Design and Implementation of Self-Repairing Component-Based Software for Stream-oriented Applications

YingHua Guo[1, a], Hang Zhou[2, b]

[1] Sci. & Tech. on Underwater Acoustic Antagonizing Laboratory, Zhanjiang, China

[2] Naval Univ. of Engineering Wuhan, China

[a]yingh_guo@sina.com.cn, [b]zhouhang@163.com

Keywords: component; self-repairing; stream-oriented; framework.

Abstract. Component-based software development has emerged as an important method in application development to solve the software-crisis. It is now becoming apparent that the technology cannot respond to such diverse requirements or technical challenges because of its insufficient flexible. In this paper the problem of building a scalable component-based system is addressed by means of dynamic reconfiguration. Specifically, considering the system response time and the throughout as the performance metrics, the performance constraint of system components can be satisfied by using more physical resources and a self-repairing component-based software for stream-oriented applications has been designed and implemented. At the end, an example is carried out to validate the design. Experimental results show that an application's properties can be scarified by adopting this design approach and dynamic properties can be achieved by dynamically adjusting components on demand.

Introduction

Component-based software development (CBSD) has emerged as an important technology in modern software development largely because it is a reused approach to defining, implementing and composing loosely coupled independent components into systems. Consequently, it has been considered as one of the most feasible methods to enhance the software production efficiency and quality, and then to solve the software-crisis. With in-depth research, an amount of component models including CORBA (Common Object Request Broker Architecture), COM (Component Object Model), .NET, and the Java-based series of technologies, including RMI (Remote Method Invocation) and EJBs have been increasingly extensive. However, Traditional component has the philosophy of "black-box" reuse, which hides the heterogeneity of underlying operating systems, networks and programming languages [1]. Though this reuse frees the developers from dealing with the heterogeneity, it is now becoming apparent that traditional component-based software development technologies cannot respond to such diverse requirements or technical challenges in a wider range of areas, such as real-time systems maintained by developers of most existing component.

Real-time systems that process large volumes of streamed data can be naturally expressed as Stream-oriented applications with timing constraint. The latency in such applications is a specific performance metric because it impacts directly on reliability and performance of the system. A stream-oriented application can many times be considered as performance aware system which means a mismatching between hardware and software may be appearance completely from the variant runtime status of software even the platform configuration is under a constant condition. In fact, the performance is degraded for the maladjustment of the software granularity and the hardware resources. One commonly used solution is based on performance aware reconfiguration, whose goal is the reduction of the system response time through a performance-aware degradation of the application, driven by the solution of performance models at runtime [1]. However, there are situations such as radar and sonar equipment where this approach is not practical. Not every system can scale by simply degrading certain performance. Another approach is based on dynamic scalability using more physical resources which meets the problem of reacting quickly to spikes in the

workload, as allocating new resources and starting new application instances is not instantaneous. Current efforts to solve this problem are focused on how the component is able to make observations, take a decision and execute a reaction previously defined. What is exactly observed, how decision is made and which actions must be performed to execute the reaction are closely related to the component and the goal given the dynamic adaptation. There are generally two approaches to implementing the adaptation: parameter adaptation and compositional adaptation [2]. Parameter adaptation modifier program variables that determine behavior, compositional adaptation exchanges algorithmic or structural system components with others that improve a program's fit to its current environment.

This paper introduces a self-repairing component framework for stream-oriented signal processing applications which has the capability to obverse its status by providing a presentation of its internals, to support the decision analysis and to allow the processing and thread to be dynamically manipulated and reconfigured for satisfying with the response time constraint.

The rest of this paper is organized as follows: Section 2 has a brief discuss on the concepts in self-repairing component-based software system; Section 3 introduces the framework of the self-repairing component; then a description about the results of experimental evaluation of the model is given in section 4; Section 5 presents the conclusions.

Problem Discussion

In the design and implementation of self-repairing component-based software, the aspects of the problem are comprised of three parts: the acquirement of system performance, the decision of self-repairing strategy instauration strategy and the realization of the project [3, 4].The first part is how to obtain the computional latency and the communitation latency during the running timem, then evalutor the throughout. The second part is how to optimize the performance, the problem is concerned with the study of enableing the platforms dynamically reconfigurable to respond to changes their environments and improve the flexibility and adaptability of component. The last part is how to dynamically reconfigure the sources such as the redistribution of process and threading.

Design and Implement of the Self-Repairing Component-Based Software

Real-time constraint is a necessarily non-functional indicator to be considered that mainly depends on the longest delay of pipeline stage.

