# PARALLELIZATION OF GENERAL PURPOSE PROGRAMS USING OPTIMISTIC TECHNIQUES FROM PARALLEL DISCRETE EVENT SIMULATION

Adam Back

Submitted by Adam Back to the University of Exeter as a thesis for the degree of Doctor of Philosophy in Computer Science in the Faculty of Science.

December 1995



I certify that all material in this thesis which is not my own work has been identified and that no work is included for which a degree has previously been conferred on me.

Adam Back

// Abstract

This thesis discusses the use of the virtual time paradigm as a mechanism for parallelizing sequential object-oriented code.

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 of dependencies in the coding. The inherent problem is that compile time data-dependency analysis presents an overly restrictive view of dependencies in a program.

The hypothesis of this thesis is that the use of causality violation detection and roll-back makes it possible to parallelize sections of code for which a conventional compiler would only be able to generate sequential code. This approach, which is based on the virtual time paradigm of parallel discrete event simulation, is generalisable to any code structures, both regular and of a more irregular structure. Its particular area of strength is in irregular program structures and execution patterns.

A design is presented for a parallelizing compiler for the C++ language. In the design, user selected C++ objects are treated as server objects obtained by transformation from the selected objects. These are distributed across the nodes of a parallel computer. The asynchronous dispatch of the server object methods introduces parallelism into the execution.

The generalisable nature of the parallelization technique 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.

# Acknowledgements

I acknowledge the help of my supervisor Steve Turner who provided guidance throught the project, and with whom I have had many discussions on details of the research. I acknowledge my second supervisor Brian Lings, and my thesis committee chair Derek Partridge for useful guidance during the research stage, and for helpful directions with the layout and form of the thesis. I acknowledge email conversations with Paul Wonnacott (DRA, Malvern) on the subject of the hybrid incremental state–saving algorithm.

This research was supported by EPSRC studentship award 92700646.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Parallelizing Compilers

This research is in the area of compilers for automatically parallelizing sequential code. There have been two main categories of parallelizing compilers: the early FORTRAN vectorizing compilers, and compile–time data–flow analyzing compilers.

Most parallelizing compilers to date have been based broadly on "compile–time data flow analysis" [78]. That is, their selection of program constructs to parallelize is based purely on data flow information obtained by examining the source code at compile time. It is not in general possible to predict actual data flow at compile time — the best that can be hoped for in the general case is to obtain a *set* of data flow paths through the program. This is a consequence of the fact that it is not generally possible to predict the branches taken in a program.

Few compilers try, or are very successful at predicting program execution beyond recognition of narrowly defined loop constructs. This problem is exacerbated by the aliasing problems in modern languages, C++, and C notably being the worst offenders, and also the most popular languages. FORTRAN is somewhat better in this respect.

Where prediction is not possible, sections of code will be treated as sequential which might otherwise have been parallelized. As an example, Figure 1 shows a loop with a conditional statement that adds a data dependency between iterations of the loop. If the expression used to decide whether the conditional branch is taken cannot be predicted at compile time, a compiler relying only on compile-time data-dependency information would have to consider all iterations of the loop to be dependent.

```
for (i = 0; i < N; i++)
{
  e();
  if (f())
  {
    g();
  }
}
```

Figure 1: Conditional dependency

As it could be the case that the conditional branch is only taken once, say on the last iteration of the loop, using optimistic execution it is possible to execute iterations of the loop in parallel. If a data dependency exhibits itself at run-time as a causality violation, then the system rolls-back and re-executes part of the code.

Early vectorizing FORTRAN compilers were amongst the first parallelizing compilers. The techniques used were mostly centered around vectorizing `DO` loops [48].

An example of a FORTRAN loop transformation is shown in Figures 2 and 3. In Figure 2 the outer loop is parallelizable, using a *loop interchange* [10] transformation, the inner loop can be made parallelizable

There are vectorization strategies which make it advantageous to have loops with

```
DO i = 1, n
  DO j = 1, n
    total[i] = total[i] + a [i,j]
CONTINUE
```

Figure 2: Original loop nest

```
DO i = 1, n
  DO j = i, n
    total[i] = total[i] + a[i,j]
CONTINUE
```

Figure 3: Interchanged loop nest

different *strides* and different granularity for the parallelizable loop. The loop interchange example is changes the stride from stride–n to stride–1. (This example was taken from [10]).

This technique works well enough for the narrow range of applications where the loops have independence between loop iterations, or transformations can be done to produce usable independence. Parallelizing matrix operations was another approach which works well on vector processor based parallel computers.

The vectorizing approach covers a particular range of problems, but there are plenty of scientific, and conventional applications problems which do not fit well on vectorizing hardware.

The vector architecture has lost favour in recent years, with shared and distributed memory multi–processors being the favoured architecture. Also communications bandwidth between workstation farms has increased, and the line between workstations clusters, and parallel computers is becoming increasingly blurred. Cheap commodity RISC processors have meant that few parallel computer manufacturers are still using custom parallel CPUs. Most scientific computation is done with FORTRAN, although C++ has started to gain some favour in this area. For commercial applications, programming is moving away from COBOL and C to C++, . There is a general trend towards the use of multiprocessor machines; multiprocessor workstations, and even multiprocessor PCs are becoming available.

Some would argue that this is signalling the beginning of the end for classical sequential, single processor architectures. Although increases in processor power have been phenomenal, and exponential over the last decades, it is believed that there is a physical limit as to how fast a sequential processor can be built. As this limit is neared, parallel architectures will be the only way to go for increased performance.

This trend is already evidenced in the design of RISC processors. High levels of micro level parallelism are being used: 4, 8, and even 64 way superscalar processors are appearing which issue up to 64 RISC instructions simultaneously. The use of multiprocessor workstations is also in evidence, and it is no longer unusual to see between 2 and 4 processors sharing memory in a single workstation.

The next stage is the convergence in performance characteristics between single and multi–processor workstation clusters, and parallel computers with dedicated high speed interconnects. This also is already in progress, LANs are moving to higher speed interconnects — FDDI, HIPPI in place of slower ethernets. Parallel computers are here to stay. Depending on the tail off in processor speed increases, and the resulting price performance advantages gained by parallel architectures, parallel computers will be increasingly used in place of sequential.

All of these hardware trends have not been followed as well by parallel programming methodologies. Apart from multi–user systems simply making use of the extra parallelism to support more users, parallel programming options used include communications libraries, supporting CSP [37] style programming, and the newer HPC++ [77], which is still in its early stages, and the corresponding high performance FORTRAN [40].

This all indicates the need for parallel programming methods. Explicit parallel programming is fraught with difficulties. Few would disagree that parallel programming is harder than sequential programming because there is another dimension to deal with: many parallel processes, their synchronization, and the ensuing dead–lock, live–lock, and communication problems. There is a large quantity of “dusty deck” code in existence, and there are many programmers who are fluent with sequential programming, but not with parallel programming.

For these reasons automatic parallelization is an important area. The use of optimistic techniques applied to general purpose programming yields an automatic parallelization method. A parallel execution can be derived from an object oriented program automatically. The parallel execution of this program can be guaranteed to have identical semantics to the sequential, and the programmer need not be aware even that there is a parallel implementation, the effect should be transparent.

## 1.2 Brief Introduction to Parallel Simulation

There is a class of simulations called Discrete Event Simulations (DESs) [29]. A discrete event simulation is a simulation in which the system is represented by simulation objects and a set of events. The events (changes in state) are simulated to occur at particular simulation–times corresponding to the real world time of the system being simulated. DESs are “discrete” in the sense that each event is simulated as if it happened at a particular simulation time, as opposed to continuous as would be the case in a simulation where simulation time progresses uniformly and continuously. In a discrete event simulation real world objects are simulated by simulation objects, real world events which affect the objects are simulated by maintaining a list of simulation events which have prescribed effects on the simulation objects' states. The discrete event simulation proceeds by repeatedly taking the event with the lowest simulation time from the event list, and simulating its effect on the indicated object. The act of simulating an effect may schedule further events. In this way simulation time in the discrete event simulation progresses in discrete steps to the simulation times of successive events.

The term "Parallel Discrete Event Simulation" (PDES) is used to describe methods for parallelizing discrete event simulations. There are two basic approaches to PDES, *conservative* and *optimistic* [32]. Both work by distributing the simulation objects over the nodes of a parallel computer. Interactions between the physical world objects are modeled by passing time–stamped messages between the simulation objects. The time–stamp indicates the simulation time of the simulation event message. Each simulation object has a logical clock which contains its local view of the current simulation time. The object's local clock is increased to the time–stamp of the event messages as they are processed.

As the system is distributed, and the notion of simulation time is also distributed, it is potentially possible for events to be executed out–of–order, and some method is required to either avoid this, or to cope with it. If messages are processed out–of–order, it is likely that an event message will at some point arrive at a particular simulation object with a simulation time lower than the logical clock of the current message. This is an error, and is known as a *causality violation*. The reason that executing events out–of–order at a particular node produces an erroneous simulation is that some of the events which have already been processed at the time the *straggler* event arrives will be in the straggler event's future. The causality principle states that events in the future can not affect events in the past. If the straggler event situation above were allowed, the corresponding event could be affected by events in its own future. This is clearly paradoxical, and will in general result in an incorrect simulation.

Conservative simulators preserve causality by avoiding the possibility of straggler messages ever arriving out–of–order. Optimistic simulators allow such errors but detect them, and undo their effects when they do occur. The act of undoing prior work is known as *rollback* in the simulation literature [46]. The optimistic approach allows for more parallelism to be exploited, but there are costs associated with maintaining the information required for the rollback operation. The principle behind optimistic simulations is known as the "virtual–time paradigm", and a well known example of such a system is the Time Warp Operating System (TWOS) [47] Details of the virtual–time paradigm are given in Chapter 3.

This research investigates methods for applying optimistic simulation techniques to the task of automatically parallelizing general purpose object oriented programs. The term "optimistic execution" will be used to describe the virtual time paradigm as applied more generally, as the "optimistic simulation" does not clearly apply to a parallel execution of a program, though there are close similarities in the underlying mechanism. The term object oriented programming will be abbreviated to "object programming", or "OO programming" interchangeably in this thesis.

## 1.3 Thesis Hypothesis

The hypothesis of this research is that it is possible to design an optimistic execution scheme which allows sequential object oriented programs to be automatically parallelized. The main claim is that such a system would be able to parallelize some programs which conventional parallelizing compilers based on compile–time data flow analysis would be unable to parallelize.

## 1.4 Research Aims

The aim of this research is to provide access to increased amounts of parallelism in areas where conventional parallelizing compilers fail to do so.

The thesis presents a design for a general purpose optimistic parallelization scheme, suited to object oriented languages, and provides the ground work and a preliminary investigation of the feasibility of this approach, and of the expected performance characteristics of such systems.

The optimistic approach used is based on a technique used successfully in the field of parallel discrete event simulation. This thesis describes techniques to apply an optimistic execution scheme to general purpose object oriented programs. This parallelization method is particularly suited to exploiting parallelism which is missed by conventional compile–time data flow analysis compilers. The essence of the approach is that selected program objects will be placed on the processors of a distributed memory parallel computer. The invocations of the methods of these objects are replaced by asynchronous remote procedure calls (RPCs). The asynchronous nature of the remote procedure calls, and the program transformations needed to enable this are an important part of the design, and what creates the parallelism in the execution. Remote procedure call 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. A incremental state–saving memory system is used to encapsulate the behaviour of the optimistic execution system state, which will include out of sequence memory accesses.

Each of the objects placed on the processors of the parallel computer acts as a server object processing asynchronous RPCs, in increasing time–stamp order. As no other synchronization constraints are imposed, it is a very flexible parallel execution, one capable of executing in parallel anything which is logically parallelizable, regardless of the ability to detect the exploited independence at compile time. A time–stamped message arriving out–of–order for a particular server object signifies a causality violation. The server object rolls–back its state to just before the time–stamp of the straggler RPC message, and re–executes from that point forward. There are optimizations of this scheme, discussed more fully in Chapter 3, which avoid rollback in certain scenarios: *lazy cancellation*, and *lazy re–evaluation*.

Evaluation of the system is mostly qualitative, with example algorithms and programs used to to show that optimistic execution can be used to parallelize programs where conventional approaches would fail. Also estimates of system overhead to aid in the interpretation of these results are gathered by taking selected quantitative measurements of components of the optimistic execution system architecture.

## 1.5 High Level Overview

The layout of the rest of the thesis is as follows. Chapter 2 gives a literature survey of research in related areas. Parallel programming models with both explicit and implicit parallelism are described, as well as parallel object oriented systems. Chapter 3 describes the techniques of parallel discrete event simulation. Chapter 4 gives an overview of the optimistic parallelizing compiler design. The components of the design are described in the subsequent chapters. The design components are quite interrelated, so part of the aim of Chapter 4 is to describe the overall system in enough detail to provide sufficient background to allow the following chapters to be readily understood, without having to cross reference the interrelated chapters on the design modules. The design chapters are Chapter 5, which describes the transformation system, Chapter 6 which describes the high level components of the run–time system, Chapter 7 which describes lower level run–time system components, the underlying message passing system, and p4 portability layer, and Chapter 8 which describes the variable length time–stamps required. Finally, Chapter 9 presents an evaluation of the system, and Chapter 10 further work and conclusions. Appendix A describes C++ specific transformations.

This research has resulted in the publication of overviews [7] and [72] of the approach of using PDES techniques for automatic parallelization, of [8] on the subject of transformations for optimistic execution as described in Chapter 5, of [5] and [71] on the port of the p4 library to the T800 and T9000 series transputers respectively, and of [6] on the subject of variable length time–stamps as described in Chapter 8.

# Chapter 2

# Parallel Object Programming Systems

This chapter describes existing research in the area of parallel object programming systems. There are two basic approaches to parallel programming in general: explicit, and implicit parallel programming systems. Explicit object programming systems are where an object oriented language has been extended, or a library written, or a new object oriented language designed, where the idioms of object oriented programming have been combined with parallel programming techniques. Examples of object languages extended with parallel functionality are extensions of eiffel [53], Smalltalk [33], and recently many experiments [77], [20], [34] with adding parallel functionality to C++ [28], [69]. This chapter will focus on parallel extensions of C++ because C++ is more widely used than the other examples, because there are more examples of extended versions of C++, and because the system described in this thesis is also based on C++.

## 2.1 Parallel Programming Techniques

There are a number of paradigms which are combined with object languages resulting in different flavours of extended languages. The parallel programming paradigms used in parallel object programming systems essentially cover the full range of conventional parallel programming:

- Message Passing
- Remote Procedure Calls
- Data Parallelism
- Data Flow
- Semaphores
- Monitors
- Barrier Synchronization
- Futures

### 2.1.1 Message Passing

There have been a number of message passing libraries, and languages based on the message passing, or communicating sequential processes (CSP) [37] paradigm. Languages include CSP, OCCAM [39]. Message passing libraries include PARMACS [16], p4 [18], PVM [12], CHIMP [56], and MPI [58]. The P4 message passing library was used as the low level message passing portability layer for this research. A transputer port of the P4 system, and the message streams abstraction based on P4 are described in Chapter 7, the low–level run–time system chapter.

The Message Passing Interface (MPI) is a new standard which emerged too late for adoption as the library used in this system. MPI has multi–vendor support, and ports are becoming available for most parallel architectures. Had it been available sooner, MPI would have made a more portable, and future proof choice of message passing interface than P4. The differences between the subset of P4 used, and the corresponding subset of MPI are minimal, and there should be little trouble converting from the use of P4 to the use of MPI.

MPI (and p4) provide a kernel of message passing calls, which allow synchronous, and asynchronous message dispatch, tagged messages and barrier synchronization. MPI provides richer functionality, being largely a superset of what was argued for by the contributing vendors, and existing message passing library authors.

Generally, synchronous message dispatch means that both the sending and the receiving process must be ready to send, and receive respectively, before the communication can proceed. Once the send has completed, the sender knows that the receiver has at least started to receive, if not completed receiving. An asynchronous dispatch, however, means that messages can be dispatched without the sending process waiting for the message to be received. The message is buffered either at the sending or the receiving end, pending the actual receive, while the sending process continues.

Tagged messages allow receives to request only messages of a particular tag. This is a useful feature where several processes with different functions are sending messages to a single process, the process can impose restrictions on the order in which messages are received according to tags. P4, and MPI also allow the receiving process to place restrictions on the messages to receive based on the sending process's id. MPI takes things one step further, and has a two level tagging system, a higher level tag indicates which application, or library, a message belongs to. This allows multiple libraries to be used where more than one of the libraries itself internally uses message passing, and be safe from cross pollution of the message space. That is this feature ensures that the user does not accidentally receive a message from, or send a message conflicting with the internals of a library. It also ensures that libraries can be set to not conflict in terms of tags, by having libraries configurable by the library tag they use.

MPI also offers more flexibility in terms of groups of processes which can be addressed in broadcasts. With P4, communication groups are restricted to being all processes in a cluster (those processes residing on a single machine), or all processes. More General communication groups could be simulated to some extent by using p4's tagged message facility, but MPI provides more general arbitrary communication groups.

Since its introduction as a standard, MPI has become a popular target message passing layer for higher level parallelization tools.

### 2.1.2 Remote Procedure Calls

Remote Procedure Calls (RPCs) are a message passing abstraction which provides the ability to remotely invoke procedures in a distributed, or parallel computing environment. The UNIX "r" services (`rsh`, `rlogin`, `rup`, `rusers`, etc) are implemented as RPCs. The UNIX RPC implementation is heterogeneous through the use of a stub compiler, and standardized message passing convention based on the eXternal Data Representation (XDR) standard [55]. XDR provides an extensible framework for transmitting values across communication networks where the hosts may be of opposite byte–endianess, and/or be of different word size. XDR is extensible in that it provides a mechanism for building XDR composite structures corresponding to program level structures. An RPC stub compiler automates much the process of creating XDR messages, by automatically building stub functions which can be called directly from C/C++ which perform the message passing, and packaging. RPCs are also found in more tightly–coupled environments as a convenient paradigm for hiding message passing. Object oriented programs are particularly suited to this view in light of their natural message passing implementation.

### 2.1.3 Data Parallelism

This term includes vector parallelism. The vector parallel programming models are the Single Instruction Multiple Data (SIMD) model [30], as targeted by parallelizing vectorizing FORTRAN compilers. Parallel variants of FORTRAN allow the parallelizing aspects of vectorizing DO loops to be user directed by hints, or by commands given explicitly via a parallel language extension, as is done in FORTRAN 90 [45].

### 2.1.4 Data Flow

Data flow machines, such as the Manchester Data Flow machine [35] provide for data–flow parallelism at the hardware level. The higher level programming language level used for data–flow machines are single assignment languages such as SISAL. UFO (Unified Functions and Objects) [66] is an object oriented implicitly parallel system. The parallelism is extracted by using data flow information.

### 2.1.5 Semaphores

Semaphores [2] are a low level method used to control concurrent access to data in a parallel shared memory environment. A semaphore, or “lock” is used to prevent other processes accessing or modifying a variable during a *critical region*. A critical region is a section of program code during which access to the critical data must be exclusive. Semaphores provide a mechanism for controlling this. Semaphores can also be used as a low–level primitive for building more complex parallel constructs such as Monitors.

### 2.1.6 Monitors

A *monitor* [38] is a mechanism for concurrency control, a higher level abstraction similar in purpose to the use of semaphores to provide critical regions. The programming language Modula2 provides monitors as a built–in language construct to allow control of hardware devices. A monitor is a module, or object which has a published set of entry points. It is presumed that the environment allows intra–object parallelism – that there may be more than one invocation of these entry points active at one time. The monitor construct provides facilities to restrict the number of concurrent activations of entry points which may be active at any one time on a per entry point basis. This allows for the creation of restrictions such as multiple reader single writer at one time for the producer consumer problem.

### 2.1.7 Barrier Synchronization

Barrier synchronization [2] is a low–level parallel synchronization construct. A barrier provides synchronization between a set of parallel processes. As processes reach the barrier instruction, they wait for all of the processes in the group to reach the barrier before proceeding. Barrier synchronization has a simple implementation in terms of either message passing, or semaphores.

### 2.1.8 Futures

Futures [21] are a mechanism for indicating that a function can execute in parallel, until its result is requested. Explicit future *collection*, allows the application programmer to indicate when the value is required, by *collecting* the future associated with the asynchronously spawned future calculation. Some implementations use the reserved words `future` and `collect` to express futures. An example of the use of futures, using this explicit notation is shown in Figure 4.

```
{
  future float f1;

  f1 = func1( /* args */ );
  :
  collect f1;
  func2( f1 ); /* use value f1 */
}
```

Figure 4: Explicitly collected futures

Other implementations rely on implicit future collection, that is the programmer does not indicate when values should be collected from futures directly, rather the use of the future variable in an expression indicates that the future should be collected. The implicit collection of futures is shown in Figure 5, this is shown using the tera [1] syntax for future variables, the tera programming model has implicitly collected futures.

```
{
  future float f1;

  f1 = func1( /* args */ );
  :
  func2( f1 ); /* use value f1 */
}
```

Figure 5: Implicitly collected futures

The tera architecture [1] has hardware support for futures. In the tera architecture's associated programming model, any variable can be declared as a future. The tera architecture has a 64 bit word, and each word in memory, and in the processors has a set of 6 tag bits with reserved meanings. At the hardware level, the property of being a variable which can be used to spawn a future is indicated by one of the tag bits. To explicitly collect a future where the code has no reference for the actual future value returned by the future, a simple reference to the future variable collects the future, this is shown in Figure 6.

```
{
  future float f1;

  f1 = func1( /* args */ );
  :
  f1;  /* force collection */
}
```

Figure 6: Forcing future collection in system with implicitly collected futures

## 2.2 Parallel Object Oriented Programming

In this section a selection of existing work on parallel object oriented programming will be presented.

### 2.2.1 CORBA

CORBA (Common Object Request Brokerage Agent) [59] is a standard which defines object oriented RPCs, and local invocation semantics in a language independent way. It is similar in purpose to the RPC and XDR stub compiler used for the UNIX “r” services as described above in the section on RPCs. With CORBA, the interface specification is in a special purpose interface language which is automatically transformed to provide prototypes in the required language, such as C++, FORTRAN, Ada. One reason for listing CORBA as a significant area of research in parallel object oriented programming, is that it defines a standard interface for object oriented languages, and allows mixed language implementations of distributed object oriented systems. The interesting CORBA capability for this purpose, apart from the language independence, is the provision of a *flatten* operation. A flatten operation is one which converts an in–core image of an object into a message stream suitable for sending across a communications stream, or writing to a file. The flatten operation is commonly associated with persistent object oriented programming systems, which augment an OO programming language with the ability to have persistent objects – objects which continue to exist outside of the life–time of the program creating them.

### 2.2.2 Parallel C++ Extensions

The following parallel languages are all based on C++. There are other examples of parallel object oriented languages, parallel extensions to Eiffel [54], and from the other direction, a parallel language extended with object oriented features: DRAGOON an object oriented extension of Ada [3]. DRAGOON also has the novel feature of presenting an inheritance mechanism for inheriting concurrency control. Several of the C++ parallel extensions below offer support for multiple paradigms of programming, so that the views can be mixed, or a suitable paradigm chosen according to the task at hand.

### 2.2.3 HPC++

The HPC++ [77] initiative is part of ESPRIT SOFTPAR. The HPC++ group is working to provide a parallel extension to C++. Multiple parallel programming paradigms are supported: message passing, Remote Procedure Calls (RPCs) and Single Program Multiple Data (SPMD), i.e. data parallel programming. This set of functionality is viewed as the core required to implement higher level models which it is envisaged will be implemented as C++ libraries. HPC++ currently supports MPI, and PVM as target message passing layers.

### 2.2.4 ES–Kit

ES–Kit [22] is a parallel extension to C++. Futures combined with object oriented programming are used to create parallelism within the system. The ES–kit system makes clever use of the operator overloading features of C++ to provide a system which can be implemented by a C++ library only, avoiding the need for a special purpose compiler.

The `->` operator is overloaded to provide inter–object communication. This hides the use of message passing, and remote procedure calls behind normal method invocation. The `new` and `delete` operators are also overloaded to provide object creation and deletion.

### 2.2.5 Data Parallel C++

Parallel Vector C++ (PVC++) [61] is an extension of Parallel C++ which is itself an extension of C++ combining Parallel C with C++. PVC++ supports data parallelism, and inherits procedural parallelism from Parallel C++. The data parallelism is provided in terms of built–in vector types based on Blelloch's vector model [14]. The details of the vector implementation are hidden from the PVC++ application programmer.

### 2.2.6 Medium Grain Data–flow: Mentat

Mentat [34] is an example of a partially implicit object programming system. The Mentat Programming Language (MPL) is a C++ extension. Mentat is based on a macro level data–driven computation model, similar in concept to the data driven computation model used in data–flow machines, but with larger granularity. The MPL allows the programmer to select program objects as suitable for parallelization. The selected objects, known as `mentat` classes must conform to the MPL compiler enforced restricted set of functionality: they must use only parameter passing by value. The methods of the selected program objects are used as the actions in the macro data flow graph which is managed transparently by the mentat transformation and run–time systems.

The mentat system is an implicit object programming system in the sense that the user does not explicitly program the futures which are used internally, but the process is not fully implicit because the programmer must select which objects to parallelize, and to make efficient use of the system the programmer must be aware of the automatic parallelization that the compiler is carrying out behind the scenes.

## 2.3 Existing Work

There has been little research in the area of using techniques from parallel Discrete Event Simulation to parallelize sequential code. Bacon investigated using optimism in the parallelization of CSP [9]. Fujimoto described some work investigating the use of optimism as a parallelization technique, using the Virtual Time Machine [31]. The current work of Winter, Kalantery and Wilson described in [76] and [75] is investigating the use of mainly conservative synchronization techniques to provide a new parallel architecture model. HOPE (Hopefully Optimistic Program Execution) [24], [25] is a language for expressing optimistic constructs.

# Chapter 3

# Parallel Discrete Event Simulation

This chapter gives some background in the areas of Discrete Event Simulation, and particularly Parallel Discrete Event Simulation (PDES). The PDES paradigm of virtual–time [46] is used as the main basis of the optimistic parallelization scheme which is the subject of this thesis. The virtual–time paradigm forms the basis of many parallel optimistic simulation systems. First an introduction to the area of discrete event simulation is presented to prepare the reader on the background of the subject before moving on to optimistic and conservative parallel discrete event simulation techniques.

## 3.1 Discrete Event Simulation

In discrete event simulation the "physical" system is modeled by a set of events. Each simulation event corresponds to a state transition of an object in the physical system. Simulation events have a time–stamp which corresponds to the time at which the event would occur in the system being modeled. On a sequential machine the simulation proceeds by taking the event with the lowest time–stamp and simulating its effect. The effect of an event may be to alter the state of an object being simulated, or to create further events which will in turn be simulated, or both. The simulation moves forwards in jumps of simulation time corresponding to the time–stamps of the events. This is in contrast to other simulation methods where time moves forward uniformly. The simulation is complete when there are no more events to simulate.

## 3.2 Parallel Simulation

In parallel discrete event simulation [32] the "physical" system is modeled by a set of processes which correspond to interacting "objects" in the "physical" system. The interactions between the "physical objects" are modeled by having processes send each other time–stamped messages corresponding to the information flow in the physical system. The time–stamped messages correspond to event creation in the sequential simulator. The parallelism is exploited by placing these processes on different processing nodes which are capable of communicating. Each process has a logical clock which denotes the simulation time of that process. A process's logical clock will be increased to the time–stamp of the message as an event is executed. In this way the logical clock or Local Virtual Time (LVT) will proceed in jumps corresponding to the time–stamps for events. The progress of the simulation is measured by the Global Virtual Time (GVT), which is the minimum of the logical clocks (and time–stamps of messages in transit).

It must be ensured that events for each process are simulated in non–decreasing time–stamp order. (In DES time–stamps are processed in *non–decreasing* order rather than increasing order because the time–stamp generation schemes give independent events the same time–stamp.) To see why it is necessary to execute in non–decreasing time–stamp order, consider a process receiving a message which has an earlier time–stamp than the logical clock time. This means that the message was sent by an event which should have been executed earlier, and that events have been executed past this point in simulation time which could have been affected by that event. It is not possible to process the late arriving event as this would amount to the current event being affected by events in the future. This is known as a causality violation, from the cause and effect principle: the fact that events in the future can not affect events in the past. There are two approaches to ensuring causality is not violated in the simulation: *conservative* and *optimistic*.

## 3.3 Optimistic vs Conservative Simulation

Conservative approaches [19] 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 approaches [46] use a detection and recovery approach: causality errors are detected, and a roll back mechanism is invoked to recover. A roll back will be invoked when a causality violation is detected by an event being executed too late. Optimistic approaches are usually based on the virtual–time paradigm. A roll back will be required when a causality violation is detected due to an event message arriving too late (this is known as a *straggler*). This situation is depicted in Figure 7. The solid lines with arrows represent event messages being passed between simulation objects on different processors in the parallel simulation. The dashed line represents a straggler event which arrives at the process on processor 1 after the message from the simulation process on processor 2 has arrived. The dashed line shows an example of a message being delivered out of normal sequential sequence, which causes a causality violation due to the fact that it arrives at the destination processors with a time–stamp in that processors past.

