Monitoring Oracle Coherence Using JMX: Challenges and Limitations

Thomas Lubinski
SL Corporation
Corte Madera, CA
October 7, 2009

Abstract – **Oracle Coherence is an in-memory distributed data grid solution for applications and application servers. However, as a powerful and complex distributed caching system, it must be managed effectively in order ensure its uptime and performance in critical applications. The product exposes extensive performance metrics via a built-in JMX interface. This paper discusses how to effectively leverage this information and some of the practical considerations that have been encountered in using these metrics in real-world monitoring situations. As part of this discussion, a breakdown of the JMX MBean schema that is provided by Coherence is reviewed, and recommendations are made to minimize the latency of MBean queries, as well as the overhead associated with collection across a large number of MBeans. Finally, the paper offers suggestions for overcoming the limitations of process nodes.**

I. Introduction

Since its acquisition by Oracle in 2008, the Coherence data grid solution has seen a steady increase in adoption. It is a powerful, yet complex component of critical enterprise software infrastructure that must be monitored and managed effectively in order to ensure uptime and optimal performance.

Many large business applications – in industries as diverse as financial services, risk management and on-line stores – use Coherence services for storing and efficiently accessing large volumes of data. A typical Coherence application consists of as many as several hundred Coherence instances or "nodes" (individual JVMs) distributing the storage and access to multiple data caches over dozens of hardware servers. This collection is referred to as a "cluster."

A significant amount of highly useful monitoring and management information is available in Coherence using a set of built-in JMX MBeans running in every instance. An application typically designates one or two Coherence instances that are configured to act as a central aggregator of the JMX information contained in all the other nodes. These nodes produce large quantities of rapidly changing, real-time monitoring data.

Many developers, especially first-time users, tend to underestimate the difficulty of monitoring the large volumes of data coming from a running Coherence installation, as well as the importance of monitoring in the first place. Oracle provides some support for managing and monitoring a cluster using its Oracle Enterprise Manager Coherence plug-in. While helpful, the OEM module provides only a part of the solution necessary to optimally monitor the applications that use Coherence.

To fill the gap, developers sometimes undertake an in-house effort to collect the JMX MBean data themselves, and assemble views using available low-level development tools. However, the complexity of the MBeans often quickly overwhelms such efforts.

SL Corporation has over 25 years of experience with monitoring and visualization applications, with particular expertise in Java. The company's RTView product has been uniquely architected to deal with real-time data produced in many different types of monitoring applications, and has features to address the most common requirements seen in these systems. Along this same vein, RTView Oracle Coherence Monitor is especially designed to deal with the complexity and volumes of real-time data seen in systems built around Coherence.

This paper presents some of the basic concepts involved in monitoring Oracle Coherence using JMX, with emphasis on the practical and effective use of the JMX information gathered. It also discusses limitations to what Coherence provides, additional requirements common to most applications, and provides suggestions for configuration options that can augment the monitoring capabilities.

While a tremendous amount of monitoring data is available, the challenge is in presenting it in a useful and effective manner. The goal of monitoring should be an enhanced ability to understand what is happening within the cluster and identify sources of trouble.

II. Understanding Coherence JMX MBeans

Due to its distributed nature, a Coherence cluster provides numerous individual "test points," or locations in the system where monitoring data may be collected. A cluster containing 100 nodes and supporting 20 caches on each node will have at least 20 x 100, or 2,000 individual collection points, one for

Copyright © 2009 Sherrill-Lubinski Corporation

each cache on each node. The data set from each test point is presented by Coherence as a JMX MBean.

Oracle provides a very useful mechanism to see the result of data collected in these MBeans. The JMX Reporter was introduced in version 3.4 of Coherence, providing out-of-the box reports that help developers and administrators manage capacity and troubleshoot problems. The reports can be extremely valuable for some purposes, but are not real time. Real-time monitoring and alerting requires that all MBean data are captured on a regular basis, aggregated and presented dynamically for immediate viewing or automated analysis. To implement such a system requires an in-depth understanding of the Coherence MBean schema.

There are six primary MBean types that provide most of the monitoring data in a Coherence application. Some of these may be instanced hundreds or even thousands of times. Several other types provide important information, but are instanced fewer times. The complex relationships between the MBeans can make it confusing to understand the MBean schema without some explanation.

