Reengineering Standalone C++ Legacy Systems into the J2EE Partition Distributed Environment

Xinyu Wang, Jianling Sun, Xiaohu Yang, Chao Huang, Zhijun He
College of Computer Science, Zhejiang University
Hangzhou, P.R.China, 310027
{wangxinyu, sunjl, yangxh, hch, hezj}@zju.edu.cn

Srinivasa R. Maddineni
Department of SSGM/Securities Trading IT
State Street Corporation
Boston MA 02110, USA
srmaddineni@statestreet.com

ABSTRACT
Many enterprise systems are developed in C++ language and most of them are standalone. Because the standalone software can not follow the new market environment, reengineering the standalone legacy systems into distributed environment becomes a critical problem. Some methods have been proposed on related topics such as design recovery, the identification of the component, modeling the interfaces of components and component allocation. Up to now, there does not exist a reengineering process for partition distributed environment, which will offer distinct advantages on horizontal scalability and performance over normal distributed solutions. This paper presents a new process to reengineer C++ legacy systems into the J2EE partition distributed environment. The process consists of four steps: translation from C++ to Java code; extraction of components using the cluster technology; modeling component interfaces and partition of the components in J2EE distribute environment. It has been applied to a large equity-trading legacy system which has proved to be successful.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering.

General Terms
Performance, Design, Reliability, Experimentation

Keywords
Software Reengineering, Reverse Engineering, Legacy System, Distributed Environment, Clustering, Component, Interface, Partition.

1. INTRODUCTION
From 1990s, many enterprise systems were developed in C++ language and most of them were standalone. However, enterprise systems are evolving from standalone mainframe systems to distributed systems. Because the standalone software can not follow the new market environment, migration of legacy systems to distributed environment becomes a critical problem in many business fields.

There are two options to migrate legacy systems to new platforms or frameworks [1]. One approach is the complete redevelopment of legacy systems with new functional specifications or use cases. The other is evolutionary migration of current legacy systems. Disadvantages of the first approach include high cost, time consuming, high risk, and the business rules in the legacy system with outdated document, thus evolutionary migration and reengineering based on legacy code are thought as a more suitable solution to enterprises.

The distributed system is more and more based on component technology. So it is necessary to recover the components of the system from the source code. Generally, reengineering legacy systems into the distributed environment includes following issues:

1) Recovery of data information through reverse engineering
2) Identifying the components in the legacy system
3) Modeling the interfaces of the components
4) Allocation of the components based on middleware or other technology

A lot of work has been directed towards the above related topics. Several methods regarding the identification of the object from legacy code were proposed [2, 3, 4]. Ways to extract components from object-oriented code or models were presented [5, 6, 7, 8, 9]. Cho, et al. defined a metrics to measure the components quality [10]. Reussner, et al. proposed the methods of modeling the interfaces of the components [11, 12]. Some solutions [13, 14] were reported to allocate the components. And some processes [15, 16, 17] were used to migrate the legacy into distributed environment. But up to now, there does not exist a reengineering process for partition distributed environment, which will offer distinct advantages on horizontal scalability and performance over normal distributed solutions. Partition framework is proposed initially in the database field [18], and it is also applied for balance of workload [19, 20].

This paper presents a new process to reengineer C++ legacy systems into the J2EE partition distributed environment. The process consists of four steps: translation from C++ to Java code; extraction of components using the clustering technology; the modeling of component interfaces and partition of the components in J2EE distribute environment. It has been successfully applied to a large equity-trading legacy system.
The rest of this paper is organized as follows. Section 2 presents the whole process to reengineer C++ legacy systems into J2EE partition distributed environment. Section 3 introduces a project which is regarded as a reference case for our process. Section 4 provides conclusions followed by suggestions for future work.

2. THE REENGINEERING PROCESS
The process consists of four steps. The first step is the translation from C++ to Java code, which aims to enhance the MoHCA-JAVA framework with a modification flag output. Because some manual work has to be done after the auto conversion, the flag is helpful for programmer to understand where and how the code should be modified. The second step is the extraction of the components using the clustering technology, where the hierarchical agglomerative clustering algorithm for grouping the classes is used. The third step involves the modeling of the component interfaces using some heuristic rules in J2EE environment. Finally partition of the components in J2EE distributed environment will be introduced in detail as the last step. Some design patterns can be applied in this step.

