GENERAL PURPOSE OPTIMISTIC PARALLEL COMPUTING

Stephen Turner and Adam Back

Department of Computer Science,
University of Exeter,
Prince of Wales Road,
Exeter EX4 4PT England
Email: steve@dcs.exeter.ac.uk

March 31, 1994

Abstract

In this paper we discuss our research into the use of optimistic methods for general purpose parallel computing. The optimistic execution of a program can allow code to be run in parallel which static program analysis might indicate was sequential. However, this also means that it may do some work which is later found to be wrong because of a causality violation. Optimistic methods use a detection and recovery approach: causality errors are detected, and a roll-back mechanism is invoked to recover. These techniques have been used very successfully in parallel discrete event simulation, but are not as yet widely used in general purpose computing.

Our research involves the development of a compiler which converts a conventional object-oriented program into a form which has calls to an optimistic run time system. The generation of time-stamps which allow for loops with an unknown number of iterations is discussed. We also describe some of the portability issues which we have addressed in this project: in particular, our use of p4, a cluster based communications library which provides the basis for portable, heterogeneous parallel computing. We describe our implementation of p4 on the transputer architecture, including some extensions to support the optimistic execution of programs, and also an outline of our work in retargetting the GNU C and C++ compilers for the transputer.

1 Introduction

Object-oriented programming has become widely accepted in recent years as a way of providing information hiding and encapsulation. Abstract data types are implemented in a language such as C++ through the *class* mechanism. A class allows a programmer to hide implementation details from the user by ensuring that all operations on an object of that class are performed using *methods* that are explicitly made *public*. Classes can be defined using an inheritance mechanism that supports the development of large programs and the re-use of code.

Because of this encapsulation, it seems natural to execute an object-oriented program in parallel by assigning different objects to different processors. Each object would have a server [15], which communicates with other servers by means of message passing. When a method is invoked, a thread is created by the server to execute that method. Thus, in addition to the parallel execution of objects, it is also possible to have parallelism within an object by executing invocations of its methods concurrently.

However, in the C++ language, methods are assumed to be invoked sequentially: in allowing methods to be executed in parallel, it is necessary to take into account possible interactions between methods. The problem here is that changes to the implementation of an object can affect the interaction patterns of the methods of that object and hence the parallelism that is possible in the user of that object. Servers for different objects must also be constrained to execute in parallel only when no data dependencies exist between those objects.

The automatic parallelization of a program involves *data dependence* analysis. For example, a statement which modifies a variable cannot normally be executed in parallel with a statement which accesses that same variable. If two statements involve method calls, interprocedural analysis is required to find the sets of variables that are accessed or modified by those methods. Interprocedural analysis is very difficult because side effects often force the analysis to make conservative assumptions. In general, it is only possible to produce summary information for each method which may be too imprecise to be of practical use.

The optimistic execution of a program can parallelize some code which static program analysis would indicate was sequential. This is because the optimistic parallel execution can presume that a particular event corresponding, say, to a branch of a conditional will not occur. A conservative approach would have to take the worst case: if it is possible for the branch to be taken it must be assumed that it is always taken. This means that the optimistic approach can execute more of the program in parallel, but it also means that it may do some work which it later decides is wrong because of a causality violation. A mechanism is therefore required to recover from such causality violations.

The remainder of this paper is structured as follows. In sections 2 and 3, we present the underlying theory of *virtual time* and the "Time Warp" mechanism for optimistic execution. Section 4 gives a simple example of the use of this approach in the parallelization of an object-oriented program. We see that general purpose optimistic parallel computing requires a flexible time-stamp allocation scheme and this is discussed in section 5. We then describe some of the portability issues which we have addressed in this project and finally present our conclusions.

2 Virtual Time

The concept of an artificial time-scale was originally proposed by Lamport [14] in 1978 as a way of defining a global ordering of events in a parallel or distributed system. A causal relation between two events A and B, such as the sending and receiving of a message, can be represented as $A \rightarrow B$ and we say that A *happened before* B. Events A and B are said to be *concurrent* if neither can causally affect the other, that is $A \not\rightarrow B$ *and* $B \not\rightarrow A$. Thus the ordering of events defined by the *happened before* relation is a partial ordering.

