Transformations for the Optimistic Parallel Execution of Object-Oriented Programs

A. Back and *S.J. Turner*
Department of Computer Science, University of Exeter,
Prince of Wales Road, Exeter EX4 4PT England

Abstract

This paper discusses the use of optimistic execution as a mechanism for parallelizing sequential object-oriented programs. Most parallelizing compilers to date have used compile-time data-dependency analysis to determine independent sections of code. This reliance on static information presents an overly restrictive view of dependencies in a program.

In this paper, a set of transformations is presented which allows the use of causality violation detection and roll-back. Selected objects are transformed into server objects, which are distributed across the nodes of a parallel computer. The asynchronous dispatch of server object methods introduces parallelism into the execution, in a way which preserves the semantics of the sequential code.

1 Introduction

Most parallelizing compilers to date have used compile-time data-dependency analysis to determine independent sections of code. Compilers relying solely on data-dependency analysis are often unsuccessful at extracting parallelism from sequential programs. This is due to failure to detect independent sections of code, either because there are no completely independent sections in the algorithm, or because some artificial data dependencies have been introduced in the coding. The inherent problem is that compile-time data-dependency analysis presents an overly restrictive view of dependencies in a program.

The motivation for using optimistic execution is that it is able to cope with *dynamic* data dependencies, for example where one section of code has a conditional dependency on another. Such a system thus avoids the restrictions associated with compilers based on static data-dependency analysis alone. Optimistic methods such as the "Time Warp" [10] mechanism have been successfully used in parallel discrete event simulation, but to date there has been little research in using such techniques to parallelize general purpose programs.

In this paper, we discuss the use of optimistic execution as a way of parallelizing programs written in a general purpose object-oriented programming language. We present a set of transformations which allow the use of the "Time Warp" mechanism in executing "sequential" C++ programs on a multicomputer architecture. With this approach, a parallel execution can be derived automatically and this execution can be guaranteed to have identical semantics to the sequential one.

2 Optimistic Execution

The optimistic approach is based on ideas taken from parallel discrete event simulation (PDES). In PDES [8], there are broadly two approaches taken to ensuring that causality (the principle that events in the future cannot affect events in the past) is preserved in the simulation: conservative, and optimistic. The conservative approach [6] strictly avoids the possibility of any causality violation ever occurring. On the other hand, the optimistic approach allows causality violations to occur, but provides a recovery mechanism which is able to roll-back and undo the effect of such violations.

In the "Time Warp" mechanism, an event message that causes roll-back may require the mechanism to perform two actions: restoring the state of the object and canceling all intermediate side effects by "unsending" previously sent messages [8]. The first action is accomplished by regularly saving the object's state, and restoring an old state on roll-back. "Unsending" a previously sent message is accomplished by sending an *anti–message* that annihilates the original when it reaches its destination. This anti–message may in turn cause another object on another processor to roll back.

In applying the optimistic execution scheme to general purpose object-oriented programs, the essence of the approach is that selected program objects are transformed into server objects and are placed on the nodes of a parallel computer. The invocations of the methods of these objects are replaced by asynchronous remote procedure calls (RPCs), thereby introducing parallelism into the execution. Remote procedure call messages are time-stamped in such a way that executing them in increasing time-stamp order would be equivalent to the sequential execution of the program.

There are a number of factors leading to the choice of object-oriented programming languages as suited to this approach:

1. Object oriented programs tend to be more modular than those written in procedural languages. This modularity enables different sections of code to be distributed across the nodes of the parallel computer.
2. There are generally well-defined interfaces controlling the use of one module's data by another module. This simplifies the data-dependencies and will help to reduce the number of causality violations.
3. There is often a multi-level object hierarchy within an object oriented program. This gives objects of varying degrees of granularity and allows suitably coarse-grained objects to be chosen as server objects.