2.1 Translation from C++ to Java Code
Many projects, such as Cappuccino [21], MoHCA-JAVA [22], C2J [23], Jazillian [24], have been carried out to work towards automatic conversion from C++/C code to Java code. But manual improvement and correction have to be done after the initial auto conversion. The amount of manual work mostly depends on the chosen conversion strategies. To minimize the subsequent work after auto conversion, we constructed the auto conversion process based on the framework of MoHCA-JAVA, and the output from Gen++ is extended with modification flags. Fig. 1 shows the conversion framework.

![Figure 1. The conversion Framework](image)

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Comma separator</td>
</tr>
<tr>
<td></td>
<td>Inline functions</td>
</tr>
<tr>
<td></td>
<td>Arrays</td>
</tr>
<tr>
<td></td>
<td>Structures/unions</td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>Signed/unsigned variable</td>
</tr>
<tr>
<td></td>
<td>Type casting</td>
</tr>
<tr>
<td></td>
<td>Function definitions</td>
</tr>
<tr>
<td></td>
<td>Memory management</td>
</tr>
<tr>
<td>Semantics</td>
<td>Reference</td>
</tr>
<tr>
<td></td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td>Enum</td>
</tr>
<tr>
<td></td>
<td>String</td>
</tr>
<tr>
<td>Library</td>
<td>IO</td>
</tr>
<tr>
<td></td>
<td>Threads</td>
</tr>
<tr>
<td></td>
<td>Multiple inherence</td>
</tr>
<tr>
<td>Design</td>
<td>Operator overload</td>
</tr>
<tr>
<td></td>
<td>Templates</td>
</tr>
<tr>
<td></td>
<td>Pointers</td>
</tr>
<tr>
<td></td>
<td>Default constructor / Destructor</td>
</tr>
<tr>
<td></td>
<td>Goto statement</td>
</tr>
<tr>
<td></td>
<td>Exception processing</td>
</tr>
</tbody>
</table>

There are several steps in the conversion process. Firstly the C++ source code is parsed with an Abstract Syntax Tree (AST) by the C front end. Next, the most important step, the AST is explored. At the same time, the converting strategies are applied, and the initial version of the Java code files is output. Then the modification flags used to guide the manual work are produced. Finally, some improvement should be done manually on the Java code with reference to the information generated by Gen++. The conversion strategies can be divided into two categories: one is the preprocess conversion including the headers, global variables wrapper and macro, and the other is common feature conversion. The conversion strategies in the first category should be applied first to facilitate those in the second one. To estimate the difficulty level and workload, the strategy in the second category will be subdivided into four classes: syntax, semantics, library, and design [22]. The syntax category involves the general changes in the source code, such as the replacement of the comma separator. The semantics category comprises all semantic changes, such as variable type and function definitions. The library category translates the function calls to the appropriate functions in Java library. The design category is about redesign of program, such as the changing for the pointer for C++ program. Most of manual work is in this category. Tab. 1 shows the major different features between C++ and Java classified by the four classes.

To minimize the manual work after the auto conversion, modification flags are output to help programmer understand where and how modifications should be made in the code. A modification flag is presented as follows:

```
[File: test Java][Line: 120][Type: Multi-inherence]
[Modification Comments]
```

2.2 Extraction of Component in Java Code
The following issues should be considered when clustering a legacy system into different components [25].
1) The entity for clustering
2) Similarity measures that can be used to clustering the similar entities into components
3) The clustering algorithm
2.2.1 Entity for Clustering

The entities identified in Java programming language are: field, method, class, and package. Package may be used if the legacy system is large and contains a large number of classes, else the use of class is favourable. Here, the regular class, inner class and interface are two main kinds of entities. There are four relationships between classes according to the UML specification [26], which are Dependency, Aggregation, Generalization and Composition. Here we won’t describe the definitions of these relationships in detail.