Lamport showed how to extend this partial ordering to a total ordering by assigning a logical clock value or *time-stamp* to each event in such a way that if $A \rightarrow B$ then the time-stamps will be ordered $T_A < T_B$. The converse is not true, $T_A < T_B$ does not imply that event A causes event B: all we can say is that event A *may* cause event B. However, if the time-stamps for two events are equivalent, $T_A = T_B$, then we know that A and B must be concurrent, that is there is no causal relation between these two events.

In 1985, Jefferson [12] introduced the idea of *virtual time* as a new paradigm for parallel computing. In many ways, this is the reverse of Lamport's approach: we assume that every event is labelled with a clock value from some *totally ordered* time-scale in a manner consistent with causality. A partial ordering may then be obtained which allows a fast concurrent execution of those events. This is achieved using an optimistic execution mechanism known as "Time Warp": each process executes without regard to whether there are synchronization conflicts with other processes. When a causality error is detected, a process must be rolled back in virtual time, and then be allowed to continue along new execution paths. Thus, the Time Warp mechanism is the inverse of Lamport's algorithm.

Although virtual time has been used in distributed database concurrency control [16], its main success has been in parallel discrete event simulation [8, 13, 19]. Here, the interactions between the objects of the simulation are modelled by the exchange of time-stamped event messages. The Time Warp mechanism allows different nodes of the parallel computer to execute events out of time-stamp order provided that no causal relation exists between them. Although events of the simulation are executed in parallel, the mechanism guarantees that the same results are obtained as would be the case with a simulator that executed the events sequentially in non-decreasing time-stamp order.

3 The "Time Warp" Mechanism

In discrete event simulation the physical system can be modelled in terms of events, each of which corresponds to a state transition of an object in the physical system. Simulation events each have a time-stamp which corresponds to the time at which the event would occur in the system being modelled. A sequential simulator proceeds by taking the event with the lowest time-stamp and simulating its effect: this may alter the state of the object being modelled, and also create further events which will be scheduled at some future simulation time. The simulation moves forwards in simulation time by jumping from the the time-stamp of one event to the next. This is in contrast to time-driven simulation methods where time moves forward uniformly. The simulation is complete when there are no more events to simulate.

In parallel discrete event simulation [8], the physical system is modelled by a set of processes which correspond to the interacting objects in the physical system. The interactions between the physical objects are modelled by the exchange of time-stamped event messages. Parallelism is achieved by placing these processes on the different nodes of a parallel computer.

Each process has a local virtual clock which denotes the simulation time of that process. A process's local clock will be increased to the time stamp on each event message as it is processed. In this way the local clock of each process will advance according to the time-stamps of events. The progress of the simulation is measured by the global virtual time (GVT), which is the minimum of the local clocks (and time-stamps of messages in transit).

We must ensure that events *for each object* are simulated in non-decreasing time-stamp order. To see why this is necessary, consider a process receiving a message which has an earlier time-stamp than the local virtual time. This means that the message corresponds to an event which should have been executed earlier, and events have been executed past this which could have been affected by that event. This is known as a causality error, from the cause and effect principle: the fact that events in the future cannot affect events in the past.

There are two approaches to ensuring that causality is not violated in a parallel discrete event simulation: conservative and optimistic. Conservative approaches [7] avoid the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event. This will be when all events which could affect the event in question have been processed.

Optimistic methods such as Time Warp [12] use a detection and recovery approach: causality errors are detected, and a roll-back mechanism is invoked to recover. A roll-back will be required when a causality violation is detected due to an event message arriving too late (such a message is known as a *straggler*). The roll-back must restore the state of the process in question to a point in time before the time-stamp of the straggler. To enable this to occur, processes must periodically save their states. After the roll-back, execution resumes from that point in time. In the course of rolling back a process, anti-messages are sent to "undo" the effect of previous event messages which should not have been sent. An anti-message will annihilate the corresponding real message and possibly cause the receiving process to roll back. This may in turn require the sending of more anti-messages.

