Decomposition of Object-Oriented Programs for Fault Tolerant Computing in Distributed Environment

Shih-Min Sheu*, Jan-Ming Ho†and Jie-Yong Juang‡

Abstract

This paper describes a program decomposition scheme that decomposes a C++ program into a set of concurrent tasks to support fault tolerant computing in distributed environments. This scheme consists of a C++ analyzer for program decomposition and a set of mechanisms to perform run-time object backup and error recovery. The analyzer uses a weighted-object approach to decompose an object into a set of concurrent tasks or combines a set of objects into a single task. The analyzer guarantee the resulting decomposition will not violate system resource constraints. A reliable message passing protocol is also generated by the program analyzer for remote object references.

Run time supports for fault tolerant computing based on the decomposed objects are discussed, which consist of an interface, a reliable message passing mechanism and a recovery mechanism. The message passing mechanism updates backup storage whenever a message is passed to avoid *Domino effect*. The recovery mechanism is to restore the computation from log file in a stable storage, or restart the task by using the spared copy in other node at the presence of hardware failures.

1 Introduction

Fault tolerant computing is a major design issue in a distributed environment. Two widely used basic techniques are program and/or data replication, and checkpointing [1, 2]. For fault tolerance, program data are usually replicated on several sites. The system maintains the consistency of these replications on runtime. Once a failure occurs, the error recovery subsystem will either mask the faulty node, or switch the control to other sites if necessary to resume normal computations. In checkpointing approach, the program checkpoints periodically or at the occurrence of a set of designated events. The checkpointing operations backup the necessary program context into a non-violate storage. Once a fault is detected, the program can be resumed at the last checkpoint by either restoring the backup information or restart the executions with the initial program image and the backup data on the other site.

Both approaches are straightforward conceptually, but difficult to implement efficiently due to the overheads of the backup operations. For example, the number of synchronization messages required for maintaining n replicated copies in a distributed environment is $O(n^2)$. It brings even more overheads if the checkpoint operations backup the entire program context. Efficient strategies should be provided to avoid communication and storage overhead. One solution to reduce the overhead dramatically is for the system to backup only the modified portion of the program context. However, it is hard to decide which part of the program context is modified for programs written in conventional programming languages. Unless with the support of paging hardwares, due to the complex data dependency and control dependency of a conventional program, it is very hard for a computer system to detect if a portion of a program has been modified.

On the other hand, object-oriented programs allows more efficient management. An object-oriented program can be viewed as a set of program entities, i.e., objects, class methods and data instances. Decomposition techniques can be used to decompose an object-oriented program into several tasks each can be scheduled independently and can run on different machines. The boundary of decomposition is determined, e.g., by the desired data granularity such that the program entities of each task are made smaller and more manageable. Invocation of a particular object also marks the object as being dirty. The checkpoint operation backups only the dirty objects. This is a way of minimizing the traffic invoked by the checkpoint operations.

This paper describes a program decomposition scheme that decomposes a C++ program into a set of concurrent tasks to support fault tolerant computing in distributed environments. This scheme consists of a C++ analyzer for program decomposition and a set of mechanisms to perform run-time object backup and error recovery.

The analyzer uses a weighted-object approach to

*S.-M. Sheu is with Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208. Part of his work is done while visiting Inst. of Info. Sciences, Academia Sinica,supported partially by NSC 81-0408-E-001-06.

†J.-M. Ho is with Inst. of Info. Sciences, Academia Sinica, Nankang, Taipei, Taiwan, *e-mail:* hoho@iis.sinica.edu.tw

‡J.-Y. Juang is with Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, *Fax:* 886-2-3628167, *Phone:* 886-2-3628009, *e-mail:* juang@csman.csie.ntu.edu.tw

0-8186-2755-7/92 $03.00 © 1992 IEEE

decompose an object into a set of concurrent tasks or combines a set of objects into a single task. A weight is assigned to each object to determine whether the object and its derived-class objects need to be decomposed or combined. The weight is calculated by the analyzer according to a set of guidelines such as backup load or reference frequency. The analyzer also guarantee the resulting decomposition will not violate system resource constraints. The constraints currently used include memory size, number of messages, CPU time, and the maximum expected recovery time. A reliable message passing protocol is also generated by the program analyzer for remote object references. The protocol maintains a consistent global state for recovery, enforces the synchronization of concurrent tasks executions and passes parameters between class instances.