2.2.2 Component Metrics to Measure Similarity

Lee defined component metrics [9], which include Connectivity Strength (CS) and complexity metrics of components. But in the definition of CS, Lee considered all of the user-defined type parameter as the same weight, which is not true in practice. Here we define a Connectivity Strength with Weight Parameter (CSWP) matrices with consideration of different weight of parameters, which derived from CS [9]. Assume system \( S \) consists of components \( (C_i, C_j, \ldots, C_k) \), CSWP shows how closely components can be defined by the following formula.

\[
CSWP \left( C_i, C_j \right) = \sum_{c_i, c_j \in S} CSWP \left( c_i, c_j \right)
\]

\[
CSWP(c_i, c_j) = \sum_{m \in CSET(c_i)} \sum_{m \in CSET(c_j)} CSWP(m, m) + \sum_{m \in CSET(c_i)} \sum_{m \in CSET(c_j)} CSWP(m, m)
\]

\[
CSWP \left( m_i, m_j \right) = \begin{cases} \Pr_i \text{Count} (m_j) \times w_{pri} + \sum_{r \neq i} w_{abs} & \text{if } m_i \text{ calls } m_j \\ 0, & \text{otherwise} \end{cases}
\]

\( C_i \): A component.

\( c_i \): A class.

\( m_x \): A method.

\( CSET \): Class set in component \( C_i \).

\( MSET \): Method set in class \( c_i \).

\( \Pr_i \text{Count} \): Count of primitive type parameter in method \( m_j \).

\( \text{Abs Count} \): Count of user-defined type parameter in method \( m_j \).

\( w_{pri} \), \( w_{abs} \): The weight value of primitive parameters and some user-defined type parameter.

2.2.3 The Clustering Algorithm

Clustering approaches can be classified as partitional and hierarchical. Hierarchical clustering techniques can be further divided into agglomerative and divisive techniques. In the divisive approach, one starts with one cluster containing all entities and divides the cluster into two at each successive step. The agglomerative approach involves a series of successive mergers.

The approach proposed here uses a hierarchical agglomerative algorithm for extracting component. All classes in section 2.2.1 will initially be put in different components. And then, all parent classes or interfaces are merged into the components where the children are in. If several children have the same parent, the parent class or interface are separately copied to all of component where the children reside. The following algorithm is used, which is an adaptation of [25, 27]:

1. Begin with \( N \) components and compute the CSWP between the initial components.

2. While all CSWP are more than a threshold

   Do

   Find the most strong pair in connectivity.

   Merge the two components into one.

   Recompute the CSWP between components.

   End

The threshold is given by the extractor, who can change the value to obtain a maximum cohesion between the components.

2.3 The Component Interface in the J2EE Environment

Some researchers have proposed the method to identify the component interfaces [11, 12]. Here we specify several heuristics rules on component interfaces in the J2EE environment.

2.3.1 Heuristic Rule 1: Session&Message Facade

Floyd Marinescu proposed the session&message façade pattern [28] in the J2EE environment. In fact, it is a good heuristic rule when we model the component interface. Fig. 2 shows the correlative call between components.

![Figure 2. The Correlative Call between Components](image)

Because the component A, B and C may be deployed in different physical nodes, the network workload is the main problem in this scenario, where session&message façade is a good solution. Fig. 3 shows the session façade. Several remote calls can be merged into one façade call.

Besides the advantage of low network workload, others are low coupling, good reusability and maintainability. Message façade is similar to session façade except that it is an asynchronous mode. Usually a JMS server is used in message mode.
2.3.2 Heuristic Rule 2: Persistent Data

In J2EE 1.4, usually the entity bean is used to interact with databases. There are two types of entity beans [29]. One is Bean-Managed Persistent (BMP), and the other is Container-Managed Persistent (CMP). They are the interface to the database.

To build more portable components, entity beans that support both CMP and BMP are written by separating business logic into a CMP superclass and BMP persistence logic into a subclass. Deployment descriptor settings can be switched between the two modes [28]. Or a third-party map library can be used at deployment time, such as Hibernate [30].

2.4 Partition of the Components in the J2EE Distributed Environment

2.4.1 General Distributed and Parallel Environment

There are two general distributed and parallel solutions in enterprise environment. One is to allocate the component with the algorithm [14,15]. One or more components may be allocated to the same site. The basic rule is to maximize the locality of the data and minimize remote communication between components. Fig. 4 shows the concept model of this kind of allocation. A, B, C are the components of a distributed system.

The other distributed solution is parallel clustering environment [31,32]. This is a symmetric clustering with all components having the same instance in every server, and any cluster member can handle any request. In general, there is a workloader to balance the requests to different servers. That means any request may be routed to any server for processing. No recovery business logic is required if a server fails. After a failure, all transactions have the rollback and the server is restarted. Fig. 5 shows the concept model of this kind of allocation. There may be other nodes which are on hot standby for high availability.