The rollback must restore the state of the process in question to a point in time before

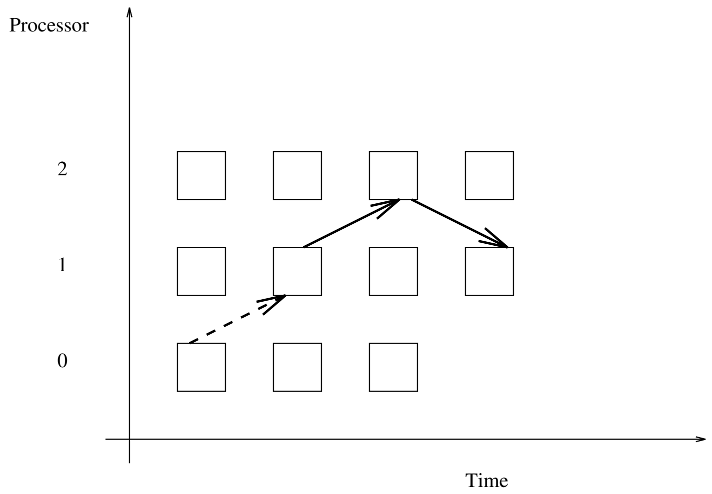


Figure 7: Straggler event arriving

the time–stamp of the straggler. After the rollback, execution resumes from that point in time.

The ability to rollback simulation objects is facilitated by having processes periodically saving their states. There are several state–saving schemes with differing efficiency characteristics which are used by optimistic parallel discrete event simulators. There is a balance which must be achieved in deciding the frequency of the state saves. Too often and state saving will become too much of an overhead, to infrequently and we will have to roll back too far. In the course of rolling back the state of the system anti–messages are sent for all messages which it is required to unsend. If the message has already been processed, the anti–message will cause the receiving process to roll back any effects which that message may have had. This may include sending more anti-messages.

The optimistic case is less restrictive and so potentially allows for more parallelism in the execution of the simulation. However, there are overheads incurred in maintaining the information necessary for roll-back, and the roll-back itself consumes execution time. In addition processor cycles will have been consumed computing work which has turned out to be unnecessary in the event of a causality violation. The optimistic approach is less restrictive because it allows events to be processed out of time–stamp order even if they could affect each other. Optimistic simulations work well where the number of cases where an actual causal relation exists between the events is comparatively infrequent.

## 3.4 State–saving

To rollback simulation objects we need to be able to undo operations on them. For some limited situations it would be possible to write an undo function which reverses the operation. This technique has been used successfully in some discrete event simulation systems, SPEEDES [68] is an example of this approach. However, it is not possible to generalize this for general purpose programming languages, as many typical operations are not reversible, and information is quite often lost. The user supplied undo function approach used in SPEEDES is referred to by some as an incremental state–saving method. For the purposes of the following discussion, where the term "incremental state–saving" is used, it should be taken to exclude the SPEEDES approach, but to mean the more general incremental state–saving schemes which are automated, and do not rely on undo functions. This type of state–saving scheme will be described below. The reason for this distinction is that the properties of the SPEEDES state–saving scheme have some differences from the more general incremental state–saving schemes, for the purposes of this discussion.

The general technique for state–savers is to save a *history* of the previous states of the objects. Implemented simplistically, state–saving can be expensive. Care must be taken to ensure that a workable balance is achieved and that the cost of the state–saving work is not too high compared to the work done by the function called between state saving. There are two basic approaches to state–saving: periodic and incremental.

### 3.4.1 Periodic State–saving

Periodic state–saving is when the state of the object is saved at regular time–intervals. A complete snap–shot of the object's state is taken and saved. As the state is only saved periodically there will points in simulation time for which no state is known. If a rollback is required to a point in simulation time for which no state–save information is stored, then the object's state is first rolled back to the time of the nearest prior state–save to the state required. Then the simulation is made to coast–forward to the required point. The "coast–forward" term means work is re–computed forwards from the given prior state–save to the required point in simulation time. The re–computed work up to the point of the simulation time is to achieve the required object's state, the computation has no other useful purpose. After the coast–forward, execution continues normally, when the straggler message is processed, execution may continue along a different path.

Periodic state saving is a compromise: there is a trading off of state saving overhead against re–computed work.

If state–saving is too frequent, the state–saving overhead can become excessively large, there will be both a time and a space overhead associated with excessively frequent periodic state–saving. Conversely, if state is saved too infrequently, large amounts of work will have to be recomputed to reach the state for the simulation time required. A systematic study which attempted to discover the optimal balance, and to model the trade–offs involved with balance of state–save frequency is [50].

One problem with periodic–state saving is that in some cases the amount of state saved can be far more than the amount of state which has changed. For instance, consider a simulation object which is comprised of a large array of floating point values. If a simulation event is the update of just one of these values, the periodic state saving technique means that the state of the entire array will be saved. Depending on the size of the array, and the number of actual state changes, and the granularity of the computation associated with the event, this can prove wasteful.

Another way of expressing this shortcoming is to state that periodic state–saving is not suited to simulations with large objects, and events which only affect small parts of those objects.

### 3.4.2 Incremental State–saving

Incremental state–saving is another state–saving technique, with different overhead characteristics. Incremental state–saving is something of an opposite to periodic state–saving. In periodic state–saving, the whole state is saved at regular intervals; with incremental state–saving, only the parts of the state which have actually changed are saved. The changes are saved on *every* change [1], so there is a complete state–saved history. This eliminates the re-doing of unnecessary work due to missing state information which occurs with periodic state–saving.

Another advantage of incremental state–saving is that it is more suited than periodic state–saving schemes to the use of an optimization known as lazy re–evaluation (see section 3.5.2).

Lazy re–evaluation is an optimization which saves re–computed work in cases where an object's state is unchanged after a straggler message has been processed. It stems from the observation that on some occasions, after rollback, execution continues along the same path as prior to the rollback. Lazy re–evaluation takes advantage of these cases by checking the saved state against the new values which would have been saved. If the program does not diverge from the saved state after the straggler message has been executed, then the state can be rolled–forward to the point in simulation time where the straggler arrived. The

[1]This is not necessarily true for incremental state–saving schemes which use undo operations, such as SPEEDES

unnecessary re–computation will have been avoided.

Neither incremental nor periodic state–saving is more efficient in all cases. Where there is a large state and relatively few parts of this state are modified, incremental state saving is likely to have lower overhead than periodic state–saving. In the converse situation, where parts of a given objects state are changed often, it is likely that periodic state saving is likely to have a lower overhead.

The problem with using incremental state–saving for a general purpose programming language is that with the naive implementation ultimately every variable or object in the entire program would need to be represented as a state–saving object.

Doing incremental state–saving on low level objects such as integers, floating point numbers, etc., adds high levels of 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, especially when the operands will typically be stored in registers, and the state save information will have to be stored in memory. So such a proposition adds a very high overhead, and although a system could still be envisaged which achieved speed up with such overheads for problems which were particularly amenable to optimistic execution, whilst being unsuited to conventional parallelizing systems, it would of course be desirable to reduce the overhead. The roll back overhead is important, in any case, because it is a deciding factor in what classes of problems speedup can be achieved.

## 3.5 Optimistic Simulation Optimizations

There are a set of optimizations which can be applied to optimistic simulation techniques, these are:

- lazy cancellation

- lazy re–evaluation
- some techniques derived from Fujimoto's space–time memory [31]

### 3.5.1 Lazy Cancellation

Lazy cancellation is based on the observation that it is possible for the rollback and forward re–execution incurred as a result of a causality violation to send the same messages as in the original execution. The optimization is thus to delay sending cancellation messages until it is determined that different messages are sent by the re–execution. If the same messages are in fact sent, a significant saving is made in that the negative effects of the rollback have been localized. In the absence of lazy cancellation in situations such as this, the messages would be canceled, and probably cause further rollbacks, and subsequent cancellation messages, which would only be re–computed when it transpired that the very same messages were re–issued in their place.

However, there is a negative side to lazy cancellation, and this is that delaying the sending of cancellation messages allows other parts of the simulation to progress further. If it turns out that the cancellation messages must be sent, more work is wasted and must be rolled back, and possibly more subsequent cancellation messages may need to be issued.

### 3.5.2 Lazy Re–evaluation

Lazy re–evaluation is based on the observation that on some occasions, after a straggler message has arrived indicating a causality violation the subsequent rollback and execution of that straggler event in its correct order leads to the same state as without executing the straggler message. That is to say, if after executing the straggler message, the state of the simulation object is unchanged, further re–computation is guaranteed to proceed along the same path as that prior to the arrival of the straggler message. The lazy re–evaluation optimization is the action of recognizing this situation, and checking the state of the object after the straggler event has been computed, if it is unchanged, a *roll–forward* can be effected. The action of rolling forward means that the simulation state is restored to that before the straggler event arrived (with the exception that the straggler message will now be recorded as having occurred in the correct sequence in the message queues). If the state diverges from the saved–state, hope of being able to roll–forward is abandoned, and normal execution is resumed.

### 3.5.3 Space–time Memories

Fujimoto proposed as part of his Virtual Time Machine [31], an abstract memory known as a space–time memory. This memory is a two dimensional device allowing memory access, and writes, by space (conventional memory location) and time. That is to say, the space–time memory retains previous values together with the time they were created, and these values can be read by supplying a time–stamp within the range of a particular variable's existence and that variable's location with the access request. Writing a value to a location with a time–stamp earlier than that of the last access causes the equivalent of a causality violation.

The interesting property from the point of view of optimizations to optimistic simulation systems, is the concept of being able to access a memory location's past values with no other penalties. This is in effect a special case of lazy re–evaluation, as lazy re–evaluation copes with read requests automatically by going through the procedure for lazy re–evaluation of the straggler event. If special provisions are made to mark read events as such, their implementation can be made more efficient by avoiding the rollback, and test for unmodified state which normally accompanies lazy re–evaluation of a straggler event. Instead the value can be read from the rollback history information for the variable in question. The fact that the read event occurred must be recorded in the usual event queues lest a future straggler write event invalidate the value read. If this happens, the normal cancellation mechanism will rollback the read event.

## 3.6 GVT

Global Virtual Time is a simulation time value which provides a measure of the progress of a parallel discrete event simulation. GVT is the minimum of the logical clocks of all of the simulation objects in the system, and of the time–stamps of messages in transit. The GVT value has important applications in *fossil collection*, and for the committing of non–reversible operations such as I/O. "Fossil collection" refers to the fact that it is possible to reclaim the memory consumed storing rollback information when it is known that the information will no longer be required. However because of the distributed, and transient location of the information required to calculate GVT correctly, it is difficult, and expensive to calculate GVT. The implementation of GVT algorithms is discussed in Bellenot's paper [13]. A comparison of the performance and communication requirements of GVT algorithms is presented in [26]. The GVT algorithm presented in [70] is suited to use in the optimistic execution system presented in this thesis because it relies on asynchronous communication only.

# Chapter 4

# Design

In this chapter the design issues will be explained, and justification for the design decisions taken presented. The aim of this chapter, together with the introductory chapters on parallel object programming and parallel discrete event simulation, is to present the background and an overview of the system to allow the rest of the thesis to be followed easily. This chapter presents the main points of the design in enough detail that the reader will be able to see where each component fits into the overall design. The components of the system are very interrelated, so we present the overall design first, with the aim of breaking the circular dependency between the actual system components, as described in subsequent chapters.

## 4.1 Basic Approach

The aim of the research is to investigate methods of applying the virtual time paradigm used in parallel discrete event simulation to an object oriented general purpose programming language. The basic approach taken is to transform selected program objects into new "server objects" which fulfill a similar function to that of simulation objects in an optimistic parallel discrete event simulation.

In the design which will be explained below, there are two main run–time components: a special purpose optimistic simulator, and a hybrid incremental state saving system. These run–time components are coupled with a set of transformations which convert the input program into a new program which can be executed in parallel, optimistically. The design criteria for the system components will be explained, and the motivation for the transformations required will be presented.

The optimistic execution system is based on the virtual time paradigm from parallel discrete event simulation. The use of an optimistic execution system is what allows parallelism to be exploited. To map sequential object oriented code on to the optimistic execution system model, a transformation system is used to transform method invocations into asynchronous remote procedure calls. This scheme is a very general one and allows many different sources of parallelism to be exploited. The other system component, an incremental state saving system, is used to provide the rollback capabilities required for the optimistic execution system.

### 4.1.1 Why OO Programs?

There are a number of motivating factors which lead to the choice of object oriented general purpose programming languages as suited to this approach.

Three important factors are that:

- Object oriented programs tend to be written more modularly than is typical for procedural languages.
- Object oriented programs tend to have designed interfaces for external use of modules' data structures.
- There is often a multi–level object hierarchy within an object oriented program.

The first factor, that OO programs are more modularly designed than programs written in procedural languages, is a stated design goal of most OO languages. Part of the language design is that data encapsulation, and encapsulation of the associated operations on the defined object are useful as a mechanism for simplifying the complexity of the code which the programmer needs to understand at one time. The second factor influencing the choice of OO languages, tightly controlled inter–module data–flows are part of this design philosophy, the control provides safety; in that it gives protection from accidental modification of that data. It also encourages cleaner designs, with interfaces to each object type being well defined, and designed at an early stage. This helps avoid spaghetti like code, and results in simpler data dependencies in the program.

Object oriented programs do not *have* to be more modular: it is still possible to write programs in a procedural style, and to ignore data encapsulation principles. Some languages provide both abstractions, procedural, and object–oriented. An example is C++, which inherits the procedural aspects of C. It is entirely reasonable to use C++ as a more strongly typed C without making use of the object oriented language features, and indeed C++ is often used in this way. Similarly the converse is true: it is possible to write very modular code, with tightly controlled interfaces in a procedural language.

From the available object oriented languages C++ was chosen because it is widely used, and there are therefore a large number of existing applications.

It is anticipated that when object oriented languages are used in the way they are intended – making use of language data encapsulation facilities – that the programs will have better defined interfaces between program modules. That is to say, use of global data is less common, and data flows between modules will tend to have programmed interfaces, rather than being just shared global variables.

In the design, as will become clear below, the modularity is useful as it provides sections of code which can be distributed across parallel processors. Less modular code is more difficult to split, as there are more data dependencies between sections of code. The criteria in deciding where to split code is that the resultant code sections should communicate as little as possible; this is important as the sections of code are being extracted for placement on different processors, and inter–module communications will be implemented by messages passed between processors.

The task of *automatically* identifying relatively independent sections of code would be harder in a procedural language, as procedural programs provide no mechanism for the programmer to specify object oriented data and task encapsulation. With object oriented languages the programmer is tasked with designing the program in an object oriented fashion, and expressing the data encapsulation rules in the program. It is part of OO design philosophy to construct programs in a modular fashion, with strict interfaces between modules. The fact that such encapsulation is supported in the language means that the program classes and objects created by the programmer provide a good starting point for heuristic categorization of sections of program code with suitably restricted interfaces.

The third advantage of object oriented programs over procedural programs is that when the inheritance mechanism is used, modular program components are available at varying degrees of granularity. It is typical with object oriented programs for there to be a class hierarchy, with problem domain abstractions coded as program objects, and higher level abstractions to be implemented in terms of lower level abstractions, as specializations of them, or by composition of several lower level objects.

Normally with specialization – inheritance – the specialized object is more complex, and hence of larger granularity than the original lower level class. Similarly, the creation of objects by composition by definition results in large grain objects. The general picture then is that there are objects of varying degrees of granularity, and that it is the nature of object oriented programs which make use of inheritance and composition that creates this.

As selected program level objects will be placed over the nodes of a distributed memory parallel computer, in a distributed message passing environment, it is important to be able to choose the granularity of the objects such that granularity is at a suitable level. If the level of granularity is too low, communication overhead may be too high as compared to the work done. There are only so many large granularity objects in a given program, and there will be cases, where if too large granularity objects are chosen, that there will be too few objects to allow a reasonable number of objects to be placed on each processor.

The assumption is that there will be fewer large granularity objects than lower granularity objects, this will in general be true where use is made of inheritance and composition. It is also assumed that it will be desirable to have multiple processes on each processor to provide some *parallel slackness* [74], in order to hide latency of communication.

## 4.2 Optimistic Execution

The parallelization of the object oriented program is achieved by transforming the original program so that it can be optimistically executed in parallel.

As described in Chapter 3, there are two broad types of PDES (Parallel Discrete Event Simulation) systems: conservative, and optimistic. Conservative schemes are based on avoiding causality violation, by various predictions. Optimistic simulations are more aggressive, and are able, in general, to exploit more parallelism. The predictions of possible events required for conservative schemes are difficult to generalize in a form suitable for automatically parallelizing sequential code. The motivation for choosing an optimistic scheme is two–fold: optimistic schemes are both able to produce greater performance in general, and their requirements are more readily satisfied in an automated manner suited to general purpose use.

This section now relates the system architecture of a parallel discrete event simulation to the message passing view of object oriented systems. The basic model of a discrete event simulation, as described in more detail in the previous chapter is of a set of events, and an algorithm for processing those events.

Events are time–stamped, and the events for each object are processed in non–decreasing time–stamp order. When an event is processed it can have some effect on the simulation object's state, and it can schedule further events.

In a parallel discrete event simulation, the simulation objects are distributed on the processors of a parallel machine. Events are scheduled on remote objects by sending a message. The messages are asynchronous; that is the sending process does not wait for a reply from the event scheduling message,

Object oriented programming has a natural parallel implementation: messages are implemented by remote procedure calls. The RPC invocation consists of an invocation message, and arguments sent with that message. The return value, and any modified parameters are then sent back as a result. The messages in an acknowledgment passing implementation of a sequential object oriented program are synchronous; the sending process, when there are return values, waits for those values. In the special case that there are no return values, the invoking message is typically synchronous also, as a message would be sent back to indicate that the remote procedure, or remote method invocation had been completed. However, if the aim of the system is to introduce parallelism there is no need to wait for the message indicating completion from the remote procedure call.

The aim is to transform the OO view so that it can be implemented with a PDES like optimistic execution. This section describes a transformation of the message passing implementation of the OO programming view which can operate with asynchronous messages only. Where the method has no return value, an asynchronous implementation can already be implemented using asynchronous messages, the problem that remains is of how to deal with return values without blocking. The basis of the system is the use of the fact that once the necessary transformations have been performed, it is known where to send the return value message before the remote procedure call is invoked.

The resulting optimistic execution system's guarantee of progress (liveness) is based on the guarantee of progress which has been shown for the virtual time paradigm [46]. The program is guaranteed to proceed provided that:

- the time–stamped set of events, if executed sequentially in increasing time–stamp order, would produce a semantically equivalent execution of the program;
- the run–time system is a correct implementation of a virtual–time system.

The *domino effect* [11] which occurs in fail–safe systems that use check–pointing and rollback for the purpose of recovery does not occur in the virtual–time algorithm because of the guarantee of progress shown in [46].

A broad model of the optimistic parallel OO view to which the sequential program will be transformed is that there is a parallel machine, and that there are a number of RPC server objects (Remote Procedure Call server objects) placed on each node. For purposes of discussion the RPC servers can be considered as processes accepting and processing the messages implementing the RPC call invocations. In the transformation, which will be presented below, a selected program object will be transformed to become a server object; invocations of the methods of this object will be transformed to RPC calls to the server object.

## 4.3 Server Objects

A server object is the run–time system object which encapsulates the functionality of state–saving, rollback, 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 main thread of control which accepts an RPC call message, decodes the message to decide which method was invoked, decodes the method's arguments, and executes that method. The program object is transformed to ensure that the server object has a disjoint set of sub–methods for each of the methods it had prior to transformation.

```
double class1::method1()
{
  x = obj1.method2();
  y = obj2.method3();
  return func1(x,y);
}
```

Figure 8: Example requiring sub–methods, before transformation

In Figure 8 suppose that `class1::method1` is implemented as a server object, and that objects `obj1`, and `obj2` are also server objects. In order to avoid blocking `class1`'s main loop should not wait for the return values from `method2` and `method3`. This is achieved via transformations by splitting the code of the methods up into sections of code within which there are no blocking RPCs. The two places where the above example blocks are waiting for the return values from the methods `method2` and `method3`. The transformed version of the code is shown in Figure 9, and shows that sub–methods `method1a`, `method1b`, `method1c` are used to implement a replacement for `method1` which uses asynchronous RPCs only.

The sub–methods can be invoked asynchronously. The code associated with handling

```
void class1::method2a(double &);
void class1::method3a(double &);

void class1::method1a()
{
  obj1.method2a(x);  // send result to object x
}

void class1::method1b()
{
  obj2.method3a(y);  // send result to object y
}

double class1::method1c()
{
  return func1(x,y);  // request values from objects x and y
}
```

Figure 9: Example requiring sub–methods, after transformation

a return–value from some other server object will be a sub–method, and this sub–method will be invoked upon return of the required value. Further, it is the task of the invoked RPC, after it has completed, to in turn invoke the relevant sub–method on the calling server object, with the returned value. This is necessary, because it may not be desirable to wait for the invoked code, if extra parallelism is required. In any case, other methods may be invoked on the invoking server object while the RPC computing the return value message is being executed.

The decision to have server objects with a single thread of control was expressed above, although a server–object with the capability of having multiple active threads might at first glance seem to be a way to improve parallelism. However, having multiple threads complicates the design when the full optimistic execution scheme is considered. The extra server object functionality of having a logical clock, and having a state–saving mechanism, mean that it would be difficult to see how a multi–threaded server could be reconciled with a single logical clock. The multi–threaded server object would in effect have multiple sub–parts with different logical clock values. The desired parallelism which it might seem could conceptually be exploited by having a multi–threaded server object can, in any case, be obtained by having a more server objects, that is, if the component objects of the server object were themselves server objects.

There are a number of ways to introduce parallelism into the optimistic execution: the two methods relevant to this discussion are as a result of an asynchronous dispatch of an RPC for a method on a remote object which has no return value, and as a result of an asynchronous dispatch of an object with a return value, where that return value is not used immediately. Indeed it would be possible, though presumably undesirable to optimistically execute an RPC for a remote method where the return value was needed immediately.

The server objects also encapsulate the functionality of causality violation detection, and rollback recovery from such violations. For this the associated functionality of state–saving, a logical clock, and time–stamps are required.

As in parallel discrete event simulations, the asynchronous RPC invocations, which correspond to the event messages in a discrete event simulation, are time–stamped. The time–stamps are chosen such that a sequential discrete event simulator executing the associated RPC calls in increasing time–stamp order, would produce an execution semantically equivalent to that of the original program. It is normal in discrete event simulators to stipulate that events be processed in non–decreasing time–stamp order, rather than in increasing time–stamp order as done here. This difference is due to the use in this research of a time–stamp allocation scheme which ensures unique time–stamps.

Each server object has a logical clock, this clock 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 arrival and processing of RPC messages. Broadly speaking, a causality violation is signalled by the arrival of a straggler message, a message with a time–stamp lower than the server objects logical clock. A causality violation indicates that the RPC should have been processed earlier in logical time than RPCs already processed by that server object. This must be accounted for, if the RPC relies on prior state changes of the object, and will read and modify the object, the server object must be rolled back, and the RPCs re–executed in the correct order.

A number of optimizations from parallel discrete event simulation can be applied to the server object; lazy cancellation, lazy re–evaluation, and avoidance of rollback for methods which do not modify the object's state. These optimizations are described in Chapter 6, on the high level aspects of the run–time system.

If a causality violation is detected, and it can not be avoided by one of the optimizations, the state of the object must be rolled back. For rollback, prior state information, a history of values must be retained. The state saving method used is an object oriented hybrid incremental state–saver.

## 4.4 State–saving

As described in Chapter 3, on PDES and optimistic execution, the simulation objects in an optimistic PDES simulation require state–saving to enable rollback. This is an integral part of the design of an optimistic execution system, to execute optimistically it must be possible to rollback erroneous states, and this requires some kind of state–saving mechanism.

State–saving can be a major overhead, it is important to minimize state–saving overhead, as compared to useful work done.

In this system an incremental state–saver has been adopted. An incremental state–saving mechanism which is able to vary its performance characteristics between purely incremental, and more periodic is presented in the high level run–time Chapter 6. This allows the state–saver to be tuned to the memory usage patterns of the object in question.

## 4.5 Time–stamps

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 the 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. For example consider the case where it is not known how many iterations of a particular loop will occur when the code is run. This can occur if the expression used to detect loop termination involves data entered at run time, or for the expression to be dependent on values which can not be predicted at compile time. There will be cases where termination, and the number of iterations can be predicted, but in general this is not possible.

The time–stamp allocation scheme is therefore more complex than those used in parallel discrete event simulation, and involve a variable length time–stamp designed to allow an arbitrary number of time–stamps to be allocated between any pair of previously allocated time–stamps. The time–stamp allocation scheme necessary to make efficient use of these time–stamps, is discussed in detail in Chapter 8.

# Chapter 5

# The Transformation System

This chapter describes the transformations required for the parallelizing compiler. The transformations are used to transform the sequential object oriented code into a version which can be executed in parallel.

Figure 10 shows the system components which are part of the transformation system within the dashed box. These are the parts of the system described in this chapter.

There are three levels to the transformation system:

- heuristic level
- annotation level
- underlying transformation system

The annotations provide an interface to allow the transformations to be directed by a heuristic analysis module, and/or to be directed manually by the programmer. Annotations can also serve as a mechanism to give hints to the heuristic system.

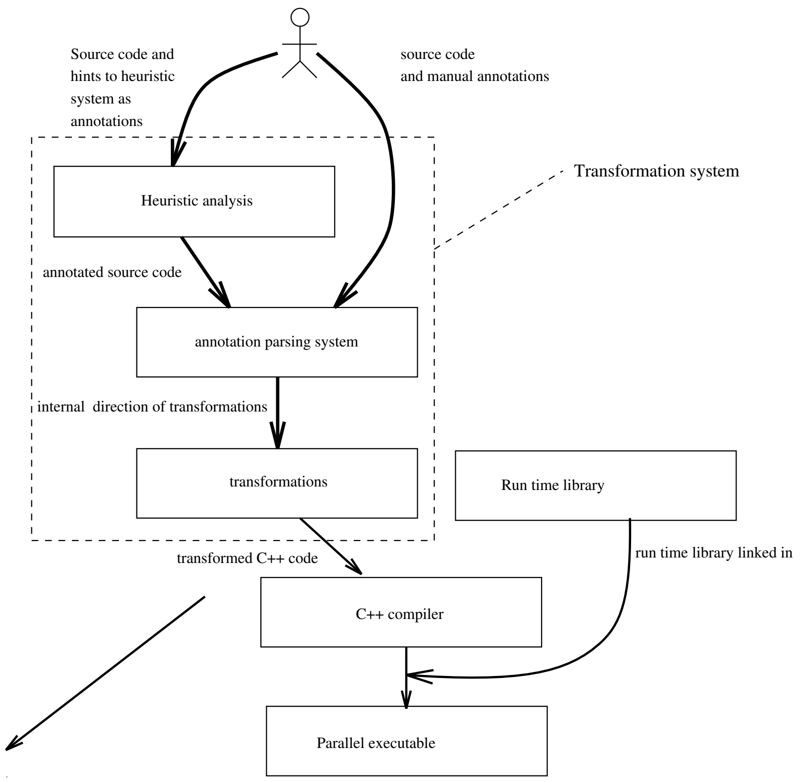


Figure 10: Transformation system architecture

An example of an annotation (this annotation requests that a state saving version of the annotated object is produced):

```
#pragma p++-state-saving("A","state_A")
class A
{
public:
  meth1();
};
```

This chapter explains how pragmas are used to direct the transformations required to produce an asynchronous optimistic execution of the program.

Heuristics can be used to set parameters of the transformation process, such as the selection of server objects, their distribution over the processors of the parallel system, and their state saving behaviour. The heuristic level produces annotations to drive the transformation system. The module responsible for heuristics is shown as the box marked “heuristic analysis” in Figure 10. As well as receiving the source code from the programmer, it is likely that it would be useful to allow the programmer to give hints to the heuristic system, as distinct from the annotations given to the annotation system.

The annotation level is the interface to the underlying transformation system. The annotations are used to direct the actual transformations via annotation commands inserted in the program text. The transformation system module responsible for managing annotations is shown as the box marked “annotation parsing system” in Figure 10. The annotation system parses the annotations which either may be from source code provided directly by the programmer, or if a heuristic analysis module is used, some, or all of the annotations in the source code may have been inserted by this module.

The underlying transformation system, the box marked “transformations” in Figure 10 is the code which performs the actual parsing, and subsequent transformations.

The other system components shown are a C++ compiler and the run–time library. The transformed code is compiled with a standard C++ compiler, and the resulting object file linked with the run–time library to provide the run–time support functions required by the parallelization system.

The choice of C++ as the OO (object oriented) source language is explained in the design overview. With C++ as the choice of language, there are some additional transformations required to make explicit some implicit C++ language features.

Within the context of C++, "implicit" language features are those where the compiler will call either a user provided, or a built–in function or conversion without there being any explicit instructions to do so in the source code.