A bottom up strategy is used in the analyzer to determine the checkpoint locations for each resulting task. The analyzer first scans the declaration of the classes to build up the class hierarchy, then estimates the computation time and the storage size of the tasks to obtain the checkpointing overhead. Based on this information a placement of checkpoints can be determined. In addition to compiler aided checkpointing, a run time checkpointing mechanism with "dirty" objects list is also provided to reduce the size of backup storage. Finally, run time supports for fault tolerant computing based on the decomposed objects are discussed, which consist of an interface, a reliable message passing mechanism and a recovery mechanism. The message passing mechanism updates backup storage whenever a message is passed to avoid *Domino effect*. The recovery mechanism is to restore the computation from log file in a stable storage, or restart the task by using the spared copy in other node at the presence of hardware failures.

This analyzer first analyzes the data dependence, inheritance hierarchy and control dependence of the target program, and uses a set of transformation mechanisms to replace some statements which are associated with partition relation semantics. The transformation breaks the program into a set of decomposed entities, a number of disjoint independent programs. Those programs are compiled with runtime support primitives, and can be scheduled for executions in a distributed computing environment. The decomposition scheme described here is different from the techniques of decomposition from specifications[3, 4]. The latter emphasizes the decomposition that uses the object-oriented methodology of the specifications.

This paper is organized as follows. In section 2, basic concepts and definitions are given. In section 3, we use an example to present a detailed description of our analyzer based on data member decomposition. In section 4, we discuss more decomposition schemes. In section 5, the implementation issues are discussed. Concluding remarks are given in section 6.

2 Basic Concepts and Definitions

This section, we present a model to describe the run time behavior of C++ programs. The characteristics and interrelation of C++ objects in the run time are explored for the object decomposition by observing the representation of C++ objects in this model.

2.1 Basic Definitions

We can view the run time behavior of a C++ program as a set of objects, a sequence of data references and procedure calls. A graph is used to represent this model. There are three types of the vertices:

- *Calling programs:* a piece of continuous storage which stores the main program or a subroutine.
- *Class methods:* A class method is defined with a piece of code that implements the user requirement. It affects the $< private\, data\, instance >$, $< friend/inheritance/global >$, $< local >$ and $< formal >$ part of the program contexts. These parts are the interactions of data dependence of a method.
- *Data instances:* A data instance can be divided into $< private\ data >$, $< protected\ data >$ and $< public\ data >$. This feature reflects the access control of the data instance inheritance of C++. In this paper, the term ***C++ object, class instance*** and ***object*** are equivalent. The term ***data instance*** emphasizes data members of the class instance , while objects represent both data and codes in a class instance.

and two types of edges:

- *Method invocation:* a directed edge from calling vertices to the called vertices. It represents the traditional caller / caller relationship of the method invocations.
- *Data reference:* a directed edge from the node uses one specific variable to the node that outputs this variable. It can show the relations of lvalue or rvalue references in the program.

The runtime behavior of C++ programs can be represented using this graph model.

Example 1: Figure 1 shows a simple C++ program; class B is a derived class of class A and only one object b in the program. The representation graph of this program is shown in figure 2.

Besides the above representation graph, we also make use of the following data structure in our analysis.

- *USE-DEF set of methods:* we extends the USE, DEF set concept [5] from a statement to a method invocation for C++. The USE, DEF set of a method is defined as:

$$\begin{aligned} USE(m) &= USE(m.I) \cup USE(m.L) \\ &\quad \cup USE(m.F) \cup USE(m.G) \qquad (1) \\ DEF(m) &= DEF(m.I) \cup DEF(m.L) \\ &\quad \cup DEF(m.F) \cup DEF(m.G) \qquad (2) \end{aligned}$$

where $USE(m) := \{v \in \boldsymbol{variables} : v$ is read in the storage of method m $\}$ and $DEF(m) := \{v \in \boldsymbol{variables} : v$ is written in the storage of method m }. $m.I, m.G, m.F$ and $m.L$ represent the instance, global/protect/friend, formal argument and local data of the method m.

