Optimized Object State Checkpointing using Compile-Time Reflection

Juan-Carlos Ruiz GarcÌa,
Marc-Olivier Killijian,
Jean-Charles Fabre
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse, France
{ruiz, killijian, fabre}@laas.fr

Shigeru Chiba
Institute of Information Science
and Electronics
University of Tsukuba
Tennodai, Tsukuba
Ibaraki 305-8573, Japan
Chiba@is.tsukuba.ac.jp

1 Introduction

Object-oriented concepts have been now used in many fault tolerant systems in order to provide better flexibility than with classical system structuring solutions. Various approaches have been used either based on inheritance [1,2], with CORBA [3,4,5,6] or using reflective ideas and metaobject protocols [7,8,9]. Most of the fault tolerance strategies implemented in these systems impose saving and restoring the state of objects during operation and recovery actions respectively. These operations are often user defined which is not acceptable: any omission in this definition leads to the total inefficiency of the fault tolerance mechanisms used. Conventional solutions based on dirty bits or flagged memory pages enable the modified state of a runtime entity to be automatically saved. Nevertheless, this solution is often hardware dependent and does not ensure an optimal amount of information to be saved.

The approach to this problem is investigated here using compile-time reflection which provides consistency of the object state saved and optimizes the amount of information necessary to checkpoint the state of application objects.

2 Delta-Checkpointing

2.1 Definitions

We consider objects as composed of an arbitrary number of attributes, some of them being reference to other objects. The idea is to identify at runtime the attributes modified by a method execution. Only the part which has been modified since the last method execution is transmitted in a checkpoint message called *Delta-Checkpoint*.

Reflection [10] is defined as *the behavior exhibited by a reflective system, where a reflective system is a computational system that includes a meta-model of itself.* Since this model is ìcausally connectedî to the real behavior of the system, its internal data structures and actions can be retrieved and altered by the user through that meta-model. The process of retrieving is called ìreificationî and the process of altering is called ìreflectionî. The meta-models of the reflective systems are often built with a set of objects. These objects are called metaobjects [11] to distinguish them from normal (base-level) objects. These metaobjects can exist at runtime, as in Open C++ v1 [12], or at compile-time, as Open C++ v2 [13]. In this later case all meta-computation is performed at compile time, so runtime penalties inserted by this metaobjects is almost zero.

2.3 Techniques Overview

We define at compile-time a collection of meta-operations, which explore each method code to determine when an attribute is modified. Each attribute has a flag whose management is performed by the meta-compilation: it is initialized to false before each method call and set to true when a write access is performed on it. The role of the compile time metaobjects is to detect how many attributes are likely to change. The necessary code for obtaining the modified attributes at runtime is added at the end of the methodís code. The modified attributes are packed into a buffer called **Delta-State** which represents the checkpoint information. This general solution can be summarized as follows: (i) attribute-flags initialization and code customization, at compile time, (ii) object's method execution, attribute-flags test and updating Delta-State, at runtime. In the remainder, we'll refer to this general solution as **Compile-Run-Time** technique **(CRT).**

In C++, the type of objectís attributes can be a built-in C++ type or a complex type (arrays, structures, etc.). However, objectís state can encapsulate the state of other objects through composition links. CRT is thus applied recursively to handle composed objects and for obtaining a Delta-State of each sub-object. The final Delta-State will be the aggregation of all the Delta-States obtained by this recursive process. The recursion stops as soon as a basic type attribute or even a complex type attribute (not an object reference) is encountered.

The detection of every object's attribute modification relies on the analysis of the object method's source code. Local variable definitions, pointers to attributes, direct or indirect attribute modification and ì†illegal†î reference pointer modification are identified. However, this is not sufficient for obtaining a 100% coverage of the modified state, which is mandatory. When a method statement do not match any of the above filtering patterns, then a C++ syntactic analyzer determines when any attribute is referenced directly or indirectly. Actually, some situations, e.g. passing an attribute reference as a function argument or performing arithmetic operations on pointers which refer to an attribute, can compromise objectís state integrity and are difficult, even impossible, to handle. In this case, we consider that some uncontrolled actions may be performed on objectís attributes and we apply the **Copy-Compare-State (CCS)** technique. All the attributes are saved beforehand and a global object's state comparison is performed at the completion of the method execution to determine the Delta-State. The drawback of this technique is that a runtime overhead is introduced. Necessary tradeoffs between state comparison and networking overheads have to be evaluated on a case-by-case basis.