One disadvantage of the first solution is that all requests will go to physical node A, which gives too big pressure to a node. This usually leads to performance problems in the enterprise system. The second solution dispatches the requests to different nodes, which ease the node A. But if there are two requests which have some relationship, the database has to be used for data synchronization and gathering, which will result in big performance problems. For a large enterprise system, it may be impossible to initialize the whole system in one physical node. For example, currently financial trading system is going to global environment, which will include million-level securities to trade in one system. Obviously initializing such a large system in one node is not practicable.
Request 1 is for EJB 1 or EJB 2, and Request 2 is for EJB 3. When component B starts up, it has to subscribe requests to the JMS server with the special partition information.

There are two main advantages for partition framework. The first one is that requests which have a relationship can be routed to the same service unit (partition). For example, in an equity trading system, a client enters a sell order of a security, and another client enters a buy order of the same security. The two requests can be routed to the same partition to trade with the partition framework, not by database synchronization which will have performance problems. The second advantage is to balance the workload of the system not only in request but in the resource used for initialization. It is obvious that partition solution has a good horizontal scalability.

But there is a disadvantage in high availability aspect, comparing with symmetric clustering model. When a node (server) fails, a controller need time to load the partitions in failed node (server) to the other nodes. So there will be a short period when the system cannot serve the request to the partition. If there is no hardware resource limitation, every node (server) can have a hot standby backup, which has a high requirement for the controller. In practice, that should not be used frequently.

2.4.3 Patterns in the Partition Environment

2.4.3.1 Partition Allocation

There are two strategies of partition allocation. The first one is to hash the partitions into the different servers with a hash algorithm. For example, there are three servers (s1, s2, s3) and six partitions (p1, p2, p3, p4, p5, p6). Using the mod (3), the p1 and p4 will be on s1, p2 and p5 will be on s2, p3 and p6 will be on s3. But this is not a good strategy when p1 and p4 are hot partitions which have much more workload than others.

The other strategy is to classify the partitions based on whether the partition will be hot requested, which may be decided by historical information in product environment. And then the partitions are allocated with the classification. If the p1 and p4 are hot partitions, the two partitions will firstly be allocated onto s1 and s2, and then the p2 will be on s1, p3 will be on s2, p5 and p6 will be on s3. The hash algorithm and strategy may be different with the special enterprise application.

2.4.3.2 Allocation of the components which cannot be partitioned

The components which can not be partitioned, such as component C in Fig. 6, should keep an instance in every server to avoid too many remote calls for communication of components. They can work based on the subscription information from the components which can be partitioned. For example, if component C is price-feeder for equity trading, it should be kept an instance in every server, but it is not necessary to every C instance provide market price for all securities. When component B start-up, it should subscribe to C, telling which securities will be served in this server, thus C will only support the market information of these securities, not all.

2.4.3.3 TaskQueue and ThreadPool

Usually the requests for the same partition have to be served in order, so a task queue is necessary, and then a thread can execute the tasks in the queue one by one. But obviously it is not logical to create a thread for every partition, for there may be hundreds of partitions in an enterprise system.

ThreadPool is a good solution to this problem. When the taskqueue in a partition includes request to wait for processing, it will be enqued into Queue-Of-Queue (QOOQ) to get an idle thread. An available thread will dequeue the taskqueue from Queue of Queue and do those tasks in it. Fig. 7 shows the taskqueue and threadpool pattern.

2.4.3.4 Partition Design and Recovery

The number of partitions should not be designed too small, since this will decrease balance ability. But too many partitions will give huge pressure to the central controller, and the performance is not good, either. So usually some partitionable units should be put into a partition. For example, there are 60k securities in a trading system; it is not reasonable to design 60k trading-book partitions. If we design 500 security trading-books in one partition, then that will be 120 partitions, which is suitable for the central controller and balance ability. Designers can give correct decisions based on a special central controller and environment.

When a node (server) fails, the controller needs time to load the partition in failed node (server) to the other node. Thus light initialization of the partition should be implemented to allow fast loading, which can reduce the recovery time.

3. PRACTICE OF THE PROCESS IN A PROJECT

3.1 Project Background

Lattice reengineering project in State Street Corporation (SSC)-USA started in 4th quarter 2001 and will take a phased approach to enhance system reliability, availability, scalability and maintainability. The Lattice equity trading system which was developed with C++ implements an electronic broker. The system...
can cross the trading-orders from clients or route the orders to stock exchange. The phases of project are described in Tab. 2.