A. *Coherence Cluster*

The figure below shows a high-level node-oriented view of a typical cluster. Conceptually, nodes are divided into "storage" nodes which store the data in caches, and "process" (or "client") nodes which access the stored data. Additionally there may be a number of "proxy" nodes which provide a pass-through capability so that other processes may "join" the cluster indirectly and act as process nodes.

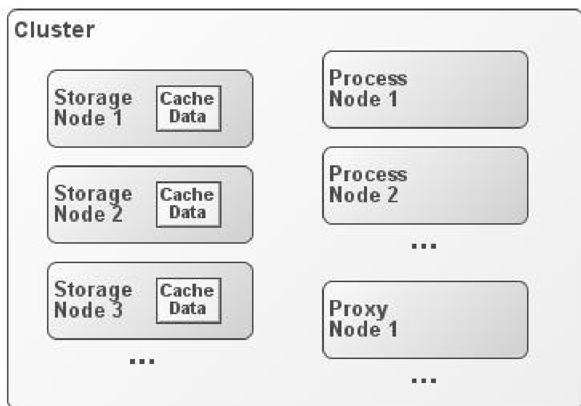

Figure 1 – Cluster View of Storage / Process / Proxy Nodes

There is a single Cluster MBean that contains information about the cluster as a whole. Each node contained in the cluster above exposes an additional Node MBean containing statistics about that node such as maximum and current memory usage, along with network packet transfer information. However, there is nothing contained in the MBean that indicates what type of node it is. That information is carried in other MBeans.

In order to perform functions within a cluster, a node may run several types of services, essentially threads within the node process. The most important of these is the DistributedCache service which can be configured in different "types" having specific operational characteristics. A node that runs a DistributedCache service can access data in any one of multiple data caches that may be defined on that service.

B. *Storage Nodes*

The diagram below shows the additional MBeans that are associated with storage nodes running such services. In this example, StorageNode 1 is running two service types – A and B – each containing two caches.

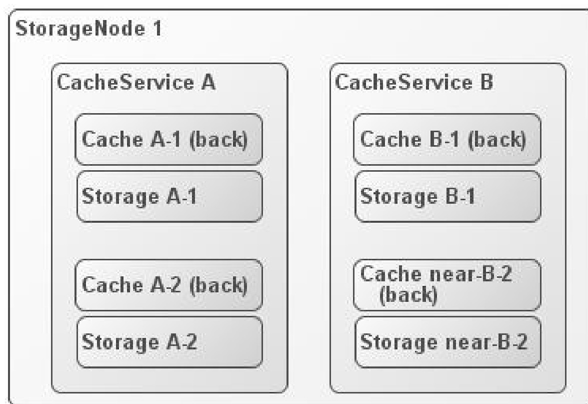

Figure 2 – MBeans Associated With a Storage Node

For each storage node, there is one Service MBean created for each service, A and B, providing data about the CPU load, request count, and so on for the service. There are also two MBeans that provide data about how that node handles each unique cache. The Cache MBeans provide information about total gets, hits, misses, size, limits, etc. for the cache, while the Storage MBeans provide detail about insertions, deletions and evictions for that cache.

In order to provide a complete picture of how a cache is performing, data contained in these MBeans must be collected from all the storage nodes in the cluster and then merged and aggregated. Multiple views can be produced that present current and historical data grouped by node, by service or by cache.

C. *Process Nodes*

Process nodes, on the other hand, make available far less monitoring data about the caches they access. The only MBean associated with the caches on a process node is the Service MBean running on the node for each service. Currently, there are no data available through Coherence JMX regarding the hits, misses, puts, etc. that are executed on a process node. Ways to overcome this limitation are discussed in a section below.

An exception to this is the case of a "near" cache, a special type of cache that provides a local "front" tier (or buffer) on the process node itself for quick access to frequently requested data contained in the "back" tier (the storage node). In this case, there is one Cache MBean

Copyright © 2009 Sherrill-Lubinski Corporation

containing information about accesses to the front local cache running on that node (since the front cache is not distributed, there is no Storage MBean).