Uses of implicitly called functions must be considered in the transformation, and hence the disambiguation tasks associated with implicit functions which are normally carried out by the compiler must be performed by the transformation system. It is necessary to transform implicit function calls, as it is necessary to be able to further transform these functions and functions calls in the main phases of the transformation process. For this reason, the mechanism used to do this is to transform the implicit C++ code into non–explicit equivalent code. The transformed, fully explicit code is then used as the input for the second and subsequent phases of the transformation process.

Appendix A gives the details of the implicit C++ language features, and shows how the transformations allow implicit calls to be expressed explicitly. These issues have been separated from this chapter because the explanation of the C++ semantics of their normal use, and the transformations to make their use explicit is lengthy, and involves the more complex C++ language features.

This chapter concentrates on the more general transformations which could apply equally to other object oriented languages. Discussions of transformations in this chapter assume that the "explicify" transformation phase has already been applied. All of the examples in this chapter are expressed in terms of C++, unless a particular example is marked as an exception, these will be for purposes of discussion where different techniques would be required in other languages.

## 5.1 Sage

The transformations where implemented using the Sage transformation tool [15]. The Sage transformation system provides a parser for C++ (and FORTRAN) and builds an internal representation of the program structure, which retains comments and pragmas. A set of C++ classes are provided which allow Sage's parse tree to be traversed, and modified. New nodes in the parse tree can be built, and existing or new code hung from the nodes. Sage also provides facilities to query the type system, and traverse the type system for composite objects, and to traverse the inheritance tree. Also symbol table lookups by name can be used to find the declaration, use of and type of an object.

Sage provides a frame work for adding user annotations (the pragmas used to drive this transformation system, and it is planned to use this feature for the transformations below). The annotations can be added to objects, and Sage provides slots to add user annotations to objects.

When the transformations are complete Sage can be told to `undump` the parse tree, and modified source code will be written to a file, retaining the original program comments.

## 5.2 Annotations

Annotations in the context of compilers are comments or hints from the programmer to the compiler or transformation system. One important requirement is that the annotations system must allow code to be written with annotations for a compiler offering particular optimizations directed by annotations, and yet still compile on a standard compiler not understanding the hints. This rules out the use of compiler specific language extensions in an uncontrolled fashion. The way this is normally handled is to put the annotations within standard comment fields prefixed by some magic string in the hope that no one accidentally triggers an annotation via a normal comment.

An example of this type of annotation would be say:

```
(* $O+ *)
```

(Borland Pascal, a directive to turn on optimizations). The presumption is that in normal usage it would be unlikely for a comment intended for humans to read to start with `$O`.

Within C++, annotations are handled by having a language construct which is used as an escape meaning that the rest of the line is an annotation. The string used is: "`#pragma`". It is defined as part of the language standard that compilers should ignore pragmas which they do not understand.

`#pragma` directives are the standard method of giving annotations within the C++ framework to allow compiler hints, in a way which should be transparently ignored by other compilers.

An example of a pragma is:

```
#pragma implementation
```

This can appear at any line in a program, but it must be the first non–whitespace character on the line it appears on. (This example is for the GNU g++ compiler).

For the parallelizing compiler described in this thesis, all pragmas are prefixed by the string "`p++`" to denote that this set of pragmas is for this compiler as opposed to others.

An example of a pragma would be:

```
#pragma p++-time-stamp-block
```

The notation used for pragmas in this system has arguments for some pragmas. An example of a pragma with an argument is:

```
#pragma p++-purify-type( x )
```

The argument in this case is the name of a program type.

This chapter is divided into sections which describe related groups of transformations. These are the important transformation annotations:

- Replace built–in types with pure replacements
  - `p++-purify-type( type )`
  - `p++-use-purified-type( var )`
- Replace compiler temporaries
- Add time–stamps
  - `p++-add-time-stamp-param`
  - `p++-time-stamp-block`
- Server objects
  - `p++-server( class, stub-class, server-class )`
  - `p++-state-saving( class, state-saving-class )`
  - `p++-use-state-saving( class )`
- Asynchronous transformations
- Rollback

## 5.3 Purify Transformations

The annotations described in this section are:

- `p++-purify-type( type )`

- `p++-use-purified-type( var )`

In order to explain the requirement for the purify transformation, the difference between pure object oriented and non–pure OO languages is explained.

In a pure object oriented language, all program objects are first class objects. There are no types outside the object oriented type system, all types can be inherited from, and even the existing primitive objects, such as integers, and the operations on them are considered as first class objects.

Some object oriented languages deviate from this pure view for efficiency reasons; C++ is an example of such a language. In C++ the built–in types are treated as a special case, and are outside the OO type system for purposes of inheritance, and syntax of operator use.

The purpose of 'purifying' an impure program is so that the resulting use of operations on such objects is explicitly expressed in terms of function calls so that such objects can be manipulated to provide rollback, and state–saving functionality. It is also necessary to have all operations expressed explicitly if the operations will be further transformed to be remote procedure calls.

Due to the use of transformations, it is required that all operations which involve server objects are transformed to an equivalent expression which expresses the operation explicitly with a function call. An example would be the use of a built–in type, such as an integer:

```
{
  int i, j;
  //...
  i += j;
}
```

If it is required that the variables `i` and `j` be server objects, it would be necessary to arrange for the operations on the variables to be expressed as function calls. An example of a situation where it would be necessary to make an `int` a server object would be if the `int` where used as the index variable to a loop where the loop termination was dependent on the outcome of method invocations of objects which had been designated to be server objects.

The requirement of having operations on built–in types expressed explicitly can be achieved by replacing the use of the built–in types with replacements. Because the replacement types are not built–in types, they will by definition be expressed in terms of member functions, and function calls. Replacement types can be written which have all of the required operations available for the built–in type, but in the form of methods, and supporting functions. It would be possible to transform the whole program to use these objects, and the whole system could then be viewed for transformation purposes as a pure object–oriented program.

However, the purifying transformation need not necessarily be applied to the whole program. It is only necessary to apply it to areas of the program where the pure OO view must hold, to enable the transformations described in the following sections of this chapter to be applied. Transforming the whole program is thus avoided as there is a significant overhead involved in using the replacement types over the built–ins. The overhead is due to a number of factors such as extra indirection for each use of the value stored in the replacement class, and the loss of opportunities to leave data values in registers. Also the C++ compiler used after the transformation phase will no longer be able to manipulate expressions involving replacement types according to semantics preserving optimizations, as there are no guarantees that user supplied overloaded operators follow arithmetic rules of associativity, and commutativity. Optimizations such as common sub–expression elimination may also be lost.

An example of the kind of transformation which is carried out to achieve a replacement for integers is:

```
class Int
{
protected:
  int value;
public:
  Int();
  Int( int );
  Int( Int& );

  Int& operator=(Int&);

  operator+=( Int );
  operator-=( Int );
  // ...

  operator int ();
};

Int operator+( Int, Int );
Int operator-( Int, Int );
//...
```

The use of overloaded operators demonstrates one of the more powerful C++ language features. A class defined in conjunction with overloaded operators can use the same mathematical notation as the built–in types. Operator overloading is described in more detail in Appendix A.

The constructors provide respectively: default construction `Int()`, so that code such as the following is legal:

```
Int i;
```

also construction with an integer literal:

```
Int i = 5;
```

or, semantically identically (a syntactical alternative), the canonical constructor syntax:

```
Int i( 5 );
```

Note that the above code involves an initialization with a copy constructor, while the following code involves a default constructor, followed by an assignment operator call:

```
Int i;
i = 5;
```

The calls are different, one involves a constructor being used, the other an assignment operator, it is just that with normal arithmetic semantics these are equivalent.

The constructor taking a built–in `int` also serves as a constructor to be used as a conversion:

```
Int i;
i = 5;
```

ie, the above code is equivalent to the following code, once the implicit functions calls, and explicit operator calls have been added:

```
Int i;
i.operator=( Int( 5 ) );
```

Also a copy constructor (`Int( Int& )`) is provided to allow initializing an `Int` with another `Int`, like this:

```
Int i = 5;
Int j = i;
```

The operator names will all be converted to normal function names, as the later transformations will require additional arguments.

Now that the purity of an object oriented language, and the transformation of an impure object to a pure equivalent object have been explained, those transformations which rely on this property can be described.

## 5.4 Replace Compiler Temporaries

There are expressions involving user defined types which force the compiler to generate temporary variables of the appropriate type. This is an example of an action being performed implicitly by the compiler.

Consider what happens with the user function multiply, `mul()`, on matrices. (It would be more elegant to use an overloaded `operator*`, but it is simpler, and more general for the purposes of this discussion to talk about the function as `mul()`).

```
Matrix mul(Matrix& A, Matrix& B)
{
  Matrix res;

  // do calculation res = A * B

  return res;
}

{
  Matrix A,B,C,D;

  D = mul(A,mul(B,C));
}
```

There will be at least one temporary generated by the compiler, as the `mul()` function expects its arguments to be references. (There would actually be two temporaries, as the assignment results in another function being called, the assignment operator, and this requires a reference argument too).

The basic approach used to transform this is to have a replacement function which takes an extra argument, the area of memory into which the result is stored. With this provision the transformation is reasonably straightforward, with the only extra requirement of the transformation system being to allocate and delete the temporaries explicitly rather than relying on the compiler to handle them.

The basic form of the transformed code, stylistically is:

```
{
  Matrix A,B,C,D;

  // create space for matrix temporaries ctmp1, and ctmp2

  D = mul1(ctmp2,A,mul1(ctmp1,B,C));

  // destruct and reclaim space for ctmp1, and ctmp2
}
```

The replacement for the function `mul()`, `mul1()` takes an extra first argument which is the address where the result is stored.

The details of this transformation are described in Appendix A, as it is specific to C++.

## 5.5 Parallelizing Transformations

There are a number of stages to the transformation, these are:

- parameterizing by time–stamp
- converting program objects to server objects
- translating to allow an asynchronous message passing implementation

### 5.5.1 Adding Time–stamps

In this section the transformations required to add time–stamps to a program are presented. The operations which are time–stamped are functions associated with server objects.

Broadly, what is done is simply to add an extra parameter. This transformation is simple once the transformations described in the previous section have been completed. That is, once the transformations which ensure that if the relevant objects are built–in, they are replaced by the use of an equivalent user class, and once any necessary transformations to make explicit the use of compiler temporaries have been performed. The explicify transformations described in Appendix A must also be performed prior to the time–stamp transformation in the case of C++.

Time–stamps are represented as variable precision binary fractions. All time–stamps have values between 0 and 1. The encoding is the numerator of the fraction. The fractional value of the encoded fraction is obtained by evaluating the expression:

$$\frac{numerator}{2^{length}}$$

where $length$ is the length of the numerator. For example with the time–stamp $0101b$ (where the b suffix indicates a binary value), the fractional value is determined by examining the length and the numerator. The length of the binary fraction is 4 bits, and the value is $0101b$. The fractional value is therefore:

$$\frac{0101b}{10000b}$$

.

The only operations on time–stamps will be comparative, their decimal or binary values will not in typical operation be relevant. Full details of the time–stamp generation scheme are given in Chapter 8.

When a time–stamp is passed to a function, the time–stamp is used to initialize a time–stamp generator of appropriate type, either an unbounded or a bounded generator. This will ensure that the time–stamps for the function all lie below the next consecutive time–stamp. The range used for the time–stamps will be $(ts, ts + 1)$, where $(x, y)$ means exclusive of the end–point values $x$ and $y$, and $ts$ is the numerator of the time–stamp. That is the values $ts$

and $ts + 1$ will not be themselves be used, but any values between these two time–stamps can be.

Consider the example time–stamp above, $0101b$. If this is passed to a function, and used to seed a time–stamp generator with a requirement for five time–stamps, the resulting time–stamps will all have the form: $0101xxxb$ where $xxx$ holds the range of time–stamps used in the function. Specifically the time–stamp values would be:

```
0101000b
0101001b
0101010b
0101011b
0101100b
```

Note that $0101b < 0101000b$ due to the fact that length is significant in comparing time–stamps.

If the function is called from different places in the program, different times–stamp arguments would be passed depending on where the function is called from. As the time–stamps allocated in the body of the function are prefixed by the time–stamp passed in, the time–stamps within the function will vary depending on where the function was called from. So in the above example the time–stamps were of the form $0101xxxb$, but in another call from a more heavily nested, or recursive position in the program's run–time call tree, the range of time–stamps used by the function could be say: $10110101001010110xxx$.

Bounded time–stamp generators allocate a fixed, predetermined number of time–stamps in a range. The example above was one of using a bounded time–stamp generator. Bounded time–stamp generators are simple because all that happens is that for a requested generator with a range of $n$ time–stamps, $\lceil log_2(n) \rceil$ bits are reserved for the time–stamp range, and the time–stamps allocated sequentially from this range.

Unbounded time–stamp generators are required for cases where it is not known ahead of time how many time–stamps will be required for the range. It is easy to arrive at situations which require such properties of time–stamps for general purpose code, because it is fairly common to have loops with an exit condition which is determined at run–time, due to being data dependent, or more indirectly data dependent, and depending on the run–time behaviour of the program.

With these time–stamps the technique used is to reserve a fixed range of time–stamps, and use the first half of this range as time–stamps, and if these have all been generated to use the remaining half to produce longer time–stamps each with a further subdivision of time–stamps. This scheme is extensible, and results in a more efficient use of bits than the naive solution, and provides a trade off choice between use of bits, and efficiency of number of time–stamps allocated as compared to the number of bits consumed initially. These variable length time–stamps, and the optimizations designed to allow space–efficient use are described in the time–stamp Chapter 8.

Time–stamps generated by an unbounded time–stamp generator can be treated the same as time–stamps allocated by a bounded time–stamp generator when used to denote a range of time–stamps for a function call. The range used by generators (of either kind) when initialized with a time–stamp is $(ts, ts + 1)$. This range is safe for time–stamps generated by either type of generator, in that it guarantees there will be no overlapping ranges of time–stamps.

Time–stamps are added as an extra argument to operations which can be rolled–back. The time–stamp parameter is added as the first parameter, this is necessary because C++ has a language feature which allows optional arguments, which are only optional starting from the last parameter to the function. That is to say, the last parameter can be optional, or the last two, or all of the parameters. When called no gaps can be left, so for an example where the last two parameters are optional either they must both be given, or the first optional parameter only, or neither of the optional parameters. The other combination, where the first optional parameter is missed, and the last given, is illegal in C++.

It is this consideration which makes it necessary to give the time–stamp parameter first. Otherwise additional transformations would be required to fill in the optional parameters, so that they were all given. The choice is for simplicity only, putting the time–stamp parameter first avoids the extra transformation of making the optional parameters explicit.

An example of generating time–stamps for a situation where bounded time–stamps are required is as follows:

```
{
  func1();
  func2();
  func3();
}
```

The transformed code would be:

```
{
  BTSG tsg( ts, 3 );
  func1( tsg() );
  func2( tsg() );
  func3( tsg() );
}
```

The bounded time–stamp generator `tsg` is initialized with the time–stamp for the block of code. The second parameter initializing the time–stamp generator is the number of time–stamps required for this block. The overloaded function operator on time–stamp generators returns the next time–stamp in the sequence.

An example of the time–stamping transformation, as applied to nested function calls, where the untransformed code is:

```
void func( int a, int b)
{
  func2( a, b );
  func3( a, b );
}

//...

{
  func( 1, 2 );
}
```

the transformed code would be:

```
void func( TS& ts_fn_start, int a, int b)
{
  BTSG ftsg( ts_fn_start, 2 );
  func2( ftsg(), a, b );
  func3( ftsg(), a, b );
}

//...

{
  BTSG tsg(ts_start,100);
  //...

  func( tsg(), 1, 2 );
}
```

In the example above two time–stamp generators are shown: `tsg, and ftsg`. The argument passed to the function is a time–stamp, and that time–stamp is used to initialize a generator internally to the function.

The basic aim is that the program will be transformed to have the appropriate calls to generate time–stamps. Some time–stamp ranges will be partly allocated, being sub–ranges of time–stamps generated at compile time.

A similar transformation could be used for a loop where it was not known at compile time how many iterations of the loop would be invoked. For example this code:

```
{
  for (expr1; expr2; expr3)
  {
    func1();
    func2();
    func3();
  }
}
```

would be transformed to:

```
{
  UTSG tsg( ts, bits );
  for ( expr1; expr2; expr3)
  {
    func1( tsg() );
    func2( tsg() );
    func3( tsg() );
  }
}
```

where the constructor argument bits for the unbounded time–stamp generator (the class `UTSG`), allows the efficiency of the time–stamp generator to be optimized for the expected number of time–stamps, and expected levels of function call nesting. If the value of expression `expr2` can only be determined at run–time, it will be necessary to the use an unbounded time–stamp generator as shown above.

The examples presented so far have involved function calls on server objects, but the surrounding code has not been dependent on the outcome of server object code. If the expression `expr2` above involved a returned value from one of the invocations of a server object, it would be necessary to transform the control code also. The way this is done is that a server object is used to represent the C++ `for` control statement. Any variables involved in the expressions now also need to be server objects, this may involve using pure replacement objects if those variables were built–in C++ types. The transformation of control constructs is described in section 5.4.4.

### 5.5.2 Server Objects

As described in Chapter 4, the system is based on transforming selected program objects to be asynchronous remote procedure call objects, and distributing those objects over the nodes of a parallel computer. The selection of which objects to use as server objects, and hence which objects require transforming relies on annotations. The annotations are supplied either automatically by heuristics or explicitly by the programmer as hints, or instructions to the compiler.

Server objects have additional functionality over the original objects:

- server objects act as remote procedure call servers;
- methods are split into sub–methods which allow an asynchronous RPC implementation;
- methods are invoked as remote procedure calls;
- server objects are able to save their state;
- they are able to rollback;
- they have a logical clock and check for causality violations;

- they invoke rollback and recovery, and the sending of any anti–messages if a causality violation is detected.

Selected program objects are transformed to server objects. This means that the methods of that object are invoked as remote procedure calls, with the calls being implemented with messages passed between server objects, on different processors.

Server objects have their own thread of control, a main loop where they accept RPC requests, and process them. The RPC server retains the actual object's functionality, and the server object main loop invokes these methods with the appropriate arguments. The server object has the task of constructing the arguments by decoding the RPC message, and of constructing resulting RPC messages to represent the return values. The server object will have been transformed to have additional properties over the original object, in that it performs state–saving in order to be able to rollback. Also all of the methods are transformed to have time–stamps as arguments. The server object also has a logical clock, which is updated with the time–stamps of RPC requests as they are processed.

References to server objects are also provided, the only data they contain is a message stream pointing to the corresponding server object. References can themselves be passed as messages over message streams. This allows references to server objects to be passed as arguments to RPCs.

Each processor has a meta–server, to allow server objects of the required type to be spawned. The meta–object is called when objects are constructed, and returns the stream attached to the newly created object. Subsequent RPC commands are sent on this stream to the server object.

This is depicted in Figure 11, where creation of the server object is triggered by the creation of a reference to a server object on processor 1. A decision has been made that the server object will be created on processor 2, so a message is sent to the meta–server on

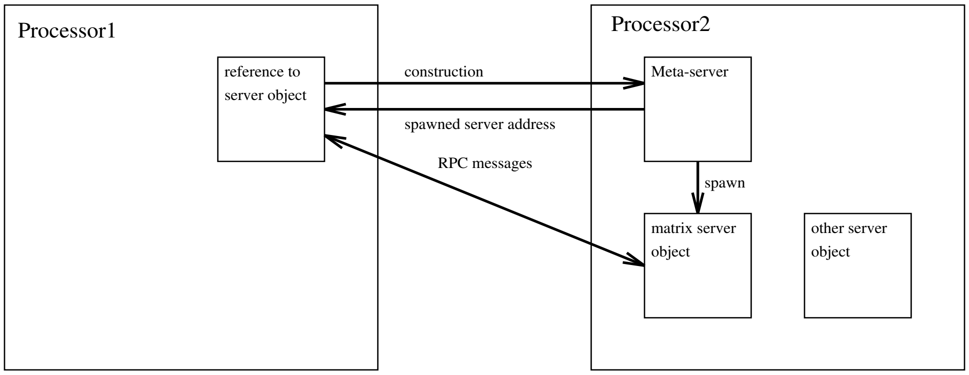


Figure 11: Spawning & using a server object

processor 2 requesting the creation of a server object of the appropriate type. The meta–server spawns the server–object, and returns the address of the newly created server object to the process requesting the construction. Subsequent interaction between the reference to the server object and the server object itself are addressed directly to the server object at the address specified by the meta–server.

The subsequent RPC calls can come from any process on any processor of the parallel machine, the reference once created, may be passed to other processes.

If we consider as an example a matrix object, which will be transformed for use as a server object, the original object with the method `inv()` and the friend function, `mul()`:

```
class Matrix
{
protected:
  double vals[10][10];
public:
  void inv();
  friend Matrix mul(Matrix&,Matrix&);
};
```

The `inv()` and `mul()` functions would be transformed to:

```
void inv(TS&);
Matrix& mul1(TS&,Matrix&,Matrix&,Matrix&);
```

The friend function is a separate function, and therefore is not invoked via the server object, although the arguments can be server objects.

Uses of the Matrix class are transformed to use references to server objects for matrices instead. The reference object looks like this:

```
class ref_Matrix
{
protected:
  omstream server_addr;
public:
  void inv(TS&);
};
```

The above class declaration for a `ref_Matrix` (a reference to a Matrix) shows that it encapsulates the fact that messages are sent to a server object, the field `server_addr` holds the address of the corresponding server object.

The `omstream` class used is a C++ class encapsulating p4 message passing between dynamically allocated processes. The functionality of the omstream class is described in detail in Chapter 7.

The code using `Matrix` objects is transformed to use instead a reference to a server object, a `ref_Matrix`, where use of a server object is required. When a `Matrix` is constructed, a message is sent to the meta–server on the required processor, and the omstream value is initialized with a stream to the server object spawned by the meta–server.

### 5.5.3 Asynchronous Transformations

This section describes the transformations which are necessary to ensure that the remote procedure calls are asynchronous.

The classical remote procedure call implementation in terms of messages is to have an invoking message which encodes the identity of the function to invoke, together with the arguments encoded as a byte stream. When the remote procedure has completed a result message would be returned. The invoking process blocks waiting for the return value while the process executing the remote procedure is busy.

This model is unsuited to an optimistic execution because it involves blocking, which restricts parallelism. For some procedure calls, those without return values, the asynchronous implementation is simple: just do not send back a return value, and do not wait for the return value in the invoking code.

For other procedure calls it is necessary to transform the code to allow the process invoking the RPC to continue with other work without having to block waiting for the result from the RPC.

If we consider the following example function `f()` with the untransformed code:

```
{
  int i = f();
  int j = g();
  int k = h( i );
}
```

There are three functions shown in the example above: `f()`, `g()`, and `h()`. The return value `i` is not used by the function `g()`, and therefore it should be safe to execute the functions `f()` and `g()` in parallel (provided that the functions are independent).

For simple cases where the return value is used to construct an object (in the example above a built–in object — an integer — which would be transformed to a pure replacement class in the transformed code) a technique can be used which instructs the invoked function to direct the return value to another server object.

In the above example, it would be necessary for `i` to be a server object, and hence one technique to allow the asynchronous dispatch of the function `f()` is to request that `f()` forward the return value to the server object for `i`. It is necessary to arrange for `f()` to send to the server object for `i` a message requesting that the server object be constructed together with the return value.

This requires the asynchronous dispatch of a larger portion of code, instead of asynchronously dispatching `f()` alone, the code portion dispatched is now:

```
int i = f();
```

If it is desirable to arrange for a larger section of code to be dispatched asynchronously as a unit within which all of the statements must be executed sequentially with the return values from each being required for the next, it can become complex to implement this with messages only. This is solved by the use of stub functions, as described below.

The aim is to exploit parallelism between sections of code which are independent despite being in the same server object.

There will be cases where the return value is required immediately, or even cases where this value decides the future course of execution.

The code below would result in an unavoidable rollback were the return value from function `g()` to be treated optimistically:

```
{
  int i = f( g() );
}
```

It would always be preferable to wait for the returned value, rather than optimistically assume something that will in all cases result in a causality violation, and rollback. The loss of parallelism resulting from blocking the thread of control of that server object can be offset by using parallel slackness. In the process of the simulation when a server object is blocked for this reason, provided that there is enough parallel slackness, all that happens is that other processes run. When the blocked thread is resumed by the waited for value arriving, it is returned to the list of runnable jobs at the OS level.

The problem of immediate use of information returned from functions is not restricted to function return values, the same problem exists for output parameters which are used soon after invocation of the function.

Output parameters are parameters with values which can be changed during the function call. These have to be server objects in the case where the function in question is a member function of a server object. What happens is that the function is passed a reference to the object which is passed as a parameter (it will have been transformed to be a server object).

In this way the function can modify its output parameters directly.

```
{
  int i = f( out );
  g( out );
}
```

In other words the output value is required immediately by the next statement. This might be a case where it would be advantageous to have a separate process which handles both functions and waits for the completion of `f()` before `g()` is invoked.

The transformations described in this section are designed to allow the parallel execution of functions invoked from a server object's flow of control. This is achieved by transforming the code so that the method dispatches can be asynchronous.

For functions, with return values, transformations are used to allow an asynchronous message passing implementation.

Server objects will be pure objects, so the return value from a method of a server object will by definition either be ignored, or be passed as a parameter to a method of another object on return. (If they were built–in types they will have been replaced by pure replacement classes).

As has been explained above, it is unnecessary to have a strictly pure source language, it is sufficient to be able to transform non–pure built–in classes to pure replacements. Also the 'purify' transformation need not be applied to the whole program, it is sufficient to apply the purify operation to server objects. Objects and control constructs which use the return value must be transformed to be server objects also. The "pure object only" restriction is just an alternative way of expressing the fact that non–pure objects must be first transformed to a form in which they can receive a message: they must be transformed to a user–defined, pure replacement object.

The way this transformation is used in the run–time system is to pass, along with the arguments for the remote object method invocation, instructions for where the result should be sent. For a simple case, such as an assignment statement, the returned value would be sent to the assignment method on the object which was on the left hand side of the assignment statement in the untransformed program. For a more complex example, where several methods are invoked and the resulting arguments when they arrive, are sent as arguments to a further method invocation, a stub process, or function may be necessary, which receives the multiple results and invokes the required method.

A stub function is used to asynchronously perform a blocking wait. A stub function will have a separate thread of control, and will be given the task of waiting for a particular set of messages to arrive, and to then send further messages as a result of this.

The need for a stub function for avoiding blocking the main thread of control of a server object is related to "futures". There arise situations where there is no sensible value which can be optimistically determined until a value is returned from an asynchronous RPC call. In such situations it is necessary to wait for the required values.

Where there is only a single value which is being waited for via the collection of the return value from one of the asynchronous calls, the situation corresponds closely to a future, and the wait for the return value.

So for example, if we consider passing the return value from a function being passed on as an argument to a subsequent function, with the untransformed code:

```
res = f( g( a ) );
```

The function `g()` is invoked with argument `a`, and the return value from `g()` is used as the argument in the invocation of function `f()`.

In terms of futures [21], this can be represented like this:

```
future g_res = g( a );
collect g_res;
future res = f( g_res );
```

where future represents asynchronous dispatch, and collect represents waiting for that value. There are however two problems with converting the above conceptual scenario into stub functions:

- with the use of classical futures, this does not scale to functions with multiple arguments, as the arguments can return out–of–order
- as this is an optimistic execution it is undesirable to block the thread of control of the server object.

To show why it is difficult to implement asynchronous RPCs with multiple arguments using classical futures, consider this example:

```
res = f( g( a ), h( b ) );
```

The example shows three functions: `f()`, `g()`, and `h()`. The return values from the invocations of `g()`, and `h()` with respective arguments `a` and `b` are passed as the two arguments to `f()`.

A simplistic implementation using futures would be:

```
future g_res = g( a );
future h_res = h( b );
collect g_res;
collect h_res;
future res = f( g_res, h_res );
```

However this is not asynchronous because the collect statements can block until the arguments come back. Also an ordering has been imposed on the receipt order of the results, whereas in practice it is undefined which result returns first. This could be accommodated by making use of the fact that the message passing system allows read requests which are tagged, and would in such circumstances buffer the other types of message. This situation would still be undesirable though because it is more efficient to process the return messages in the order in which they arrive.

The other problem, which might be apparent, is that in some cases the returned value is actually required immediately for further computation to proceed in the invoking method. For the assignment statement example, perhaps this would not be obvious, as the actual value may not be used for some time. However in some cases, the value is required immediately, such as the case when the result is directly passed in to a further remote object method invocation. In such cases there is little point optimistically executing the further remote method invocation, because it is known that there will be a causality violation in all cases when a null or default object is passed as an argument.

The solution is to split the invoking method into sub–methods, each acting as an entry point, with the required value which would otherwise cause the object to block as an argument.