**Table 2. The Phases of Lattice Project**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lattice porting and reverse engineering</td>
</tr>
<tr>
<td>2</td>
<td>Lattice refactoring with multithread technology</td>
</tr>
<tr>
<td>3</td>
<td>Reengineering the C++ legacy system to J2EE partition distributed environment</td>
</tr>
<tr>
<td>4</td>
<td>To be determined</td>
</tr>
</tbody>
</table>

Phase 1 was to port Lattice from an old UNIX to another UNIX Platform with good support. With adopting current SSC compliant technology, fixing critical bugs to improve reliability, leveraging the capacity of the new platform to improve performance and rebuilding the documentation, Lattice system was well qualified.

Phase 2 was refactoring Lattice system with modern technology so as to improve its performance, vertical scalability and maintainability. In this phase, multi-thread technology was adopted to improve performance by leveraging SMP capability of the new UNIX platform; Core part of Lattice system was refactored using a loose-couple architecture to improve its vertical scalability and maintainability. Some important business requirements were also implemented in this phase. But the system is still a standalone system.

Phase 3 is to reengineer legacy Lattice system into the J2EE partition distributed environment. This project is named Equity Connect Next Generation (EQCNG). The target system will adopt all current State Street compliant technologies and some leading technologies to substantially improve its performance, high availability, horizontal scalability and maintainability.

### 3.2 Reengineering Process in EQCNG

The reengineering process proposed in the paper has been applied in EQCNG. The legacy system involves 2244 files which include 860658 lines of C++ code. The system has been in existence as a product for more than 15 years. The legacy system has ever been refactored twice, but still stay in standalone model and some core modules lack proper documentation for some reasons. Because the business is going to globalization, the legacy system need be reengineered to improve its performance, high availability, horizontal scalability and maintainability.

#### 3.2.1 Translation from C++ to Java Code

The process introduced in Section 2.1 was used to translate the legacy C++ code into Java code. Because of business changing, some classes in C++ code were removed manually before automatic conversion. Finally we obtained 1875 classes in Java code. Tab. 3 shows the statistics of the conversion.

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature</th>
<th>Class Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Comma separator</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Inline functions</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>Arrays</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Structures/unions</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td>Signed/unsigned variable</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Type casting</td>
<td>1088</td>
</tr>
<tr>
<td></td>
<td>Function definitions</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>Memory management</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Reference</td>
<td>1312</td>
</tr>
<tr>
<td></td>
<td>Constant</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td>Enum</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>String</td>
<td>1012</td>
</tr>
<tr>
<td></td>
<td>IO</td>
<td>915</td>
</tr>
<tr>
<td>Semantics</td>
<td>Multiple inheritance</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Operator overload</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Templates</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>Pointers</td>
<td>1859</td>
</tr>
<tr>
<td></td>
<td>Default constructor / Destructor</td>
<td>1220</td>
</tr>
<tr>
<td></td>
<td>Goto statement</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Exception processing</td>
<td>402</td>
</tr>
</tbody>
</table>

After the initial auto conversion, manual improvement and correction have to be done on some features, such as multiple inheritance in design category. Modification flags were output to help programmer understand where and how modification should be in the code. A modification flag sample of multiple inheritance is presented as follows:

```plaintext
[Source File: Order.hxx/Order.cxx][Line: 56][Type: Multi-inheritance]
[class Order : public KOrder, public PriceUser]
[Comments: the target files Order.java and PriceUser.java should be checked. The PriceUser class in source file was converted into interface.]
```

#### 3.2.2 Extraction of Component in Java Code

The hierarchical agglomerative algorithm was used for clustering, and the class was used as clustering entity. Some building block modules and business modules were obtained with CSWP metrics to measure similarity. It has been proved that CSWP can have the more effective result than the normal CS. Some modules extracted are presented in Fig. 8.
### 3.2.3 Modeling the Interfaces

Session facade and message facade were used as interfaces of the modules in the EQCNG. Because we used JMS to communicate with applications outside of EQCNG server, message facade had the advantage of low network workload. The performance data of EQCNG will be presented in Section 3.2.5. Though there was no remote call between nodes in EQCNG, session façades were still designed to have low coupling, good reusability and maintainability.