The optimistic approach is less restrictive and so potentially allows for more parallelism in the execution of the simulation. This is because it allows events to be processed out of time-stamp order even if there is a possibility of one event affecting the other. Optimistic methods work well if the number of cases where a causal relation actually exists between events is comparatively infrequent. However, there are overheads incurred in maintaining the information necessary for roll-back, and the roll-back itself consumes execution time. We have to be careful that the roll-backs are not too frequent or we could lose more than we gain. There is a balance which must also be achieved in deciding the frequency of state saving. Too often and state saving will become too much of an overhead, too infrequently and it is necessary to roll back further.

We can take any general purpose computation and split the program into blocks of code which will correspond to events. This idea forms the basis of our research into automatic program parallelization: by assigning a time-stamp to each program control structure, the program may be executed in parallel using an optimistic mechanism, but will give the same results as it would do if the control structures were executed sequentially in strict time-stamp order, in the same way as a parallel simulation using the Time Warp mechanism gives the same results as a sequential one.

In applying the optimistic technique to general purpose parallel computing, the program becomes the "physical" system and the optimistic simulation of this system the execution of the program. The aims are to make use of more of the available parallelism than is possible with automatic parallelization schemes based on static program analysis. Some work has been done by Bacon [3] on the optimistic execution of CSP (Communicating Sequential Processes) [10], but the use of optimistic execution as a parallelization tool has been largely unexplored.

4 An Example of Parallelization

To illustrate these ideas, we present an example which shows the parallelization of an object-oriented program involving matrix multiplication. Figure 1 shows how each object of class *Matrix* is implemented as as array of *Vector* objects. The multiplication of two matrices $A \times B$ is implemented by setting element i, j of the result to the inner product of the i^{th} row of A and the j^{th} column of B.

Each object (i.e., an instance of a class) is treated as a Time Warp process with its own local virtual clock. When a method is invoked, the virtual clock of that object is increased to

```
/* Vector implemented as an array of float */
class Vector {
  float *vec;
  int size;
public:
  Vector(int n);
  float innerprod(Vector& v);
  float& operator[](int i);
  ...
};

/* Inner product of vector with another */
float Vector::innerprod(Vector& v) {
  int k;
  float temp = 0.0;
  for (k = 1; k <= size; k++)
    temp = temp + vec[k] * v[k];
  return temp;
};

/* Matrix implemented as array of Vector objects */
class Matrix {
  Vector *mat;
  int size;
public:
  Matrix(int n);
  Matrix& multiply(Matrix& m);
  Vector& column(int i);
  Vector& operator[](int i);
  ...
};

/* Multiplication of matrix with another: element i,j
   given by the inner product of row i and column j */
Matrix& Matrix::multiply(Matrix& m) {
  int i, j;
  Matrix& mtemp = *new Matrix(size);
  for (i = 1; i <= size; i++)
    for (j = 1; j <= size; j++)
      mtemp[i][j] = mat[i].innerprod(m.column(j));
  return mtemp;
};

main()
{ Matrix A(N), B(N), C(N), D(N), E(N), F(N), G(N);
  ...
  E = A.multiply(B);
  F = C.multiply(D);
  ...
  if (condition)
    G = E.multiply(F);
  ...
}
```

Figure 1: Parallelization involving Matrix Multiplication

the time-stamp of the invocation. This, in turn corresponds to the time-stamp of the object which invoked that method. If a method is invoked with a time-stamp earlier than that of a previous invocation, the object must normally be rolled back to recover from the causality error. An exception is where the methods are *read-only* and do not affect the state of the object. Such methods may be executed out of time-stamp order using a principle similar to that of *lazy cancellation* [9].

In figure 1, we can see that the Time Warp mechanism would allow the multiplication of $A \times B$ to proceed in parallel with that of $C \times D$, since these are method invocations on different objects, each with its own local virtual clock. No roll-backs would result since there are no data dependencies between these two statements. It is also possible to execute in parallel with the matrix multiplications, the succeeding statements in the program, including the conditional statement. If the condition is true, the statement which assigns a value to G may be executed before the correct values of E and F have been computed. However, the assignment method which is invoked on E will then have a smaller time-stamp than the multiply method and the Time Warp mechanism will recover from the causality violation.