- *Class hierarchy graph:* a graph to show the inheritance of classes. Many C++ inheritance features, such as inherited data allocation and class scope, are directly related to this data structure.
- *Object invocation trace:* a sequence of object invocations that are in the same order in the program invocations. We can view an object invocation trace is an instance of program control flows.

The following sections will discuss the decomposition with more details.

3 Data Member Decomposition

In a decomposition scheme, the program is decomposed into several tasks according to the decomposition boundaries specified by users or by daemon processes according to system resource management requirements. After decomposition, the tasks can be scheduled for concurrent executions or be used as the unit for replications to improve the system reliability. The decomposition described here differs from those in specification decompositions[3, 4]. The input to the specification decomposition is a set of application requirements, while input to ours is an object-oriented program. Our work emphasizes the decomposition of runtime structures into tasks of disjoint name space ready for fault-tolerant computing.

For conventional programming languages, decomposition of a program can be performed either by programmer or automatically. In a manual approach, a programmer decomposes his application by using special programming constructs such as ***parbegin*** and ***parend*** statements. A programmer could also perform the same activity by using system calls provided by an operating system. Fork in Unix, and other runtime primitives, such as remote procedure call, are good examples. This approach requires users to have sufficient knowledge of the decomposition details such as run time behaviors and the synchronization of the decomposed programs. However, the programming productivity is limited. The automatic approach performs similar tasks by using a compiler or an analyzer. A programmer writes a sequential program without decomposition. The compiler then scans the target program, analyzes the data and control dependency and transforms the program automatically. Parallel compilers are good examples. However, most existing automatic tools for conventional languages apply only to a limited set of program constructs such as loops.

In this paper, we propose an automatic scheme to decompose object-oriented programs. Data encapsulation and inheritance features of OOPL makes the decomposition boundary clear and can be used to reduce the complexity of dependency analysis. Furthermore, the object-oriented programs are easier to maintain and reuse. For specific application requirements, such as concurrency and fault tolerance, automatic tools may be devised to increase software productivity. In this section, we use a program example to illustrate the necessary tasks for automatic program decomposition with respect to data members in a class instance. In subsection III.A, the formal description of the scheme is presented. III.B shows the decomposition. III.C presents a scheduling scheme for separated executions of the decomposed program. III.D demonstrates how the result is applied to improve the implementation of checkpoint operations.

3.1 The Scheme

A data member in a class instance consists of a set of variables. The data members in a class provide a vehicle for an user to composite data structures in the program. In this section, we present a basic decomposition scheme based on the decomposition of the data members. Given a decomposition of the data members in a class instance into several disjoint sets of variables, the scheme decomposes the object-oriented program without changing the control flow of the original program. The decomposed program can then be partitioned into several tasks and distributed to several sites for concurrent executions or replicated into several copies to improve availability. This scheme can also be used in a single processor system. Several groups of decomposed objects are designated by different information such as access control information to improve system performance or to meet other requirements. With these information, the system resource management daemons can also take management actions. For example, the checkpointing operations of read-only objects can be omitted to reduce the checkpoints overhead without affecting the checkpointing consistency.

The decomposition boundaries are determined by the users or by the system resource based on management policies. For example, if the objective of decomposing data and programs is to make replications for fault tolerant computing, the boundaries between data members could be determined in order to maximize the robustness degree in the system.

The following is a formal description of the data member decomposition scheme. The decomposition boundary lies between R_1 and R_2, which partition R. The original class C has a data member set R and a set of methods, $f_1, \ldots, f_n$.

The original input:

```
Class      C {
             R;
public:
             f1 ;
             .
             .
             .
             fn ;
}
```

We want to decompose the data members of the instance of class C into two disjoint sets R_1 and R_2. The result is the following program:

```
Class     C1 {                              Class   C {
          R1;                                       R2;
};                                          };

Class     C : public C1, public C2 {
public:
          f1;
          ⋮
          fn;
}
```

After the decomposition, the data members in a data instance of class C break into two parts. The decomposition is made through the following transformations. We define two new base-classes, C_1 and C_2 which consists of R_1 and R_2 as their sets of data members respectively. The set R of data member is removed from the original definition of class C. The class C inherits the two new base-classes. Since the access control of both R_1 and R_2 is *protected*, all the methods in class C can still access the data members in R_1 and R_2. Therefore, by taking advantage of the information-hiding feature of C++, no other transformation is needed for data references in $f_1, \ldots, f_n$ and its derived classes.

