Runtime Compositions and Optimizations of Parallel JavaBean Programs on Clustering Environments *

Cheng-Wei Chen Chung-Kai Chen Jenq Kuen Lee
Department of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan

Abstract *In this paper, we describe a on-going research work to attempt to provide a software infrastructure for the runtime composition and optimization of parallel javabean programs. Our system includes several essential components. First, a software module called, component annotator, is employed. An application-level javabean program (component) can be annotated to produce trace information at runtime. This information can be used for further program optimizations. Second, application components are integrated with a container component proposed by us before execution. In the runtime execution, the container component can act as an active program unit to analyze the trace information, perform component optimization, produce the new version of a chosen application component, and perform runtime composition of application components by replacing the original application component by the new component. Our system works on the component level, deals with parallel optimization issues, and optimize the programs by utilizing application characteristics. In an application scenario, our system can be used to replace a component from parallel javabean programs by a new version of component at runtime for efficiency. We illustrate such an optimization for a fluid application component with unstructure grids and sparse computations on IBM SP-2. We think our system gives an important framework and direction in the optimization of parallel Java components at runtime.*

*This paper is supported in part by NSC of Taiwan under grant no. NSC 89-2213-E-007-049 and by MOE/NSC program for promoting academic execellence of universities under grant no. 89-E-FA0414.

Keywords: Parallel JavaBean, Runtime Component Builder, Container and Adaptor, Bytecode Optimizer, Clustering Environments.

1 Introduction

In this paper, we report an on-going work to provide a software infrastructure for the runtime composition and optimization of parallel javabean programs. Our system includes several essential components. First, a software module called, component annotator, is employed. An application-level javabean program (component) can be annotated to produce trace information at runtime. This information can be used for further program optimizations. Second, application components are integrated with a container component proposed by us before execution. The container component in our environment mainly consists of two software modules, the component adaptor and component optimizer. In the runtime execution, the container component can act as an active program unit to analyze the trace information, perform component optimization, produce the new version of a chosen application component, and perform runtime composition of application components by replacing the original application component by the new component. The component optimizer will help on the trace analysis, component optimization with trace information, and generating the new version of an application component. The component adaptor will help the runtime composition of application components by replacing the existing application component by a new version.

In contrast to JIT compilers, our system works on the component level, while JIT compiler works on the applet or bytecode levels. Our system is also designed mainly to deal with parallel issues and to optimize the programs by utilizing application characteristics, while the JIT compiler mainly optimize bytecodes without considering much the performance effects of parallel issues. In addition, our system works on the optimization on application characteristics and parallel issues by replacing a component bytecode with a more efficient component bytecode. The JIT compiler works on translating a bytecode program into native codes. In our design, our system can work with a JIT compiler coherently. For example, after the composition of a new component into the application component, in the resumed execution, the JIT compiler can be employed again to work on the newly optimized bytecode programs.

To illustrate the effect of our proposed system, we consider the following application scenario. Suppose we have a fluid application with unstructured grids. The application include a graph partition component, a component for solving fluid equation, and a component for a linear system solver. In the program execution, initially, the graph bisection component adopted to partition the unstructured grids on distributed memory parallel machines, IBM-SP2. It then adopts the component to perform stencil pattern (SIMPLE) computation with the finite volume scheme. After the stencil pattern computation, it then executes the component to solve a linear system. Throughout the computation, the trace information and performance statistics are collected and the optimizer in our proposed container component work actively to attempt for optimizations. Suppose, in the original component for solver, a dense linear system solver was employed. Throughout performance profilings, our analyzer can figure out a sparse linear solver can best utilize the system performance for the solver of unstructured grids. The adaptor in our container component can be used to re-compose programs with a sparse version. Currently, in our design we also support the application description file to be associated with the component. The adaptor and container can use this information to choose the proper component for replacement by consulting the runtime trace information and the application description file. Our system is really one-step towards the automatic compositions of parallel javabean programs with the considerations of parallel optimizations, component optimizations, and application characteristics. In our work, we will illustrate the preliminary effects with the application of our environment for a fluid application component with unstructure grids applications on IBM SP-2.

The remainder of the paper is organized as follows. Section 2 gives the overview of our runtime composition and software environments. Section 3 gives some technical information for conducting runtime component compositions. Next, Section 4 describes preliminary performance results. Finally, Section 5 concludes this paper.