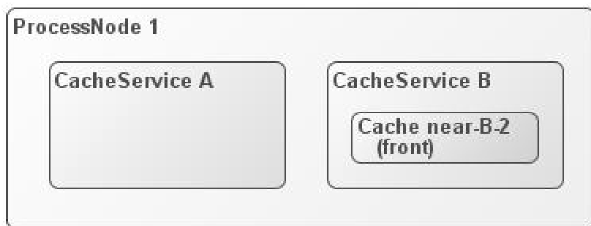

Figure 3 – MBeans Associated With a Process Node

Proxy nodes are typically configured to run one or more proxy service(s), for which the ProxyService MBean provides information about total throughput, CPU load, etc. For each external process that connects to the cluster via the proxy, there is a Connection MBean that provides detail about the number of bytes transferred through that proxy during any given time interval, along with the total bytes transferred and other metrics.

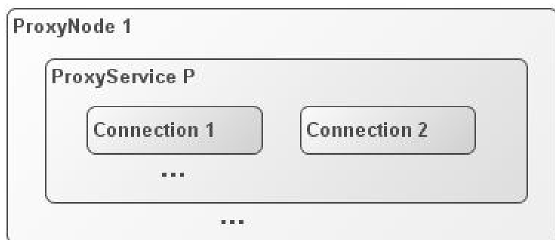

Figure 4 – MBeans Associated With a Proxy Node

Besides the MBeans discussed so far, there are several other monitoring MBeans available, including the PointToPoint MBeans and Connection Manager MBeans. These will not be discussed here.

D. Management Nodes

In a Coherence Cluster, one or two special nodes may be configured to perform a "management" function, using environment variables in the startup command. In this case, the node runs no cache services and provides no storage capabilities. It acts strictly as an aggregator of the JMX information that it collects from all the nodes in the cluster. A monitoring application connects to this management node using a JMX port, RMI, or directly using a local MBean connection. The techniques for doing this are well documented in the Coherence knowledge base.

As of version 3.4, Coherence has been able to act as an MBean aggregator for arbitrary MBeans in addition to its own MBeans. In practice, this feature has been used to collect the standard JVM MBeans from each node in the cluster, augmenting the Coherence statistics with information about heap memory details, including garbage collection pause times and post-GC memory consumption (very important to detecting problems in a cluster). There are over 30 standard JVM MBeans; the challenge is that collecting these from every node greatly increases the number of MBeans that must be processed.

In a cluster containing large numbers of nodes (>100) and many caches (>20), the number of MBeans can go into the tens of thousands. As a practical matter, the total number of MBeans collected can be a source of overhead and latency when monitoring larger clusters.

III. Minimizing MBean Query Latency

There are a number of techniques available for minimizing the overhead associated with querying MBeans. It is important to understand the source of the overhead, and to measure it effectively so optimizations can be performed. The first step is to determine a formula for calculating the total number of MBeans requiring monitoring in a cluster.

Using the schema described in the previous section, it is possible to construct a simple table showing elements of a formula for calculating the total MBean count. As an example, take a typical cluster containing 100 storage nodes "SN," 50 process nodes "PN," and 15 "near" caches "C" each on 2 services "S," evenly distributed across all storage nodes (ignoring proxy nodes for now):

Table 1 – Calculating MBean Count – Example

Component	Formula	Sample	Total
Node beans	SN + PN	100 + 50	150
Service beans	S * (SN + PN)	2 * (100 + 50)	300
Cache beans (back)	2 * C * S * SN	2 * 15 * 2 * 100	6,000
Cache beans (front)	C * S * PN	15 * 2 * 50	1,500
Storage Beans	C * S * SN	15 * 2 * 100	3,000
JVM beans	3 * (SN + PN)	3 * (100 + 50)	450

The total count in this (medium) example is about 12,000 MBeans. Installations have been encountered with even larger counts. Collecting such a large amount of data using JMX is clearly a source of concern in a large cluster.

One option is to avoid querying all of the MBeans at the same time, or to query them only on demand. However, it is a common requirement to collect all monitoring data on a regular interval and archive the data to a database. Doing this, current values can be compared against historical, or capacity planning can be implemented. To get a complete picture of the size, activity, and performance levels of the cluster, all of the MBeans must be queried so they can be aggregated and stored; querying on demand is limited in its usefulness.

Another option is to poll them all, but only do it once every 5 minutes or so to minimize load on the system. But,

Copyright © 2009 Sherrill-Lubinski Corporation

when a cluster encounters a problem, there is often a cascade effect that may take a matter of seconds. To get a good understanding of what has happened in a failure case, it is important to have granular metrics. Thus, it is necessary to query the metrics data as often as possible without overloading the system.

