Optimistic Network Computing and its Performance Control

Stephen J. Turner
Department of Computer Science, University of Exeter,
Prince of Wales Road, Exeter EX4 4PT England

Abstract

Optimistic execution is a powerful technique that can enable an application to make more efficient use of resources that are distributed across a computer network. Such a mechanism may be used to provide automatic parallelization of a sequential object-oriented program, allowing it to be executed concurrently on a network of processors. Optimistic execution can be of particular benefit to distributed applications, because the increase in concurrency can be used to hide the latency of inter-node communication.

In this paper, a set of program transformations is presented which allows the use of optimistic execution through causality violation detection and roll-back. The asynchronous dispatch of method invocations introduces concurrency into the execution, in a way that preserves the semantics of the sequential code. An important part of such a system is the adaptive control mechanism that will select objects at run-time for optimistic execution. This is achieved by means of a reflective computation which monitors the behaviour of the objects, dynamically introducing concurrency where appropriate.

1 Introduction

Optimistic execution is a powerful technique that can improve the performance of an application by introducing concurrency into the program. As a parallelization tool, it is able to cope with *dynamic* data dependencies, for example where one section of code has a conditional dependency on another. Most existing parallelizing compilers are based on compile-time data dependency analysis to determine independent sections of code, but these are often unsuccessful at extracting parallelism. This is due to their 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.

An optimistic system allows sections of code to be executed concurrently based on the assumption that there is no interference between those sections. If this assumption is true, the optimistic computation is retained, otherwise a causality violation will be detected and the optimistic computation will be rolled back. Such a system 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 for general purpose programs.

In a distributed environment, optimistic execution techniques can be of particular benefit because of the high latency of inter-node communication in the network. By introducing or increasing concurrency in a distributed program, it is possible to hide the latency of communication. For example, consider the sequence of code $S_1; S_2$ where S_1 is a remote method invocation. By transforming this synchronous invocation into an asynchronous one, it is possible to execute S_1 concurrently with S_2. If these statements are independent, the latency of the remote method invocation will be hidden. If there is a data dependency, a rollback will occur and the semantics of the sequential code will be preserved.

The aim of this paper is to present a design for a general purpose optimistic parallelization scheme, suited to object oriented languages, and to show the feasibility of this approach by describing the implementation of a transformation tool and associated run time system. The design includes a mechanism for selecting those parts of the program to be executed optimistically, so giving a fully automatic approach. Since the parallel execution can be guaranteed to have identical semantics to the original sequential code, the effect is transparent and the programmer need not be aware that there is a parallel implementation.

The rest of this paper is structured as follows. Section 2 gives an overview of optimistic execution and discusses related work in this area. Section 3 describes the design of our system, including the transformations and time-stamp allocation scheme. A prototype of this system has been developed for the optimistic execution of C++ programs [2] on a transputer network. This system is now being re-implemented for the execution of Java programs, to provide optimistic computing on the Web. Section 4 discusses the run-time system, and shows how an adaptive control mechanism may be used for selecting constructs dynamically for optimistic execution. Finally, section 5 presents the conclusions.

2 Optimistic Execution

The optimistic approach is based on ideas used successfully in the field of parallel discrete event simulation (PDES). In PDES [9], 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 [5] 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 causality violations.

The most widely known optimistic execution technique is the "Time Warp" mechanism and this forms the basis of our system. An event that causes roll-back may require the mechanism to perform two actions: restoring the state of the object and cancelling all intermediate side effects by "unsending" previously sent messages [9]. 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.

Although there have been a number of proposals in the literature for using such techniques to parallelize general purpose programs, there are few descriptions of successful implementations. Fujimoto was one of the first to suggest the use of optimism as a parallelization technique, using the "Virtual Time Machine" [8]. Tinker [13] presented an approach to scheduling tasks using rollback synchronization. Bacon and Strom discussed the use of optimism in relation to CSP [3]. Winter *et al* [15] have proposed the use of mainly conservative synchronization techniques to provide a new parallel architecture model. Cleary *et al* have presented the design of an optimistic CPU known as the "Warp Engine" [6].