The invocation which returns the value which is required is sent instructions to invoke the next sub–method. In this way the server objects are kept single threaded. At this stage it would seem possible to extend the model to include multi–threaded server objects, however this complicates the design when the full optimistic execution scheme is considered. The extra server object functionality of having a logical clock, and having a state–saver, mean that it would be difficult to see how a multi–threaded server could be reconciled with a single logical clock. The single server object would in effect have multiple sub–parts with different logical clock values. The desired parallelism which it might seem could be exploited by having a multi–threaded server object, can in any case be obtained by having more server objects, if the component objects of the server object were themselves server objects.

As was stated earlier, the aim of the object transformation is to allow an asynchronous implementation of this distributed message passing OO programming view. As should hopefully be clear by now, this can be achieved by having each method invocation RPC be implemented as an asynchronous message dispatch. The invoking server object can then continue execution. When the server is idle it must be ready to receive other RPC messages in it's main loop. It is important that the server be ready to receive any message at this point because, as this is an optimistic execution, it will not be known which method invocation will arrive next. In the normal course of operation of the system, methods can be invoked out–of–order.

### 5.5.4 Control Flow Depending on Server–objects

For control flow constructs where the flow of control is determined by the result returned by a method of a server–object, the control statement must itself be transformed to be a replacement control flow object, which has the properties of a server–object. It is only necessary to transform the control construct where the control flow is determined by the result from a server object. Thus, the example in Figure 12 would *not* require the `for` control construct to be transformed. The Matrix class is presumed to be transformed to be a server–object, so that all of the elements of the array `m[]` are server–objects. The optimistic execution of this example allows independent execution of the iterations of the loop. The data–dependent nature of the array accesses means that this loop could not be parallelized by a parallelization technique which relied only on compile time data–flow analysis. Because the bounds of the loop are determined at compile time the control construct is independent of the values of the server objects. Whatever rollbacks occur, and whatever RPC messages are sent between the server objects in array `m[]`, the number of iterations, and value of the index variable is unaffected. So in this example it is known that the `for` construct will not be rolled–back (except as part of some larger construct).

```
class Matrix;

{
  Matrix m[100];
  for (int i = 0; i < 100; i++)
  {
    m[i] = m[ f(m[ i ]) % 100 ];
  }
}
```

Figure 12: Control construct involving server–objects

The example presented in Figure 13 shows the opposite case, where the control construct outcome does depend on the result from a server–object. In this case it is not known at compile time how many iterations there will be of the `for` loop control construct, since that value is determined by server–object return values.

For this reason, it is possible that the optimistic execution of the code section may result in certain iterations of the control construct being rolled–back. To enable this, the control construct must be transformed to be a server object. The transformed `for` construct would be represented by the `server_for` server–object shown in Figure 14. Figure 14 shows the reference to a `for` server object, the actual server–object would handle the RPC requests, and therefore have a less useful interface from the point of view of showing the operations available.

Each iteration of the for loop sends itself a message to invoke the test of the terminating

```
class Matrix;

{
  Matrix m[100];
  for (int i = 0; i < m[i] && i < 100; i++)
  {
    m[i] = m[ (m[ i ] + i) % 100 ];
  }
}
```

Figure 13: Control construct involving server–objects in termination expression

expression (the `for_expr()` method). If the expression evaluates to true, and the loop does not terminate, the `for_expr()` method sends itself (the `server_for`) a message requesting the invocation of the next iteration (`for_iter()`). In this way, if at a later time a causality violation occurs and the value of `m[i]` assumed by the `for_expr()`, and hence the iteration invoked by that expression (`for_iter()`) is modified, the RPC representing the read of `m[i]` will itself be canceled via the sending of a cancellation message. The cancellation message will cause the `for` construct to rollback, and this will result in the erroneously initiated iterations being rolled–back. The `for_expr()` and `for_iter()` methods may themselves be split up into sub–methods depending on the complexity of the expressions, statements contained within the iteration, and reliance on server object–values. This splitting process follows the same rules as for normal sub–method creation.

Figure 15 shows the RPC messages involved in the overall view of the server–object `for_server` implementation of the for construct.

The RPC messages in Figure 15 are represented by solid arcs which are labeled by numbers in circles, and by the corresponding RPC method names. The dotted lines indicate the flow of control internally to the `for` server object. RPC message 1 represents the RPC message invoking the initialization (`init`) section of the `for` server object. RPC message

```
class Server_for
{
public:
  for_init();
  for_expr();
  for_iter();
};
```

Figure 14: Example `for` construct server object

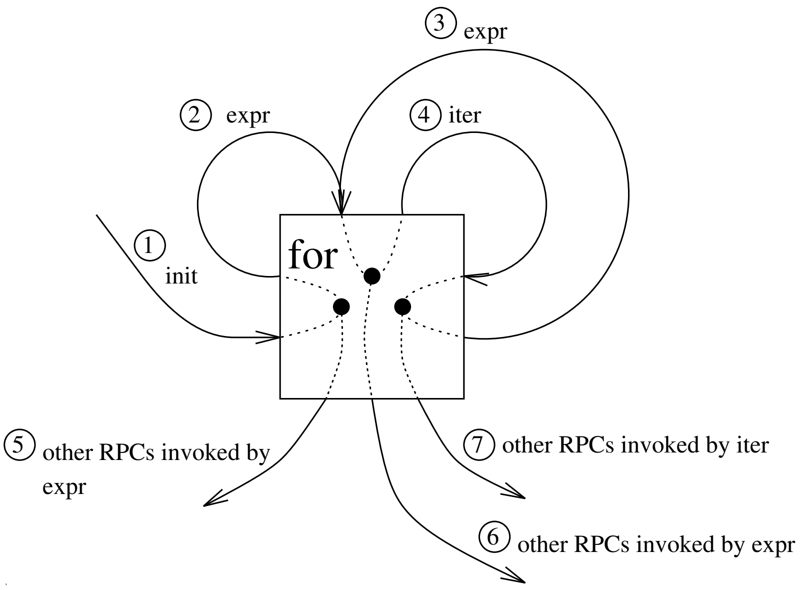


Figure 15: RPCs with replacement `for` construct server object

2 represents the invocation of the termination expression test (`expr`) by the initialization section. RPC message 4 represents the invocation of the body of the iteration (`iter`) in the event that the test succeeded. RPC message 3 represents the invocation of the terminating expression test by the iteration body (`iter`). RPC messages 5, 6 and 7 respectively represent the set of RPC messages which may be sent as a result of the initialization section (`init`), the expression test section (`expr`), and the iteration body (`iter`) respectively.

The RPC messages 5, 6 and 7 are drawn as coming from black dots in the Figure to indicate that the diagram is incomplete, and that for a given `for` server object, there may be 0, 1 or more messages resulting from those sections, and also 0, 1 or more further sub–methods to remove any blocking behaviour which may arise from these further RPCs.

## 5.6 Rollback and Logical Clock

As part of the transformation of selected program objects to server objects, it is necessary to add state–saving and rollback capability. The state–saving mechanism is described in more detail in the chapter on the high level run–time system.

The pragma used to annotate this transformation is:

- `p++-state-saving(class,trans)`

This requests that class `class` is transformed to have state–saving ability, and that the transformed class is named `trans`.

Consider the example:

```
#pragma p++-state-saving("A","state_A")

class A
{
public:
  meth1();
};

A::meth1()
{
  // code
}
```

This results in a new class called `state_A` which is transformed to incrementally save state on exit from each of the methods. The replacement for `A::meth1()` calls the state–saving method on exit:

```
state_A::meth1(TS& ts)
{
  // code
  _save(ts);
}
```

The methods will be transformed to have time–stamps also, as state–saving requires time–stamps. The time–stamp of the invocation of the method will be passed on to the state–saver. There is a base state–saving class from which the replacement class `state_A` will be derived.

```
class state_A : protected _state_saver
{
public:
  meth1(TS&);
};
```

The state–saver class provides functions to save the state of a block of memory, to rollback to specified time–stamps, and to read the state of the object from a specified time–stamp in the past.

```
class _state_server
{
public:
  _state_saver(TS&,void*,size_t);
  save(TS&);
  rollback(TS&);
};
```

For server objects which are composite objects, each of the contained objects can be chosen to do its own state saving, or to be state–saved by the current object.

```
#pragma p++-server("C","stub_C","server_C")

class C
{
public:
#pragma p++-use-state-saving("a")
  A a;
  B b;
};
```

A call to the low level state–saver function is added at the end of each method. A rollback method is constructed for the transformed object, this method merely calls the low–level roll back method. The size argument will be: `sizeof(object)` where `object` is the object being transformed, and the start of object argument will be: `this`, the C++ reserved word meaning a pointer to the start of the containing object. For non–polymorphic objects, it is actually unnecessary for the `this` pointer to be passed in, as the objects grow down wards from the `this` pointer in conventional inheritance. With virtual inheritance, as required for possible future multiple inheritance, fields are assigned above the `this` pointer of the base class, so the object's space can grow upwards under certain conditions also. This necessitates passing the `this` pointer for the actual class, which avoids all confusion.

Each server object has a logical clock, which is updated to the time–stamp associated with RPC requests as they are processed. The server object is transformed to include such a logical clock. It is this time which is updated accordingly as RPC requests are processed, and when rollbacks are executed due to causality violation.

## 5.7 Example

The example given in Figure 16 below has a dynamic dependency. Static analysis of the loop in the example would reveal that the loop has a possible dependency and hence must be executed sequentially. An optimistic execution is able to parallelize the loop despite this possible dependency. The code is using a matrix class which has the usual operations.

The `Matrix` class is transformed to become a server object `Server_Matrix`, and the invocations of `Matrix` class methods are transformed to be RPCs. The flow of control taken by the `if` statement depends on the state of a server object and so is transformed into the `Server_If` server object. `det()` invokes the `eval1()` method of server object `if1`, sending that object its return value. `eval1()` uses the return value of `det()` to determine whether to invoke the `then1()` method which implements the then part of the if statement.

The original code:

```
class Matrix {
protected:
    // representation
public:
    Matrix();
    Matrix( const Matrix& );
    Matrix& operator=( const Matrix& );
    Matrix& inv();
    double det();
    // other operations
};

{
    Matrix w[ 100 ];
    ...
    for ( int i = 1; i < 100; i++ ) {
        if ( w[ i ].det() == 0 )
            w[ i ] = w[ i - 1 ];
        w[ i ].inv();
    }
}
```

The transformed code (simplified):

```
class Server_Matrix;
class Server_If1;

{
    Server_Matrix w[ 100 ];           // placed over processors
    ...
    for ( int i = 1; i < 100; i++ ) {
        Server_If1 if1;               // server for if statement
        w[ i ].det( if1, eval1 );     // RPC, result sent to if1
        w[ i ].inv();                 // RPC call
    }
}

Server_If1::eval1( double res ) {    // receive result from det()
    if ( res == 0 )
        then1();                      // RPC to self to do body
}

Server_If1::then1() {
    w[ i ].operator=( w[ i - 1 ] );  // RPC to assign to w[ i ]
}
```

Figure 16: Example code, and its optimistic transformation

If the determinant `w[i].det()` is only equal to 0 for `w[3]`, then a possible optimistic execution of the above would be for a number of iterations to have been executed before the `if1` server processed its `then1` message for `w[3]`. The `then1` code would send an RPC to `w[3]` requesting an assignment, this would cause the server object `w[3]` to rollback as it will have already processed the RPC request for `w[3].inv()`. Table 1 below shows the sequence of events for server object `w[3]`. Time–stamps $t$ in the range $i < t < i + 1$ have been used for iteration $i$ of the loop. When the `w[3]` server object has rolled–back to time 3.1, it resumes processing its event queue, executing the straggler event 3.2 and then re–executing event 3.3. The other server objects are not affected by `w[3]` rolling back.

| logical clock | RPC message | message time–stamp |
|---|---|---|
| 0 | det() | 3.1 |
| 3.1 | inv() | 3.3 |
| 3.3 | op= (assign) | 3.2 (CAUSALITY VIOLATION) |
| 3.3 | ROLLBACK TO | 3.1 |
| 3.1 | op= (assign) | 3.2 |
| 3.2 | inv() | 3.3 |

Table 1: Time–stamps of example optimistic execution

# Chapter 6

# The Run–time System

This chapter describes the run–time system. The run–time system provides the primitives for the code output by the transformation system, and the optimistic execution system support. Other parts of the run–time system include: the message passing system, causality violation detection mechanism, lazy cancellation and lazy re–evaluation, and references, and GVT calculation and server objects.

## 6.1 Rollback Support

The rollback systems used in optimistic parallel discrete event simulations are described in Chapter 3. It is important, in general, with optimistic execution systems, to minimize the overhead of maintaining the state–saving information required to enable rollback.

A novel object oriented approach to reducing state–saving overhead can be used for this system. As has been expressed in Chapter 3, neither types of state–saver — incremental, or periodic – wins in the general case. There are situations where either wins.

The insight then is to use a different type of state–saver on a per–object basis, depending on that object's memory usage patterns.

### 6.1.1 Hybrid State–saving

The hybrid state–saving approach seeks to provide a trade–off which achieves a system which though still incremental, does the state–saving of objects at arbitrary levels in the object oriented class hierarchy. This allows states to be saved less frequently if this would be desirable depending on the characteristics of the object.

The hybrid incremental state–saving scheme has the ability to vary its performance characteristics to suit the object's state–saving pattern on a per object basis. Its characteristics can be varied between being purely incremental, and being some way towards periodic. Thus the hybrid incremental state–saving scheme will not save all states, but choose to save states at a particular granularity. The characteristic of not saving all states is shared with periodic state–saving schemes. It would be possible to use coasting–forwards, or recomputation (see section 3.4) to achieve an effect much like the coasting forward used in pure periodic state–saving schemes. Using coasting–forwards would make the design more complicated, but would allow memory to be traded off against compute–time (and added complexity). A simpler approach is to use the level of objects which can be rolled–back to arbitrary method invocation entry points as the level of computational granularity for object methods. Only objects that can be rolled–back without resorting to coasting–forwards are expected to be used in the design.

The decision not to use coast–forward simplifies the design, and the desired effect of having lower granularity object methods can be achieved by selecting the smaller granularity objects as the level to use as server–objects. Server–objects are objects which have their methods executed optimistically and are discussed further below.

To illustrate how the hybrid state–saving scheme works, we will consider the areas of strength of both the periodic state–saving scheme, and the purely incremental scheme. This exercise should demonstrate the flexibility of the hybrid scheme, and how it is able to provide some of the advantages of both the incremental, and the periodic schemes.

### Using hybrid state–saving to approximate the periodic scheme

The area of strength of periodic state–saving is in situations where the simulation object has a state which is changed many times in the course of the computation associated with an event. Periodic state–saving can win over automatic incremental state–saving in situations such as this because it avoids the excessive state–saves performed by the incremental state–saving scheme. (There are also state–saving schemes which are known as incremental which give the user control over the time at which an incremental change is saved, an example being the SPEEDES state–saving scheme which is based on user supplied undo functions.)

The hybrid state–saving scheme can achieve a similar effect, with similar savings to those made by periodic state–saving schemes, by making the object a state–saving object in its own right. That is, all of the objects that make up this object's state are not able to save their state. This means that state will be saved after each method invocation of the containing object, rather than at the end of method invocations of contained objects. Effectively this is a periodic state–save, with a variable period between state–saves, and where the state–save operation for the object is invoked at the end of each method invocation. The state–saving operation can still be classified as an incremental state–saving scheme, it is just that the state changes are regarded as the invocation of the object's methods. The lower level operations which make up the objects methods, and the changes they make are not counted for the purposes of state–saving. This means that the whole object's state will be saved after each method invocation.

### Using hybrid state–saving to behave like the incremental scheme

The area of strength of incremental state–saving schemes is in situations where the simulation object has a large state, only parts of which are changed during the course of the computation associated with an event. Incremental state–saving can win over periodic state–saving in situations like this because it avoids the wasteful saving of the entire (in this case large) state at the end of the method invocation, even when only small portions of that state have changed. As an example, consider an object which consists of a large array of smaller objects, only a few of which are typically changed by the method. Incremental state–saving would mean that each of the lower level objects that make up the array elements would be an incrementally state–saving object. In this way only objects in the array with state which changed would be saved.

### Hybrid use

The fact that decisions on state–save method can be made per object means that different objects within the simulation, or even within the same higher level object, can have different state saving mechanism, tending more to periodic, or to incremental depending on the object, the size of its state, and the number of changes to contained objects its methods make.

This gives the hybrid scheme the advantages of both schemes, periodic, and incremental, plus the advantage of allowing mixtures as required by the program. The resulting state–saving system should be more efficient than either method in isolation, as it can be tailored to suit the characteristics of the program.

If the lowest level objects are translated to be state–saving objects, the hybrid state–saving scheme degenerates to a pure incremental state–saving scheme. The opposite extreme, having as large as possible state–saving objects approaches, but does not equate to a periodic state–saving scheme. It tends to be more periodic than the incremental state–saving scheme, but it is not a true periodic state–saving scheme. The main difference is the actual time at which the state is saved, rather than being at fixed intervals as would be the case in a true periodic state saving system, state is instead saved after each of the operations on the simulation objects.

### 6.1.2 Implementation

In this section the implementation of the rollback mechanism will be presented. The example implementation is for the C++ programming language.

The incrementally saved states are marked with time–stamps as described in Chapter 5. There are a number of mechanisms to describe, the raw byte-wise time–stamped state saving mechanism, the generalized transformation of an object to transform the object in to an object which is able to rollback, and the method by which rollback to a particular point in simulation time is achieved for such an object.

#### Low Level State–saving Mechanism

The low–level state–saving mechanism is a run–time system component which is able to save state in the form of an untyped variable sized sequence of bytes. Each operation to save state has an associated time–stamp. A preliminary implementation was written which uses a simple linked list to hold the state–saved groups of bytes.

A rollback operation on this low level state–saver was also written, this is in the form of a function which searches back through the saved state for a state with a time–stamp less than or equal to the time–stamp requested by the rollback operation.

**Transformations**

As described in the previous chapter, there are a number of parts to the transformation, firstly parameterizing by time–stamp, then adding the calls to the raw state–saver at the end of each method, and finally choosing to use either transformed state–saved objects, or untransformed objects, for composition to build hybrid state–saving objects.

The parameterizing by time–stamp transformation is basically to add an extra parameter to each of the member functions of the object, so that when it is called its time–stamp is passed to it, and the state–save at the end of the method is given that time–stamp. The only complications are overloaded operators which are called implicitly or explicitly. C++ considers overloaded operators to be just ordinary functions, but there are restrictions on the numbers of parameters they may have, to fit in with their semantics. Another problem is time–stamping destructor calls. Destructors are usually called implicitly, and in any case can have no parameters. Several other implicit parts of C++ cause similar problems by virtue of restrictions on the number of parameters. A transformation phase called explicify, makes explicit all such language features, and converts them all to ordinary function calls. This transformation is discussed in detail in Appendix A. After the explicify operation, the adding of an extra time–stamp parameter is straightforward.

The next part of the transformation is the calls to the low level state– saver. The object which is being transformed is additionally transformed to derive the low level state–saving capability from the run–time system component. This automatically gives it the capability of rolling back, once the appropriate calls to the state–save operation, and composite rollback operation are added.

The constructor of the transformed object is given an extra parameter, the time–stamp of the object's creation. This is passed in to the constructor of the run–time system low–level rollback base–class.

Also a call to the low level state–saver function is added at the end of each method. A rollback method is constructed for the transformed object, this method merely calls the low–level roll back method, with the appropriate size argument, and start of object argument (see Section 5.5).

The above discusses transforming a program level object into an incrementally state–saved object. The last part of the transformation is the combination of base–level incremental objects, and untransformed objects into hybrid state–saving objects. Each object can be composed of or derived from combinations of incrementally state–saved objects, or untransformed program level objects. The choice is expected to be driven by annotations, or heuristics.

## 6.2 The Message Passing System

Server objects are placed on different processors of a distributed memory parallel computer. The message passing system is required to transport the messages between simulation objects. To facilitate parallel slackness, multiple processes are placed on each processor, so the message passing system must also be able to handle messages between processes on the same processor. Messages are time–stamped in accordance with the requirements of the optimistic execution mechanism.

An implementation of a message passing system was built on top of the p4 message passing library. P4 was chosen as a portability layer, to insure that the system could be ported to a wide range of parallel computers.

The p4 library is discussed in Chapter 7. P4 has the functionality of tagged messages, with messages buffered at the receiving process. Messages may therefore be received out–of–order, the call to receive a message has as a parameter the tag type to receive. However, p4 has the restriction that dynamically allocated processes cannot receive messages.

The processes in an optimistic execution system are required to be created dynamically. The message passing system built on top of p4 allows dynamically allocated processes to exchange messages. In addition to this functionality, the messages are time–stamped, and logged. Message logging is required to allow cancellation messages to be constructed in event of causality violation during the execution of the system. The messages are received (i.e. taken out of the buffer) in increasing time–stamp order, which means that they may potentially be received in a different order to that in which they are sent even if they have the same tag. That is to say, if there is more than one message in the receive queue, the message returned by the receive message call is the one with the lowest time–stamp.

This is to facilitate the parallel discrete event simulation requirement that messages are required to be processed in non-decreasing time–stamp order. For the sending process, messages are logged (copies are kept until at least GVT has passed the time–stamp of the message), so that their antimessages may be sent in the case of rollback and there is information to compare against for lazy cancellation. At the other end the received messages are also kept until after GVT has passed as the messages are required for possible re–processing after a rollback.

## 6.3 Causality Violation Detection

Causality violation is detected purely on the processing sequence of the time–stamped messages. It is the processing sequence which is important for purposes of violation detection. Messages could actually arrive out of order, and be processed in order, as messages are sorted in increasing time–stamp order on arrival. A causality violation occurs when the next message available for processing has a lower time–stamp than the logical clock of the simulation object receiving the message. There is code in the message passing part of the run–time system which checks for this situation, and invokes the sending of anti–messages, and the rolling back of the object to a point in simulation time just before the time–stamp of the straggler message.

For the actual cancellation messages, only the time–stamp is sent in the anti–message. This is an optimization proposed in [65]: only the time–stamp is required, as this is sufficient for cancellation where time–stamps are unique. The uniqueness of time–stamps is not traditionally adhered to with parallel discrete event simulation, but the close mapping to the equivalent sequential execution used in this thesis for automatic parallelization already ensures that this is the case.

Thus a causality violation is detected by a message being processed out of time–stamp order, anti–messages can be sent by just sending the logged time–stamps of the messages to be canceled.

An optimization to this scheme is lazy cancellation. This optimization stems from the fact that sometimes, even though a message arrives late, no difference in the messages sent results. If this turns out to be the case, the anti–messages will have caused unnecessary rollbacks. The way that this optimization would be implemented would be to delay sending the cancellation messages, rollback the object in question, re–process the messages, and check to see if any messages sent are different to the logged messages from the optimistic execution. If there is no difference in the messages, the anti–messages do not need to be sent, and the knock–on rollbacks will have been saved. However, if the messages do turn out to be different, at this point the anti–messages are sent. The negative aspect is that the objects for which the anti–messages were destined will have had time to do more work; work which will need to be rolled back.

Another optimization to the optimistic execution scheme is lazy re–evaluation, even when an object has been rolled back to take account of a straggler message, it is not certain that the object's state will be affected. To determine this for lazy re–evaluation, its execution, and the state–saving system is switched to a different mode; one where the state–save operations compare against the already existing saved state. If a difference in state is observed, the state–saver is switched to normal saving mode, and possibility of lazy re–evaluation is abandoned. If, however, no differences in state are detected during the re–evaluation of the code corresponding to the straggler message, it is known that the complete re–evaluation can be avoided. The system is rolled forward to the point where execution was taking place when the straggler message arrived, and the object resumes execution from that point.

The read only straggler optimization described in Chapter 3 is used as an optimization on the basic time–warp system. The optimization, which is derived from Fujimoto's Space–Time Memory, as used in his Virtual Time Machine [31], allows a straggler event which it is known will only perform a read operation to be executed with greater efficiency than that possible for lazy re–evaluation.

## 6.4 Server Objects and References

Only server–objects can be referenced across processor boundaries. A server object reference is possible because the local stub object encapsulating the remote procedure calls is already a reference to the remote server–object. As described in section 7.8, message streams can themselves be passed over message streams. The reference object half of a server–object client–server pair, is little more than a message stream pointing to the corresponding server–object, and server–object references can also be passed on message streams. This means that references to remote server–objects can be passed relatively transparently.

Operations on the referenced server–object are forwarded to that server–object irrespective of the actual processor on which the server–object reference resides. Requests for a value copy of the server–object are themselves operations on the object embodied by the server–object. Such requests can result in the copy constructor making copies of the server object. These values would also normally be server objects because they must also be able to rollback, as the value obtained from the copy constructor, or assignment operator of the respective server object may later turn out to be incorrect due to the arrival of a straggler message at the server–object, causing a causality violation and rollback.

### 6.4.1 Smart Pointers

As with references, the only cases where memory can be accessed on remote processors is where the remote access is to a server–object. A pointer to a server object is implemented as a reference to a message stream pointing to the server–object. Complex pointer based objects such as linked lists, and trees can be built as normal. Operations on that pointer value are sent as RPC requests to the corresponding server-object. Pointers are restricted to pointing at server objects.

## 6.5 Calculation and Use of GVT

GVT is the calculation of the time in the optimistic execution before which events can not be rolled back. GVT is defined as the minimum of the local logical clocks and the messages in transit. GVT algorithms are discussed in Chapter 3. No GVT algorithm has been implemented, but there are standard algorithms available in the literature. The algorithm in [65] is suggested to fit in with this design because it has the advantage of allowing cancellation messages to be formed by just the time–stamp of the message (rather than the whole message).

### 6.5.1 Memory Management

The run–time system also contains a specialized set of memory management routines, which have the property of being able to undo memory management operations. This is necessary for the implementation of the optimistic execution scheme, because it is necessary to be able to rollback memory management operations as well as all other program constructs. In C++ nomenclature, there are two memory management operators `new`, and `delete`, these are respectively the operations used to allocate memory, and to free previously allocated memory.

For purposes of rollback, the rollback of an allocate operation requires only the freeing of the memory. The rollback of a free operation, is however more complex than re–allocating it. To unfree a previously freed memory area requires that all of the previous values be retained. The way that this is implemented is not to actually delete the memory area when it is freed, but rather to just mark it as pending deletion. In this way a rollback of a free operation can be implemented by just removing this mark, so that the memory is no longer considered to be pending deletion.

The actual deletions, which are necessary for the heap not to be used up quickly, are delayed until GVT. When the time–stamp of the deletion is passed by GVT, the space can be safely reclaimed, and returned to the free store.

### 6.5.2 Exceptions

Exceptions, in the sense of operations which cause the program to halt with an error condition, also need to be delayed until GVT. This is for the reason that they may have been erroneously raised. A later straggler event may cause the exception to be rolled back. So things like dereferencing a null pointer, resulting in a segmentation violation, are trapped, and held pending GVT. If the violation still exists at GVT, the exception is reported, and the program terminated.

## 6.6 Input / Output

Input and output are also delayed until GVT. This is implemented by having special versions of the C++ classes in the run–time library which have the property of delaying processing until GVT. This ensures that they will never be rolled back.

In a limited sense is possible to rollback output: if the output is cached, the data may be changed before the cache is flushed. In general, however, it is not possible to rollback output. Rolling back input is similarly restricted. Provided that the program is not interactive, input may be rolled back by providing a mechanism to return the data to an input buffer for re–reading. In the general case, however, neither input, nor output can be rolled back. This is not a problem, as this property of I/O can be coped with by deferring the I/O actions until GVT. This is the same approach as taken by TWOS [47].

The system requires special purpose libraries to delay I/O until GVT. Where library source code is not available, program correctness can be ensured by delaying all library functions until GVT with wrappers around those functions. Greater efficiency can be obtained where the library source code is available, by only delaying those functions which do actual I/O, and taking account of library functions which retain state (for instance some string manipulation functions eg. `strtok`, and memory allocation functions: `malloc`, `calloc`, `free`).

## 6.7 Fossil Collection

Fossil collection refers to the fact that memory used to store rollback information, and event queue information can be reclaimed when GVT passes the time–stamp of that information. It is known that no straggler events will arrive before GVT, so it is safe to discard any information pertaining to rollbacks prior to this time. The information which can be discarded therefore includes: incremental state–saved information, entries in event queues kept in case of rollback, delayed input and output, information retained to allow free store operations to be reversed.

# Chapter 7

# The Run–time Libraries

This chapter describes run–time library components which are sufficiently self–contained to be separated from the preceding high level run–time system chapter without damaging the logical flow of that chapter.

The components discussed in this chapter are ordered in a bottom up sequence: starting with the lowest, portability level, the p4 library, moving up to a set of run–time library calls allowing communications between dynamic processes, then the C++ streams interface on top of this, and then on to remote procedure calls and the associated flattening operations.

In more detail, the section on the portability layer, p4, describes first the standard p4 cluster parallel programming model, and the extensions necessary to use p4 for optimistic execution schemes. Issues involved in porting p4 onto transputers are also discussed.

Next the object oriented streams layer is described, together with it's implementation, and the transformations which are required for automatically producing input / output operations for user defined objects. The related issues of the workings of remote procedure calls and the accompanying issue of the "flattening" of program data structures to enable them to be passed over message streams are finally discussed.