Access control of the data members must remain the same in order not to violate the original program semantics. In the above example, this seems trivial. But this is not true in general. Consider a private variable **a** in R similar to the above example. After decomposition, let it be moved to a new class. If the access control of **a** in the new class is still private, then methods in class C can no longer access variable **a**. If the access control is changed to protected, then not only the methods can access the data members, but also the methods in derived classes of C can access variable **a**. This obviously violates the desired private control rule. The solution is to this dilemma is to declare C as a friend class of C_1 and C_2. In this way, only methods in C can access the private variable a, while other derived classes of C will not be able to access them. Other access control mechanisms must be taken care of similarly in order to maintain the original semantics, refer to [6] for more details.

3.2 Example 2

We use example 2 to explain the decomposition of a C++ program in more details. The original program contains an instance c of class C, and defines three methods. The first is a constructor which calls **GetInputValue** to set the initial values of two integer array, **a** and **b**. The second method, **RandomSum**, accumulates the sum of **a** and resets it with random values. The last method, **OriginSum**, gets the sum of the original input values.

```
class C {
    int a[100];
    int b[100];
public:
    C();
    int RandomSum();
    int OriginSum();
};
// constructor.
C::C()
{
    for (int i = 0; i < 100; i++)
        a[i] = b[i] = GetInputValue();
};
int C::RandomSum()
{
    int sum, tmp;

    for (int i=0; i < 100; i++) {
        tmp = random() * 100 ;
        if ( i < tmp ) sum += a[i] ;
            a[i] = tmp;
    }
    return sum;
};
int C::OriginSum()
{
    int sum;

    for (int i = 0; i < 100; i++)
        sum += b[i];
    return sum;
};
main()
{
    C c;

    while (1) {
        .
        .
        c.RandomSum();
        c.OriginSum();
        .
        .
    }
    .
    .
}
```

After the decomposition, the program is transformed to the following. We only show the difference between the two programs.

```
class C1 {
    friend class C;
    int a[100];
};
class C2 {
    friend class C;
    int b[100];
};
class C: public C1, public C2 {
public:
    C();
    int RandomSum();
    int OriginSum();
};
// The following remains the
// same as the original program.
        .
        .
        .
```

Figure 3 shows the the rum time structures of this example before and after the decomposition.

We observe from the above example that our transformation modifies only the definition of data members in an object-oriented program, while the definition of the control flow of the decomposed program remains unchanged. Thus, the execution result also remains the same. Decomposition rules that we use to transform C++ programs always maintain an equivalence of control flow and data flow between the two versions of programs. Correct synchronization protocols are also necessary for maintaining the equivalence in distributed systems.

Based on this decomposed program, several runtime schemes can be further applied for different application targets, for example concurrency and fault-tolerance. However, besides the information embedded in the above decomposed program, other runtime structures are necessary for the decomposed program to be placed to processors in a distributed environment. In the next section, we will briefly discuss issues related to the run time structure.

3.3 A Run Time Partition Scheme

The objective of applications determine how a decomposed program is partitioned into concurrent tasks. There are several alternatives in placing the replications of the main program body, methods and data members on the same or different sites. To schedule the partitioned entities for executions needs the support of other runtime primitives such as message passing mechanisms. Many solutions exist for these primitives[7, 8]. An example of run time partition of example 2 is shown in Figure 4. The decomposed program breaks into several copies and are appropriately replicated. One daemon is linked with each copy on each site. The daemon handles the runtime behaviors of concurrent executions, such as object creations, synchronization and interfaces to the other system services including interprocess communications.

The above run time partition of a decomposed program illustrates several advantages.

- *Concurrency:* Under certain circumstances, methods that access different data members can execute concurrently. Since the data members in DEF and USE sets of these two invocations are disjoint, consider `c.RandomSum` and `c.OriginSum` in this example which access integer array `a` and `b` respectively, the methods can concurrently execute without affecting the result. The concurrency thus achieved is an important reason for decomposition.