Finally, for attribute modifications that can be detected at compile time we have defined the **Compile-Time-Only (CTO)** technique.

3. Implementation Issues and Concluding Remarks

The implementation of these techniques using compile time reflection is done with Open C++ V2. It enables source-to-source translation from extended C++ to regular C++ driven by compile time metaobjects. Any class of object can be associated to a class metaobject which defines how this class is translated. The associated metaobject knows exactly the definition of the object class which is necessary to detect each member object call. During the analysis of each methodís code, we look for statements where attributes or pointers to attributes are used, we detect any assignment of an attribute address to a pointer, enabling the technique to be determined for each attribute. The resulting code is compiled using a standard C++ compiler (like gcc).

Handling object state in object-oriented fault tolerance application is a major issue and relying on user-defined virtual functions is not acceptable. Relying on dirty bits is not optimal and orthogonal to the object model. Compile time reflection enables objects state to be analyzed and the object code

to be prepared to determine attribute modifications at runtime. As far as possible, the proposed techniques optimize the object's state to be transferred possibly to a different node, but performance tradeoffs have to be evaluated according to object's attributes number and size, memory space, network bandwidth. However, the major contribution of such techniques is that the object's state is automatically handled. The efficiency of fault tolerance mechanisms at upper layers is not annihilated by an incorrect definition of the recovery state.

Reference

[1] Detlefs D., Herlihy M.P. and Wing J.M., ìInheritance of Synchronization and Recovery Properties in Avalon/C++î, *Computer*, 21 (12), Dec. 1988, pp. 57-69.

[2] Shrivastava S.K., Dixon G.N. and Parrington G.D., ìAn Overview of the Arjuna Distributed Programming Systemî, *IEEE Software*, 8 (1), 1991, pp. 66-73.

[3] Landis S. and Maffeis S., ìBuilding Reliable Distributed Systems with Corbaî, Theory and Practice of Object Systems, (special issue on the future of Corba 3), vol. 3 (1), 1997, pp. 59-66.

[4] P. Felber, B. Garbinato, and R. Guerraoui ìThe design of a CORBA group communication serviceî, in *Proc. of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake (Canada), October 1996.

[5] Narasimhan P., Moser L.E. and Melliar-Smith P.M., ìExploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Toleranceî, in *Proc. of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland (Or, USA), June 1997, pp 81-90.

[6] Moser L.E. and Melliar-Smith P.M., "The Interception Approach to Reliable Distributed CORBA Objects," P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Panel on Reliable Distributed Objects, *in 3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, (Or, USA), June 1997, pp 245-248.

[7] Agha G., Fr¯lund S., Panwar R. and Sturman D., ìA Linguistic Framework for Dynamic Composition of Dependability Protocolsî, in *Proc. of DCCA-3*, 1993, pp. 197-207.

[8] B. Garbinato B., Guerraoui R. and Mazouni K., ìImplementation of the GARF Replicated Objects Plateformî, *Distributed Systems Engineering Journal*,2), March 1995, pp. 14-27.

[9] J.C.Fabre, T. PÈrennou, ìA Metaobject Architecture for Fault Tolerant Distributed Systems: The *FRIENDS* Approachî, *IEEE Transactions on Computers*, Special Issue on Dependability of Computing Systems, Jan. 1998, pp. 78-95.

[10] Maes P., ìConcepts and Experiments in Computational Reflectionì, in *Proc. of OOPSLA'87*, Orlando, USA, 1987, pp. 147-155.

[11] Kiczales G., des RiviËres J. and Bobrow D.G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[12] Chiba S. and Masuda T., ìDesigning an Extensible Distributed Language with Metalevel Architectureî, in *Proc. of ECOOP'93*, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993, pp. 482-501.

[13] Chiba S., ìA Metaobject Protocol for C++î, in *Proc. of OOPSLA'95* (ACM Conference on Object-Oriented Programming, Systems, Languages and Applications), Austin (TX-USA), October 1995, pp. 285-299.