## 7.1 The P4 Standard Library

The p4 parallel library [18, 17] which originates from 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.

### 7.1.1 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.

The p4 model of computation has been implemented at Argonne (with contributions from other researchers) on UNIX workstations, shared memory multi-processors, distributed memory computers and virtual shared memory computers. The p4 library provides heterogeneous operation across these architectures, so that a single p4 program may be run on a combination of architectures simultaneously.

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 multi-processors, the shared memory can be implemented in hardware between p4 processes which are running concurrently, 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.

With virtual shared memory, two options are possible:

- to have processes in the same cluster placed on the same processor, where virtual shared memory is used for message passing only;
- to make use of the virtual shared memory provided by the architecture to allow processes in the same cluster to be placed on different processors.

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 tagged: it is possible to request a message with a particular user defined tag, the p4 system will automatically buffer messages with different tags 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 requested.

P4 provides facilities to define a set of machines and a description of how clusters map on to these 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 initialized. There is also provision for dynamic process creation, although such processes are not able to communicate via message passing.

A global barrier mechanism and a cluster barrier mechanism are provided by p4, implemented by message passing and semaphores respectively. With the cluster barrier mechanism, it is possible to wait for all, or a subset of the cluster processes to synchronize.

### 7.1.2 P4: Extensions

The extensions made to p4 [5] were to add a pre–emptive multipriority scheduler to the transputer implementation. The transputer architecture normally provides only two priority levels. The extra priority levels are used to prioritize the current simulation processes on a processor by their logical clock value. The processes with the lowest logical clock are executed with highest priority. The library extension also allow the priority of a process to be modified at run–time, this allows the priority to be modified as the logical clock of the process changes.

## 7.2 P4: Transputer Port

The transputer implementation [71] differs from the generic UNIX implementation in two areas: in the way processes are created, and the message passing mechanism.

### 7.2.1 Fork

The Argonne implementation of p4 makes use of the standard UNIX fork, which means that processes have their own copy of the data segment and of the heap segment.

The T800 series transputers, however, have an unsegmented architecture, and are unable to directly support a segmented process model. It is possible to provide a fork using a compiler which generates extra code to emulate segmented memory, as is done in the Helios O/S [60]. However there are two costs incurred here: the overhead of the software emulated segments, and the memory and time overhead of copying forks.

The T9000 series transputers [73], [52] have a segmented architecture, and should theoretically be able to provide a full segmented UNIX process model. However no support for this is provided in the current Inmos C toolset, and using these facilities would involve resorting to the assembly level.

The transputer model of parallel processing has differences from the UNIX process model. Transputer processes are orders of magnitude more lightweight than their UNIX counterparts. With the transputer architecture, process creation becomes a primitive operation at the hardware level — creating a process is literally a machine instruction.

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.

Process creation on transputers does not involve copying the data segment or the heap segment. The heap is not copied because the T800 series transputers do not have a segmented architecture, and this makes copying a heap difficult. It would be possible to provide a copy of the heap, although this requires the software simulation of segment registers, and adds overhead to any pointer dereferences.

The only way to provide a copying mechanism for the data segment which fully preserves the semantics of the UNIX fork() is to use a similar mechanism, namely software emulation of segment registers. This adds overhead to any accesses to global data, all data accesses to the data segment, heap segment and stack segment require an extra level of indirection.

The standard process creation function in the INMOS toolset, `ProcRun()` does no segment copying; both the data and heap segment will be shared by parent and child process. This function is implemented directly in terms of the hardware process creation instructions, and has very low overhead compared to a standard UNIX copying `fork()`. It should be noted that there are a number of variants of UNIX `fork()` in use on different UNIX machines, which provide optimizations over the basic copying `fork()`. Also some UNIX vendors provide lightweight threads packages which come somewhere between standard fork() and transputer processes in efficiency. Nevertheless, process creation on UNIX machines is a much more costly affair than on transputers.

The problem is that the p4 model requires a small amount of per p4 process state, and with the transputer process model there is no easily identifiable area to store this in based only on the transputer notion of a process.

To understand the solution to this problem requires some more insight into the transputer assembly language programming model.

### 7.2.2 Transputer Architecture Programming Model

The relevant aspect of the transputer assembly language programming model [43, 42] is that the CPU has a *workspace pointer* which doubles as a conventional stack pointer and a process descriptor.

The transputer architecture has a hardware scheduler, and during the normal operation of this mechanism, transputer processes are descheduled pending I/O, or waiting for semaphores, or external events. The workspace pointer is used to identify a process for these purposes. The transputer has no general purpose registers in the conventional sense, only a three deep register stack, on which all arithmetic operations are performed. A strange feature of the architecture is that process switching can not (generally) occur between arbitrary instructions. The purpose of this is to avoid copying the register stack on a process switch, this design gives the transputer a very fast context switch time. There is a defined set of instructions which can be interrupted, and a defined machine state which can be interrupted. In the normal course of operation it is typical for the register stack to become empty, as there are only 3 operands which can be stored on it. It is theoretically possible for a process to issue instructions which will ensure that the stack is never empty, and hence to never relinquish control. However, there are a set of compiler writer's guidelines which delineate rules for ensuring this does not happen, and indeed it would be rare for this to happen accidentally, as the limited depth of the stack, and the existence of instructions which require all three stack values as operands means that it would be a strange usage pattern which permanently kept the stack non–empty.

Whilst processes are descheduled, the instruction pointer and information pertaining to the reason for the process's descheduling are kept at negative offsets from the workspace pointer. For purposes of reference in any process synchronization or communication the process may have entered into, the process is internally referred to by its workspace pointer.

The unusual consequence of this is that the CPU's natural way of referring to processes, by workspace pointer, changes over time. This is because the workspace pointer, during its normal course of use as a stack pointer will move with depth of procedure invocation, as is typical of conventional RISC and CISC processors.

This defines the problem — a process has no concept of self. There are no internal process tables, or process ids which could be used to associate the information p4 wishes to store for each process with its id.

### 7.2.3 Aligned Workspace Allocation

The problem specified above 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, as described above, 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. The negative offsets from the workspace pointer are only used when the process is descheduled so that this use is guaranteed not to interfere with the use of the workspace pointer as a stack pointer.

A process can determine the value of its **WPtr** (workspace pointer) using a single CPU instruction. The solution is to allocate a process's workspace at an aligned address. The stack grows downwards from the top of a process's workspace area. If we restrict the size of the workspace to being less than the alignment of the process, we can extract the lower limit of the workspace from the current workspace pointer. To see why this is, consider a workspace region **wspace** aligned on a boundary which is a power of 2, say **ALIGN = 64k**, so **wspace** *mod* **ALIGN = 0**. Say we have allocated **WSIZE = 10k** of workspace for the process, then the workspace pointer **WPtr** for the process will start at **wspace + WSIZE**, as the stack grows downwards. We can see from this that the legal values of **WPtr** will be from **wspace + WSIZE** down to **wspace**. We can calculate **wspace** from **WPtr** at any time during the life of the process based on the relation **wspace = Wptr** *and* **MASK** where **MASK =** *not* **(ALIGN - 1)**.

Based on this relation we can place **DATA** bytes of per process data at locations **wspace** to **wspace + DATA** by increasing the size of the allocated workspace to **WSIZE + DATA**.

### 7.2.4 Arbitrarily Aligned Pointer Allocation

The system provides a heap implementation which allows the allocation of arbitrarily aligned blocks of memory. As well as the usual `malloc()`, and `free()` it provides `amalloc()` which takes a second parameter, the alignment boundary required for the block of memory requested. A possible problem here is the memory efficiency of allocating blocks on large memory boundaries. If we allocate on 128k boundaries comparatively small objects of say 10k, we would seem to be wasting the remaining 118k of heap space. The

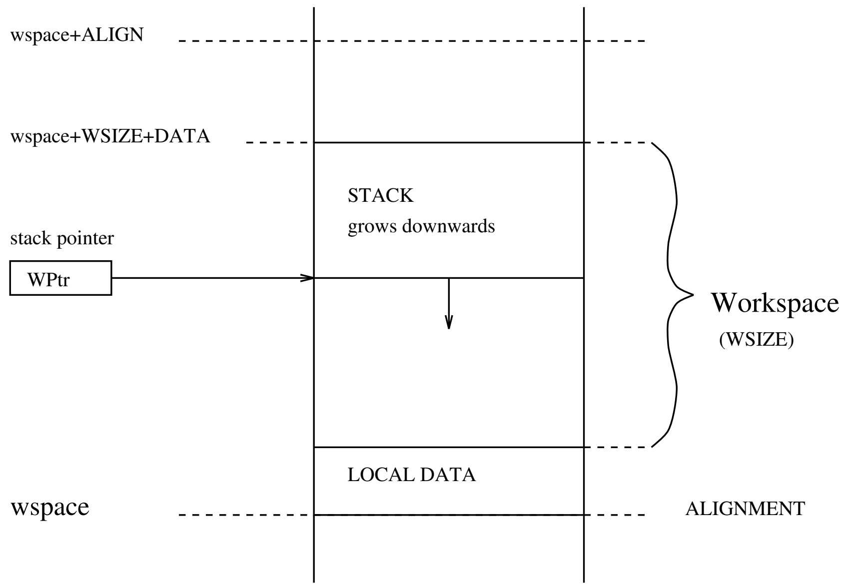


Figure 17: Aligned workspace

heap implementation gets around this problem by returning the remaining space to the free list for the allocation of objects with no, or smaller alignment requirements.

With this system in place, it is possible to retain the transputer's advantage in fast dynamic process creation, provide a p4 model which requires a small amount of per process state, and at the same time minimize the memory overhead.

### 7.2.5 Transputer p4 Memory Model

The result of these implementation decisions is that the p4 model now has only a p4_shmalloc() - the function which allocates memory from the shared memory area. The system provides a p4_malloc(), with the same functionality as p4_shmalloc(), the memory it allocates will not be copied for p4 processes.

The other major implication for p4 programs is that the data segment is no longer copied.

### 7.2.6 Message Passing: Channels

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. The p4 message passing calls have been implemented in terms of the Inmos VCR (Virtual Channel Router) software [44].

P4 messages provide point–to–point communication, whereas the transputer hardware provides placed channel communications.

The communication capabilities of the hardware are limited to the placing of only one communications channel on each of the hardware links. Further more the hardware provides no facilities to send a message which would have to traverse several hardware links.

There is a hardware solution to these problems: the T9000 transputer and - C104 router [36], [52], [73] and on the current generation of transputers several software solutions to these problems: Tiny [23], Southampton VCR [27] and the Inmos VCR [44].

The Inmos VCR system provides virtual channels which can be placed between processes on any processor in transputer network, the necessary through–routing and multiplexing is performed by the VCR software.

In order to implement p4's point to point communication in terms of virtual channels a software configuration file is provided which links each processor to each other processor in the network. Then a multiplexor and a demultiplexor process is provided for each virtual channel. The multiplexor for a particular channel will forward all messages destined for the remote processor via the virtual channel.

There will be a virtual channel $V_{AB}$ from processor A to processor B, and a multiplexor $M_B$ on processor A; also a demultiplexor $D_A$ on processor B. (see Figure 18.)

The steps involved in a message passing asynchronously from process 1 on processor A to process 2 on processor B are as follows:

- Process 1 inserts the message into multiplexor $M_B$'s queue
- Process 1 resumes execution
- Multiplexor $M_B$ forwards the message to demultiplexor $D_A$ via virtual channel $V_{AB}$
- Demultiplexor $D_A$ receives message and inserts in its message queue
- Process 2 removes message from demultiplexor $D_A$'s message queue

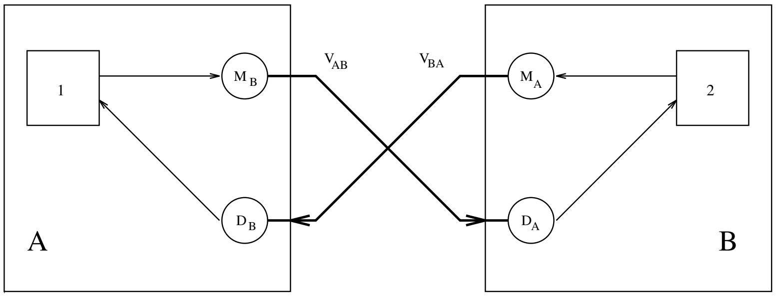


Figure 18: P4 communications

There is however a problem with this communication scheme as it stands:

Consider processes 1 and 2 on processor A; processes 3 and 4 on processor B. Process 1 sends a large message to process 3 which ties up the virtual channel for a significant amount of time. Now process 2 sends a message to process 4, but the transmission of this message must wait until the large message has been delivered.

To alleviate this problem we place two virtual channels between each pair of processors: one for large messages and the other for small messages.

There are now two paths a message can take depending on the message size, this means that messages will not necessarily arrive in the same sequence as they were sent.

Consider processes 1 and 2 on processors A and B respectively, and that process 1 sends two messages to process 2, a large message followed by a small message. It is possible for the small message to be delivered to process 2 before the large message even though it was sent later.

To get around this problem we use information about the number of undelivered messages for a process on the large and small message queues for each multiplexor. If there are any undelivered large messages for that process then sending a small message is delayed until that message has been sent. Similarly if there are any undelivered small messages for that process, sending of large messages is delayed.

## 7.3 Object Oriented Message Streams

The run–time system contains an abstraction of the raw time–stamped message passing system, which is based on the requirements of the object oriented design. This object oriented design includes support for transformations which produce a flatten operation, which allows objects to be communicated to remote processors.

This system is based on the design of the C++ streams abstraction which allows objects to be output on file and output streams. The basics are that a capability for input and output of each basic object type is built. So all of the C++ built–in objects, or basic types, have an output and an input method provided for them in the standard streams library `iostream`. A similar capability is provided to output objects to files, the `fstream` class.

However both of those abstractions provide human readable input and output methods. For objects to be passed to different processors over message streams, a binary method is more efficient.

The problem has similarities with the idea of persistent storage. There are various systems for persistent storage of objects, these are designed to create objects which will, as the name suggests, out live the program which created them [4]. This typically means that they will be transparently stored to disk, and re–created by reading back them from the disk the next time the program is run. For such persistent storage schemes, an associated problem known as object flattening is recognized.

The run–time system contains implementations for the input and output of the basic C++ types over message streams as described below. This includes support for different endian processors via p4 XDR facilities, this is required to enable the use of different endian processors in a heterogeneous network.

### 7.3.1 Input/Output of Message Stream References

Given the run–time library support for heterogeneous input and output, arbitrary user defined C++ objects can be constructed by simple transformation rules. The basics are that an input and an output method should be defined for each object. There are two methods of building objects in C++, by inheritance, and by containment. For either of these methods of construction, an input and an output method for each of the objects being built on will be available according to the transformation rules. They will either be built–in objects in which case their input and output capabilities are defined in the run–time library, or they will be previously defined user objects, in which case they, in turn, will have been constructed in a similar manner: either from other user objects or built-in objects.

Basically all the input and output operators have to do for each contained or inherited from object, is to call the object's input or output operators in some defined order, The order chosen is inherited from objects first, then contained objects by lexical order of inclusion in the object definition.

# Chapter 8

# Time–stamps for Optimistic Execution

## 8.1 Introduction

This chapter presents the particular requirements of time–stamps for the optimistic execution scheme discussed in previous chapters, and describes the time–stamps adopted to meet these requirements. This chapter is fairly self-contained, but ties in with the chapters on the design of the overall system, and with the transformation chapter. Time–stamp allocation is a well defined problem in the field of parallel discrete event simulation, and has simple solutions in this setting. In PDES, time–stamps are allocated corresponding to simulation times of events in the simulation model. There are a number of extra problems which arise when generating time–stamps for events corresponding to the execution of statements in a general purpose programming language.

There are other types of time–stamp allocations schemes which have been proposed, for example the vector and matrix clocks described in [62].

## 8.2 Requirements

Time–stamps must be allocated dynamically as the program runs, this is necessary because it is impossible in general to predict the actual flow of control which will be taken by the program when executed. The halting–problem [57] mandates just this: it is impossible in general to predict the behaviour of a program, and as a result of this it is impossible to tell if the program even halts.

Following on from the dynamic allocation requirement of time–stamps, a second related requirement is that it be possible to allocate an arbitrary number of time–stamps between any two previously allocated time–stamps. The necessity of this requirement can be derived from the halting–problem applied recursively to the sub–program parts which must themselves be time–stamped (such as functions, and procedures). More concretely, consider an example in which a given procedure call is followed by a second procedure call, where the optimistic execution requires that both calls be executed simultaneously. It is not possible to tell how many time–stamps are required for the first procedure at compile–time, and yet the time–stamp for the invocation of the second procedure call must be allocated prior to the termination of the first call.

Another area which mandates the use of arbitrary precision time–stamps is loops with unknown termination. Determining how many iterations of the loop will be required, and hence how many time–stamps to allocate can again not in general be predicted. We may wish to allocate a time–stamp for the instruction following the loop before termination of the loop.

In C/C++ notation an example showing a loop where it is not possible to predict the number of iterations which will be executed, might be:

```
{
    while ( exp )
    {
        stmt1;
    }
    stmt2;
}
```

The assumption is that the expression `exp`, is not a value which can be determined at compile time. There are two basic reasons why this might be the case: complexity, and run–time behaviour directed by input data. The complexity of analyzing programs often means that even for examples where it would be reasonable to expect a human to be able predict the number of iterations, a general automated method of performing this task is difficult. This is because often meta knowledge is required to make the prediction, so a proof by induction, or knowledge of the algorithm being implemented, would be needed. The second problem is that programs are often dependent on data input at run–time. This data may be in the form of a data–file, or obtained by user interaction, in either case this information is not available at compile–time. In summary it is desirable, and in many cases necessary to be able to allocate time–stamps with arbitrary precision to avoid the problem of unpredictable numbers of iterations of control flow constructs.

Referring back to the example, the specific problem is the task of allocating time–stamps for the three executable pieces of code in the example: `exp`, `stmt1`, and `stmt2`, before it is known how many times the loop will iterate. The reason that it may be desirable to be able to time–stamp `stmt3` is that if it could be executed concurrently with the execution of the (as yet undetermined number of) iterations of the loop.

To see how arbitrary precision time–stamps solve the problem, consider marking up the program with time–stamps represented by floating point numbers ranging between 0 and 1, 0 is program start, 1 program end.

The aim is to time–stamp `exp` and `stmt2` prior to termination of the loop. Example time–stamps for `exp` and `stmt2` are shown in Figure 2.

| time–stamp | statement |
|---|---|
| 0.0 | exp |
| 0.5 | stmt2 |

Table 2: Time–stamps using real numbers

Given the time–stamps in Figure 2, any time–stamps which fall with in the loop must be in the range $0.0 < x < 0.5$. So for the first iteration of the loop a time–stamp of 0.25 could be used for `stmt1`, for the `exp` corresponding to the next iteration a time–stamp of 0.375, could be used. This time–stamp allocation scheme allocates progressively smaller intervals, the first few time–stamps would be as shown in Figure 3.

| time–stamp | statement |
|---|---|
| 0.0 | exp |
| 0.25 | stmp1 |
| 0.375 | exp |
| 0.4375 | stmt1 |
| : | |
| 0.5 | stmt2 |

Table 3: Real number allocation

From a mathematical point of view this time–stamp allocation scheme is sufficient as it is demonstrably feasible to generate an arbitrary time–stamp between any consecutive pair of existing time–stamps. The new time–stamp is the average of the two time–stamps, the mid point. If the two time–stamps are `x` and `y` respectively, then to generate a time–stamp z lying between these two time–stamps the created time–stamp would be computed by the function:

$$z = (x + y)/2$$

However whilst correct, this method of allocating time–stamps is impractical. In a binary representation of the number, a bit of precision is consumed for every sub–division of consecutive time–stamps. Using a fixed point binary representation, the cost of allocating the time–stamp can be demonstrated clearly. Consider two consecutive existing time–stamps $x = 0.0$, and $y = 0.5$, represented in fixed point binary:

$$x = 0b$$

$$y = 0.1b$$

$$z = 0.01b$$

Subdividing $x$ and $y$ to get $z$ makes the representation one bit longer. A further subdivision of $x$ and $z$ to get $w$ a time–stamp falling between $x$ and $z$ demonstrates this further:

$$x = 0b$$

$$z = 0.01b$$

$$w = 0.001b$$

Considering the typical applications for which these time–stamps will be required, the cost of one bit per sub–division is unacceptable, a loop with a single statement which iterated for 64k times would result in a time–stamp 16 kilobytes long. The time–stamps described in this chapter meet the arbitrary precision requirements, and improve on the efficiency of the simplistic time–stamp generation scheme described above.

## 8.3 Representation of Time–stamps

The representation of the time–stamps is chosen with consideration of the criteria of providing good efficiency for the operations compare, and generate. It is important that the operation of comparing arbitrary time–stamps, and of generating sequences of time–stamps be efficient, as these are the most commonly used operations on time–stamps. A variable length time–stamp is described in this chapter which satisfies these requirements. The use of multi–level time–stamps to sub–divide the event time–line for the integration of events into active object–oriented databases [51] suggested the use of variable length time–stamps.

We define the variable length time–stamp as a pair $(length, value)$ where *value* is a non–negative binary number of length *length*. 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

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

An ordering relation on time–stamps is defined:

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

$(l_a, v_a) < (l_b, v_b)$

$$\text{iff either: } \begin{cases} f_a < f_b \\ \text{or } f_a = f_b \text{ and } l_a < l_b \end{cases}$$

The time–stamps can be thought of as variable precision binary fractions. The time–stamps can be represented pictorially, in Figure 19 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. The left child of time–stamp $(l, v)$ is $(l+1, 2v)$ and the right child $(l+1, 2v+1)$.

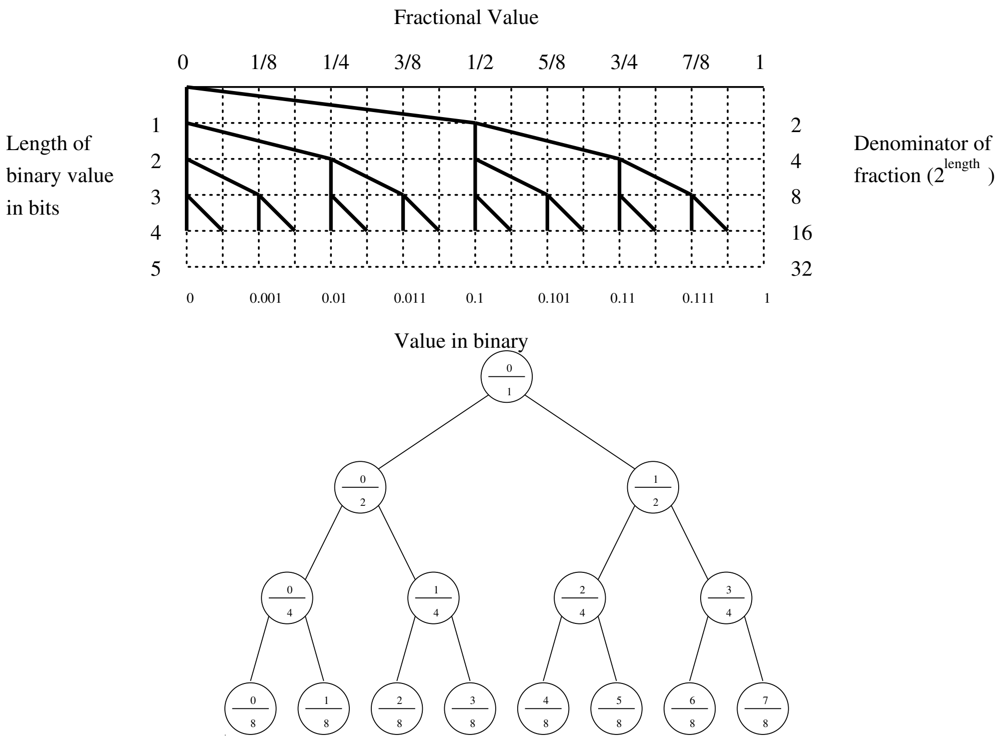


Figure 19: Variable length time–stamps

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:

$$
\begin{aligned}
&(l, v) < \\
&\quad \left\{(l + d, 2^d v) < \ldots < (l + d, 2^d v + 2^d - 1)\right\} \\
&\qquad\qquad\qquad\qquad\qquad < (l, v + 1)
\end{aligned}
$$

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. The time–stamps have the property that for their comparison, trailing zeros in their fractional representation become significant in the ordering, so that

$0.10 \neq 0.1$. Figure 4 shows some example values and their arithmetic, and time–stamp relations.

| *fractional values* | | *arithmetic* | *time–stamp* |
|---|---|---|---|
| 0.0 | 0.1 | $<$ | $<$ |
| 0.0 | 0.00 | $=$ | $<$ |
| 0.10 | 0.1 | $=$ | $>$ |

Table 4: Arithmetic & time–stamp relations

To clarify the ordering on these variable length time–stamps the following Figure 5 shows the complete ordered set of time–stamps up to length 3 bits:

| time–stamps |
|---|
| 0. |
| 0.0 |
| 0.00 |
| 0.000 |
| 0.001 |
| 0.01 |
| 0.010 |
| 0.011 |
| 0.1 |
| 0.10 |
| 0.100 |
| 0.101 |
| 0.11 |
| 0.110 |
| 0.111 |

Table 5: Ordered list of time–stamps up to length 3

## 8.4 Allocating Time–stamps for Control Structures

In this section the allocation of time–stamps for typical program control structures is described. First the naive use is considered, and subsequently more efficient time–stamp generation schemes are presented. As an example of the type of program structure for which time–stamps will be generated, consider:

```
loop A(3)
  loop B(<unbounded>)
    statement C
  end
end
```

The term "unbounded" is used to describe a loop construct where the number of iterations of the construct which will be executed is not predictable. Typically this means that it will not be possible to predict the number of iterations of the loop at compile–time. A simple example is where the terminating condition depends on input from the user at run–time:

```
{
  cin >> limit;

  for (i = 0; i < limit; i++)
  {
    // body
  }
}
```

With this example it is known once the information has been obtained from the user (the value of the variable `limit` has been read), how many iterations of the loop there will be. For examples such as this, it would be possible to envisage situations where advantage could be taken of the fact that the loop termination could be better described as a case of late binding, rather than arising from unpredictable run–time program behaviour.

The more general case is where the terminating expression is simply too complex for the compiler to predict, either due to complexity alone, or a combination of complexity and user input as exemplified by the above example. A contrived example showing this is:

```
{
  for (i = 0; i < exp; i++)
  {
    // body affecting exp
  }
}
```

It is presumed that the code in the body of the `for` loop construct affects the value of the terminating expression in some non–trivial way. Similar examples could be constructed which would show similar problems with the unpredictable nature of other constructs, and with recursion. Control constructs with unpredictable termination will be referred to in the rest of this chapter as “unbounded constructs”. The term “unbounded loops” will be used to refer specifically to loops whose termination is unpredictable.

Returning to the original example of this section, the tree representing the sequential execution ordering of this code block, is shown in Figure 20:

The graphical representation of the code fragment shown in Figure 20 allows the tree to be arbitrarily expanded for the unbounded loop. Arbitrary N–ary trees can be represented by binary trees without loss of information. In Figure 21 the same code block is shown represented as a binary tree rather than an N–ary tree. The representation of N–ary tress as binary trees is a general property, and will work for arbitrarily complex N–ary trees. It is possible to generate time–stamps by mapping the binary tree shown in Figure 21 on to the

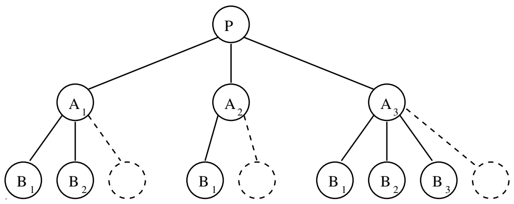


Figure 20: Code block

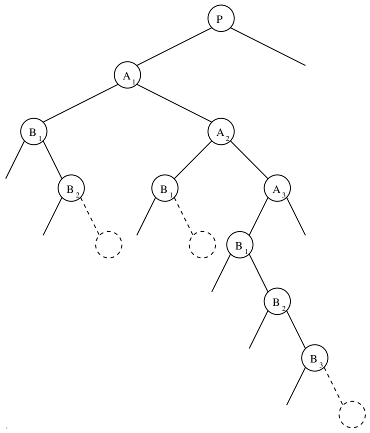

Figure 21: Code block, as binary tree

binary tree representing possible time–stamps shown in Figure 19. All that is needed is to assign the corresponding time–stamp for each node in the tree.

It is possible to use this scheme to assign time–stamps for arbitrary programs containing any number of unbounded constructs. Unfortunately, the efficiency in terms of length of time–stamps required for a given problem is poor. The number of bits required to represent the time–stamp varies as O(n) where n is the number of iterations of a loop being time–stamped. For example, a loop `A(128)` requires a sequence value with 128 bits. This is a costly solution, in terms of bits per time–stamp.