A third solution used in some cases is to query only a subset of some types of MBean. For example, by querying only the Node and Service MBeans, some information may be obtained about the cluster. This approach, of course, leaves out key information and thus is only a partial solution.

In Coherence versions 3.3 and earlier, collecting a large number of MBeans in a large cluster was problematic and limited the usefulness of the JMX monitoring data. The MBean management node collects MBeans one at a time. In the standard Java JMX implementation, an MBean query waits to return until the data are obtained, and uses CPU cycles while waiting. Querying 12,000 MBeans, the wait time on an average Linux box could be as much as 60 or 120 seconds. The only solution was to query at a slower rate, e.g. 90 or 150 seconds to give the CPU some idle time.

In version 3.4, Oracle introduced a clever mechanism controlled by a new property called the "refresh policy" and a value "refresh expiry time." This is a method by which the Coherence node acting as the MBean aggregator can make assumptions about the access pattern for the MBean data. If there is a regular access pattern, the system can be placed in a "refresh-ahead" mode. In this mode, MBeans from all nodes are collected prior to the next expected query using a fast internal protocol. The next time the query is made, the MBeans are available and can be returned with little delay. In the example discussed here, the query time can be reduced to just a few seconds by using the appropriate setting for the refresh expiry time and adjusting the MBean query interval.

Additionally, 3.4 introduced a refresh timeout for JMX queries that prevents a node, which is experiencing a long delay due to garbage collection or other CPU-intensive activity, from holding up the query. This helps to make the query times more predictable.

Tuning the policy and setting the proper expiration period is not trivial. In the RTView Oracle Coherence Monitor, tools are provided to aid in this tuning. Typically, queries must be done slowly at first until the management node "learns" the access pattern, at which point the access interval can be shortened. Once tuned, MBeans access can be very quick, providing a high level of granularity in the monitoring metrics.

IV. Reducing Total MBean Counts

Even with the refresh-ahead optimization, there is reason to give attention to the overhead associated with collecting such a large number of MBeans. While the elapsed time is shorter, permitting one to collect data at a higher rate, large amounts of data are transferred from every node to the aggregator node from all other nodes. This network overhead is not large compared with other forms of Coherence overhead such as deserialization and cluster repartitioning, but it is nonetheless an area to consider for additional optimization.

Coherence developers are familiar with using custom cache configurations to control the behavior of services and caches in the cluster. For example, configuring the High Units setting on a specific service type can limit the amount of memory used by the caches on that service in order to prevent OutOfMemory JVM errors if too many objects are inserted into a cache.

Interestingly, the configuration of caches and services across cluster nodes can also affect quite dramatically the number of monitoring MBeans that are created when the cluster runs. This is one place to look for ways to reduce the total MBean count and minimize overhead associated with monitoring.

A. Use "Near" Caches Carefully

In the example above, "near" caches were defined on both services and on all nodes. This was done deliberately to highlight the way in which monitoring MBeans are created for this type of cache. Near caches are useful for creating a "double-buffered" cache with better performance, but they do introduce additional monitoring overhead.

In Table 1 above, note that the formula for the "back" caches is **2 * C * S * SN**. The "2" in the formula is necessary because, for a "near" cache, there are two Cache MBeans created for every cache, representing two tiers, "front" and "back." However, on the storage nodes, only the back MBean carries important information, even though a front MBean is created. Additionally, every process node creates a front Cache MBean for its local cache.

It would be better to define near caches only when they are specifically required, for performance reasons. For example, if only 4 near caches were defined instead of 30, the Cache MBean count for the back tier would be reduced to 3,400 and the front tier to 200. The result is a combined 3,600 MBeans rather than the 7,500 seen initially.

Often, users create a cache configuration file and, for simplicity in deployment, apply it to all the nodes identically. Then all caches and services are run the same way on all nodes. This simple example illustrates the significant cost associated with ignoring the impact of this on MBean counts.

B. Control Service Configuration Across Nodes

One technique sometimes used to provide control over cache capacity and memory utilization in large clusters is referred to as "heterogeneous scaling." Rather than running every Cache Service on every node in the cluster, services are started on-demand in order to supply additional capacity when required by an application with dynamically changing storage requirements.