3 Design Overview

A server object is a run-time system object which encapsulates the functionality of state-saving, roll-back, and causality violation detection for the optimistic parallel execution of a transformed program object. A server object is a remote procedure call (RPC) server for an object. It has a single thread of control which accepts asynchronous RPC messages, decodes the message to decide which method was invoked, decodes the method's arguments, and executes that method.

Each server object has a logical clock, which is set to the time-stamp of RPC messages as they are processed. In this way the logical clock increases in discrete steps with the processing of RPC messages. A causality violation is signalled by the arrival of a "straggler" message, a message with a time-stamp less than the server object's logical clock. This indicates that the straggler should have been processed earlier in logical time than RPCs already processed by that server object.

If a causality violation is detected, the state of the object must normally be rolled back. To do this, a history of states must be retained. The state-saving method used is an object-oriented hybrid incremental state-saver. This can perform state-saving on objects at arbitrary levels in the object-oriented class hierarchy. The overhead of roll-back may often be reduced or eliminated by applying one of a number of optimizations [8]: these include lazy cancellation, lazy re-evaluation, and avoidance of roll-back for methods which do not modify an object's state.

The time-stamps required for the optimistic execution of general purpose code have more demanding requirements than those normally used for parallel discrete event simulation. This is as a result of the fact that programs are not predictable, and yet ranges of time-stamps need to be allocated in advance for sections of code. In general, the time-stamp allocation scheme must allow an arbitrary number of new time-stamps to be allocated between any pair of previously allocated time-stamps. An allocation scheme for generating and manipulating time-stamps efficiently is discussed in detail in [1].

4 The Transformations

There are three levels to the transformation system, as shown in figure 1: the heuristic level, the annotation level, and the underlying transformations. The annotation level provides an interface which will, in the future, allow the transformations to be directed by an heuristic analysis system. At present, these annotations are provided manually by the programmer. The areas where it is intended that heuristics will be used include the selection of program objects to use as server objects, the distribution of those objects, and the parameters which determine the state-saving behaviour of objects.

The box marked "transformations" in figure 1 is the level which performs the parsing of C++ code and the subsequent transformation of that code to allow optimistic execution. These transformations were implemented using the Sage++ transformation tool [3]. The other components shown are a C++ compiler and the run-time library, which is linked with the transformed code and provides the run-time support functions required for parallel execution. The run-time library is built on top of the p4 message passing system [4] which provides portability.

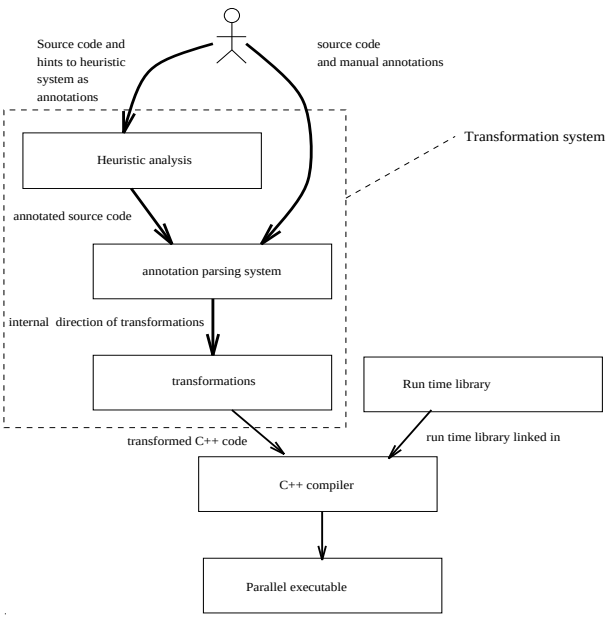

Figure 1: The Transformation System

There are a number of transformation phases which are applied to the sequential C++ program:

Explicify With C++ as the choice of language, there are some preliminary transformations required to make explicit some implicit language features, such as where a

C++ compiler will call either a user provided, or a built-in function or conversion. As implicitly called functions must be considered in the subsequent transformations, the "explicify" phase must carry out the disambiguation associated with implicit functions and replace them by explicit code.