## 8.5 Optimization of Time–stamps

For loops with fixed bounds, and for sequences, it is known that $n$ time–stamps are required, for some fixed $n$. This information can be used to allow use of less bits per time–stamp. To generate $n$ time–stamps for parent time–stamp $(l, v)$ a complete tree rooted at $(l, v)$ with depth $\lceil \log_2 n \rceil$ is created. The $n$ time–stamps are allocated as the leaf nodes in this tree. This allows the number of extra bits required for a loop or sequence of size $n$ to be reduced from $n$ bits to $\lceil \log_2 n \rceil$ bits.

### 8.5.1 Unbounded Loops

For unbounded loops the number of time–stamps to allocate can not be predicted. An initial number of time–stamps $N$ will be allocated initially, where $N = 2^b$, and $b$ is the number of bits which will be used by the initial time–stamp allocation. The first half of these time–stamps will be used for the first $2^{b-1}$ time–stamps. If more than $2^{b-1}$ time–stamps are required complete binary trees of depth $b$ are constructed under all of the remaining $2^{b-1}$ time–stamps. This will create $2^{2b-1}$ additional time–stamps. Again, the first half of these time–stamps ($2^{2b-2}$) will be used first, and the second half reserved for further time–stamps. This process is repeated to produce time–stamps of arbitrary length, and arbitrary precision.

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 $b$. The cost $\mathcal{C}$ of an allocation scheme is defined as the number of bits used to represent time–stamps in that scheme. Also, the efficiency $\mathcal{E}$ of the allocation is defined as the actual number of time–stamps used divided by the total number of time–stamps available for that cost, that is:

$$\mathcal{E} = \frac{n}{2^{\mathcal{C}}}$$

The cost in bits $\mathcal{C}_{\text{static}}$ and efficiency of static allocation $\mathcal{E}_{\text{static}}$ is given by:

$$\mathcal{C}_{\text{static}} = \lceil \log_2 n \rceil$$

$$\mathcal{E}_{\text{static}} = \frac{n}{2^{\mathcal{C}_{\text{static}}}}$$

The cost in bits $\mathcal{C}_{\text{dynamic}}$ and efficiency of dynamic allocation $\mathcal{E}_{\text{dynamic}}$ is given by:

$$\mathcal{C}_{\text{dynamic}} = Lb$$

$$\mathcal{E}_{\text{dynamic}} = \frac{n}{2^{\mathcal{C}_{\text{dynamic}}}}$$

where $L$, the number of levels of dynamic allocation required is:

$$L = \left\lceil \frac{\mathcal{C}_{\text{static}}}{b-1} \right\rceil$$

A balance must be struck in the choice of $b$, too small and the efficiency will be poor for large unbounded loops as many levels will be required and time–stamps will be lost at each level. However, too large a value of $b$ and there will be large numbers of unused time–stamps generated for unbounded loops which happen to have a small bounds. It would be possible to tune the time–stamp generation scheme on a per control construct basis for a particular program. However this information may not be available, and so an intermediate value may in practice be chosen to provide reasonable efficiency for both small and large ranges of time–stamps.

## 8.6 Implementation

On a machine with a word size of $w$ time–stamps (*length*, *value*) are represented as an array of integers, where the first element in the array is the length of the time–stamp in bits.

Time–stamp $t = \{\mathit{length}, a_1, a_2, \ldots, a_n\}$ where $n = \left\lceil \frac{\mathit{length}}{w} \right\rceil$, as $n$ words are required to store a value of length *length*.

An efficient function to can be defined to compare time–stamps. The compare function computes:

$$\{l_a, a_1, a_2, \ldots, a_{n_a}\} = \{l_b, b_1, b_2, \ldots, b_{n_b}\}$$

$$\text{iff } l_a = l_b \text{ and } a_i = b_i \;\; \forall\, i,\; 1 < i < n_a$$

$$\{l_a, a_1, a_2, \ldots, a_{n_a}\} < \{l_b, b_1, b_2, \ldots, b_{n_b}\}$$

$$\text{iff} \begin{cases} l_a = 0 \text{ and } l_b > 0 \\ \text{or } a_1 < b_1 \\ \text{or } a_1 = b_1 \text{ and, recursively} \\ \{l_a - w, a_2, \ldots, a_{n_a}\} < \{l_b - w, b_2, \ldots, b_{n_b}\} \end{cases}$$

### 8.6.1 Time–stamp Generators

A C++ implementation of the variable length time–stamps described in this chapter was written. The implementation provides variable length time–stamps with the described properties, and described machine representation (array of word sized integers). A C++ time–stamp class `TS` is provided, together with a set of overloaded operators operating on objects of class `TS` which provide the standard arithmetic comparison operators: `< , <=, ==, !=, >=, >`

The implementation also provides a pair of time–stamp generating schemes implemented as C++ classes: a Bounded Time–Stamp Generator (class `BTSG`), and an Unbounded Time–Stamp Generator (class `UTSG`). The `BTSG` class provides the functionality required for time–stamping code sections where the number of time–stamps required is known at compile–time. The class prototype provides a constructor, which takes two arguments, the first argument is a time–stamp specifying the start of the range that the time–stamps will fall into, the second argument is an upper limit on the number of time–stamps which will be generated. The time–stamps generated will fall in the range $start < t < start + 1$ where $start$ is the numerator of the first constructor argument, and $t$ represents the time–stamps generated. The second constructor argument determines how many bits will be used. For instance a range requesting $n = 100$ time–stamps will extend the length of the $start$ time–stamp by $\lceil \log_2 n \rceil = 7$ bits.

Time–stamps can be specified individually by the transformation process, this example shows this taking place:

```
{
  BTSG tsg1( start, 3 );

  func1( tsg1( 0 ) /* , args */ );
  func2( tsg1( 1 ) /* , args */ );
  func1( tsg1( 2 ) /* , args */ );
}
```

The example above shows the static allocation of the three time–stamps within the range defined by start. The actual allocation occurs at the time this code block is invoked, and the $start$ time–stamp is defined. The allocation of time–stamps within the range defined by $start$ is determined by the calls to the generator `tsg1()` as shown above. The argument to the time–stamp generator requests the generation of the i–th time–stamp within the range where `i` takes the values 0, 1, and 2 in the above example.

The time–stamp generator argument can be generated for loops with known bounds in a similar manner, this example:

```
BTSG tsg( start, 100 );
for (int i=0; i < 100; i++)
{
  func( tsg( i ) /* , args */ );
}
```

This shows the function `func()` being called with a time–stamp argument. The call to the time–stamp generator is shown as `tsg( i )`, this returns the i–th time–stamp in the range covered by the time–stamp generator.

### 8.6.2 Unbounded Time–stamp Generators

Unbounded time–stamps `UTSG` provide a similar class interface. The first argument to the `UTSG` constructor is as with the `BTSG` class, the $start$ time–stamp. All of the time–stamps generated by the time–stamp generator will again fall into the range defined by $start < t < start + 1$. The second argument to the `UTSG` class is the $bits$ parameter to the dynamic time–stamp generation algorithm described earlier in this chapter. This parameter selects the balance of the bits consumed by the initially allocated time–stamp range, and the bits consumed by each subsequent sub–division when the time–stamps within the currently allocated range are used up. The `UTSG` class provides a single other function: one to increment the generator to the next time–stamp in the range. The `UTSG` generator thus always allocates time–stamps in sequence, whereas with the `BTSG` generator, time–stamps can be allocated out of sequence.

Consider the following example, which shows the use of a `UTSG` generator in combination with a `BTSG generator`. The `BTSG` generator is used to time–stamp areas of the code which have predictable number of iterations. The time–stamps for the sequence within which the unbounded loop occurs is thus allocated by a `BTSG` generator, as is the sequence within the unbounded loop. The unbounded loop itself is time–stamped by a `UTSG` generator:

```
{
  BTSG tsg1( start, 10 );
  func1( tsg( 0 ) /* , args */ );
  UTSG tsg2( tsg( 1 ), 8 );
  for ( int i = 0; i < exp; i++ )
  {
    BTSG tsg3( tsg2(), 5 );
    func2( tsg3( 0 ) /* , args */ );
    :
    func3( tsg3( 4 ) /* , args */ );
    ++tsg2;
  }
  :
}
```

In the above example, generator `tsg1` has 10 time–stamps in its range, generator `tsg2` has an unbounded number of time–stamps, and generator `tsg3` has 5 time–stamps. Each time–stamp generator is initialized with a time–stamp generated by the previous; this means that all of the time–stamps in the generated range fall within a subrange of the previous range. The relationship between the three time–stamp generators is shown graphically in Figure 22.

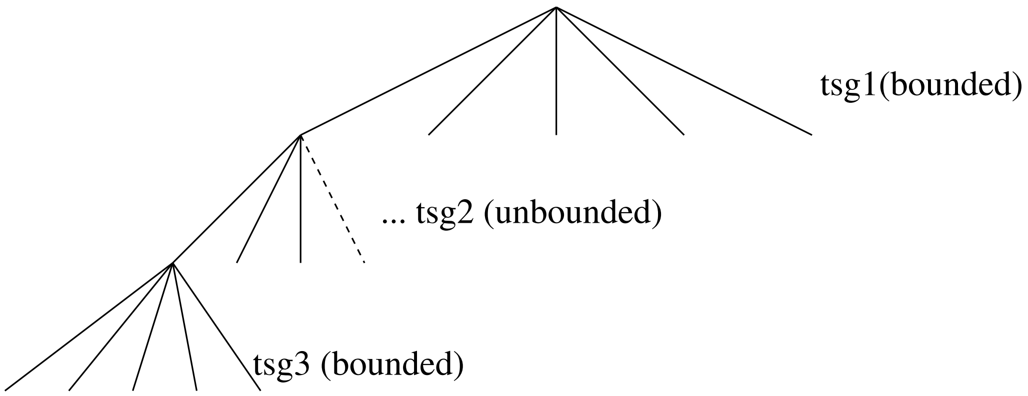


Figure 22: Time–stamp generators

### 8.6.3 Recursion

Recursion is another area where unbounded time–stamp generators must be used if it is not known how deep the recursion will nest. It may be possible to place a limit on the recursion by knowledge of the machine resources — the depth of recursion will be limited by physical or virtual memory on the machine. A more general solution which removes the need to estimate the maximum level of recursion is to use an unbounded time–stamp generator to generate the time–stamp for each level of recursion. A bounded time–stamp generator would be used to generate the time–stamps for the body of the recursively called procedure. Consider a simple head–recursive procedure:

```
int recur( /* params */ )
{
  if ( exp1 )
  {
    recur( /* args */ );
  }
  /* body */
  return exp2;
}
```

This could be time–stamped naively with bounded time–stamp generators by using a bounded time–stamp generator to generate time–stamps for each level of the recursion. The above code using `BTSG` generators only would look stylistically like this:

```
int recur( TS ts /* , params */ )
{
  BTSG tsg( ts, 10 );
  if ( exp1 )
  {
    recur( tsg( 1 ) /* , args */ )
  }
  /* body */
  return exp2;
}
```

The problem with the solution using only `BTSG` generators is that the number of bits consumed for recursion of depth $n$ varies as $O(n)$. For moderate depths of recursion, the time–stamps would reach impractical lengths. The time–stamps generated at each depth, presuming a start time–stamp of 0, would be of the form:

| recur–depth | time–stamp |
|---|---|
| 0 | 0.0000 |
| 1 | 0.00000000 |
| 2 | 0.000000000000 |
| 3 | 0.0000000000000000 |
| : | : |

A better solution is to use a `UTSG` generator for the recursion. The resulting transformation stylistically would look like:

```
int recur( UTSG utsg /* , params */ )
{
  UTSG tsg( utsg(), bits );
  if ( exp1 )
  {
    recur( ++utsg /* , args */ );
  }
  /* body */
  return exp2;
}
```

Presuming that the `UTSG` passed in to the top–level invocation of the recursive procedure `recur()` was initialized with its $bits$ parameter set to 8, this would mean that 128 levels of recursion would be available before the length of the time–stamp would increase. The time–stamps would be of the form:

| recur–depth | time–stamp |
|---|---|
| 0 | $0.00000000_{xxxx}$ |
| 1 | $0.00000001_{xxxx}$ |
| : | : |
| 127 | $0.01111111_{xxxx}$ |
| 128 | $0.1000000000000000_{xxxx}$ |
| 129 | $0.1000000000000001_{xxxx}$ |
| : | : |
| 16383 | $0.1011111111111111_{xxxx}$ |
| 16384 | $0.110000000000000000000000_{xxxx}$ |
| 16385 | $0.110000000000000000000001_{xxxx}$ |
| : | : |

For many programs, it will only be necessary to use time–stamps with $length \leq w$ where w is the machine word size. To give an idea of the complexity of program which can be represented with these time–stamps before an extended length time–stamp is required, consider:

```
loop <fixed>
  recur(A)
  end
end
```

With a choice of $b = 6$ for dynamic loop allocation, and assuming a word size $w = 32$, the recursion could reach depth 1,000 in a fixed loop of 1,000,000 before an extended time–stamp would be required.

With double length time–stamps more complex structures are possible:

```
loop <fixed>
  recur(A)
    loop <unbounded>
    end
  end
end
```

For a fixed loop of 1,000,000 and recursion of 30,000 the unbounded loop could reach iteration 1,000,000 before it would be required to extend it further.

# Chapter 9

# Evaluation

This chapter presents an example program involving matrices which a conventional parallelizing compiler would not be able to parallelize. Timings are then provided for a number of aspects of the parallelizing compiler run–time system, and compared with the primary operation in the example (matrix multiplication). The purpose is to demonstrate that it should be feasible to obtain speedup using this parallelizing approach, where conventional compilers would be unable to provide speedup.

## 9.1 Example

The example code fragment shown in Figure 23 is based on a set of operations on an array of matrices. There are 100 matrices in array `w`, and each matrix is a square matrix with bounds of `DIM`. The granularity of the matrix operations is determined by these bounds. (Matrix<DIM,DIM> is C++ template notation indicating that a templated matrix class is being used).

The example would be difficult to parallelize using conventional parallelizing compiler techniques because each iteration possibly depends on the output of the previous, and

```
{
    Matrix<DIM,DIM> w[ 100 ];
    ...
    for ( int i = 1; i < 100; i++ ) {
        if ( func( w[ i - 1 ] ) )
            w[ i ] = w[ i - 1 ] * w[ i ];
        w[ i ] = w[ i ] * w[ i ];
    }
}
```

Figure 23: Example where data flow analysis would fail

because the existence of that dependency is determined by a function which itself depends on the previous iteration.

## 9.2 Overhead Estimates

The following subsections present an estimate of the run–time system overheads incurred by optimistically executing the example programs including that shown in Figure 23. These overheads are compared with the compute time as the granularity of the matrix multiplication varies.

The categories for which estimates of overheads are gathered are:

1. using replacement types for built–in objects;

2. adding state–saving to replacement types;

3. adding time–stamps as well as state–saving to replacement types;

4. state–save overhead for larger objects;

5. time–stamp comparison;

6. bounded time–stamp generation;

7. unbounded time–stamp generation;

8. p4 messaging system.

The analysis of the overhead of using replacement types is to isolate the overhead incurred just as a result of this type of operation. The state–save overhead for larger objects is particularly important as this is a large run–time overhead. The time–stamp generation, comparison overheads are included, as they are part of the run–time system overhead. Some experiments were also carried out to evaluate the performance of the p4 message passing system.

## 9.3 Replacement Types

These results were obtained with the GNU g++ compiler (version 2.6.0), on a 100Mhz R4600 based SGI Indy under IRIX 5.2, with `-O3` selected. Inlines were disabled with `-fno-inline` because the GNU compiler otherwise automatically inlines small functions even if there is no explicit `inline` keyword marking that function as inline.

The transputer network was not used because the available compilers were unable to cope with the template functions used heavily in the transformation system.

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 [64] encrypting a single block (of size 8 bytes) using a 128 bit key. The number of rounds has been increased to 2048 over the recommended 12 to increase the accuracy of the timings. The application was chosen for its intensive use of bit–wise integer operations.

Operations on `int`s were selected to demonstrate the worst case performance of the

| Replace | Inlines | References | time (ms) | time / operation | additional times slower |
|---|---|---|---|---|---|
| | | | 33 | 16.5 ns | - |
| √ | √ | √ | 73 | 36.5 ns | 2.2 x |
| √ | √ | | 83 | 41.5 ns | 1.3 x |
| √ | | √ | 375 | 187 ns | 4.5 x |
| √ | | | 677 | 338 ns | 2.1 x |

Table 6: Overhead of replacement types: count1 (2000003 useful operations)

| | calls w. no refs | calls w. refs |
|---|---|---|
| copy constructor | 2000002 | 0 |
| less than | 1000001 | 1000001 |
| pre–increment operator | 1000000 | 1000000 |
| construct with int | 2 | 2 |
| total operations | 4000005 | 2000003 |

Table 7: Tally of replacement class calls: count1 example

state–saving system were the lowest compute to state–saving rate is likely. RC5 was chosen as an example of an algorithm which uses bit manipulations on `ints` intensively.

| Replace | Inlines | References | time (ms) | time / operation | additional times slower |
|---|---|---|---|---|---|
| | | | 19 | 22.3 ns | - |
| √ | √ | √ | 26 | 30.6 ns | 1.4 x |
| √ | √ | | 28 | 32.9 ns | 1.1 x |
| √ | | √ | 162 | 191 ns | 5.8 x |
| √ | | | 204 | 240 ns | 1.3 x |

Table 8: Overhead of replacement types: rc5 (850338 useful operations)

The first set of timings are for using a replacement type for integers. No other overheads are included, so there is for instance no state–saving of the replacement integers. This

| class operation | calls w. no refs | calls w. refs |
|---|---|---|
| copy constructor | 344232 | 0 |
| assignment operator | 61484 | 61484 |
| default constructor | 14 | 14 |
| construct with int | 391352 | 391352 |
| bit–wise and | 40972 | 40972 |
| divide | 1 | 1 |
| test greater–than | 1 | 1 |
| bit–wise left shift | 32780 | 32780 |
| test less-than | 4098 | 4098 |
| test less–than–or–equal | 4099 | 4099 |
| minus | 24585 | 24585 |
| minus–equals | 4098 | 4098 |
| multiply | 3 | 3 |
| test not–equal | 12295 | 12295 |
| bit–wise or | 32780 | 32780 |
| plus | 129090 | 129090 |
| plus–equals | 2050 | 2050 |
| pre increment | 45076 | 45076 |
| mod–equals | 24588 | 24588 |
| bit–wise right shift | 32780 | 32780 |
| bit–wise xor | 4096 | 4096 |
| bit–wise xor–equals | 4096 | 4096 |
| total operations | 1194570 | 850338 |

Table 9: Tally of replacement class calls: rc5 example

experiment is just to demonstrate the approximate overhead of using replacement types.

The first row of Tables 6 and 8 are the timings for the unmodified code, without replacement types. The remaining rows all show timings for replacing integers in the two applications. The options are whether or not the calls are inlined, and whether or not the replacement types took reference parameters. Using reference parameters avoids the use of copy constructors, as the parameters to each operator call are accessed directly from the class. This is the difference in performance between pass by value, and pass by reference. Pass by reference is faster because the overhead of copying parameters is avoided. Tables 7, and 9 show a tally of the replacement integer operations called. The effect of using references to avoid the overhead of copy constructors can be seen in the third columns of Tables 7, and 9 where using references in both cases reduces the number of copy constructor calls to 0.

For the timings where calls of replacement integer operations are inlined, the loss of efficiency arises from the loss of knowledge about common sub–expression elimination, the loss of the normal arithmetic rules of associativity, and commutativity. The compiler does not know that the replacement types obey normal arithmetic rules, indeed it would be perfectly feasible, and legal, to replace the operators with functions which did not obey normal arithmetic rules. Also it is likely that compilers would not do a perfect job of inlining, so some efficiency may be lost in this area also.

For the timings where calls of replacement operations are not inlined the additional loss of efficiency arises from the extra overhead of function calls. From Tables 6 and 8 it can be seen that the function call overhead is by far the largest factor in the loss of efficiency.

The exact loss of efficiency will depend heavily on the mix of operations used by the application, on the architecture of the machine used, and on the compiler used. The efficiency losses will depend on the relative costs of the built–in operations compared to function call overheads. The loss of efficiency from lost optimization opportunities will also depend on the effectiveness of the compiler's optimization algorithms.

It is expected that some of the losses could be recovered by using a special purpose compiler, as the code could be optimized for the specific purpose, rather than relying on the compiler's general purpose optimizer. Also the loss of knowledge of arithmetic relationships between operators could be remedied, enabling common sub–expression elimination, and other optimizations.

The subsequent use of the replacement operators will be likely to have a performance below row two of the Tables 6 and 8. This is because even though inlining can be done in the body of the server–object main loop, and any arguments can be passed by reference, in practice it is expected that the overhead of the copy constructor invocations avoided by the use of references will be replaced by the overhead of decoding and then creating an object to reference from the RPC message stream. It is likely for this reason that the performance will be below that shown in row 3 of Tables 6 and 8. Also it is expected that the effect of the inlining will not be as marked in the server–object main loop as in normal code because of the effects of avoiding jumps, and hence pipeline flushes and cache misses. For this reason it is expected that the performance will be similar to, or possibly better than, that shown in row 5 of the Tables 6 and 8. Following on from this discussion of likely performance, the performance timings in the next section will be based on the same assumptions as row 5 of Tables 6 and 8. This is the most conservative estimate in any case.

### 9.3.1 Adding State–saving

In this section the additional overhead of collating rollback information for replacement built–in types is evaluated, and compared to the overhead purely from using replacement types.

The state–saving costs are independent of the rate of rollback, except in the sense that new state must be saved when re–executing events which are rolled back. State–saving costs are related to the number of states saved and the sizes of those states.

The figures in Table 10 show timings using the standard allocation operator `operator new()` for state–saving, and no deallocation. As stated at the end of the last section, the tables shown in this section (starting with Table 10) are based on no inlining, and no use of references, for both timings with replacement types only, and for timings including state–saving.

| program | replacement | additional from state–saving | combined overhead |
|---|---|---|---|
| count1 | 322 ns | 9.14 us | 9.46 us |
| rc5 | 218 ns | 7.07 us | 7.29 us |

Table 10: Overheads compared, useful operations: replacement types vs state–saving

| program | saves | save overhead | overhead per save |
|---|---|---|---|
| count1 | 7000008 | 18.2 s | 2.60 us |
| rc5 | 2034667 | 6.00 s | 2.95 us |

Table 11: State save overhead per save

There is a distinction made with the state–saving system between reads, writes and updates. Each saved state is marked as one of these three. A read is where the object's value is read, but not modified, a write is where the object's value is written, but not read, and an update is where the object's value is first read and then written. These actions are denoted 'R', 'W', and 'U' internally representing read, write, and update respectively. As examples: the value of `b` is *read* ('R' state–save) in the expression `a = b`, the value of `a` is *written* ('W' state–save) in the same expression (ie the value of `a` is not read first, it is simply over–written), and the value `c` is *updated* ('U' state–save) in the expression `c+=d` (this is because the resulting value of `c` depends on `c`'s previous value, and hence `c` must be read first, and then written).

The figures in Table 13 and Table 14 show timings using a replacement operator `operator new()` for state saving, and no deallocation. As before, timings are based on no inlining, and no use of references, for both timings with replacement types only, and for timings including state–saving. The improvement is due to a special purpose memory allocation algorithm.

| program | R saves | W saves | U saves | total |
|---|---|---|---|---|
| count1 | 4000004 | 2000004 | 1000000 | 7000008 |
| rc5 | 34832 | 1157677 | 842158 | 2034667 |

Table 12: Number of state–save operations

The special purpose memory allocation algorithm is able to out perform the general purpose memory allocators because it is tailored to the application. The approach is to allocate memory in buckets, and to allocate from a bucket until it is used up. When the bucket is consumed a new bucket is allocated. Deallocation is also simplified by keeping count of the number of allocated areas with in a bucket, allocation increases that number, and deallocation decreases it, when the number reaches zero, the bucket is deallocated. This removes fragmentation, and the associated performance degradation over time.

| program | replacement | additional from state–saving | combined overhead |
|---|---|---|---|
| count1 | 322 ns | 3.96 us | 4.28 us |
| rc5 | 218 ns | 2.94 us | 3.16 us |

Table 13: Overheads of replacement types vs state–saving

| program | saves | save overhead | overhead per save |
|---|---|---|---|
| count1 | 7000008 | 7.92 s | 1.13 us |
| rc5 | 2034667 | 2.50 s | 1.23 us |

Table 14: State save overhead per save

### 9.3.2 Adding Time–stamps to Rollback Overhead

| program | replacement | additional state–saving | additional time–stamp | combined overhead |
|---|---|---|---|---|
| count1 | 322 ns | 3.96 us | 768 ns | 5.05 us |
| rc5 | 218 ns | 2.94 us | 602 ns | 3.76 us |

Table 15: Overheads replacement types vs state–saving

Finally, the additional overhead for storing a time–stamp in the objects state is shown in Table 15.

## 9.4 State–save Overhead for Larger Objects

This experiment is designed to show the approximate overhead of saving the state of larger objects, and to show how this overhead varies with object size. The example investigated is that of state saving a square matrix with rows and columns varying from 10 to 100. The elements of the matrix are double precision floating point numbers which occupy 8 bytes each. The smallest matrix with 10 rows, contains 100 doubles, and the largest with 100 rows contains 10,000 doubles.

Figure 24 shows how the state saving overhead varies with the size in bytes of the matrix, and Figure 25 shows how it varies with the size (as measured in rows or columns). The data is presented in this format so that it may be more easily compared against the matrix multiplication figures in the next section. As expected, state–save overhead is proportional to the size of the object, and hence to the square of the number of rows and columns of the matrix.

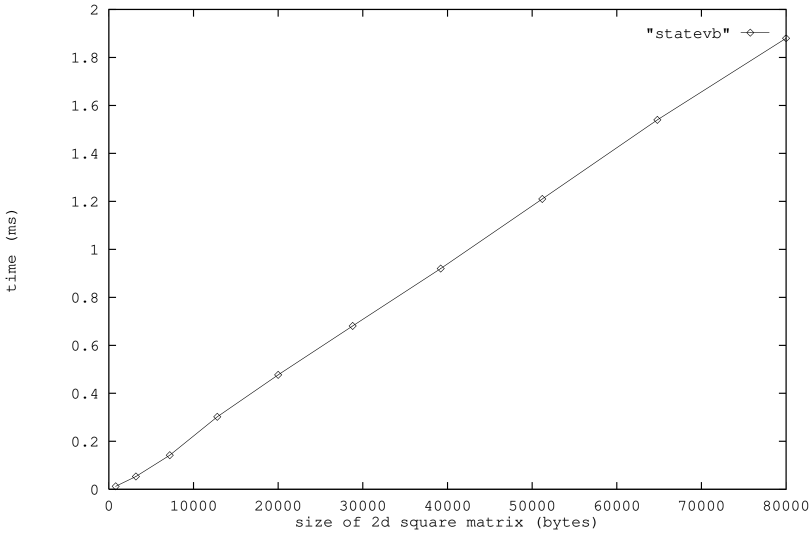


Figure 24: State–save overhead vs size of object in bytes

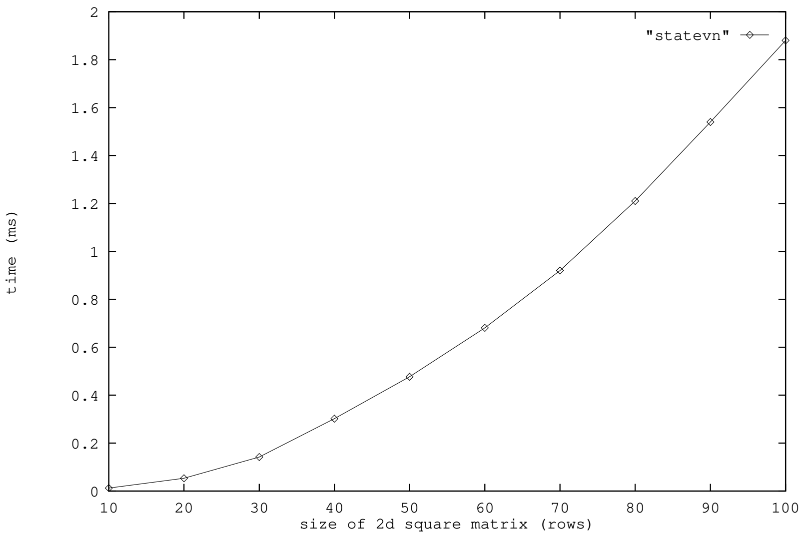


Figure 25: State–save overhead vs matrix size (rows)

## 9.5 Matrix Multiplication Granularity Measurement

The granularity of the task of multiplying square matrices of sizes varying from 10 x 10, to 100 x 100 are calculated to allow the granularity for this task to be compared against state saving overhead.

Figure 26 shows the overhead of multiplying square matrices as compared to the number of rows, or columns. The most simplistic matrix multiplication method is used, and the complexity, as expected is $O(n^3)$ in the number of rows or columns of the matrix.