An interesting implementation of these ideas is provided by the HOPE programming environment [11]. This provides a number of optimistic language constructs which enable the programmer to make optimistic assumptions which must then be verified at a later stage. Cowan [7] shows how optimism may be used to enhance the performance of distributed programs by avoiding remote communication delay. This work differs from our approach in that optimistic assumptions must be specified explicitly in HOPE, whereas our system automatically selects those parts of the program to be executed optimistically.

3 System Design

In applying an optimistic execution scheme to general purpose object oriented programs, the essence of the approach is that selected program constructs will be transformed into Time Warp (TW) objects. Synchronous method invocations on these constructs are replaced by asynchronous remote procedure calls (RPCs). RPC messages are time-stamped, in such a way that if the program were executed in increasing time-stamp order, an execution semantically indistinguishable from the sequential execution is obtained, just as executing a simulation using the "Time Warp" mechanism gives exactly the same results as a sequential simulation.

A TW object encapsulates the functionality of state-saving, roll-back, and causality violation detection. 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 TW 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 TW object's logical clock. This indicates that the straggler should have been processed earlier in logical time than RPCs already processed by that TW 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 overhead of roll-back may often be reduced or eliminated by applying one of a number of optimizations [9]: these include lazy cancellation, lazy re-evaluation, and avoidance of roll-back for methods which do not modify an object's state.

3.1 The Transformations

To map sequential object oriented code on to the optimistic execution model, a transformation system is used to transform method invocations into asynchronous remote procedure calls. This is required as the aim of the system is to introduce concurrency: there is no need for the sending object to wait for a return value or acknowledgement message indicating completion of the remote procedure call. The flexibility of the optimistic execution scheme enables different sources of concurrency to be exploited, but in a distributed environment, the most significant performance improvement will be obtained by hiding the latency of communication.

There are a number of transformation phases which are applied to the sequential object oriented program:

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

TW Objects Selected program constructs are transformed to TW 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 TW objects, possibly on different processors. In addition, TW 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 TW 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.

Control Flow Objects For control constructs where the flow of control is determined by the result returned by a method of a TW object, the control construct itself must be transformed to a replacement control flow object, which has the properties of a TW 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 `TW_for` object, as shown in figure 1.

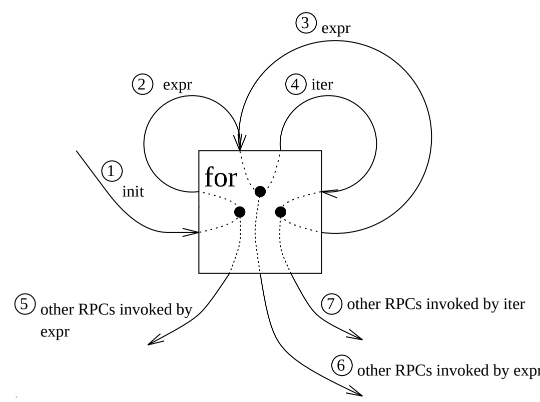

Figure 1: RPCs of replacement `TW_for` object

State-Saving A TW 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.

3.2 Time-Stamp Allocation

With the optimistic execution of a general purpose program, a problem arises which does not occur in parallel

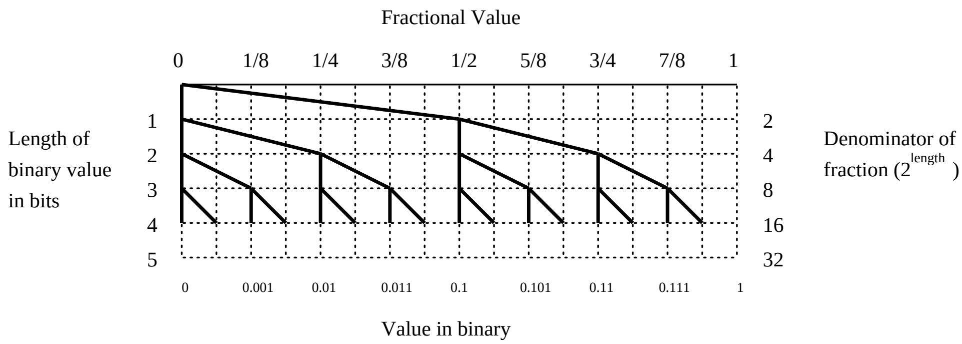

Figure 2: Variable Length Time-stamps

discrete event simulation: it may be necessary to allocate an arbitrary number of new time-stamps between any pair of previously allocated time-stamps. For example, with a program control structure consisting of an unbounded loop, it would be possible to execute the individual iterations of that loop on one processor in parallel with the events corresponding to the code following the loop on another processor. It is therefore necessary to allocate the time-stamps for the events following the loop before all of the time-stamps have been allocated for the iteration events.