This technique has an additional benefit that can be used effectively when monitoring large clusters with many caches.

Copyright © 2009 Sherrill-Lubinski Corporation

A service that supports many small caches can be started on only a subset of the available nodes, significantly reducing the number of Cache and Storage MBeans that are created.

In the table showing the total MBean count for our sample cluster, the Cache MBeans make up the largest percentage of the total. This count is the product of the storage nodes, services and caches running on those nodes. Reducing any one of these multipliers dramatically reduces the total.

For example, if Service B in the example supported caches that were relatively small (object count and memory size) and were not “near” caches, it might be started on only 10 storage nodes instead of 100. The total count for the Cache MBeans would be modified as shown here:

Table 2 - Calculating Cache MBean Count - Second Example

Component	Formula	Sample	Total
Cache beans (service 1)	C1 * SN1	15 * 100	1500
Cache beans (service 2)	C2 * SN2	15 * 10	150

Now the 7,500 Cache MBeans seen at the start has been trimmed to 1,650, a reduction of over 75%, simply by supporting the second set of caches on a smaller number of nodes and limiting near caches. As long as the number of nodes supporting the cache is adequate to provide data backup and a safe cluster, this technique can be used effectively for reducing overhead in a large cluster. In our example, the number of caches is only 30, but in some installations there may be hundreds of caches and the effect is greatly magnified.

There is a tradeoff in the use of this technique as it increases the complexity of the cluster configuration. However, the value in terms of capacity management and reduction in monitoring load is often worth the extra effort.

V. Overcoming Limitations Of Process Nodes

The MBeans associated with the storage nodes provide the bulk of the metrics available for monitoring in Coherence. In practice, the behavior of the process nodes is equally, if not more, important. An application may be performing poorly, yet the storage nodes all seem to be running fine. In this case, the problem may be in the process nodes, but there is not much information available to help determine the cause. Only one MBean is available on a process node, the Service MBean.

There are, however, several techniques that may be used to help isolate the cause. Three of these are described below.

A. *Define Unique Service for Important Caches*

The first requires a custom service configuration in which one or more important caches are assigned uniquely to their own services. In other words, define multiple services in such a way that only a single cache is run on each. In this configuration, the data contained in the Service MBean is known to be specific to the single cache running on that service. When multiple caches are running on a service, there is no way to know which cache is causing a problem.

The Service MBean contains a lot of useful information, such as CPU utilization for that thread, a count of messages executed on the service (a measure of activity, usually equating to gets or puts), and information about task backlog and available threads. Using this technique, a great deal of information can be determined about the behavior of a single cache being accessed by each specific process node.

The caveat is that there is overhead associated with running multiple services on a node. While it is common to run several different services on nodes, it is not clear what the effect would be of having 30 or more separate services supporting a single cache on each. Clearly, it would be wise to limit the use of this technique to the more important and heavily used caches in an application.

B. *Make Use of Proxy Nodes*

In a Coherence cluster, proxy nodes may be utilized as a way to decouple the processing applications from the cluster itself. A proxy node does no processing itself, but rather acts as a gateway to external applications that communicate with it over a traditional TCP socket. This provides a measure of security in that the external processing applications cannot directly use the Coherence API and exact damage to the cluster.

Proxy nodes offer an additional, almost unintentional, benefit when it comes to monitoring behavior in a cluster. Standard process nodes provide no information about their interaction with the cluster other than the service information described above. A proxy node exists between the cluster and the external process nodes, and as a result can provide useful information about data transfers to and from the cluster.

A proxy node exposes metrics about the quantity and rate of data transfer, as well as CPU utilization for all activities passed through it. Additionally, there is another MBean exposed for every client process that connects to the proxy node, providing yet more information about activity going through the node.

In versions 3.3 and earlier, proxy nodes suffered from a number of performance issues. As of version 3.4, these issues have been addressed and it is possible now to take advantage of proxy nodes as another way of collecting statistics about cluster processing.

For example, multiple proxy nodes could be created, one for each important cache. All access to these caches could be directed through the proxy in order to gather metrics about the data transfer rates and quantities. Again, this is not something that should be overdone. It is simply one more tool available for gathering information about cluster behavior.