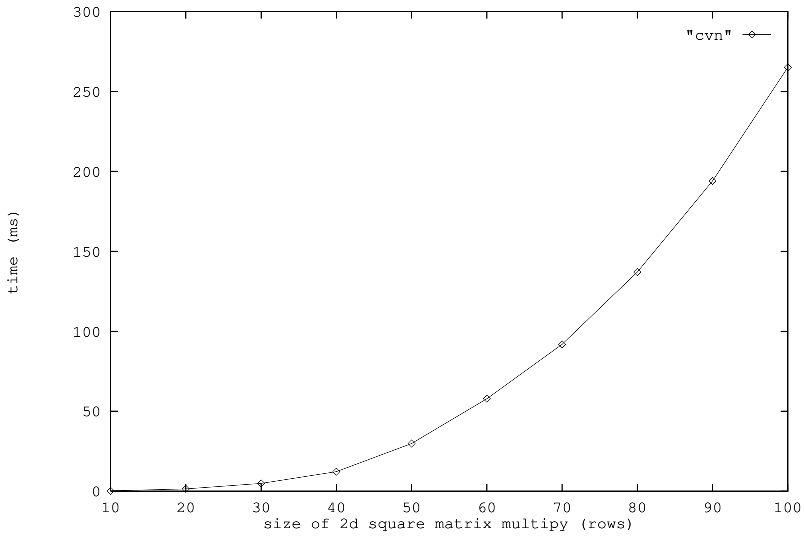


Figure 26: Multiply operation vs matrix size (rows)

Figure 27 shows an estimate of the percentage of CPU load that would be attributable to state–saving for the example of matrices. As the complexity of matrix multiplication rises faster than that of state–saving, the efficiency improves for larger matrices.

## 9.6 Time–stamps

Figure 28 shows the overhead of comparing time–stamps, which increases with length of the time–stamp in bits. The time–taken increases in steps as an extra word is consumed for the representation of the time–stamp. Time–stamps of different length in bits, but with the

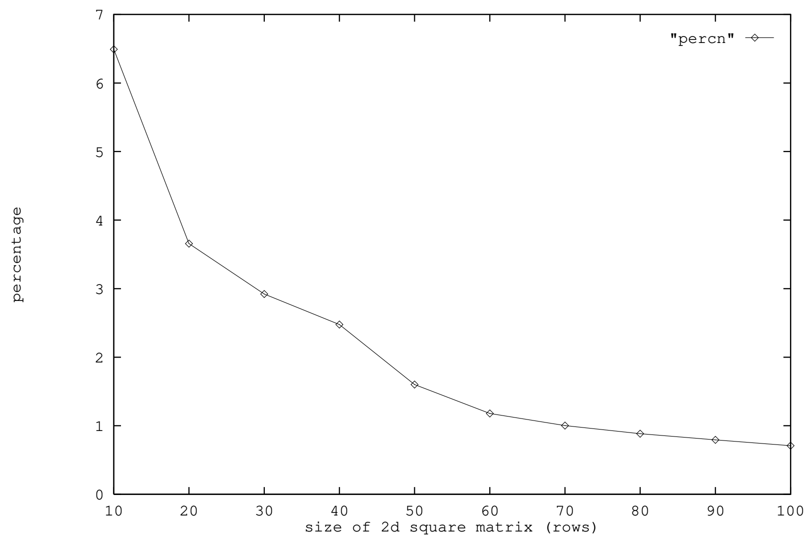


Figure 27: Percentage of CPU consumed by state save vs matrix size (cols)

same number of words in their representation take the same amount of time to compare.

### 9.6.1 Time–stamp Generation

Figure 29 shows the overhead of time–stamp generation for ranges which fall within time–stamps of differing length. The overall trend is that the amount of overhead increases with length of time–stamp, but the overhead function is spiky because the implementation relies on bit shifting, and masking operations and the precise method used varies as the range of the time–stamps falls at the beginning of a word boundary in the representation of the time–stamp. The generation function is also different when the range overlaps a word boundary to when it lies entirely within one word.

Figure 30 shows the overhead of time–stamp generation for unbounded ranges of time–stamps. These time–stamps grow in length as time–stamps are allocated, as described in Chapter 8. The time–stamps must be generated in sequential order, unlike the time–stamps

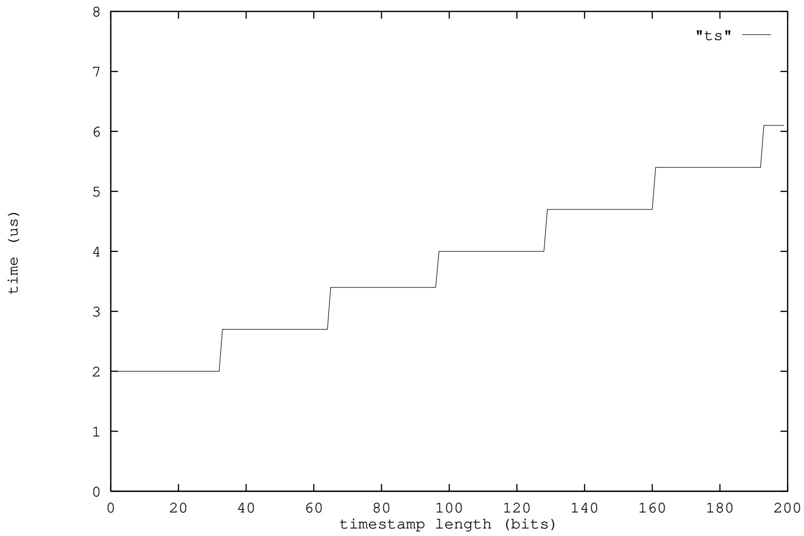


Figure 28: Time–stamp comparison: inlined and non–inlined

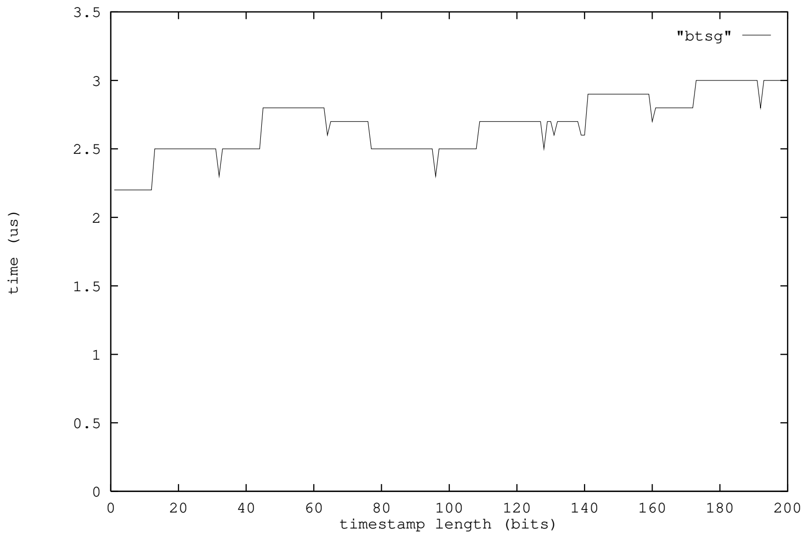


Figure 29: Time–stamp generation: bounded

generated by the bounded time–stamp generators, and the operation is just to increment an arbitrary long integer, with checks for when the current range of time–stamps are all consumed. When the current range is consumed, the length of the time–stamp is extended, and time–stamps are allocated from the new larger range.

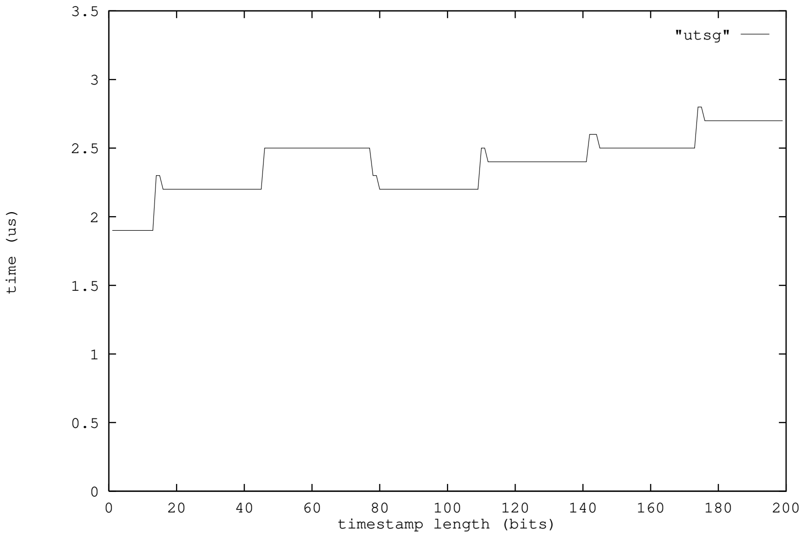


Figure 30: Time–stamp generation: unbounded

## 9.7 P4 Messaging System

The p4 figures presented here are those for a T800 transputer network connected with a C004 programmable switching unit. These figures are felt to be more useful than workstation figures, because they show typical communication latencies on a real parallel architecture as opposed to the high latency performance resultant from ethernet connected workstations.

The p4 implementation on transputer networks is described in Chapter 7. The communications overhead of p4 is reasonable, and provides additional functionality over the existing Inmos virtual channel routing software. It is felt that using the p4, or the newer

MPI to obtain portability will not prove to be a serious overhead.

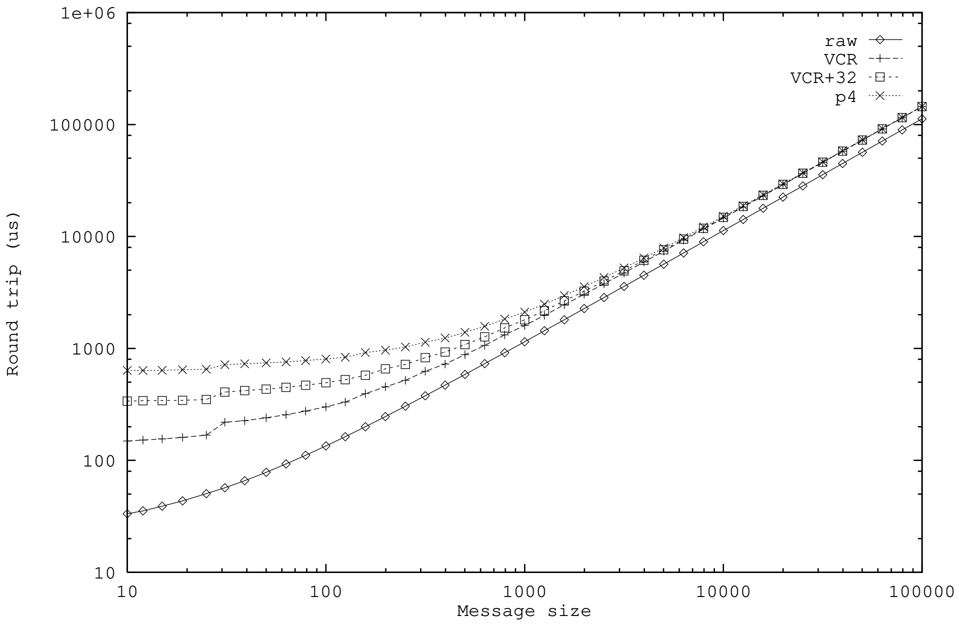


Figure 31: Ping 2 nodes: round trip time

Figure 31 demonstrates the efficiency of the p4 system as compared to the basic Inmos C communications library using virtual channels.

The lines labeled “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 channel. “VCR+32” shows the performance of variable sized messages where a 32 byte header is sent as a separate message before the actual message. This is the size of header used by p4 and would in p4 contain information about the message size, type, senders p4 process id, whether XDR is to be used etc. This is shown for comparison with the actual p4 communication performance labeled “p4” as this is the overhead p4 incurs for variable size packets alone, the extra overhead is due to demultiplexor and multiplexor buffer management and the provision of typed messages.

Figure 32 shows the corresponding bandwidth for p4, VCR, VCR+32 and raw communication.

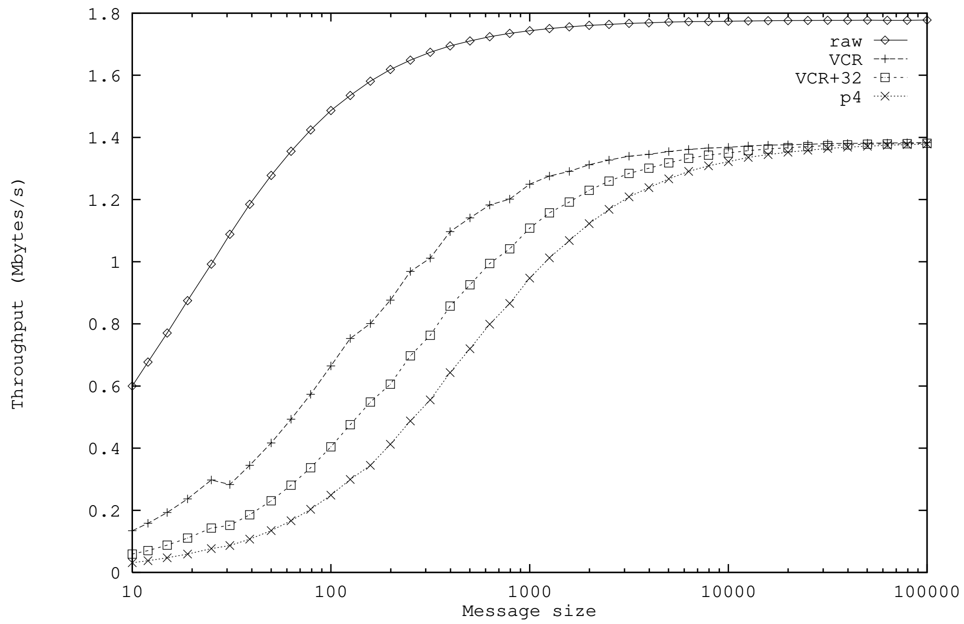


Figure 32: Ping 2 nodes: throughput

## 9.8 Summary

The cut off point for efficiency in any particular example will depend on the granularity of the program objects chosen as server objects, on the state saving overheads (estimated above per byte of object), the rate of causality violations, and the rollback overhead, and the load balance of the system.

The example presented in this chapter, together with the timings, indicates that with suitable choice of matrix, it should be possible to reduce overheads sufficiently to produce speed up in situations where a conventional parallelizing compiler would fail to do so.

# Chapter 10

# Further Work and Conclusions

This chapter presents future research areas which could be explored in connection with the optimistic parallelizing system described in this thesis, and presents the conclusions to this research.

## 10.1 Empirical Studies

The work in the thesis has been presented largely on a qualitative basis. One area which should be explored further is system performance using a large sample of existing applications and algorithms. A systematic approach to this would facilitate discovering actual performance for typical problems.

The class of problems the system is designed to exploit are those which are not typically handled well by conventional parallelizing compilers. A class of programs that it would be useful to evaluate the system with would be programs exhibiting irregular behaviour due to having run–time dependencies. This would allow a comparison of speed up attained for such problems as compared to conventional parallelizing compilers.

The rate of rollback is highly program and data dependent. The rate of rollback is the determining factor in the performance of the system. Extensive experimentation with sets of example programs would be required to enable quantitative measurements of rates of rollback typical for programs implementing different classes of algorithms. Throttling is one technique used to reduce problems of excessive rollbacks, and experiments could also be conducted on the effect of different throttling algorithms on the rollback rate.

The optimistic approach, because it relies on rollback, and the associated state saving and rollback overheads, may perform less efficiently than a conventional parallelizing compiler for programs where conventional compilers can do a good job of extracting parallelism. Programs which fall into these categories will be those which are amenable to compile time data flow analysis.

No GVT algorithm has been implemented in the experimental work discussed in Chapter 9. GVT algorithms are available in the literature [26] and we have selected the GVT algorithm described in [65] because it is based solely on asynchronous messages, and because it reduces the messaging overhead by sending only time–stamps in cancellation messages.

## 10.2 Heuristics

There are several areas of the system which currently rely on programmer–provided hints in the form of code annotations. The long term goal of providing a fully automatic system requires this information to be derived automatically by the parallelizing compiler system. Due to the nature of the task the information gathering techniques will take the form of heuristic algorithms.

The design includes a framework for heuristic components to be built–in to the system. It is expected that the heuristic system is given the source code to analyze, and that the heuristic system will produce a set of recommendations about the source code which can be used to drive the transformations. The information derived will be passed to the transformation system in the form of annotations, in a similar manner to the interaction between the programmer and the transformation system through manually coded annotations in the partially automatic transformation system.

The information required by the transformation system to make a fully automatic parallelizing system is the following:

- server object selection
- object placement
- object state–saving characteristics

Heuristics for each of these categories will be discussed in the following sections.

### 10.2.1 Server Object Selection

It is necessary to select objects for the transformation system to transform into server–objects. The selection process must attempt to take into account the granularity of the objects. Selecting objects with coarse grain methods will improve the efficiency of the system, by providing the best work to rollback and system overhead ratio. Also, the selection should take into account the state–saving characteristics of the object, objects which generate small amounts of state–saving information for relatively large amounts of computational work are desirable, those with higher state–saving demands are less desirable.

The heuristic process of automatically selecting objects with these properties based only on compile–time information is difficult. Examples of the sorts of metrics which are available at compile–time are the number of lines in the object's member functions, the objects place in the class hierarchy, and recursively, the estimated granularity of functions, and other objects used, and contained by that object. Complexity metrics can add estimates of costs weighted for loops – for loops whose bounds can be predicted at compile–time this can be reasonably accurate, whereas for loops with unpredictable bounds some experimentally determined average value will be the best that can be done.

It is expected that the object's location in the class hierarchy will on average prove a useful indicator of complexity. In situations where there are a limited number of objects, and there is a multiple layered class hierarchy, the objects which are instances of the highest level classes will tend to be higher granularity than the lower level objects. The class hierarchy for the purposes of this kind of evaluation also needs to consider objects containing other objects, as the operations on the contained objects will add to the computational complexity of the containing object.

The state–saving characteristics of the object also affect the suitability of the selection of a given object as a server object. The heuristic estimation of state–saving overhead, and selection of state–saving characteristics to use to direct the transformation system to build the most space efficient state–saving mechanism on a per object basis is described in the section below on state–saving heuristics. The overhead metric resulting from estimation of state–saving overhead for objects may be useful in deciding the selection of server objects. The observation that there is a balance which must be achieved between state–saving overhead, and effective work done by an object, and that this balance should maximize work done, and minimize state–saving overhead suggests that the estimated state–saving behaviour of objects should be taken into account in the server object selection heuristics. This means that it may not always be optimal to select the object with the highest granularity as a server object, if for instance its state saving characteristics are poor, and would result in too high a state–saving overhead.

Another approach to determining estimates for object complexity, is to perform profiling runs — run the program as a sequential program with profiling turned on to determine statistics about the program experimentally [41, 67] The profiling figures would then be fed into the parallelization stage. This technique is becoming more frequently used for optimizing compilers, allowing, for example, better defaults to be chosen for branches.

### 10.2.2 Object Placement

The server objects are distributed over the processors of the parallel machine. The placement decisions will affect the performance of the optimistic execution for a number of reasons. If the parallel architecture has a notion of locality for communications overhead, placement should take into account the frequency and volume of communication between each pair of processes in the system. Objects which communicate frequently should if possible be placed on the same, or on close processors. The placement of the processes on the processors of the parallel machine is known as static load–balancing, because the processes are placed on nodes at start up time, and the placement decisions are made at compile–time. This is the start up placement: because dynamic server–object creation is catered for in the system, it may also be necessary to place objects at run–time.

Both the placement of program level server–objects which are visible at compile–time, and the placement of server–objects which are created during the execution of the program are expected to be controlled by annotations in the initial system. The optimal placement of processes in a parallel system is a separate research issue.

Another approach to the load–balancing problem is to provide a dynamic load balancing system [63], with a load function associated with each processor indicating when processes should be migrated to other less loaded processors. This can allow for more even load–balancing: as some programs will have characteristics which vary at run–time, dynamic load–balancing will tend to even the load across the machines, and to work to rectify changes in program behaviour which may occur in different program phases.

Dynamic load balancing however has its own drawbacks, the first being that maintaining the load estimate for each processor itself consumes processing time. The other problem is that the concept of load is not as well defined in an optimistic execution as in a conservative simulation system, or in a system based on explicit parallelism. Actual processor load, in terms of how busy the processor is may be a useful metric in some situations, but in others, the processors may always be busy, as there may be enough optimistic execution to saturate all of the processors. Saturation does not necessarily mean that the best distribution of work has been achieved because there needs to be a concept of *effective work* [63], and a weighted measurement of work depending on how far into the future it is being optimistically executed. Effective work is work that is committed, as opposed to processor load generated by rollbacks. The measure of how loaded a processor is should include a weighted discounting of work which is progressing too far into the future, as the further into the future the task is being optimistically executed, the higher the probability that it will be rolled back. The concept of work weighted by distance into the future is a concept that is used in the "throttle" optimization. With a "throttle", the aim is to penalize work which is too optimistic, by lowering its priority, so that it progresses more slowly or not at all, until the rest of the simulation has caught up. Also a commonly used approach is to have a scheduling algorithm which executes events with the lowest virtual time first (at least locally).

There is a problem in even measuring how far in the future a task is with the time–stamping scheme used in this system (see Chapter 8 for a description of the time–stamp generation scheme used with the variable length time–stamps). Because the time–stamps do not increase uniformly, the absolute difference between GVT and Local Virtual Time

(LVT) may be a poor indicator of distance into the future. Research attempting to provide a dynamic load balancing scheme would need to consider the problem of measurement of load, and of measurement of degree of optimism.

### 10.2.3 Object State–saving Characteristics

The hybrid state–saving scheme is based on incremental state–saving. For base objects (replacement objects for built–in objects), there is only one option, that the object's complete state is saved after each operation. For objects which are constructed by containing, or by being derived from state–saving objects, or other composite objects, there are two options in the state–saving method used for the containing objects. The options are either that the whole object is saved after every operation (this is the same as the state–saving method for built–in objects), or that the state–saving is delegated to the contained objects. The process of selecting one or the other containment and state–saving method can be arbitrarily nested.

In the partially automatic system, the decisions about whether objects are state–saved by contained state–saving objects, or by performing their own state–saving is delegated to the programmer to select by means of annotations. For a fully automatic system these decisions must be made by heuristics. The criteria for making the decision is based on the expected state–saving characteristics of the object. The state–saving characteristics are determined by the expected number of invocations, and hence state–save operations, of the member functions of contained and derived objects. There will be a trade off where if there are a sufficient number of invocations, it will become cheaper in terms of state–saving overhead for the containing object to save its whole state after each of its own operations rather than relying on the incremental state–saves of the contained objects. Knowledge of the size of the objects involved is also required for the calculations.

The task of the heuristic system is then to estimate the number of invocations, and to combine this information with source derived knowledge of the object size to arrive at a recommendation for the appropriate method of composing the object. The sorts of metrics used to arrive at an estimate for the number of invocations will be based on similar metrics to those described in the section on server object selection above, where techniques for estimating member function granularity are discussed.

### 10.2.4 Investigating and Evaluating Heuristics

Apart from the suggestions given above, which are based on intuition, the actual parameters and heuristic cost functions must be arrived at via experimentation. A series of experiments with a representative set of C++ applications is a suggested approach for obtaining evaluations of cost functions, and average parameters about programs, via profiling and other metrics of the programs. Heuristic cost functions can be tested using this test environment.

## 10.3 Combining Data–flow Analysis Information

One area which has not been addressed in this thesis is the use of data–flow information obtained from compile time analysis in combination with the optimistic execution system. The implementation work for the prototype system has been based on a pure time–warp based system. Such a system is general in that it able to produce an optimistic execution which is semantically indistinguishable from the original sequential execution.

The time–warp system relies on causality violation detection, and on rolling back when violations are detected. A pure time–warp system is able to produce a semantically correction execution, but situations can occur where actions are taken by a pure–time warp system which will always rollback, and some of these situations would be predictable with information derived from compile–time data–flow analysis.

There are a number of research issues involved in this area, they are:

- interprocedural data–flow analysis
- combining this data–flow analysis information with an optimistic execution

The first problem, interprocedural data–flow analysis, is known to be a hard problem [49]. The complexity of the task can be reduced by using an object oriented language which enforces data encapsulation, or having equivalent user stipulations for efficient operation in languages where there is no syntactic enforcement of encapsulation.

The complexity of performing data–flow analysis can be reduced by using enforced data–encapsulation. With data–encapsulation the interfaces between objects are restricted to be through procedure calls only. The resulting data flow analysis is still complex because references to contained objects can in effect be used to allow access to encapsulated data. However data–flow analysis is simpler when encapsulation is used because accesses to non local data are explicitly expressed within the program, and the access restrictions are compiler enforced. When data–encapsulation is used programs are more modular, and typically less use is made of global data, and typically fewer external accesses to internal class data are made.

The second research issue is in combining compile time data–flow analysis information with the pure time–warp system. There are two types of information which can result from data flow analysis: information which shows that there will never be a rollback in a given situation, and information which shows there will always be a rollback if the construct is executed optimistically. For the situations where it is known that there will be no rollback, this information can be taken advantage of by not state–saving in that area. Situations where there will always be a rollback, can be avoided by embodying the synchronization requirement into the code so that the dependency will never be optimistically executed. This is advantageous because it can reduce the rollback rate and hence improve efficiency.

## 10.4 Conclusions

This thesis has laid down the groundwork in showing that the approach of parallelizing sequential programs through optimistic execution is feasible, and has demonstrated that it should be possible to obtain speed up for programs where conventional parallelizing compilers would fail.

The parts of the system design which are novel are the use of optimistic execution based on object oriented programming, and the transformations necessary to produce a remote procedure call implementation which uses asynchronous messages only. The time–stamps and time–stamp generation schemes which satisfy the requirements of allocating an arbitrary number of time–stamps between existing pairs of time–stamps are another novel aspect of the system. The time–stamp scheme's performance and characteristics may be useful for other areas of research such as distributed databases.

The hybrid state–saving system is also new, and should allow for improved state–saving performance over either purely incremental or purely periodic state–saving mechanisms, as the state–saving performance characteristics can be tuned per object.

The system guarantees a parallel execution which is semantically indistinguishable from the original sequential execution. The system as currently presented in this thesis is based partly on annotations, but a framework for a fully automated system has been laid down, and an approach for experimenting with heuristic algorithms to fully automate the previous sections of this chapter.

The example, and overhead estimates presented in Chapter 9 show that the system is feasible, and predict that the system should be capable of providing improved speedup in situations which are normally considered unsuitable for parallelization by existing parallelizing compilers which are based solely on compile time data–flow analysis.

# Appendix A

# C++ Specific Transformations

C++ has a number of language features which result in implicit invocation of user supplied functions in certain situations. It is necessary to first understand which functions are being called implicitly, because they may need to be further transformed. The first stage of the transformation system is to make all implicit calls explicit.

## A.1 Explicify Transformation

In C++ it is quite often possible for calls to built–in and user supplied functions to be added by the compiler to fulfill particular functions, even though no code is included by the programmer to call the function in the particular code fragment. Such function calls are called "implicit function calls", because the user knows, or expects, that these functions will be called to achieve a particular syntax, or particular type compatibility, even though no code is explicitly given by the applications programmer. Implicit function calls are used in two areas of the language: to provide syntactic sugar to improve the degree of syntactic control the programmer has in constructing user defined types, and as a mechanism for controlling type compatibility with user defined types. Both purposes are designed to improve the elegance of use of user defined types, to the point where their use can be equivalently elegant to the built–in types.

An example of syntactic sugar is the use of functions which can be given names corresponding to various arithmetic operators, and the same precedence, and associativity rules. This allows for the creation of things like user defined matrix classes with normal notation for multiplication. This form of implicit function call is known as operator overloading, and will be described in more detail below. Use of this language feature can allow the user to create a matrix class which could be used as shown here:

```
{
  matrix A,B,C;

  A = B * C;
}
```

Another example of the use of implicit function calls is to allow type compatibility between user defined types and built–in types, and between user defined types and other user defined types. The following code would for instance allow the creation of a user type `bigint`, which allows arbitrary precision arithmetic, and yet is still type compatible with the built in scalar types. The language feature which allows this is the creation of user defined conversion operators, and the use of constructors for the same. These language features, which will be described in more detail below, allow code such as the following:

```
{
  bigint b;
  int i;

  b = pow(i,100) % 53;
  i = b;
}
```

The example shows a user defined type `bigint` which allows arbitrary precision arithmetic being used in expressions intermixed with the built–in type `int`, the variable `i` in the example.

It is necessary to replace implicit function calls with explicit calls before the rest of the transformations can be done. This is because the implicit functions themselves may need to be transformed, and this would otherwise be noticed by the rest of the transformation system as no explicit code is included in the actual code.

The purpose of the "explicify" transformation is to take an existing C++ program and convert it into an equivalent program with no implicit assumptions. Fortunately C++ has equivalent explicit syntax for most of these language features. This transformation has been logically separated from the rest of the transformations, as it is relatively self contained, and this transformation is done first, as a first pass of the transformation system.