The *for loop* of the matrix multiplication may itself be executed in parallel since it invokes the innerprod method on different objects, in this case the different rows of the matrix. Again, this will cause no roll-backs since there are no data dependencies between the individual iterations of the loop. However, it is not necessary to perform static analysis to be sure of the absence of data dependencies: if such dependencies were to exist, the Time Warp mechanism would take the appropriate recovery action.

There is a particular problem with the optimistic execution of a general purpose program that does not arise in parallel discrete event simulation: it may be necessary to allocate an arbitrary number of new time-stamps between any pair of previously allocated time-stamps. If we consider a program control structure consisting of an unbounded loop, we could be executing the individual iterations of that loop in parallel with the events corresponding to the code following the loop. We therefore need to allocate the time-stamps for the events following the loop before we have allocated all of the time-stamps for the iteration events.

5 Representation of Time-stamps

In this section, we discuss a solution to the problem of time-stamp generation, which allows us to allocate a sequence of time-stamps of any length between any pair of previously allocated time-stamps. Our representation of time-stamps must also be such that the operations to compare and generate the next time-stamp in a sequence are efficient. A variable length time-stamp is proposed which satisfies these requirements.

We define our variable length time-stamps in the following way:

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

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. The fractional value of a time-stamp (l, v) is

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

We define an ordering relation on time-stamps:

$(l_a, v_a) = (l_b, v_b)$ iff $f_a = f_b$ and $l_a = l_b$

$(l_a, v_a) < (l_b, v_b)$ 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. We can represent time-stamps pictorially in figure 2 as the nodes of a binary tree.

The ordering defined by the pre-order traversal of the time-stamp tree is the same as the ordering defined above. The time-stamps which fall between a pair of consecutive time-stamps of a particular length are the descendants in the time-stamp tree of the first time-stamp.

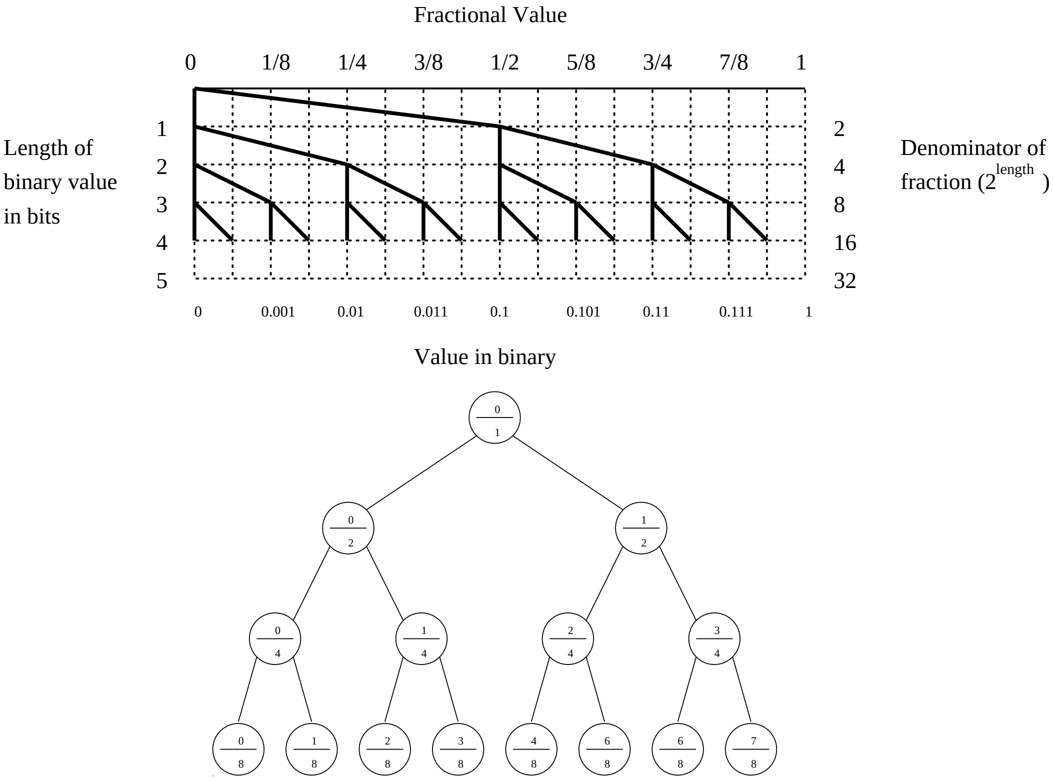

Figure 2: Variable length time-stamps

The left child of time-stamp (l, v) is $(l + 1, 2v)$ and the right child $(l + 1, 2v + 1)$. More generally at a depth d below l, there is a set of time-stamps at level $l + d$, which fall between (l, v) and the consecutive time-stamp at the same level $(l, v + 1)$:

