatable standard test(s) where measurements can be made by which others can be compared, evaluated, and judged.’ From the beginning, every benchmark must start with a set of goals. When the Movex NextGen benchmark was first defined, the benchmark had the following goals:

- Simulate a real world workload
- Create an environment that is easy to define and recreate
- Measurable metrics
- Configuration definitions
- Use industry standard tooling to perform and measure the benchmark

Simulate a Real World Workload

Most, if not all, application benchmarks would like to boast that they closely model a real world environment. Although this should be a goal of the benchmark, the end result will not mirror the many ways different customers will use the software. When designing the Movex benchmark, we hand picked several customers who have a ‘typical’ Movex workload. With their consent, we examined how these customers utilized Movex. We recorded which transactions were performed, as well as the frequency in which the transaction occurred. With these statistics, we created a table that contained each transaction, as well as the number of times it is called. We found that 90% of Movex transactions performed at the customer site could be summarized by 4 or 5 different flows.

<i>Flow</i>	<i>Movex transaction</i>	<i>% of customer load</i>
Customer Order	OIS100	45
Purchase Order	PPS200	15
Work Order	PMS100	30
Manufacturing Order	MMS100	10

The next step on defining a flow is what steps are performed when creating a customer order, for instance. Using Movex, there are many ways to create a customer order. Again, using the actual customer as a reference, we chose a flow that was the most common. Each of the 4 flows that we chose had a specific definition on what that transaction looked like. Here is an example of the detail needed to define a flow for a customer order:

Step	Panel	Description
1	OIS100	Start Customer Order (OIS100 - Panel A)
2	OIS100/A	Delete customer order number (if it exist).
3	OIS100/A	Enter a Customer number (1000)
4	OIS100/A	Type '011231' date in the "req. dly dte"
5	OIS100/A	Press <Next>
6	OIS101/A	Press <Next> to confirm.
7	OIS101/B1	Type item (Random 1030 – 1199)
8	OIS101/B1	Type '1' in "Quantity"
9	OIS101/B1	Type '10' in "Sales Price"
10	OIS101/B1	Press <Finish>
11	OIS101/B1	Press <Finish>

Now, even with the different flows, we still had to define the database that exists for these typical customers. For instance, a seemingly useless metric such as having a set number of customers for customer orders becomes quite important.

- The sheer number of customers within a database table can affect performance on the time it takes to retrieve customer information.
- If there are a limited number of customers, locking considerations occur when several orders are processed for the same customer at the same time.
- Depending on the algorithm, the amount of memory needed could increase significantly, as the number of customers grows in size.

The data from the customers were modeled in the benchmark database and saved away as the data used for all Movex NextGen benchmarks.

An Environment That is Well-Defined and Easy to Recreate

A good benchmark should be well-defined and easy to recreate, regardless of the platform on which the test is run. This rules out any speculation.

Once the preparation for the benchmark was finalized, the environment was cemented in time. The database was saved off to file, the application code was saved off, and system settings were recorded.

In order to have an environment that is easily recreatable, the exact contents of the environment cannot be in flux. The benchmark was performed on the same level of code that was shipped to customers. Any changes that are to be applied must be examined with scrutiny, in order to prevent many temporary fixes to the environment, some of which could hinder the actual performance. Also, these changes must be clearly documented.