- *Fault tolerance:* Replicating data members in this structure can make the decomposed program to tolerate hardware failures. Program and data on site 1, 2 and 3 are primary copy and the backup copies are on site 4, 5 and 6 respectively. Whenever an invocation is made in the main program, the site 1 daemon replicates the invocation to the backup site to keep the primary and backup data consistent. If site 2 fails and the recovery subsystem detects the event, the recovery system will notify the daemons on the other normal sites. Site 1 daemon masks the related information of site 2 and replaces it with that of an alternative site, in site 5 in this example, to continue the operation. In this way, the goal of fault-tolerance is achieved.

- *Data availability:* Distribution of data members in an instance to different sites can improve the data availability. In case of simultaneous failures of site 2 and 5, the data `a` is unavailable. However, data `b` is still available.

4 More Decomposition Schemes

In section 3, we present each component of the analyzer for decomposing a C++ program to support fault-tolerant distributed computing based on the decomposition of data members. C++ programs can also be decomposed in other dimensions. According to the type of boundaries, 10 different decomposition schemes are listed below. In the following list, each item is given by first describing the definition of the given type of boundary, and followed by brief descriptions.

1. *Between variables within an instance:* This is the type as presented example 2.

2. *Between statements within a method:* The resulting decomposition is the same as those in parallel program transformations of conventional programming languages.

3. *Among program, subroutines and class instances:* The resulting decomposition realizes the program structure of the object model[9].

4. *Between methods and data members of an instance:* This decomposition will result the data server model. The methods request data from the server and load the desired data to its local storage.

5. *Among data instances in the same class:* This scheme can form the *SIMD* or *MIMD* program structure and provides the basic model for distributed computing. Example 2 demonstrates a good example for this decomposition. Other example is the data partition for executions for parallel algorithms.

6. *Among unrelated data instances and methods:* Although this decomposition is a general form of 3,4 and 5, its application is quite different. One is for the system management . The operating system can omit the details of the logical meanings of the objects, and just perform this kind of decomposition for less application dependent purpose such as load balance.

7. *Between methods within the same class:* In most object-oriented programming implementation, the methods are not replicated but shared with all the instances. This decomposition provides a function decomposition of the C++ programs. It improves the concurrency in an instance.

8. *Between methods in derived classes:* This decomposition is similar to the above scheme.

9. *Among instances with same virtual base class:* This is for multiple inheritance feature. Decompose the data members in *generic* class from inherited classes in an instance.

10. *Between virtual functions in same derived classes:* Decompose the overloaded functions on different derived classes which are within the same generic class.

The transformation rules based on the above types of boundaries are presented in [6]. We are preparing another paper to describe the transformation mechanisms for

From the above list, we can observe the fact that different definitions of boundaries implies the requirements of different applications. For specific application, for example data replication for fault-tolerance, the type of decomposition boundary can be manually assigned and passed to the program analyzer. The analyzer analyzes the input of C++ program and decomposes it according to the desired configuration.

5 Implementation Issues

The key tasks to implementing the decomposition scheme is divided into several steps. The first step is to analyze the program structures such as class hierarchy and number of invocations. The next step is to determine the boundaries according to user-specification or other system specific constraints and runtime status. Once the boundary is determined, program transformation is performed to partition the target program into several tasks ready for distributed executions. Some statements are replaced with transformation primitives and a set of run time primitives are linked with the program by the analyzer. Run time servers will monitor system state information to convey synchronization and communication protocols.

Detailed descriptions of implementation issues are omitted due to space limitation. Interested readers may refer to [6] for more details.

6 Conclusion

In this paper, we have studies the object-oriented program decomposition. A new scheme containing an analyzer and proper runtime supports are proposed. An example based on data member decomposition is used to explain the details of our scheme, a complete list of decomposition dimensions are also presented. This scheme reduces the checkpointing overheads and improves the data availability for fault-tolerance when applied in distributed environments. Related implementation issues are also given.

In the future, we are going to study the problem of automatic determination of the decomposition boundaries according to some functional requirements. Specifically, automatic weight-assignment of objects is studies for this purpose.