Copyright © 2009 Sherrill-Lubinski Corporation

C. *Instrument Client Applications With JMX MBeans*

Last, but probably most important, is a recommendation to take time to instrument the processing applications with JMX MBeans (or some other methodology). There is no better way to obtain performance data than to collect it at the site where is it used and make it available in real-time to a monitoring application like RTView.

One of the most commonly requested metrics is information about the time it takes for a process node to get data from a cache or to put data into it. There are (currently) no MBeans available in the Coherence nodes that will provide this information. It is possible to get information about how long the storage nodes took to perform these operations, but one cannot tell from this which process nodes were affected. Often there are other factors influencing the performance of specific process nodes.

By measuring the time it takes to perform critical operations in the application and exposing this information via JMX MBeans, displays can be created to correlate observed behavior in each process node with metrics available from Coherence. The result can be a highly effective monitoring system that can be used to head off problems before they occur, as well as troubleshoot when something does go wrong.

VI. Other Considerations

There are some very basic configuration options provided by Coherence that are important to keep in mind in order to provide an effective monitoring solution.

For example, associating a unique "member" name with each node in the cluster is crucial to being able to track activity on specific nodes across a node or cluster restart. When nodes are created, they are assigned a unique "ID," but this ID can change from run to run and cannot be used to track activity on the node. The command line option **"–tangosol.coherence.member=NNN"** can be used to assign a unique name that is retained across invocations so a node's activity can be stored in a database, for example, and analyzed over time. Additionally, assigning a machine ID in a similar way can be helpful in clarifying cluster topology.

Several other topics are deserving of complete treatment in papers of their own, and will be discussed only briefly here.

Understanding memory utilization in Coherence clusters is complicated. The number one culprit is JVM memory heap utilization and garbage collection. Memory usage reported by Coherence does not account for garbage in memory and cannot be relied upon to understand true memory consumption. This data must be correlated with JVM metrics on garbage collection in order to get a complete picture of current memory usage, the trend of which can be an important trigger for possible cluster failures.

Additionally, it is often important to understand how memory is allocated and consumed in each cache on each service on each node. This is complicated by the fact that users must configure cache services to report memory in bytes instead of objects; front caches report usage only in terms of objects and thus the memory usage can only be estimated by multiplying number of objects by average object size (obtained from back caches).

A further complication is that Coherence only reports the amount of memory consumed by primary data. One has to calculate the memory consumed by back storage by accounting for the configured backup count parameter. Index data for caches is another problem. There is currently no way to determine precisely the amount of memory consumed by index data associated with a particular cache, and these data can be as large as the data stored in the cache.

Coherence also provides no information about where specific data are located (on which node or partition). A commonly seen issue is related to "hot keys" or specific objects that are accessed heavily. It is not easy to see what these are and to troubleshoot the effects on the cluster of access patterns that are not balanced.

Each of these areas deserves further study so that adequate solutions can be developed to assist Coherence users in the future.

VII. Summary

Oracle Coherence is a superb piece of technology that implements distributed caching. There are numerous mission-critical applications that simply would not be possible or could not perform adequately without the underlying Coherence cache infrastructure. However, there are significant challenges to monitoring Coherence effectively to ensure uptime and performance.

This paper has attempted to demystify the complex JMX MBean schema that is provided by Coherence, highlighting ways that the data produced can be used effectively in a monitoring application.

Additionally, some of the limitations inherent in present versions of the Coherence JMX MBeans were discussed. While there is a lot of information about storage behavior, there is not as much about process behavior. This paper suggested several techniques that can be used to help identify and isolate causes of trouble in the cluster.

References

[1] Oracle - Coherence Knowledge Base, Coherence 3.4 User Guide 2009, http://coherence.oracle.com/display/COH34UG/How+to+Manage+Coherence+Using+JMX

[2] Oracle - Coherence Planning: From Proof of Concept to Production, An Oracle White Paper, November 2008

[3] Oracle - Coherence 3.4 Documentation, Interface Registery Javadocs, http://download.oracle.com/otn_hosted_doc/coherence/340/com/tangosol/net/management/Registry.html

[4] Lubinski, Thomas – Practical Considerations When Instrumenting Applications with JMX, Information Week, July 2008

[5] Lubinski, Thomas – Business Activity Monitoring: Process Control for the Enterprise, www.sl.com, SL Corporation, 2008

Copyright © 2009 Sherrill-Lubinski Corporation