Measurable Metrics

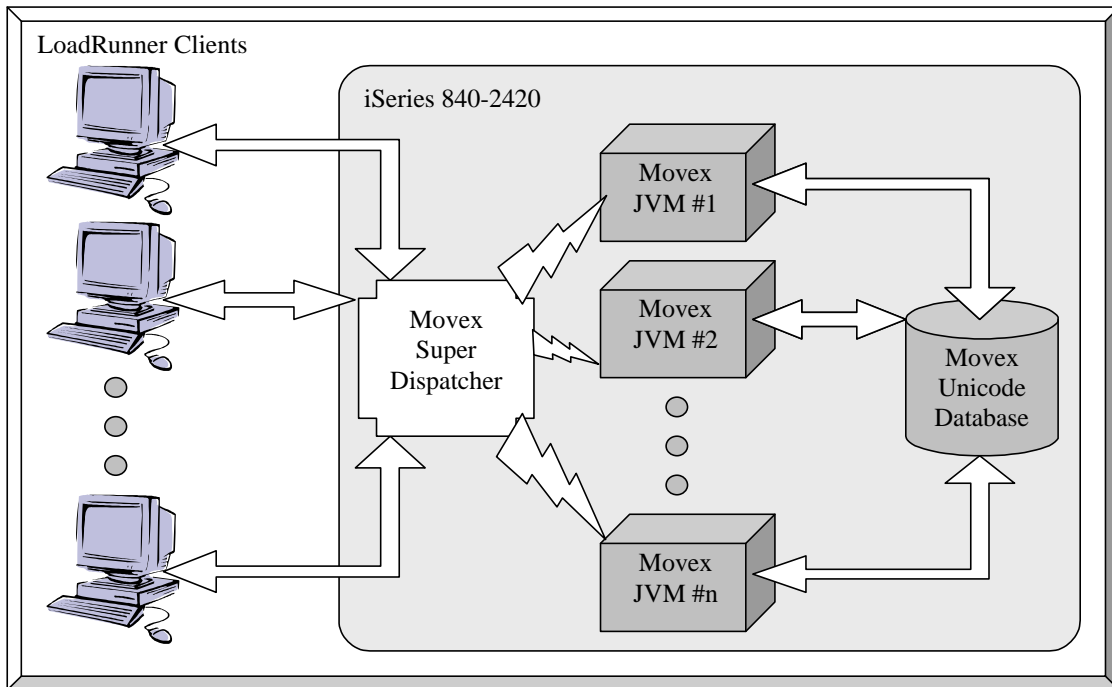
The key metric of the Movex benchmark was the number of transactions fully executed during an hour. To get this number, the application was ramped up for at least 15 minutes. Between 15 minutes and 45 minutes, the actual measurement was taken. This value was multiplied by two to get the final measurement.

Average Script Response Time is the amount of time spent during a transaction, once a submit request was sent, and when the request is received. Each script contains multiple submits/receives. The Average Script Response Time is the average of all the submit/receive times within the script.

The maximum response time is also very interesting in a Java benchmark. As the garbage collector starts to run, it could severely affect the performance of interactive threads on some platforms. This could stem as very long response times for the unlucky soles who are running when the garbage collector is running.

Two-Tier Environment

For the Movex benchmark, all transactions were performed using a two-tier environment. This involved multiple LoadRunner clients, modeling the Movex Explorer front end interface, connected to a single iSeries 840-2420 machine as the backend application server and database.



The setup of the iSeries 840 is listed in the table below:

Server Hardware Configuration			
System Name	S105HYTM	Disk Configuration	
Model	840-2420	Total internal disk capacity	900GB
Processor type/clock rate	Risc 500 MHz	ASP 1. RAID-5	120 Disks. Operating System + Movex + DB
RAM	90 GB	ASP2. Mirroring	20 Disks. Movex Journals
Server Software Configuration			
Operating System	OS/400® V5R1	JVM	IBM JDK 1.2
Database	DB2 UDB® 400	Movex Database Drivers	MvxRLA 1.0.4 MVXSQL 1.15

Use Industry Standard Tooling to Perform and Measure the Benchmark

For the Movex benchmark, we used Mercury Interactive's LoadRunner application. LoadRunner allowed us to simulate thousands of simultaneous users, while maintaining several important statistics. These statistics included response times, measured network load, and kept track of any failures that occurred.

Benchmark Chronology

Problem determination for an application of Movex's size is a more daunting process during the benchmark phase, as opposed to software development phase. During a benchmark, you have the entire solution in your hands, with all parts of the software interacting with each other. Even with the correct tools, this can be an overpowering experience. Luckily, for our benchmark, we had exceptional tools on the iSeries to simplify this process. We divided the benchmark into four distinct stages.

Stage 1: Starting the Benchmark

When you first start a benchmark, several things must be done up front. First, validate the test cases and the environment. Each script that is part of the benchmark should be run individually. It is not only important that the test passes according to the accounting tool used, but that the script performs every step as defined in the script definition. After this point, save off all information in a format that is easy to restore at later stages.

The next step is to take a baseline measurement by running the official benchmark of all scripts against the official code base, using the default application and system configurations. This baseline measurement should be recorded and referred to throughout the benchmark.