We assume:

Throughout: the throughput of pipeline, it is generally less than the given threshold;

Dataset: the data set of pipeline, the value is commonly fixed for a specific system;

Latency: the response time of pipeline, the time interval between the input data and the next one.

Then, the calculation formula for the pipeline throughput

$$Throughout = \frac{Dataset}{Latency} \tag{1}$$

The formulation show that the Throughout is dependent on the Latency.

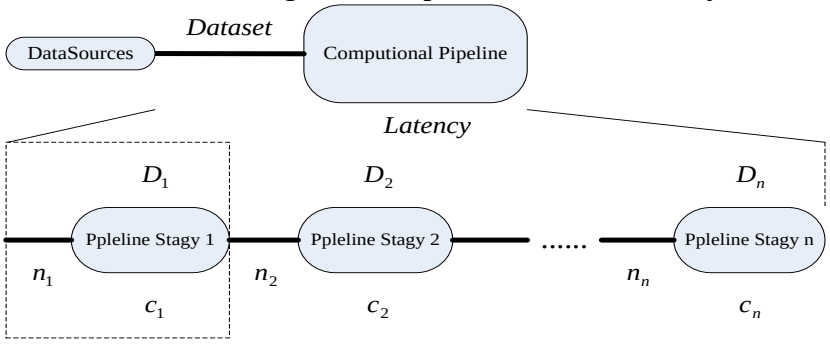

Fig.1 Throughout model of pipeline

Figure 1 shows the throughout model of pipeline [5].The symbol in the figure is defined as:

Di: the data set of the pipeline stage i, that may increases or decreases dictated by the certain signal processing algorithm

Ci: the computational latency of the pipeline stage i, which is determined by the algorithmic complexity

ni: the communication latency of the pipeline stage i, that is concerned with the network topology, bandwidth and so on.

According to the formula (1), the throughput of the pipeline stage i is as follows:

$$Throughout_i = \frac{D_i}{c_i + n_i} \tag{2}$$

Note that the processing time of a dataset in each pipeline stage should be approximately equal to ensure the pipeline is in balance, otherwise a bottleneck is created. This implies that when

$$(c_1 + n_1) \approx (c_2 + n_2) \approx \ldots \approx (c_n + n_n) \approx Latency = \frac{Dataset}{Throughout} \tag{3}$$

the pipeline is stable. It seemed that the emphasis on performance evaluation is computational and communication latency of the pipeline stage i. According to the evaluation result, a judgment whether a performance fault has been occurred or not can be made.

The Measure of the runtime states and internal behaviors. As mentioned, the important criterion that must be met for a pipeline to be implemented is the computational latency of each computational component. The Measure of the runtime states and internal behaviors process is consists of the following steps:

The acquirement of component latency : the technique is to obtain the runtime of the circulation function, which consisting of three portions: data receiving, data processing and data sending. The timer function is setting by two functions. The setTimeA() function recording when the process commences and setTimeB() function while it completes. The difference is considered as the component latency, and is transmitted to the Threading Controller and Manager Controller in the performance parameter distributed stage.

On-line calculation of the pipeline's throughput measurement: As well as the computational latency, throughput is a general criterion to take into account. When processes are distributed in the same node, the data transferred in memory while the inter-process communication is relied on the internet when the MPI process is located in different cluster nodes. Therefore, the throughput of the MPI process has some relationship with the data flow in different network hardware. The measurement of MPI process throughput is then transformed into the measurement of network flow problem.

The acquirement of network information in the single node is depended on the calculating formula of the network throughputs:

$$\text{Network Throughput} = \frac{\text{the amount of network data per unit time}}{\text{time interval}} \tag{4}$$

In the Linux operating system, the network information can be obtained though reading system information files. The most important information are total bytes either received or sent by the web service.

The source code following is to obtain the necessary information:

```
if((fp = fopen("/proc/net/dev", "r")) == NULL) //open status files
scanf(context, "%s %s %s %s %s %s %s %s %s ", RecvBytes, info1, info2, info3, info4, info5, info6, info7, SendBytes);
long recv = atol(RecvBytes);       // Convert the bytes received
long send = atol(SendBytes);       // Convert the bytes sended
long bytes = recv + send;
int load = (bytes - bytes_last) / 1000; // Get network traffic through calculation, unit (KB/s)
```

bytes_last = bytes;

The global information of clusters' network throughput is contributed by each node in the cluster system with the MPI_Gather () function. The method adopted is the main gthering process running in the master node receives message of external nodes while the external nodes run gthering process for collecting the network status and transmits those message to the main process.