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

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 value of a pair of time-stamps is 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. What we are proposing is that trailing zeros become significant in the ordering, so that $0.10 \neq 0.1$.

For unbounded loops we cannot predict the number of time-stamps to allocate. We therefore allocate an initial amount of time-stamps N where $N = 2^b$ and b is the number of bits which will be used by the initial time-stamp allocation. We use the first half of these time-stamps for the first 2^{b-1} time-stamps. If more time-stamps are required we create complete binary trees of depth b under all of the remaining 2^{b-1} time-stamps. This will create 2^{2b-1} additional time-stamps. We again use the first half of these (2^{2b-2}) and reserve the second half for further time-stamps.

The choice of b is important in this scheme as the efficiency of dynamic allocation compared to the static allocation used for fixed bound loops depends on the choice of b. Further details of our time-stamp mechanism may be found in [2].

6 Portability Issues

In this section, we discuss the suitability of p4 as a portable base for our research into the use of general purpose optimistic parallel computing. The p4 parallel library [5, 6], which originates from the Argonne National Laboratory, provides a portable programming model for a large set of parallel machines. The model combines message passing with shared memory to form a cluster based model of parallel computing.

We also present our extension to p4, "lightweight p4", designed to allow programming using lightweight pre-emptive processes while retaining portability. This was designed to provide a way of controlling the scheduling policy at run time, as required by the Time Warp mechanism. We wished to be able to do this in a lightweight process environment, but at the same time we did not wish to lose the portability of our system. Finally, we outline our work in retargetting the GNU C and C++ compilers for the transputer.

6.1 Clusters

The cluster model in p4 groups together processes into clusters. All p4 processes can communicate via message passing, and p4 processes within the same cluster can share memory. With standard p4 on Unix workstations, the p4 process corresponds to a Unix process, and a cluster is formed by a set of Unix processes sharing memory. Communication between processes in the same cluster is implemented using shared memory, and between processes in different clusters by using sockets.

With shared memory multiprocessors, the shared memory can be implemented in hardware between p4 processes which are running in parallel, as opposed to the time sharing of processes on a single processor Unix machine. On a distributed memory architecture, the p4 processes within a cluster will all reside on the same processor. Message passing is implemented using the underlying communications mechanism provided by the hardware.

The p4 system includes monitors within the programming model to control shared memory. It provides synchronous and asynchronous message passing facilities between p4 processes. Messages may be typed: it is possible to request a message of a particular user defined type, and the p4 system will automatically buffer messages of different types until they are requested. Similarly, it is possible to request a message from a certain p4 process, and messages from other processes will be buffered until required.

Facilities are provided to define a set of machines and a description of how clusters map on to those machines. This configuration is pseudo-dynamic in that it can occur at run time, but it must remain fixed after the p4 library has been initialised. There is also provision for dynamic process creation, although such processes are not able to communicate via message passing.

6.2 Heterogeneous operation

The p4 system provides a framework within which it is possible to have a combination of machines of different classes presenting a common model. It allows us to execute programs on combinations of parallel machines, workstations and multiprocessors. In this way we could develop an application which would run simultaneously on a set of p4 clusters. Some clusters could be located on shared memory multiprocessors (for example SGI multiprocessors), some on Unix workstations (for example SUN Sparcstations) and some on distributed memory parallel machines (such as transputers).

To allow messages to be passed between machines of different architectures, p4 uses the XDR (eXternal Data Representation) library. XDR [18] provides a standard representation for float, double, int, long into which messages must be translated on send and from which they must be translated on receive. In this way messages can be passed between little-endian and big-endian machines, and between machines which do and do not use the IEEE floating point format.