Stage 2: Tuning the Benchmark

After the benchmark has been run to get a baseline, the real work of the benchmark can progress. This stage involves performance problem solving and tuning of the application and system. You must be careful at this stage to only change one thing at a time, document the change, then gather the performance data. If more than one thing is changed at a time, you will have no idea whether change A or change B improved performance. In the worst case scenario, change A improved performance, while change B degraded performance, and the net result was the same, thus it was wrongly determined that neither change had an impact. Before every run, the environment should be restored to its original condition, both from the database and the application layers.

During our baseline benchmark, we noticed several glaring problems with the application running on iSeries. When we first ramped up the number of scripts, we hit a knee around 50,000-100,000 transactions per hour. To reiterate, the final benchmark run contained over 1,000,000 transactions per hour. The following list highlights some performance problems we hit. The subsequent section lists how these problems were identified and includes ways to solve the problem.

- The amount of memory to the iSeries pool dedicated to the JVM proved insufficient.
- Spotlight HotSpots in Java code
- Excessive Object creation and Object deletion

- Java object leakage
- Optimum workload per JVM

Insufficient memory to run JVMs

One rule of thumb that we found was to ensure that the heap for the JVMs did not grow past the amount of RAM memory available. If this happens, a massive performance degradation occurs. Since the Java garbage collector will touch objects all over the heap, the disk I/O became intensive as the JVM heaps grew past the size of memory available. It was obvious that this was the problem, due to the low CPU usage, and the high disk I/O activity.

To determine the size of the heap on a running JVM, two things can be done on the iSeries. First, when you start the JVM, the option **-verbosegc** can be specified. Then, each time the garbage collector runs, several statistics are written to standard output. One of these statistics is the size of the Java heap. Secondly, you can run the iSeries command **DMPJVM** on a Java job that is currently running. Again, the size of the Java heap can be displayed. Both of these commands have other useful features that will be addressed later on.

Spotlight Hotspots in Java Code

On the iSeries, Performance Trace Data Visualizer (PTDV) can be used to profile the Java code to find out where the application is spending most of its time. (Documentation for PTDV can be found at the <http://www.alphaworks.ibm.com> website. Search for 'Performance trace'.) PTDV profiling let us determine where in the code the application spent most of the time during the benchmark. PTDV also highlighted any kind of lock waits that were performed during the run to spotlight any possible problems.

Excessive Object Creation and Object Deletion

Almost all Java performance articles deal with the creation of short-lived Java objects. Notoriously, Java String objects are targeted as the number one contributor to this problem, and articles stop just short of not using String objects all together. Intentia made it a religion to have their developers concentrate on suppressing the unnecessary creation of Java objects in their mainline code. Although their efforts were superb in this area, the benchmark still showed some object creation problems.

One way to discover if the problem of short-lived object creation exists in your application, it is necessary to study the behavior of the garbage collector as the application is running. As the number (and size) of short-lived objects that are created increases, the more often the garbage collector has to run. In order to keep tabs of how often the garbage collector runs, start the JVMs with the option of **-verbosegc** on the Java command line. As the garbage collector finishes a collection run, statistics will be dumped to standard output. One statistic will be the number of milliseconds that the garbage collector ran for. From this, it is easy to calculate the amount of time spent in the garbage collector, for an interval. This total time should not be greater than 10-15% of the total time of the interval, as a standard rule of thumb. Of course, each application is different, but generally the lower the number the better. Movex NextGen has tuned their application to a point where the garbage collector ran for less than 1% of their time during this benchmark.

Once it is determined that short lived objects are causing the garbage collector to run too often, two things can be done to address this. First of all, on the iSeries, Performance Trace Data Visualizer (PTDV) can be used to find out what objects are created. (Documentation for PTDV

can be found at the <http://www.alphaworks.ibm.com> website. Search for 'Performance trace'.) PTDV will not only find out which objects are created, but where in the code they are created.

Another way to play with the amount of time that the garbage collector runs on iSeries is to play with the initial heap size. On the iSeries, the initial heap size is also a threshold value as well, to determine when to run the garbage collection. For instance, if this value is set to the default of 32M, the garbage collector will kick in when the heap grows to 32M. If the garbage collector collects 20M of objects, the heap will then contain 12M of used objects. The garbage collector will next run when the heap reaches <Last used heap size> + <initial heap size>, which in our example would be when the heap reaches 44M. This proved to be extremely important in our benchmark, since it was determined that this value was much too small for our case. If this value is too small, the garbage collector will be constantly running, picking up small amounts of memory each time. If this value is too large, the time it takes to run the garbage collection can be noticeable. The net is that each application will have an optimum setting of this value. Not only does this affect performance, but the size of the heap will be smaller, with a lower value. We found that setting this value to 1024M was the correct value for Movex NextGen, to maximize the performance running the benchmark. The initial heap size can be tuned on the command line by the **-Xms** option.