CMP was used as the interface with database in three ways. The first way is read-only. Some common data were loaded in this way. The second way is MultiReader-MultiWriter. Some shared data were used in this way. All partition could read or write the data. The third way is OneReader-OneWriter. Because one partition could only be allocated in one node, some private data of the partition were used in this way.

### 3.2.4 Partition Design

IBM Websphere Application Server 5.1 XD-WPF [33] was used in EQCNG as partition distributed environment. WPF can dynamically assign partitions to a server based upon policy script. HA Manager monitors the status of partition in any time. If a cluster member fails, then partitions in the failed member are redistributed across the remaining active members.

Fig.9 shows the architecture design in EQCNG with using WPF.

There is a trading-book for every security as trading plant, and 65k trading books are created. The trading-books are separated in different partitions. There do not exist duplicated trading-books among the servers, but all of ClientSessions and ExchSessions will be created in every server to reduce the remote communication. The BrokerSession and AdminSession are not demonstrated in Fig.9. They use the same pattern with ClientSession and ExchSession. Every server keeps an instance for FeedConnect for market price information. We had a hash algorithm to balance the partition into the different server. When trading-book partitions start-up, they subscribe information to ClientConnect, ExchConnect and FeedConnect, which work based on the subscription information. ClientConnect and ExchConnect subscribe the corresponding topics in JMS server and communicate with clients and exchanges. The FeedConnect is using the client-library of the feed vendor to connect to the FeedServer.

### 3.2.5 Performance Comparison

Two performance test cases were selected for comparison between the legacy system and the target system. One is “order entering” case which can measure the speed of trading instructions entering from client application to EQCNG server. The other test case selected is “matching” which can measure the speed of crossing the buy orders and sell orders.

The test environments of the legacy system and the target system are listed in Tab. 4.

<table>
<thead>
<tr>
<th></th>
<th>CPU Speed</th>
<th>Memory</th>
<th>Harddisk Speed</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy System</td>
<td>2.4G HZ</td>
<td>8G</td>
<td>80M BPS</td>
<td>1</td>
</tr>
<tr>
<td>Target System</td>
<td>2.4G HZ</td>
<td>2G</td>
<td>80M BPS</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 9. Architecture Design in EQCNG
The partition architecture need cluster environment which includes multiple nodes, so we have no way to do the performance comparison test with the same hardware resource. Tab.5 shows the performance test data with the environments in Tab.4.

Table 5. The Performance Test Data

<table>
<thead>
<tr>
<th>Legacy System</th>
<th>Order Entering</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>150–170 orders/s</td>
<td>40–50 matches/s</td>
<td>120–150 matches/s</td>
</tr>
<tr>
<td>Target System</td>
<td>410–500 orders/s</td>
<td>120–150 matches/s</td>
</tr>
</tbody>
</table>

The EQCNG in the J2EE partition environment improved the performance about 200% comparing with the legacy system. EQCNG also had good horizontal scalability. Fig. 10 shows the performance data with the different number of nodes. It can show the good horizontal scalability in EQCNG. The performance was improved almost linearly with adding nodes into the J2EE partition environment until JMS server became a bottleneck.

Figure 10. The Horizontal Scalability in EQCNG

4. CONCLUSION
This paper presents a new process to reengineer C++ legacy systems into the J2EE partition distributed environment. The process consists of four steps: translation from C++ to Java code; extraction of the components with the cluster technology; modeling component interfaces and partition of the components in the J2EE distributed environment, which will offer distinct advantages on horizontal scalability and performance over normal distributed solutions. It has been successfully applied to a large equity-trading legacy system. The process is also suitable for other partitionable C++ legacy systems, such as foreign exchange, bond and fix income financial systems.

In the future work, translation from some C++ code directly to J2EE code can be considered. The design patterns in J2EE partition environment should be enhanced. This process should be applied on the legacy systems in the other business fields as further practice.

5. ACKNOWLEDGMENTS
This work is part of Lattice® Reengineering project which is an attempt to reengineer a large complex legacy system. Lattice Reengineering is funded by State Street Corporation, USA. The project is a collaborative effort between Zhejiang University, China and StateStreet Corporation, USA. Lattice is a registered trademark of State Street Corporation, or its subsidiaries, affiliates, officers, directors or employees.

6. REFERENCES