The time-stamps required for the optimistic execution of general purpose code thus have more demanding requirements than those normally used for parallel discrete event simulation. Two types of time-stamp generator have therefore been implemented: bounded time-stamp generators provide a fixed, pre-determined 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 our representation of time-stamps, the operations to compare and generate the next time-stamp in a sequence must be efficient. A variable length time-stamp has therefore been adopted which satisfies these requirements. This is defined as:

$$\text{a time-stamp is the pair } (\mathit{length}, \mathit{value})$$

where *value* is a binary number of length *length* whose value will be non-negative. Time-stamps can be viewed as variable length binary fractions, whose values fall in the range 0 to 1, as shown in figure 2. The fractional value of a time-stamp (l, v) is

$$f = \frac{v}{2^l}$$

An ordering relation is defined on on time-stamps:

$$(l_a, v_a) = (l_b, v_b) \text{ iff } f_a = f_b \text{ and } l_a = l_b$$
$$(l_a, v_a) < (l_b, v_b)$$
$$\text{iff either: } \begin{cases} f_a < f_b \\ \text{or } f_a = f_b \text{ and } l_a < l_b \end{cases}$$

The time-stamps can be thought of as variable precision binary fractions. The left child of time-stamp (l, v) is $(l+1, 2v)$ and the right child $(l+1, 2v+1)$. More generally at depth d below l, the set of time-stamps at level $l+d$ which fall between (l, v) and the consecutive time-stamp at the same level $(l, v+1)$ is given by:

$$(l, v) < \{(l+d, 2^d v) < \ldots < (l+d, 2^d v + 2^d - 1)\} < (l, v+1)$$

Informally, for time-stamps with different fractional values, integer compare can be used on the *value* component to correctly order the time-stamps. However, if the fractional values of a pair of time-stamps are equal, the shorter time-stamp is taken to be smaller. Only if both the fractional values and the length of time-stamps are equal are the time-stamps considered equal, that is, trailing zeros become significant in the ordering, so that $0.10 \neq 0.1$. A full discussion of the time-stamp allocation mechanism can be found in [1].

4 Run-time System

In our prototype implementation, the run-time system was implemented as a library which was linked with the transformed code [2]. A novel approach is being adopted in the new version, in which the Time Warp mechanism is encapsulated at the meta-level of a reflective system. Computational reflection is the process of reasoning about and acting upon the computational system itself [14]. The advantage of a reflective system is that it supports the ability to monitor and modify the behaviour of objects dynamically. This forms the basis of our adaptive performance control mechanism.

A reflective system must contain some data which represents the structural and computational aspects of the system itself. There are a number of approaches to constructing the self representation of such a system. At one extreme, it would be possible to have a single

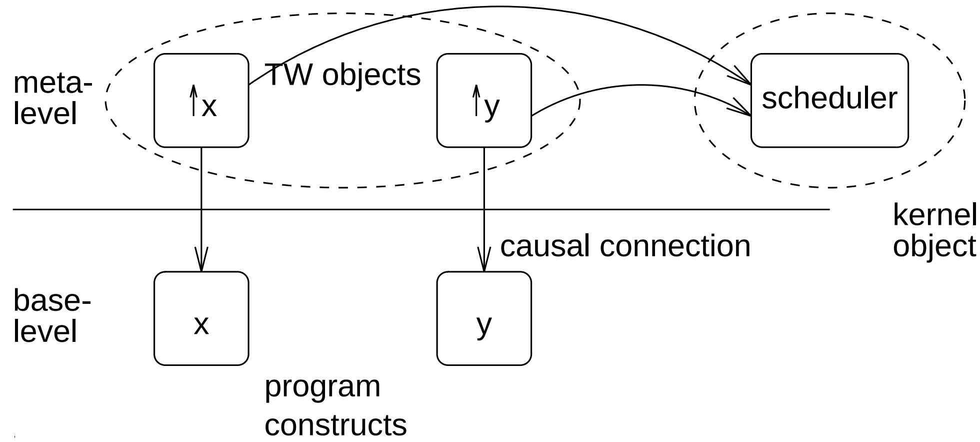

Figure 3: Hybrid Group Architecture

data structure that represents the whole system, while at the other, each object may be represented by its own *meta-object*. A construct of a program at the base-level is said to be *reified*, or exposed, when it is causally connected to its self representation at the meta-level.