Java Object Leakage

Java, when first released, was touted as a language where memory leakage was a thing of the past. With C and C++, memory leakage is very frequent due to the programmer being responsible for freeing any storage that they allocate. Since Java will automatically free storage for an object once all references to an object disappear, object management was greatly simplified. But, even though Java simplifies this, there are ways to cause object leakage to occur. For instance, one common way is to have a global hashtable Java object. As objects are placed in this hashtable, a reference within the hashtable is created on the Java object. So, although the programmer thinks that all references to the object have gone out of scope, one reference still remains inside the hashtable. Only if the hashtable reference goes away will this object be freed.

To determine if you have major Java object leakage, you will have to look at your heap usage over time. Before leaving for the day, we wrote down the size of the heap and let the benchmark continue over night. The next morning, we looked at the heap again, and noticed that it doubled in size. Since the number of users remained constant, and the load was fully ramped up before we took the first measurement, we felt confident that an object leak was the cause of the problem. To find the object responsible for the leakage, we performed a **DMPJVM** command before leaving for the day, and a **DMPJVM** command when we arrived the next morning. **DMPJVM** lists all of the object types that are within the Java heap, and reports the number of each object. Using this information, we compared the before and after to have an educated guess on which Java objects are to blame. From this point, we could use PTDV to determine which methods are responsible for creating these objects, to further pinpoint the problem.

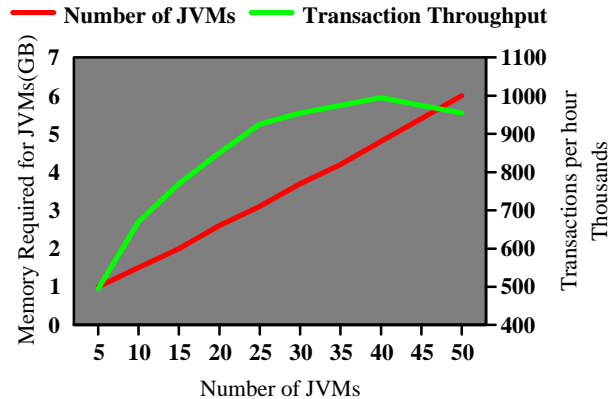
Optimum Workload per JVM

Movex NextGen contains a module called the Movex Super Dispatcher which takes a new request from a client, then does one of two things.

1. Routes the request to a running Movex NextGen JVM
2. Spawns off a new JVM, then routes the request to the new JVM.

To make this decision of which option to use, Movex has a series of tunable parameters within their application to determine if a JVM currently is doing enough work, and if so, whether to spawn off a new JVM as new requests come in.

Number of JVMs effect on Performance



In the chart above, the same amount of clients were running against Movex NextGen. From the chart above, two things are noticeable. First, as the number of JVMs increase, the amount of memory required increases linearly. This is obvious, due to each JVM having its own separate heap, and runtime environment. Secondly, and more interestingly, the number of transactions per hour hit an apex when Movex NextGen had 40-45 JVMs running at one time. Why would this be?

It makes sense that as the number of JVMs increase, Movex is able to handle more load. Each JVM has a certain overhead to it, involving all of its threads, memory management, garbage collection, class loader, security overhead, etc. As the activity in the JVM increases, a bottleneck could occur in one of the operational components that make up a JVM. Spreading the load across multiple JVMs reduce the load on a particular component.

Just as you can saturate a JVM with a workload, you can saturate a machine with too many JVMs. Since many threads within a JVM are required, a certain amount of overhead comes with it. This overhead taxes the resources from the iSeries, thus as the number of JVMs becomes too large, the overall throughput can decrease.

At the apex, a typical Movex JVM had approximately 100-150 active threads running within the JVM. This number should not be used as a rule of thumb, but the amount of workload per JVM should be a tuning factor when maximizing your Java throughput on the iSeries, as well as other platforms.