Our transputer implementation is thus able to operate in a heterogeneous manner with Unix workstations, and shared memory multiprocessors. Transparently to the p4 applications programmer, communication in our example network will be taking different paths depending on the location of the p4 processes involved:

- Transputer $\leftrightarrow$ Transputer, via transputer hardware links,
- Transputer $\leftrightarrow$ Unix machine, via transputer socket library and Unix sockets.
- Unix machine $\leftrightarrow$ Unix machine, via Unix socket library.

6.3 The Transputer Implementation

The transputer implementation differs from the generic Unix implementation in two areas: in the way processes are created, and the message passing mechanism.

In implementing p4 processes on a transputer architecture, there is a problem in associating a local data area for each p4 process. This local data area is required to hold the p4 process id, XDR data buffers and other data which is required on a per process basis. This is not a problem in the standard Unix implementation of p4 as Unix fork() is used to spawn p4 processes. Unix fork() copies both the data segment and the heap segment of the parent process. In this way the child process has its own copy of all global variables, and all program data which is stored on the heap.

The transputer process creation functions are implemented in hardware and are orders of magnitude more lightweight than the standard Unix fork(). Process creation on transputers does not involve copying the data segment or the heap segment. The problem of finding a place to store local per process data in the transputer process environment is solved efficiently by using an aligned workspace area.

A workspace pointer is used by a process as its stack pointer, and varies according to the current depth of function invocation. The workspace pointer also has a second function as a process identifier. When a process is descheduled by the hardware scheduler its workspace pointer is used to identify it. Negative offsets from the workspace pointer are used by the scheduler and some transputer instructions, to form a linked list of processes and to store information about the process while it is descheduled. A process can determine the value of its workspace pointer using a single CPU instruction. By allocating a p4 process's workspace at an aligned address, it can determine the start of its workspace easily. The p4 per process data is then stored at the start of the p4 process's workspace, while the process's stack will grow down from the top of the workspace with the workspace pointer pointing to the stack top.

The generic Unix implementation of p4 uses sockets for communication. Vendor specific communication libraries are used for communication on parallel machines with specific communication hardware. We have implemented the p4 message passing calls on the transputer using the Inmos VCR [11] system, which provides virtual channels which can be placed between processes on any processor in a transputer network, the necessary through routing and multiplexing being performed by the VCR (on the T805, this is performed by software).

In order to implement p4's point to point communication in terms of virtual channels we provide a configuration which links each processor to each other processor in the network. Then we provide a multiplexor and a demultiplexor process for each virtual channel. The multiplexor for a particular channel will forward all messages destined for the remote processor via the virtual channel. Further details of the implementation of the message passing mechanism may be found in [1].