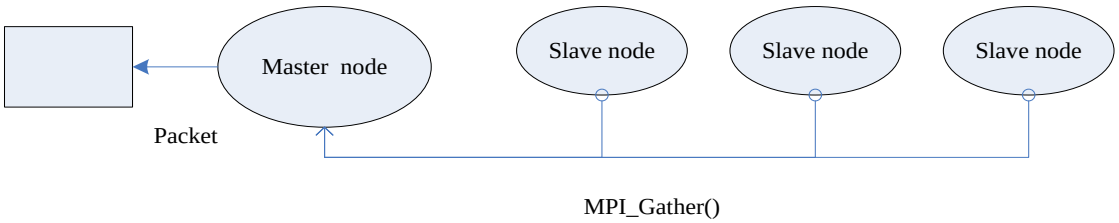

Fig 2 Data collection

Composition of performance Fault-Tolerant manager. The performance fault-tolerant manager contains all mechanisms that can be defined independently of the content of the service functional components. The Evaluator component decides whether the system should be dynamically reconfigurable or not with the observation and manipulation of the runtime states and behaviors internal of platform. The Frequency Controller component prevents the long-lasting instability status of the system by limiting the over-frequency reconfigurable. The Planner establishes a suitable plan given to the executor.

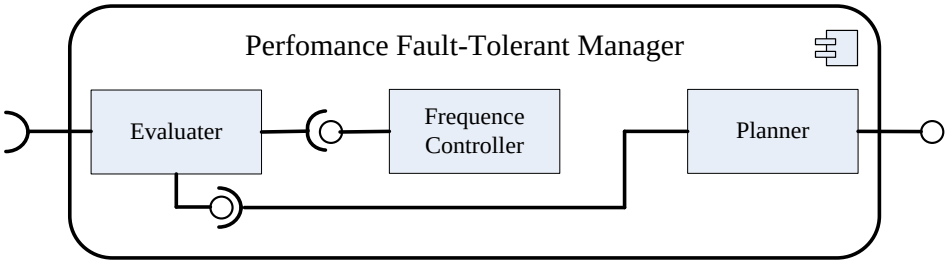

Fig.3 Component graph of performance Fault-Tolerant manager

The relocation strategy implementation. According to the above analysis, the Performance Fault tolerant is a two-level process. The relocation of threads is self-controlled under the preinstalled parameter in each computational component. The relocation of process is centralized controlled by the performance Fault-Tolerant manager. The concrete realization is introduced in the following discussion.

The relocation of the threads：When the computation component is initialized, the granularity has been described using <granularity> in the XML files.

```
<thread number="1">
    <inc>30</inc>
    <dec>15</dec>
</thread>
```

The <inc> represents the upper bound of the latency for increasing the threading and the <dec> denotes the low bound for decreasing the threading. The trigger level can be modified by the self-repairing manager. Such certain code display as follow：

```
ThdsRelocate(cid)
{
    //relocate the threads，cid : a unique number for the link
     xmlr.appList.component[cid].granularity_thread_inc = 20;
     xmlr.appList.component[cid].granularity_thread_dec = 10;
}
```

After determining the limit value, the chkNumThds () calculates the delay time parameters for judging whether thread is relocated or not.

The relocation of the process：The decision algorithm of process relocation is in the management interface and the algorithm of thread relocation is performance fault-tolerant manager .

when the computation component is initialized, the granularity has been described using < process > in the XML files.

```
<process number="1>
    <node>1</node>
</process>
```

This indicates that the component is a single process configured in the No.1 node of the cluster. And the process granularity information is modified by the performance Fault-Tolerant manager

```
ProcRelocate() {                    //relocation of process，cid: component serial number
    xmlr.app.co[cid].granularity_process_number = 2;
    xmlr.app.co[cid].granularity_process_node.push_back(hid);
    Info Convert();
}
```

After the code execution, a replication of orginal component is required and placed in the hid node.

Example and Analysis

A test is performed on a Linux Cluster with 5 Blades. Every blade, also known as a node with IBM HS21, contains two Intel Xeon E5450 processor CPUs that each has 8GB of memory and 4 cores. The software platform includes Red Hat Linux Enterprise 5 and Intel C Compiler 11.083. This test is just a simple signal processing example to verify this scheme's feasibility by measuring the run-time states of real-time components in a dynamic setting. As Figure 4 shown, it contains four periodic real-time components—COM1, Com2, COM3 and Com4 and the component graph is naturally expressed in terms of a pipe-and filter paradigm.