Purify In a pure object-oriented language, there are no types outside the object-oriented type system. Some object-oriented languages, such as C++ , deviate from this pure view for efficiency reasons: the built-in types are treated as a special case, and are outside the type system for purposes of inheritance, and syntax of operator use. The purpose of "purifying" is so that the resulting use of operations on such objects is explicitly expressed in terms of method invocations.

This is achieved by simulating the built-in types by providing replacements which are specially defined: these allow inheritance, and have all of the required operations available as methods. It should be noted that the purify transformation need not be applied to the whole program, but only to selected objects, where it is necessary for such objects to be manipulated to provide roll-back, and state-saving functionality.

Server Objects Selected program objects are transformed to server objects. This means that the methods of that object are invoked as asynchronous remote procedure calls, with the calls being implemented as messages passed between server objects, possibly on different processors. In addition, server objects have a logical clock and check for causality violations: they invoke roll-back and recovery, including the sending of anti-messages when a causality violation is detected.

Asynchronous Messages This phase is concerned with transforming methods which have a return value, so that they may be invoked asynchronously. In a "pure" object-oriented program, the return value will by definition be passed as an argument to a method of another object. The transformation is therefore to pass, along with the arguments for the RPC, instructions for where the result should be sent. For example, in an assignment statement, the returned value would be sent to the assignment method of the object which was on the left hand side of the assignment in the untransformed program.

Adding Time-stamps Time-stamps are added as an extra argument to RPC messages for server objects, as these operations can be rolled back. If the method is invoked from different places in the program, different time-stamp arguments would be passed depending on where the method is called from. As the time-stamps allocated in the body of the method are prefixed by the time-stamp passed in, the time-stamps within the body will also vary in different invocations.

Bounded time-stamp generators provide a fixed, predetermined number of time-stamps in a given range. Unbounded time-stamp generators are required for cases where it is not known how many time-stamps will be required for the range, for example, with loops with an exit condition which is determined at run-time. To meet these requirements, a variable length time-stamp scheme has been adopted [1].

Control Flow Objects For control constructs where the flow of control is determined by the result returned by a method of a server object, the control construct itself must be transformed to a replacement control flow object, which has the properties of a server object. For example, with the parallelization of a `for` construct, it may be the case that certain iterations of the loop may need to be rolled back. To enable this to happen, the `for` construct must be transformed into a `server_for` object, as shown in figure 2.

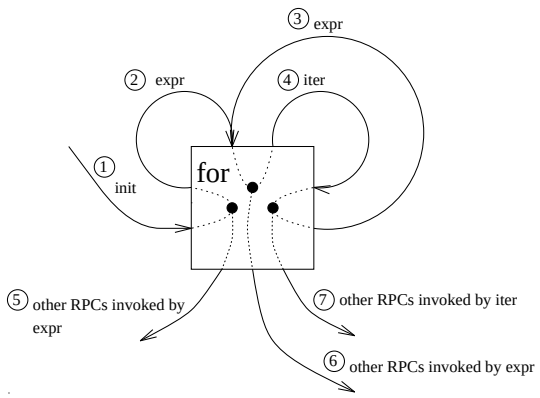

Figure 2: RPCs of replacement `server_for` object

State-Saving A server object either knows how to save its own state or relies on the state-saving abilities of its contained objects. A state-saving object is a transformed object with the same functionality as the original object, but which automatically saves its state as it is used, and with extra methods to invoke roll-back. There is a base state-saving class from which the replacement class is derived. Its methods will be transformed to have time-stamps also, as state-saving requires these.

As objects are built from component objects, the selection of state-saving mechanism is made by choosing between building the object from the functionally equivalent state-saving objects, or from unmodified objects. The selective use of state-saving objects at various levels in the class hierarchy allows for a hybrid system, which is capable of varying its characteristics between purely incremental state-saving, and more periodic behaviour.