References

[1] B. A. Kingsburg and J. T. Kline. "Job and process recovery in a Unix-based operating system". *USENIX – Winter 89*, pages 355–364, 1989.

[2] W. K. Fuchs C. J. Li. "CATCH - compiler-assisted techniques for checkpointing". In *Fault-Tolerant Computing: 20th International Symposium*, pages 74–81, Los Alamitos, CA, June 1990. IEEE Computer Society Press.

[3] M. W. Alford. "A graph model based approach to specifications". In *Distributed Systems*, pages 131–201, Berlin Heidelberg, 1985. Springer-Verlag.

[4] H. Dean. "Object-oriented design using message flow decomposition". *Journal of Object-Oriented Programming*, pages 21–31, May 1991.

[5] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[6] S. M. Sheu. *Decomposition of Object-Oriented Programs for Fault-Tolerant Computing.* PhD thesis, Northwestern University, 1992. In preparation.

[7] S. Wilbur and B. Bacarisse. "Building distributed systems with remote procedure call". *Software Engineering Journal*, pages 148–159, September 1987.

[8] R. Floyd E. F. Walker and P. Neves. "Asynchronous remote operation execution in distributed systems". In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 253–259, 1990.

[9] R. S. Chin and S. T. Chanson. "Distributed object-based programming systems". *ACM Computing Surveys*, 23(1):91–124, March 1991.

```
class A {
          .
          .
          .
public:
          .
          fj() {...};
};
A::fj() {
          .
          .
};
class B:A {
          .
          .
public:
          fi();
          .
          fk();
          .
};
B::fi() {
          .
          A::fj();
          .
};
B::fk() { ... };
```

```
B b;

main () {
          .
          b.fi();
          sub1();
          .
          .
          subn();
          .
};
sub1() {
          .
          .
          b.fk();
          .
          .
};
          .
          .
subn() {
          .
          b.fk();
          b.fi();
          .
};
```

Figure 1: Example 1.

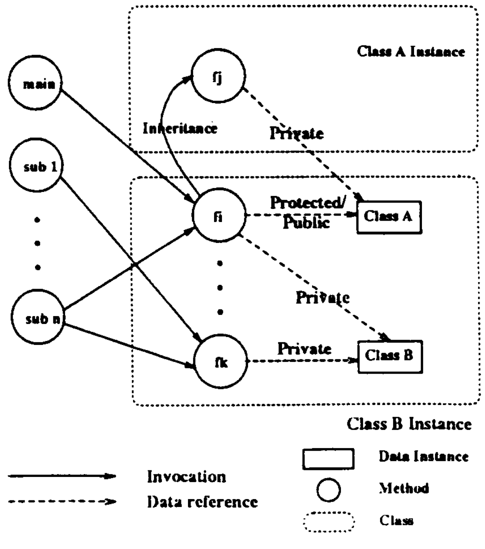

Figure 2: Representation graph of example 1

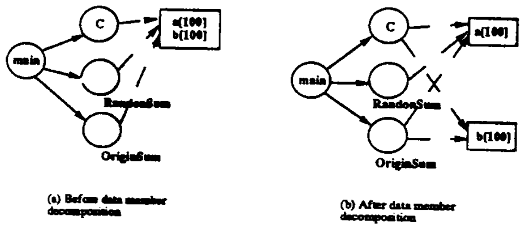
main
C
a[100]
b[100]
RandomSum
OriginSum
(a) Before data member decomposition
main
C
a[100]
RandomSum
b[100]
OriginSum
(b) After data member decomposition

Figure 3: The run time model of example 2

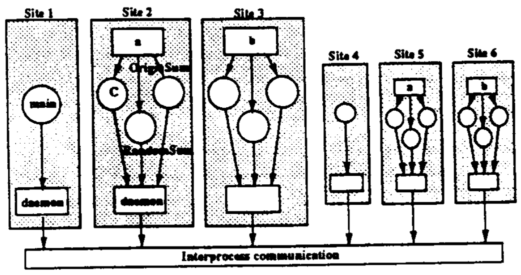

Figure 4: A partition scheme for example 2