2 Overview of Software Infrastructures

In our system, a software component can be first annotated with our annotator so that it can produce some forms of trace information at runtime for optimization purposes. Figure 1 illustrates such a idea. The application component is also integrated with a container component proposed by us. After the annotation, the program can be executed, and a series of state transitions can be done for optimization purposes. Figure 2 illustrates the state transition of the program exeuction of a parallel javabean. First, the program is executed. Two program units are actively executed, the application javabean and the container component. The container component consists of two modules, the component optimizer and the component adaptor. In the next state, the execution of an application component is producing profiling and performance traces, and the component optimizer gets started to analyze the trace information for optimizations. In the third state, the component optimizer is producing a new version of one component from the application javabean programs. In our example, the component C (say Bean C) is analyzed and a new version C' (say Bean C') with better efficiency is produced. In our design, a new version can be a refinement of the original component by the optimizer, or it can be picked up from the specification pool by picking up a component with the same functionality, but with better efficiency. For example, if the profiling indicates the data file for Bean C is really sparse data set, a new component which works best for sparse dataset can be chosen for the new version. In addition, a new version fits best on an architecture can also be chosen by the optimizer for re-composition of application components, a communication component does well on an idle network (of a dual-connection network) at the execution time will be chosen, or a new release version of a component of the application components can be chosen for the re-compositions.

In the fourth state of Figure 2, the adaptor of the container component gets started to replace the old component by the new component. This needs to be done with the help of serialization, as this is done at runtime. The adaptor works similarly as the javabean visual builder, but it's now at runtime and without visual interfaces. More technical information related to the container will be given next section. Next, it shows the new component is installed at runtime, and then the program is resumed for execution from the previous continuation point.

In our current design, the new version of the component can be obtained in two ways. First, it can be produced directly by the bytecode optimizer in our system as a direct refinement of the original bytecode programs. Second, it can come from a specification file which we call "application description file" associated with each component. The application description file mainly consists of attributes and location

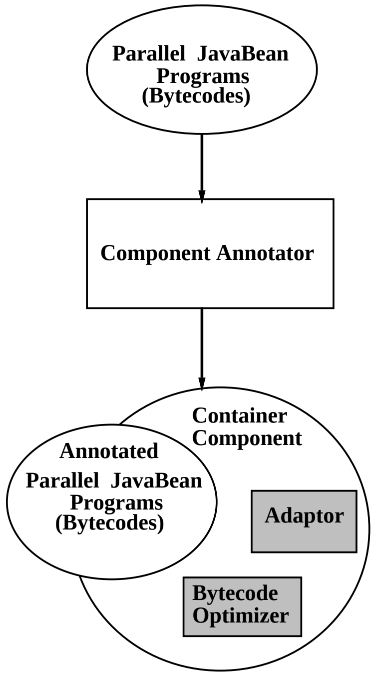

Figure 1: Component Annotations

of component versions. The attribute include a set of keywords related to application and performance characteristic. The set of keywords include parallel, sequential, communication networks, sparse, dense, etc. The location describes where to find the equivalent version of the execution component but with different performance characteristics. The adaptor should pick up the right version specified by application programmers according to performance characteristics and trace information collected and analyzed for the application component.

3 Container Component

Our container component includes two parts, the bytecode optimizer component and the adaptor component for composition of bytecode components. We give the essential idea below.

3.1 ByteCode Optimizer

We are in the process of a bytecode optimizer for for application and parallel optimizations