5 Experimental Results

This section presents an estimate of the overheads incurred as a result of the transformations required to allow optimistic execution. These overheads include: using re-

placement types for built-in objects, adding state-saving to replacement types, and adding time-stamps as well as state-saving. These results were obtained with the GNU g++ compiler (version 2.6.0), on a 100MHz R4600 based SGI Indy under IRIX 5.2

Two examples were chosen: the `count1` example is just a program to count up to $1,000,000$, with no other code, the `RC5` example is RSADSI's RC5 symmetric key block cipher encrypting a single block using a 128 bit key. In general, the exact overhead will depend on the mix of operations used by the application, the relative costs of the built-in operations compared to function call overheads, the machine architecture, and the compiler used.

Figure 3 shows the approximate overheads, per useful operation, due to replacement types, state-saving and the addition of time-stamps. These figures are based on no inlining, and no use of reference parameters. A special purpose memory allocation algorithm is used as this gives an improvement over the standard `new()` operator.

example	replacement	state-saving	time-stamp	combined
count1	322 ns	3.96 us	768 ns	5.05 us
rc5	218 ns	2.94 us	602 ns	3.76 us

Figure 3: Overheads per useful operation

These results show that by choosing suitably coarse-grained objects for the server objects, the ratio of overheads to operation costs may be reduced to an acceptable level.

6 Conclusions

This paper has shown how it is possible to extend the ideas of parallel discrete event simulation to the execution of general purpose object-oriented programs. By assigning time-stamps which are consistent with a sequential execution, a conventional object-oriented program may be executed in parallel using the Time Warp mechanism, with a guarantee that the same results are obtained as would be the case if the program were executed sequentially.

To date, there has been little research in using such techniques to parallelize general purpose code: papers in this area have generally been proposals rather than descriptions of implementations. Bacon and Strom discuss the use of optimism in relation to CSP [2]. Fujimoto [7] has also suggested the use of optimism as a parallelization technique, using the Virtual Time Machine. Winter, Kalantery and Wilson [11] have proposed the use of mainly conservative synchronization techniques to provide a new parallel architecture model.

Our research also differs from the numerous parallel C++ projects, for example CC++ [5] or Mentat [9], in that these projects generally aim to define extensions to the C++ language. Such extensions provide explicit ways of expressing parallelism, whereas our approach is to extract parallelism from an object-oriented program, whilst preserving the semantics of the original sequential execution through the use of optimistic methods.

The generalisable nature of our approach makes it a conceptually attractive alternative to conventional parallelization techniques which are typically special purpose, and limited to particular problem classes. Full dependency analysis is a hard problem, so the avoidance of this problem is a useful result.

References

[1] A Back and S Turner. Time-stamp generation for optimistic parallel computing. In *Proceedings of the 28th Annual Simulation Symposium, Phoenix, AZ*, pages 144–53. IEEE Press, April 1995.

[2] D F Bacon and R E Strom. Optimistic parallelization of communicating sequential processes. *ACM Sigplan notices*, 26(7):155–166, 1991.

[3] F Bodin, P Beckman, D Gannon, J Gotwals, S Narayana, S Srinivas, and B Winnicka. Sage++ : An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. *Object Oriented Numerics*, 1994.

[4] J Boyle, R Butler, T Disz, B Glickfeld, E Lusk, R Overbeek, J Patterson, and R Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[5] K M Chandy and C Kesselman. Compositional C++ : Compositional parallel programming. In *Proceedings of the Fourth Workshop on Parallel Computing and Compilers*, 1992.

[6] K M Chandy and J Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Engineering*, S.E.5(5):440–452, 1979.

[7] R M Fujimoto. The virtual time machine. In *SPAA (Symposium on Parallel Algorithms and Architectures)*, pages 199–208, 1989.

[8] R M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[9] A Grimshaw and J Liu. Mentat – an object-oriented macro data flow system. In *OOPSLA Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 35–47, 1987.

[10] D R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[11] S Winter, N Kalantery, and D Wilson. From BSP to a virtual von Neumann machine. In *BCS Parallel Processing Specialist Group Workshop*, pages 92–99, 1993.