6.4 Performance

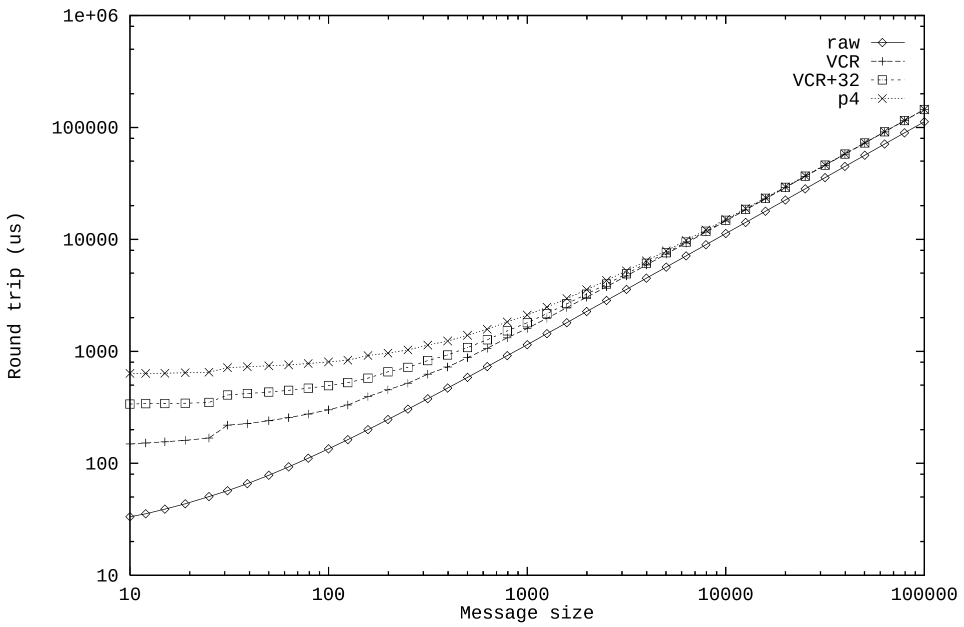

Figure 3: “Ping” 2 nodes: round trip time

Figure 3 demonstrates the efficiency of the p4 system on T805 transputers, as compared to the basic Inmos C communications library using virtual channels. These times are obtained with a simple "Ping" application where a message is sent from one transputer to an adjacent transputer and then back to the originator. This is repeated a large number of times in order to obtain accurate measurements. It can be seen that the communications overhead of p4 is reasonable, in view of the additional functionality over the virtual channel routing software that p4 provides.

The lines labelled "VCR" and "raw" show the performance of fixed sized messages with and without the Inmos VCR. The performance difference between raw and VCR messages is due to packetization and multiplexing of multiple "virtual" channels on a single hardware link. "VCR+32" shows the performance that would be achieved with variable sized messages where a 32 byte header is sent as a separate message before the actual message. This is the size of the header used by p4 and would contain in p4 information about the message size, type, sender's p4 process id, whether XDR is to be used etc. This is shown for comparison with the actual p4 communication performance labelled "p4" to distinguish the overhead p4 incurs for variable size packets from the extra overhead due to buffer management and the provision of typed messages.

6.5 The "Lightweight p4" extension

We provide a lightweight process programming library where each process has a priority level which determines the scheduling. Processes with a priority level equal to that of the highest runable process are executed using a round robin policy. When processes at the current highest priority all become blocked or die, processes at the next lower priority run. If at any time a higher priority process becomes ready, the lower priority process is pre-emptively descheduled. The library is able to pre-emptively schedule and deschedule processes within two transputer time-slices. That is, if a higher priority process becomes ready while a lower priority process is running, the lower priority process will run for at most two time slices before being descheduled.

A multi-priority scheduler with the capability of pre-emptively descheduling processes has been constructed based on the work of Shea *et. al.* [20]. The library allows processes to be externally suspended, resumed and killed, and also allows control over each process's priority.

6.6 Retargetting the GNU C and C++ compilers

We wish to be able to develop programs that are portable across a wide range of parallel architectures. In particular, we would like to be able to transfer our programs, without modification, between transputer systems (both T805 and T9000 [17]), networks of workstations, and shared memory multiprocessors. We also wish to be able to develop programs which run on a heterogeneous system involving any or all of the above architectures.

With this in mind, we have retargetted the GNU CC compiler (an integrated C and C++ compiler) so that it can generate code for the transputer (currently the T805, but we intend also to develop a code generator for the T9000). The GNU compiler is in the public domain and has been designed to make it easily portable to new systems. The front end of the compiler produces abstract code in a LISP-like language called *RTL* (Register Transfer Language). Code for a particular machine is generated by pattern matching the RTL file produced by the front end against a *machine description* file which specifies the actual code for each abstract instruction.

One of the main considerations in retargetting the GNU compiler is that it should generate code which utilises the standard Inmos C libraries [11]. To do this, it is necessary to adopt a stack layout which is compatible with that used by the Inmos compiler. Also the global and static data must be accessed in a compatible way, using a *GSB* (Global Static Base) which points to a list of *LSB* (Local Static Base) pointers for each local static area.

One of the assumptions made by the designers of the GNU CC compiler was that the target machine would have several general purpose registers available to it. This is not the case with the transputer architecture. Since this is a fundamental part of the compiler, it is necessary to define a set of pseudo-registers which can be accessed very quickly. These are placed in the first 16 positions above the current workspace pointer, so that they can be accessed with a short instruction. Pseudo-registers 0 to 6 are special purpose registers, such as the frame pointer, etc., whereas those from 7 to 15 are general purpose. Details of the GNU CC abstract machine model for the transputer are given in [4].