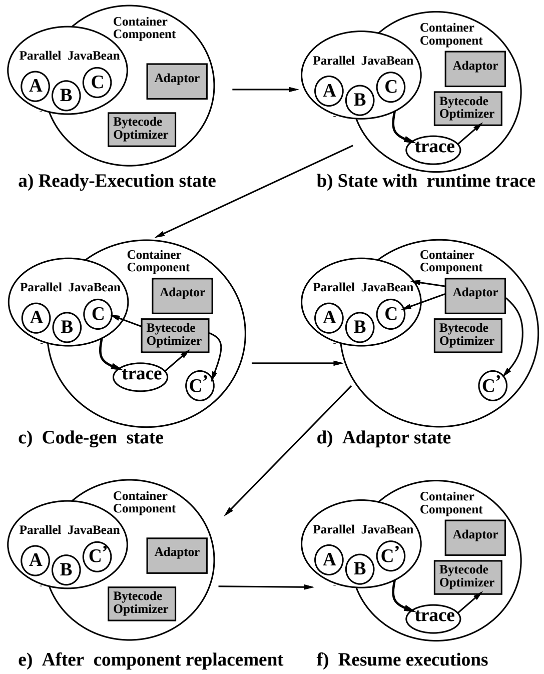

Figure 2: State transitions in runtime composition systems

based on a package called JavaClass API[2, 3]. Our software component uses `JavaClass API` to do the static analysis and dynamic creation or transformation of Java class files. The `JavaClass API` is a general purpose tool for Java bytecode engineering.

In addition, We design the TraceMointer, a Java interface, described below to collect the execution statistics. Every bean can provide the execution information and trace information for the bytecode optimizer in our container. The trace information can be gotten by application component implementing TraceMonitor or by our bytecode annotator to annotate this information into application bytecodes.

```
interface TraceMointer {
  int getTraceInfo(TraceInfo trace)
      {   ...  }
  ...
}
```

3.2 Adaptor

The Java interface AdaptorInterface is the exported interface to deal with the adaptor. As long as the beans implement the interface, the adaptor can invoke the methods of beans to let them serialize into a particular stream or to resume executions from a stream. Thus, the adaptor can choose the suitable bean among the similar beans for replacement.

```
interface AdaptorInterface {
  int performSerialize() {  ...  }
  int performResume() { ... }
  ...
}
```

To make the bean class persistent, we define the class as implementing java.io.Serializable. Here is an example,

```
public class BeanC implements java.io.
           Serializable, TraceMointer,
           AdaptorInterface {
  int getTraceInfo(TraceInfo trace)
      {  ...  }
  int performSerialize() {  ... }
  int performResume() { ...   }
  doWork() {  ...   }
  ...
}
public class NewBeanC extends BeanC {
  int getTraceInfo(TraceInfo trace)
      { ...  }
  int performSerialize() {   ...  }
  int performResume() {  ...  }
  doWork() { ...  }
  ...
}
```

In the code above, the component BeanC illustrates how an application should be implemented to work in our system for runtime component compositions. The component BeanC gives an example for the component to be replaced in Figure 2. Similarly, the component NewBeanC gives an example for the new component of component C looks like in Figure 2. The component NewBeanC can be gotten by the bytecode optimizer described above, or by the application description file associated with an application component.

4 Preliminary Experiments and Discussions

As mentioned earlier, the benefit of our system can be illustrated by the application scenario. Suppose we have a fluid application with unstructured grids. The application include a graph partition component, a component for solving fluid equation, and a component for a linear system solver[14]. This is also a joint work with Power Mechanics Department, Tsing-Hua university, to develop scalable methods concerning the the parallelisation of pressure correction method on unstructured grid and large sparse systems[14]. In the program execution, initially, the graph bisection component adopted to partition the unstructured grids on distributed memory parallel machines, IBM-SP2. It then adopts the component to perform stencil pattern (SIMPLE) computation with the finite volume scheme. After the stencil pattern computation, it then executes the component to solve a linear system. Throughout the computation, the trace information and performance statistics are collected and the optimizer in our proposed container component work actively to attempt for optimizations. Suppose, in the original component for solver, a dense linear system solver was employed. Throughout performance profilings, our analyzer can figure out a sparse linear solver can best utilize the system performance for the solver of unstructured grids. The adaptor in our container component can be used to re-compose programs with a sparse version.

Below we illustrate the performance benefits in transforming a dense linear system solver into a sparse solver when in fact the dataset is a sparse dataset. For example, in our experiment on IBM SP-2 with conjugate gradient solver, we illustrate one example dataset for parallel java programs on IBM SP-2. The dataset is a sample from the toolbox of Matlab and a well-known test case[11, page 352, Fig. 4][1]. Potential benefit exists in such a sample example. For example, if the system uses a sequential dense java component, it will need to consume 22.6 seconds for just 10 iterations. However, if according to the trace information, the proposed container figures out that the data set is a sparse dataset, and replaces the dense solver component by a sequential sparse dense solver. The performance in the rest of iterations can be improved by almost 9 times. The performance can be further improved if the composer is to compose a parallel sparse component by replacing the original sequential and dense component.