The reflective architecture that we have adopted is similar to the hybrid group architecture of ABCL/R2 [12] and is shown in figure 3. Each construct that is selected for optimistic execution has its own meta-object which models both the structural and behavioural aspects of that construct. A special *kernel* object is introduced on each processor for scheduling those objects. Since the kernel interacts with the meta-objects of the group, it exists only at the meta-level. The kernel objects are also responsible for calculating global virtual time [4] or GVT, which is the minimum of the logical clocks (and time-stamps of messages in transit). No event with a time-stamp smaller than GVT will ever be rolled back, so storage used by such events (e.g. saved states and messages) can be discarded.

The selection of program constructs for optimistic execution should take into account the granularity of the construct and the potential improvements that might be obtained through hiding the latency of communication. Statistics may be collected at run-time by the meta-level objects and used to determine whether an optimistic computation is likely to be worthwhile. For example, with an unbounded loop, a guess could be made as to how many iterations to execute optimistically. Similarly, with an `if` construct, either the `then` or `else` part could be executed before the boolean condition is known.

Another area concerns the selection of state saving mechanism to be used for a TW object. One problem with using incremental state-saving for a general purpose programming language is that with the obvious implementation ultimately every variable or object in the entire program would need to be represented as a state-saving object. Performing incremental state-saving on low level objects such as integers, or floating point numbers, adds a significant overhead compared to the simple task performed, that of multiplication, addition, etc. Saving a previous value in these cases is many times more expensive than the operation itself, especially as the operands will typically be stored in registers, while the state-saved information will have to be stored in memory.

Taking advantage of the object oriented nature of our system, our approach to reducing this overhead is to adopt a hybrid scheme. A choice may be made between building an 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, and with containment, allows for a hybrid system, which is capable of varying its characteristics between purely incremental state-saving, and more periodic behaviour. For example, figure 4 shows the two possibilities when an object `A` contains two objects `B` and `C`. Either `A` can be composed of state-saving objects `B` and `C`, or `A` itself can be state-saving and `B` and `C` are unmodified objects.

5 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 ex-

Figure 4: State Saving Options

ecuted sequentially.

A novel part of our design is the encapsulation of the Time Warp mechanism at the meta-level of a reflective system. This provides a mechanism by which statistics about the execution of the program may be collected, in order to determine which parts of the program should be executed optimistically. The reflective system uses a hybrid group architecture, in which a kernel object on each processor is responsible for scheduling the objects, as well as taking part in global computations such as the calculation of global virtual time.

References

[1] A Back and S J 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] A Back and S J Turner. Transformations for the optimistic parallel execution of object-oriented programs. In *Proceedings of Parallel Object-Oriented Methods and Applications Conference (POOMA '96)*. Los Alamos National Laboratory, 1996.

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

[4] S. Bellenot. Global virtual time algorithms. In *Proc. SCS Distributed Simulation Conference*, pages 122–127, 1990.

[5] 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.

[6] J G Cleary, M Pearson, and H Kinawi. The architecture of an optimistic CPU: The Warp Engine. In *Proceedings of HICSS*, volume 1, pages 163–172, 1995.

[7] C Cowan, H Lutfiyya, and M Bauer. Performance benefits of optimistic programming: A measure of HOPE. In *Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC-4)*, 1995.

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

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

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

[11] H Lutfiyya and C Cowan. Optimistic language constructs. In *Proceedings of the ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages*, 1995.

[12] S Matsuoka, T Watanabe, and A Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP 91 Conference*, pages 231–250, 1991.

[13] P Tinker. Task scheduling for general rollback computing. In *Proceedings of 1988 International Conference on Parallel Processing*, pages 180–183, 1988.

[14] T Watanabe and A Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA 88 Conference*, pages 306–314, 1988.

[15] 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.