7 Conclusions

In this paper, we have shown how it is possible to parallelize an object-oriented program using optimistic execution techniques based on the concept of virtual time. Optimistic methods use a roll-back mechanism such as Time Warp to recover from causality violations: this allows code to be executed in parallel which static program analysis might indicate was sequential.

In order to reduce the overheads of the Time Warp mechanism, it is still useful to perform some static analysis. There is little point in executing code optimistically if it is certain that a roll-back will occur. Static program analysis might also indicate situations where roll-back will never occur, in which case state saving can be avoided. However, to obtain such precise information about data dependence is very difficult, particularly when it involves inter-procedural analysis.

It is where the information that is provided by static analysis is uncertain or incomplete that the optimistic approach comes into its own. It does not matter if we make an incorrect assumption about the data dependence between two statements: if we gamble that the two statements are independent and execute them in parallel when in fact there is a causal relation, the Time Warp mechanism will detect the causality violation and recover from it.

This paper has also shown how it is possible to provide a programming environment in which parallel programs may be developed in a way which is independent of the particular architecture. We believe that the cluster based model of computation, as provided by p4, is a useful model of parallel computation for a wide range of applications. It presents the programmer with a uniform model which enables the development of efficient, portable applications that can be run in a heterogeneous environment.

Our results suggest that it is possible to avoid any significant loss of efficiency even on platforms such as the transputer which support a fast process switching mechanism and high speed communication links. The "lightweight p4" extension that we have presented makes use of the fast process creation and switching mechanism of the transputer and provides a convenient framework for our research into optimistic execution mechanisms. Finally, the retargetting of the GNU C and C++ compilers ensure the portability of our approach and the ability to run in a heterogeneous environment.

8 Acknowledgements

The authors would like to acknowledge the work of Chris Berry in retargetting the GNU C and C++ compilers for the T805 transputer. We are also grateful to Inmos for information on the ANSI C compiler and its run-time system.

References

[1] A Back and S J Turner. Portability and parallelism with lightweight p4. In *BCS PPSG Conference on General Purpose Parallel Computing*, 1993.

[2] A Back and S J Turner. Time-stamp generation for the parallel execution of program control structures. Technical report, R289, Department of Computer Science, Exeter University, 1994.

[3] D F Bacon. Optimistic parallelization of communicating sequential processes. *Association of Computing Machinery*, 1991.

[4] C Berry, A Back, and S J Turner. A GNU CC compiler for the transputer. Technical report, R295, Department of Computer Science, Exeter University, 1994.

[5] R Butler and E Lusk. User's guide to the p4 parallel programming system. Technical report, ANL-92/17, Argonne National Laboratory, 1992.

[6] R Butler and E Lusk. Monitors, messages, and clusters: the p4 parallel programming system. Technical report, Argonne National Laboratory, 1993.

[7] K M Chandy and J Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Engineering*, SE5(5):440–452, 1979.

[8] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1989.

[9] A Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Procedings SCS Distributed Simulation Conference*, pages 61–67, 1988.

[10] C Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[11] Inmos. *ANSI C Toolset User Guide*, 1992.

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

[13] JPL. *Time Warp Operating System User's Manual.* Jet Propulsion Laboratory, 1991.

[14] L Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] T G Lewis and H El-Rewini. *Introduction to Parallel Computing.* Prentice Hall International, 1992.

[16] M Livesey. Distributed varimistic concurrency control in a persistent object store. Technical report, University of St. Andrews, 1990.

[17] M D May, P W Thompson, and P H Welch. *Networks, Routers and Transputers*. IOS Press, 1993.

[18] SUN Microsystems. *Network Programming Guide*, 1990.

[19] M Presley, M Ebling, F Wieland, and D Jefferson. Benchmarking the time warp operating system with a computer network simulation. In *Procedings SCS Distributed Simulation Conference*, pages 8–13, 1989.

[20] K M Shea, M H Cheung, and F C M Lau. An efficient multi-priority scheduler for the transputer. In *Proc. 15th WoTUG Technical Meeting (Aberdeen)*, pages 139–153. IOS Press, 1992.