Table 1: The time (seconds) spent for conjugate gradient solver with 10 iterations for 479 $\times$ 479 case on IBM SP-2.

Procs.	Dense Matrix Class	Sparse Matrix Class
1	22.6	2.6
2	11.8	2.1
4	6.5	1.7
8	3.9	1.6

Next, we consider the application component with data-intensive applications. We consider the application related to OLAP problem and group-by operations[12, 13]. The relationships among the arrays in group-by operations of OLAP can be represented as a lattice. Performance benefits and effects can also be achieved similar to the previous example. Suppose, in the original component for group-by operation, a dense data cube and operation were employed. Throughout performance profilings, if the analyzer finds out that the dataset is quite sparse, a sparse component can be re-composed at runtime. In OLAP problem, it's mainly dealing with cube data information, and quite a few of them are large sparse data set. In addition, a parallel version can be composed at runtime as well if the trace information indicates that the program is a longlasting job and more processors are available during the executions. Our early work with experiments done on Fortran 90 environments also illustrate the benefits in replacing a sparse program for a desne program when the dataset is actually sparse[9, 10]. However, the previous result is done on Fortran 90 and done at static environments. With parallel javabeans and our component optimizer, we hope to illustrate the effects done on Fortran 90 for parallel Javabean programs and to do it at runtime at component levels. We are in the process of converting our early application programs on Fortran 90[10, 9] into parallel java programs to work with our framework.

5 Conclusion

In this paper, we describe a design for a software infrastructure for the runtime composition and optimization of parallel javabean programs on clustering environments. Currently, we are implementing and evaluating this technologies with parallel Javabean programs on IBM SP-2 and network-based PC clusters. Our system works on the component level, deals with parallel optimization issues, and optimizes the programs by utilizing application characteristics. In an application scenario, our system will be able to automatically replace a component from parallel javabean programs by a new version of component at runtime for efficiency. We also illustrated the application scenario for a fluid application component with unstructure grids and for OLAP applications on IBM SP-2.

Acknowledgement

We would like to thank Dr. Tyng-Ruey Chuang at Academia Sinica, Taiwan and Rong-Guey Chang in our lab for many suggestions on sparse optimization issues and applications.

References

[1] Rong–Guey Chang, Cheng–Wei Chen, Tyng–Ruey Chuang, and Jenq Kuen Lee. Towards automatic supports of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience*, 9(11):1101–1111, November 1997.

[2] Markus Dahm, "Byte Code Engineering with the JavaClass API", *Technical Report B-17-98*, Freie University Berlin, Institute for Information, 1999.

[3] Markus Dahm, *JavaClass API*, Freie University Berlin, This package is made available at `http://www.inf.fu-berlin.de/~dahm/JavaClass/index.html`.

[4] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[5] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley, 1999.

[6] Jon Meyer and Troy Downing, *Java Virtual Machine*, O'Reilly, 1997.

[7] Bill Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 1998.

[8] Rong–Guey Chang, Cheng–Wei Chen, Tyng–Ruey Chuang, and Jenq Kuen Lee. Towards automatic supports of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience*, 9(11):1101–1111, November 1997.

[9] Rong–Guey Chang, Tyng–Ruey Chuang, and Jenq Kuen Lee. Efficient Support of Parallel Sparse Computation for Array Intrinsic Functions of Fortran 90, ACM International Conference on Supercomputing, Melbourne, July 13-17, 1998.

[10] Rong–Guey Chang, Tyng–Ruey Chuang, and Jenq Kuen Lee. Compiler Optimizations for Parallel Sparse Programs with Array Intrinsics of Fortran 90, International Conference on Parallel Processing, 1999.

[11] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333—356, January 1992.

[12] Sanjay Goil and Alok Choudhary. High Performance OLAP and Data Mining on Parallel Computers, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[13] Mahesh V. Joshi, George Karypis, and Vipin Kumar. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[14] C. M. Lin, Yue-Chee Chuang, Jenq Kuen Lee, K. L. Wu, C. A. Lin, *Parallelizing Pressure Correction Method on Unstructured Grid*, Parallel CFD '98.