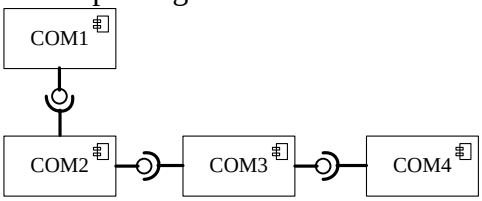

Fig 4 the component graph

COM1 sends volumes of data per 4ms. COM2 receives that data and transmit it when it completes the processing in 2ms. COM3 implies that tuple in a certain time which is depended on a parameter noted as α, Figure 5 illustrates the plot of execution time as a function of parameter α. COM4 just displays the results.

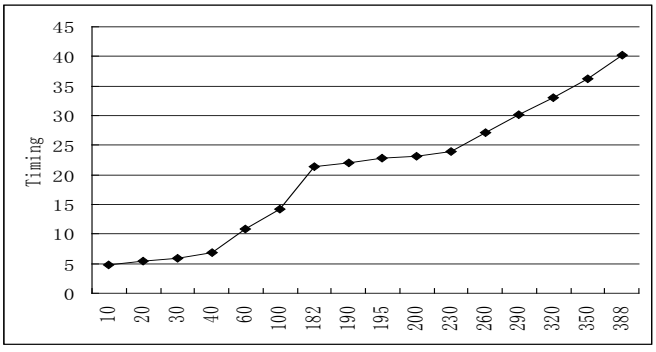

Fig 5 Relation between parameter α and execution time

Because the time increases as the accretion of α, if we assume the timing constraint of the system is 23ms, it suggests that when α is greater than 210, COM3 would not achieve the non-function capability for its excess execution time. A reconfiguration should then be taken place: a replication of COM3 node as COM3_1 is required and placed in a free node.

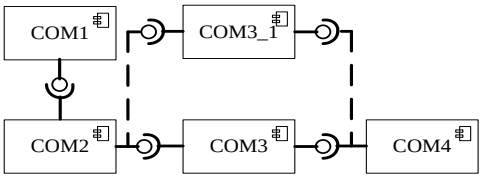

Fig 6 the component graph after reconfiguration

Figure 7 displays the utilization factor of CPU operation is approximately linear decreased along with the increment of thread number, when the thread binding is used. When α is less than 210, the system matched well with the timing constraint. The utilization factor of CPU in original node is smoothly increased from 8% to 10% while the expand node remained 0%. When α is more than 190， the reconfiguration occurred in the running time. The replication of COM3 is placed in the expand node which cause the utilization factor of CPU in expand node is changed from 0% to 8%, while the original node decreased from 10% to 8%.

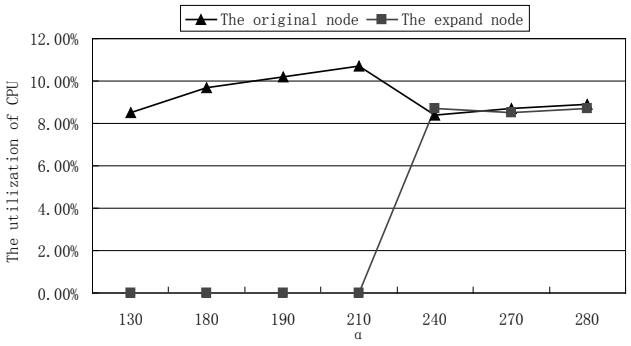

Fig 7 Utilization factor of CPU in original and expand node

Summary

This paper designs a kind of component framework of signal processing applications, suitable for the self-repairing component-based development. The reusable model matches the characteristic of stream-oriented application. It is a particular configuration of components that can be selected at reconfigured during runtime for performance aware. The experiments results indicate that our design approach is feasible; it is convinced that the approach provides he agility that applications require.

References

[1] Rajkumar Buyya, “High Perfomance Cluster Computing: Architectures and Systems” vol. 1.Pearson Education, 1999, pp. 661-663

[2] R. Sudarsan, C.J. Ribbens, Efficient multidimensional data redistribution for resizable parallel computations, in: Proceedings of the International Symposium of Parallel and Distributed Processing and Applications (ISPA ’07), Niagara falls, ON, Canada, 2007, pp. 182–194.

[3] G. Henkelman, G.Johannesson and H. Jónsson, in: Theoretical Methods in Condencsed Phase Chemistry, edited by S.D. Schwartz, volume 5 of Progress in Theoretical Chemistry and Physics, chapter, 10, Kluwer Academic Publishers (2000).

[4] U. Aßmann, Invasive Software Composition, (Springer, 2003).

[5] M. Korch and Th. Rauber, Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining, J. Par. and Distr. Computing, 66, 444–468, (2006).