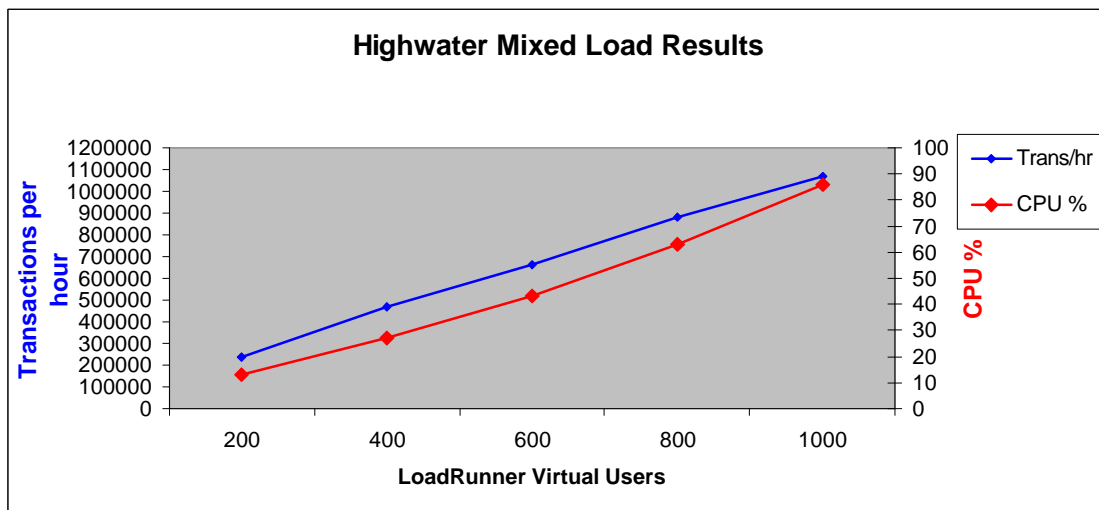
Stage 3: Run the Benchmark

Depending on how you were running the tuning exercises, you might already have ran the actual benchmark. During the run of Movex, each tuning run was performed in a known environment, which corresponded to a valid Movex benchmark run. In other words, our benchmark run occurred when the last tuning run was performed!

Regardless of how the tuning has occurred, this stage should be the shortest stage of them all.

Stage 4: Analyze the Results

This stage usually occurs during the days and weeks following a benchmark. Depending on what the benchmark metrics were, this can be a fairly trivial exercise. For the Intenia benchmark, the most important metric was the number of transactions per hour. Below are the results as we ramped up the workload to 1000 Virtual Users (A virtual user does not coincide with a real life user). The most impressive statistic of this graph is the linear nature of the number of transactions per hour, as we ramped up the number of virtual users. The maximum number of transactions that occurred was 1,071,672 fully processed “mix load” business transactions per hour.



The average response time of each script is also listed below:

Business Transaction	Description	Average Response Time
MMS100	Manufacturing Order	0.23
OIS100	Customer Order	0.39
PMS001	Regular Work Order	0.24
PPS200	Purchase Order	0.29

Conclusion

Intentia's Movex NextGen benchmark was a crowning achievement to 3 years of work by Intentia on the performance of their ERP application. The results on the iSeries were outstanding, considering that over 1 million mixed transactions were performed in one hour time. Intentia has shown that Java can support a huge application that can rival other programming languages if the application is written and debugged correctly.

Intentia designed Movex NextGen to run great on Java and the iSeries is one of the best Java platforms. The IBM eServer iSeries will compete against other platforms for new Movex customers for the first time and the iSeries community should welcome the competition. Intentia has implemented NextGen in a way that will not only take advantage of iSeries, but will help maximize the benefits of Java.

About the Author

Scott Moore is currently a Staff software engineer with the IBM Rochester Laboratory. Scott has worked with IBM Rochester for over 8 years, working in fields such as OS security, POSIX APIs, Java, Performance, WebSphere®, and Linux. Scott is a Magna Cum Laude graduate of Oregon State University with a bachelors degree in Computer Science.

Trademarks and Disclaimers

References in this document to IBM products or services do not imply that IBM intends to make them available in every country. The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AS/400	IBM(logo)
AS/400e	iSeries
e (logo) business	OS/400
IBM	

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Movex, NextGen and Intentia are trademarks of Intentia International AB. Other company, product and service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products does not constitute an endorsement of such products by IBM. IBM cannot confirm the accuracy of performance,

capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

Performance is based on measurements and projections using benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here