First operator overloading will be described in more detail, and as a prelude to this, overloaded functions will be described. Overloaded functions are made explicit by a process known as name mangling. This process is already carried out by most C++ compilers, to allow the continued use of older linkers which do not handle type information. This mechanism is used to allow type safe linkage, as mangled functions, and variable names will not match for purely lexical reasons if they are of different type, this is used to emulate type safe linkage. In this way, the linkers used are not required to be aware of type information.

Overloaded functions are presented first, as they are related to overloaded operators, and hence described here as an introduction to overloaded operators. Overloaded operators are a specialization of overloaded functions, they are overloaded functions with unusual names, and with special additional rules for their use, and syntactic rules which mean that they can be used in their implicit form.

Other implicit features in C++ are:

- user–supplied conversion operators
- constructor calls used as conversions
- compiler generated temporaries for user classes
- virtual method invocation

## A.2 Overloaded Functions

Function overloading allows more than one function to have the same name provided that there are a different number of arguments to each of the overloaded functions or the arguments are of different types. The correct overloaded function can be selected by examining the number and types of the arguments when one of the set of overloaded functions is called.

For example the following two functions are resolvable because they have different numbers of arguments:

```
int example( int );
int example( int, int );
```

So the resolution of which functions to call in the following code fragment:

```
{
  example(1);
  example(1,2);
}
```

is done by examining the number and types of the arguments. The first call is to function `example( int )`, and the second is to `example( int, int )`. The two functions with name `example` are said to be overloaded because they have the same name, the function name is *overloaded*.

Another example, this one with differing types: the following two functions have the same *number* of arguments, but the arguments are of different type:

```
int example2( int );
int example2( float );
```

An example code fragment using the overloaded functions `example2()` is:

```
{
  example2( 1 );
  example2( 3.14 );
}
```

The first call resolves to `example2( int )`, and the second call resolves to `example2( float )`. The differing types of the arguments allows the compiler to determine which function to call in each case.

Return values are not counted in the process of resolving overloaded operators, so the two functions shown below would be considered to have the same signature, and hence cause a compile error.

```
int example3( int );
float example3( int );
```

If the compiler did accept both function prototypes, there would be an ambiguity error when a function of this name was called. For example this call:

```
{
  int i = example3( 2 );
}
```

would be considered to be ambiguous as without taking into account the return type, the function prototypes are equivalent. The reason that C++ does not consider return types as part of a function's signature is that there are other complications to do with built–in conversions, and user conversions which would make the overall resolution process overly complex if return values were considered. Built–in and user conversions will be described further below.

Overloaded functions are made explicit, by name mangling the functions name, so that it depends on the type and number of arguments. An example transformation of the above two examples (excluding the incorrect `example3`) with the name mangling transformation would be:

```
{
  example1I(1);
  example1II(1,2);
  example2I(1);
  example2F(3.14);
}
```

That is integer arguments are represented by an 'I', and floating point arguments by a 'F'. Multiple arguments are represented by multiple uses of these single letter codes. The mechanism used here is exactly analogous to that used for the name mangling used in type safe linkage. The names will be re–mangled by the C++ compiler used to compile the transformed code, there is no need to use the same mangling convention here that would be used by the compiler. Indeed it is better that a different mangling scheme be used to avoid clashes between the two confusing the compiler's link phase.

## A.3 Overloaded Operators

Operators are just functions with unusual symbols for names, and particular restrictions on the number of arguments, and special rules for invocation. The term "overloaded operator" is used because all of the operators for which the user can provide operator functions already have at least one function, that of the built–in functionality, of arithmetic, logical, or bitwise operations on built–in types. The operators are overloaded in this sense as the compiler must apply similar rules about deciding which overloaded operator was intended from the context, as are applied for conventional overloaded functions.

The language feature of operator overloading is provided to improve the self extensibility of C++. C++ is designed to be extensible, in the sense that the user is able to create user classes with similar expressive power to built–in types. Operator overloading allows functions to be given names corresponding to all of the built–in operators, such as simple arithmetic operators: '`+`','`-`','`*`','`/`','`%`' for addition, subtraction, multiplication, division, and modulus. Some of the symbols for built–in operators are multiply overloaded already, so '`-`', and '`+`' also represent unary minus, and unary '`+`', the '`*`' symbol is overloaded to mean both multiplication and pointer dereference.

### Built–in Operators

Operators which can be overloaded are restricted to a subset of the built–in operators. The built in operators can *only* be used with what would be implicit syntax for user defined overloaded operators. This code is legal, for integers:

```
{
  int i,j;
  i = j + k;
}
```

but this code would not be:

```
{
  int i,j;
  i = operator+( i, j );
}
```

Some common C++ operators, their signature, and semantics are:

<table>
<tr><th>operator</th><th>result type</th><th>use</th><th>name</th></tr>
<tr><td>+</td><td>scalar</td><td>scalar + scalar</td><td>addition</td></tr>
<tr><td>−</td><td>scalar</td><td>scalar − scalar</td><td>subtraction</td></tr>
<tr><td>*</td><td>scalar</td><td>scalar * scalar</td><td>multiplication</td></tr>
<tr><td>/</td><td>scalar</td><td>scalar / scalar</td><td>division</td></tr>
<tr><td>%</td><td>integer</td><td>integer % integer</td><td>modulus</td></tr>
<tr><td>&</td><td>integer</td><td>integer & integer</td><td>bitwise-an</td></tr>
<tr><td>|</td><td>integer</td><td>integer | integer</td><td>bitwise-or</td></tr>
<tr><td>ˆ</td><td>integer</td><td>integer ˆ integer</td><td>bitwise-xor</td></tr>
<tr><td><<</td><td>integer</td><td>integer << integer</td><td>shift-left</td></tr>
<tr><td>>></td><td>integer</td><td>integer >> integer</td><td>shift-right</td></tr>
</table>

The assignment operator can be combined with the above operators to produce an operation combined with assignment. The operator '+=' is related to '+', for example this code:

```
{
  int i;
  i += 5;
}
```

(for all built–in types) has equivalent semantics to:

```
{
  int i;
  i = i + 5;
}
```

Similarly, this syntax applies to the other simple operators listed above, to give this set of operations:

| operator | result type | use | name |
|---|---|---|---|
| += | scalar& | scalar& += scalar | plus-equals |
| −= | scalar& | scalar& −= scalar | minus-equals |
| *= | scalar& | scalar& *= scalar | multiply-equals |
| /= | scalar& | scalar& /= scalar | divide-equals |
| %= | integer& | integer& %= integer | modulus-equals |
| &= | integer& | integer& &= integer | bitwise-and-equals |
| \|= | integer& | integer& \|= integer | bitwise-or-equals |
| ^= | integer& | integer& ^= integer | bitwise-xor-equals |
| <<= | integer& | integer& <<= integer | shift-left-equals |
| >>= | integer& | integer& >>= integer | shift-right-equals |

An additional bitwise operator is bitwise negation:

| operator | result type | use | name |
|---|---|---|---|
| ~ | integer | ~integer | bitwise negation |

There are relational operators for scalar values:

| operator | result type | use | name |
|---|---|---|---|
| < | bool | scalar < scalar | less-than |
| <= | bool | scalar <= scalar | less-than-or-equal |
| == | bool | scalar == scalar | equal |
| != | bool | scalar != scalar | not-equal |
| >= | bool | scalar >= scalar | greater-than-or-equal |
| > | bool | scalar > scalar | greater-than |

and logical operators:

<table>
<tr><th>operator</th><th>result type</th><th>use</th><th>name</th></tr>
<tr><td>&&</td><td>bool</td><td>bool && bool</td><td>logical-and</td></tr>
<tr><td>||</td><td>bool</td><td>bool || bool</td><td>logical-or</td></tr>
<tr><td>!</td><td>bool</td><td>!bool</td><td>logical-not</td></tr>
</table>

In addition there are a number of unary operators:

| operator | result type | use | name |
|---|---|---|---|
| − | scalar | −scalar | unary-minus |
| + | scalar | +scalar | unary-plus |
| * | object | *object-ptr | pointer-dereference |
| . | object | struct.field | field-select |
| -> | object | struct-ptr->field | deref-and-field-select |

and some more unusual operators:

| operator | result type | use | name |
|---|---|---|---|
| , | any | any , any | comma operator |
| ? | any | scalar ? any : any | question mark operator |
| ++ | scalar | scalar++ | post-increment |
| -- | scalar | scalar-- | post-decrement |
| ++ | scalar | ++scalar | pre-increment |
| -- | scalar | --scalar | pre-decrement |

### User Overloadable Operators

Of the built–in operators described above all are user overloadable except for the '.' field-select operator, which is in any case a special operator, it is field selection within a structure, and hence the 'value' on the right hand side of the '.' operator is a lexical field value. Here is an example of how an operator is overloaded. To illustrate a typical mathematical example of operator overloading, the example is for overloading `operator*` to implement matrix multiplication for a user defined matrix class.

First a matrix class is defined:

```
class matrix
{
protected:
  double values[20][20];

public:
// operations...
};
```

The example matrix class has space for a 20x20 matrix, the comment indicates that in practice there would be some operations declared as member functions.

Next, an overloaded `operator*` operating on matrices would be written like this:

```
matrix operator*(matrix& a,matrix& b)
{
  // implementation...
}
```

This matrix multiplication function, and `matrix` class would allow code such as the following to be written using the overloaded operator explicitly, like this:

```
{
  matrix A, B;
  A = operator*( A, B );
}
```

Or, this code which results in an implicit call of the matrix `operator*`.

```
{
  matrix A, B;
  A = A * B;
}
```

The compiler can determine the operator to use by looking at the types of the operands. As they are matrices, and an overloaded `operator*` is available, the `operator*` for the user defined type `matrix` will be chosen. To make this explicit it is also necessary for the transformation system to use name mangling, as there can be several overloaded operators with the same name. For example if the programmer also defined a `vector` class, and a multiply operation for vectors, the notation:

```
A = operator*(A, B);
```

would be ambiguous as it would not be known whether the function called was matrix multiply, or vector multiply. It is not possible to directly mangle the operator's name "operator*" as there are specific syntactical rules which govern the naming of operator functions. They must be formed by the reserved word "operator" with one of the built in operator symbols concatenated to this word.

To mangle overloaded operators, the function is first renamed to have an ordinary textual name. So the above example becomes:

```
A = op_mul(A, B);
```

Once this is done the name can be mangled to specify which of the overloaded operators was intended. For matrices, the name would become:

```
A = op_mul_matrix_matrix(A, B);
```

to indicate that the operator takes two matrices as arguments.

The renaming of the operators is required in any case, for the adding of time–stamp arguments, a further transformation specified in chapter 4. It is a restriction of overloaded operators that they can only have a legal number of arguments as defined by the built–in arithmetic operation.

So, if the more explicit version of the operator call is used, the following would be illegal code:

```
A = operator*(time_stamp,A,B);
```

The presumed reason for this restriction is that there would be no equivalent implicit notation, and hence little to gain by allowing this notation in terms of notational convenience.

### Member Functions

In the object oriented framework there are member functions, and there are ordinary functions. The distinction is that member functions must be applied to a particular instance of the class for which the member function is defined, and in effect has a first parameter provided in an unusual syntax. Ordinary functions have all their arguments passed as arguments.

An example to clarify this, first an ordinary function taking as arguments two integers:

```
int func1(int,int);
```

which is called as follows:

```
int x = func1(1,2);
```

Next an ordinary function, but taking a user defined class as arguments:

```
{
  T A,B;

  func2(A,B);
}
```

where `T` is a user defined class.

Lastly member functions, which have by definition to be of user defined classes, as there are no built–in types with member functions (although there are some standard library classes). Consider the class `D`, with a single member function `mfunc1`:

```
class D
{
public:
  void mfunc1(D);
};
```

To call this member function, an object which is an instance of that class is required:

```
{
  D A,B;

  A.mfunc1(B);
}
```

That is, the member function is invoked with a similar syntax to that used for field selection in C structures. It is additionally possible to specify the class the member function is called from with the call. This is more explicit than just the code:

```
A.mfunc1(B);
```

as it is not known which class the member function is being called on without determining the type of the object (`A` in the example).

The more explicit syntax is:

```
A.D::mfunc1(B);
```

this makes explicit the type of object `A`.

It is also possible to have overloaded operators as member functions. So the following is a member function equivalent to the matrix multiplication operator used as an example above:

```
class matrix
{
 //...
public:
  matrix operator*(matrix&);
};
```

The explicit call of this operator looks like this:

```
{
  matrix A,B,C;

  A = B.operator*(C);
}
```

as opposed to the explicit notation for the non–member function multiply operator above:

```
{
  matrix A,B,C;

  A = operator*(B,C);
}
```

The implicit use of the the member function overloaded operator is the same as for the non–member function operator, namely:

```
A = B * C;
```

There is one difference between member and non–member function operators, and that is that no user–conversions can be applied to the left hand side of a member function operator. This means that if user conversions are being used, it would be more typical for a non–member overloaded operator to be used.

The same explicit notation as shown above for member functions can, in general, be used for overloaded operator member functions, applying this to the above function yields:

```
A = B.matrix::operator*(C);
```

Again, as with non–member functions, it is necessary to name mangle the overloaded operators (by first renaming) to make the call fully explicit. That is, it is fully possible, and indeed often useful to have an two overloaded operators of the same operator in the same class with different arguments. To give a concrete example, consider the situation with a matrix class, and a vector class which are interoperable, that is matrices can be multiplied by matrices, vectors, and built–in scalars. In the class definition for a matrix, the `operator*=` is an overloaded member function (it is realistic to have the `*=` operator as a member function, thought not as realistic to have `operator*` as an operator).

```
class matrix
{
  //...
public:
  matrix& operator*=(double);
  matrix& operator*=(vector&);
  matrix& operator*=(matrix&);
};

matrix operator*(double,matrix&);
matrix operator*(vector&,matrix&);
matrix operator*(matrix&,matrix&);
matrix operator*(matrix&,vector&);
matrix operator*(matrix&,double);
```

The combinatorial explosion of overloaded `operator*` operators, is due to the fact that for full flexibility all of the combinations must be catered for. It is possible to reduce this effect with the use of user conversions which would allow, say, the automatic expansion of a vector to a diagonal matrix with 0 valued elements on all but the identity diagonal, or expansion of a scalar to a multiple of the identity matrix. However for efficiency reasons, it is quite common to see an expansion as shown above, the expansion of code results in more code, and larger executables, but the extra work is done by the compiler at compile time, and so efficiency is retained.

The above example matrix operations would be translated to the following:

```
class matrix
{
public:
  matrix& op_mul_D(double);
  matrix& op_mul_vector(vector&);
  matrix& op_mul_matrix(matrix&);
};

matrix op_mul_D_matrix(double,matrix&);
matrix op_mul_vector_matrix(vector&,matrix&);
matrix op_mul_matrix_matrix(matrix&,matrix&);
matrix op_mul_matrix_vector(matrix&,vector&);
matrix op_mul_matrix_D(matrix&,double);
```

### A.3.1 Conversion Operators

A user supplied conversion operator allows the user to provide type conversions, typically between user defined classes and built–in types, but potentially between different user defined classes. These conversions will be applied appropriately by the compiler.

For example consider a user defined class `T` and a conversion operator back to the built–in type `int`:

```
class T
{
// ...
public:
  operator int ();
};

T::operator int ()
{
  // ...
}

{
  T t;
  int i;

  i = t;
}
```

This involves the compiler applying the conversion operator implicitly as class `T` and `int` are otherwise not type–compatible.

There are several other situations in which a conversion operator can be applied automatically by the compiler. The implicit conversion above is to allow assignment to a built–in type. The purpose of providing a conversion operator from a user defined class to a built–in type is to make it interchangeable with that built–in type. It can be assigned to the converted–to built–in type, and it can be used as an argument to a function expecting a built–in type. The other situations will be detailed briefly below.

The explicit transformation of the above code just shows the conversion call explicitly:

```
i = t.operator int();    // explicit user conversion to int
```

It is also possible to explicitly specify which class the member function is a member of in the conversion operator call, like this:

```
i = t.T::operator int();
```

This is the most explicit syntax for a conversion operator, and removes all ambiguity, as there can only be one conversion operator for a particular type for any given class.

### A.3.2 Constructors as Conversions

The compiler will also use a constructor as a conversion operator if an appropriate constructor exists. If we have a constructor with a single argument, like this:

```
class T
{
// ...
public:
        T(int);
};
```

a situation arises where a conversion is required but no conversion operator exists, a suitable constructor will be considered instead. In the following example the constructor of class `T` which has a single argument of type `int` will be used implicitly by the compiler as a conversion from `int` to `T`.

```
{
  T t;
  int i;

  t = i;
}
```

The above code involves an implicit construction of a temporary variable, followed by an assignment from the temporary.

The conversion can be shown explicitly like this:

```
  t = T(i);
```

This is legal code, and it does make explicit the fact that there is a conversion between the type of `i` and that of `T`. But the method of conversion is still implicit, as it is not possible to tell without examining the context whether the conversion is done by a constructor, or by a conversion operator. This syntax can be used to represent either use of a conversion operator, or conversion by use of a constructor.

It is possible, though more difficult, to make situations such as this explicit by making the temporary that the compiler will be using to do the conversion explicit.

The following would be the explicit version of the use of the constructor above as a conversion operator.

```
{
  T compiler_tmp(i);
  t = compiler_tmp;
}
```

However there is an additional complication, in that in the C++ standard it was only recently decided when the compiler generated temporaries should be destroyed, and hence their destructors, if any, called. Prior to this, the behavior was undefined, this state of affairs was problematic because if the destructors are called too early, undefined behavior can result. Also when they are called at different points with different compilers, if the destructors have side–effects, the undefined behavior could potentially cause the program to behave differently with different compilers.

There were three main points at which compilers called destructors prior to the time before compilers were updated to follow the standard, and prior to the time when the standard was decided. The points of destruction were:

1. after the immediate expression for which the temporary had been created (the earliest point)

2. or at the end of the largest containing expression (somewhere in between the other two)

3. at the end of the containing block (this was the latest point)

The standard destruction point was defined to be not before the largest containing expression. This makes compilers implementing 2, and 3 conforming, and compilers which implement 1 non–conforming.

The end of the largest containing expression is the earliest point at which the temporary can be safely destructed while still keeping the temporary's value around long enough to keep the containing expressions semantics well defined. It is desirable to have the temporary destroyed as soon as possible, as this minimizes memory consumption, and for larger objects this could be an important consideration.

The semantics required for compliance with the standard for handling of compiler temporaries are satisfied by the example transformation shown above in the example involving the use of a constructor as a conversion operator with a single assignment. However the requirement for the temporaries to be constructed part way through the expression for more complex expressions would mean that the expression would have to be split up, and the explicify transformation would become complex. It is not possible to just declare all of the constructed temporaries prior to the expression, and destruct afterwards, because the order of the constructor calls as compared to the rest of the expression must be preserved in cases where either parts of the expression, or the construction of the temporary have side effects. That is to say the point at which the temporary is constructed can be significant.

Fortunately, this whole issue can be avoided by adopting a convention for the denotation of explicit constructor conversions. As was made clear above, a conversion operator can be expressed in two semantically equivalent ways with different amounts of explicitness.

For the classes `T1` and `T`:

```
class T1
{
// methods...
  operator T2();
};

class T2
{
// methods...
};
```

The conversion operator `operator T2()` which can be used to convert from objects of type `T1` to objects of type `T2`, this code fragment demonstrates implicit use of the conversion operator:

```
{
  T1 t1;
  T2 t2;

  t2 = t1;
}
```

the use of the conversion operator can be expressed like this:

```
t2 = T2( t1 );
```

however, this is not fully explicit, as if class `T2` had a constructor with a single argument of type `T1`, the above syntax could equivalently represent a conversion by use of a constructor.

A fully explicit notation for the conversion operator is:

```
t2 = t1.T1::operator T2();
```

It should therefore be clear that for conversion operators, the notation of:

```
obj1 = type( obj2 );
```

will never be used as the explicit transformation of any conversion operator. As it has no other use, this notation can be reserved to mean that a constructor was used. This allows much more straight forward transformations. The expression can be transformed in place without the use of an explicit compiler temporary. In general, an assignment:

```
obj1 = obj2;
```

with a conversion using a constructor, would look like this:

```
obj1 = type( obj2 );
```

and with a conversion operator, would look like this:

```
obj1 = obj2.operator obj1();
```

A more complex expression is presented here to show an example which would cause a problem for the transformation using an explicitly constructed temporary.

Consider classes `X`, and `Y`, and values `x1`, and `y1` and `y2` of those types respectively, and a function `f1` which expects a single argument of type `X`, and returns a value of type `Y`, so that its prototype is:

```
Y f1( X );
```

Also presume that there is a conversion operator from `Y` to `X`: `operator X()`. Consider the expression:

```
{
  X x1;
  Y y1, y2;

  y1 = f1( y1 ), f1( y2 );
};
```

Note that the semantics of the comma operator are that the left expression is evaluated first, and its value discarded, then the right expression is evaluated and its value returned.

Using an explicitly constructed compiler temporary this would look like this:

```
{
  X x1;
  Y y1, y2;

  {
    X compiler_tmp1( y1 );
    f1( compiler_tmp1 );
    X compiler_tmp2( y2 );
    y1 = f1( compiler_tmp2 );
  }
}
```

This transformation is much different, and not easily achievable in an automated way. The comma operator is no longer even in the expression.

Using the convention described above, a much more straight forward transformation can be achieved:

```
{
  X x1;
  Y y1, y2;

  y1 = f1( X( y1 ) ), f1( X( y2 ) );
}
```

## A.4 Virtual Methods

Another more advanced language feature is virtual methods. These also result in implicit code selection, this time at run–time, this is a form of late binding, and the purpose of this is to allow specialization of classes, where behavior of existing methods can be modified by replacing called functions without modifying the code.

An example of the use of a virtual member function, a class `A`, with member functions `mfunc1`, and a virtual member function `vfunc1`:

```
class A
{
// ...
public:
  mfunc1()
  {
    // ...
    vfunc1();
    // ...
  }
  virtual vfunc1() { cout << "A" << endl; }
};
```

As shown in the example the virtual member function `vfunc1` is called by `mfunc1`. When the class `A` is derived from to produce a class `B` which is a specialization of `A`, if a new implementation is supplied for virtual function `vfunc1`:

```
class B : public A
{
//...
public:
  virtual vfunc1() { cout << "B" << endl; }
};
```

then the function `mfunc1` if invoked on an object which is an instance of class `B` will call the new `vfunc1` even though `mfunc1` has not been specialized in class `B`.

That is the following code would result in "B" being printed out:

```
{
  B b;

  b.mfunc1();
}
```

This piece of code would also result in "B" being printed out:

```
{
  A& ap = *new B();

  a.mfunc1();
}
```

This is perhaps more surprising, as `a` is declared to be a reference to an object of type `A`. However, because of the late binding it is the actual type of the object which counts rather than the declared type.

It is not necessary to predict which of the incarnations of a virtual function is called (and indeed impossible, as the type of the object can be decided at run time). However, it is necessary to be able to flatten the object, and send it as a message. It is necessary to be able to send the object's state as a message; part of the object's state is the type of the object. Therefore, it is necessary to be able to obtain the information as to what a given object's type is, in order to know which of the late bound virtual methods needs to be called if the methods of the object must be called on a remote processor.

The selection of the appropriate virtual function is typically implemented with a hidden field which holds the derivation number, which is used to index into an array of pointers to member functions.

In the new C++ standard language features have been added which allow the type of an object to be determined. However the interface to these functions only allows type comparison, so this method would require interrogating the derivation against the required type. The type interrogation would take O(n) in the depth of the derivation.

It is quite possible to implement virtual functions using C++ non–virtual functions. The calls of the virtual functions `vfunc1` are handled manually by replacing calls to that function to go through a virtual method table, `vtable`, which is an array of pointers to member functions, which is initialized with the versions of the virtual method for all of the classes in the derivation tree. The constructors for the classes have an additional parameter added which is the derivation depth. The derivation depth is stored in the field `derivation`, and this is used to index the Virtual Method Table (VMT) for calls to virtual functions.

This implementation is very closely analogous to the mechanism used by most compilers to implement virtual methods. The overhead of compiler generated virtual method tables is avoided, and replaced with the manually implemented virtual method tables. As the virtual method tables, and invocation is implemented in the same overall way the only difference in efficiency caused will be the fact that the compiler will not know that the manually created VMT is a VMT, and hence may miss some optimizations which are possible on virtual functions, and VMTs.

The use of an explicit derivation field means that the transformation system has access to it, and hence it can easily be included in the information which is sent with an object when it is sent as a message.

### A.4.1 Compiler Temporaries

There are expressions involving user defined types which force the compiler to generate temporary variables of the appropriate type. This is an example of an action being performed implicitly by the compiler.

Consider what happens with the user function multiply, `mul()`, on matrices. (It would be more elegant to use an overloaded `operator*`, but it is simpler, and more general for purposes of this discussion to talk about the function as `mul()`).

```
Matrix mul(Matrix& A, Matrix& B)
{
  Matrix res;

  // do calculation res = A * B

  return res;
}

...

{
  Matrix A,B,C;

  A = mul(B,C);
}
```

A temporary will be generated to hold the result of the `mul() function call`. It might seem in this case that a temporary is not necessary because the compiler could put the result in variable A. However the language semantics require that the above code assigns the result of `A * B`, so the above code is equivalent to:

```
{
  Matrix A,B,C;
  {
    Matrix compiler_tmp;
    mul(compiler_tmp,B,C);
    A.operator=(compiler_tmp);
  }
}
```

The `mul()` function is shown as a procedure, as internally user defined functions are often compiled as procedures with the location to store the return value as the first argument. This is necessary because the code can be used in several different scenarios. It can be used as an argument to a function, or as a an argument to a constructor.

When a call to the function is used as an argument to a function (as in the example above), the compiler generates a temporary. When it is used as an argument to a constructor, the return value is constructed into the constructed value directly. For example, this code does not involve a compiler generated temporary:

```
{
  Matrix A,B;
  Matrix C(mul(A,B));
}
```

That is the space for `C` is used by `mul()` to construct its return value into.

What is happening internally here, is something like this:

```
{
  Matrix A,B;
  Matrix C;    // where this would not be constructed yet
  mul1(C,A,B);
}
```

The examples above have shown an approximation to what occurs internally to the compiler with the implementation of the respective code fragments. More formally functions which have a return value are instead given an extra reference parameter of the same type as the return value.

So transformed version of `mul()` named `mul1()` above is implemented like this:

```
Matrix& mul1(Matrix& _return, Matrix& A, Matrix& B)
{
  Matrix Res;
  // do calculation Res = A * B
  return _return(res);
}
```

The first parameter must be a reference to an uninitialized Matrix, it is possible to force C++ to produce uninitialized references, though this is not normal usage.

Also it is necessary to be able to force C++ to later initialize such a reference, as during normal usage it is not possible to initialize a reference after its declaration, as this would normally include initialization, and for references the compiler enforced rule in C++ is that only a single initialization is allowed.

Finally it is necessary to be able to force the destruction of the object.

One way to create uninitialized references is to create the space with an array of `chars`, as there is no default constructor for built–in types. This space can then be cast to a matrix reference, which achieves the desired effect of having allocated space for a matrix without calling the constructor.

This would be achieved by this code:

```
char space[sizeof(Matrix)];
matrix& uninit_ref = *(Matrix*)space;
```

Enough space is created for a `Matrix` without initializing it by using an array of `chars`. Next the space is cast to be a reference to the appropriate type, a `Matrix`.

It is also necessary to force the construction of the uninitialized reference, the way to achieve construction of previously allocated space is using the placed new syntax for the new operator. The syntax is:

```
new (memory) type( constructor-args );
```

the semantics of the placed variant of operator new is related to those of the use of the normal non–placed operator new:

```
new type( constructor-args );
```

the example operator new (the unplaced version immediately above) means the space is allocated from the heap, treated as type `type`, and constructed with the given constructor arguments.

The placed syntax simply means that instead of the space being allocated from the heap, the memory area is instead supplied by the user. This allows the user to request construction of arbitrary locations of memory.

Finally there is a syntax provided within C++ to allow the explicit destruction of an object. This language feature is required in C++ as a complement to the placed `operator new()` syntax, otherwise there would be no way to destruct the object. The syntax for explicitly deleting an object is:

```
object.~class();
```

or more explicitly (explicitly giving the type of the object):

```
object.class::~class();
```

where `object` is a variable of type `class`.

Using the placed syntax new, and technique of obtaining unconstructed space of the required size by using arrays of chars, the functioning of `mul()` can be shown:

```
Matrix& mul(Matrix& _return, Matrix& A, Matrix& B)
{
  Matrix res;

  // do calculation res = A * B
  // construct res into _return

  return *new (&_return) Matrix( res );
}
```

The code using the transformed `mul()` function changes also, for code such as this, where a compiler temporary is required:

```
{
  Matrix A,B,C;
  A = mul(B,C);
}
```

the transformed code is:

```
{
  char compiler_tmp_space[ sizeof(Matrix) ];
  Matrix& compiler_tmp = *(Matrix*)compiler_tmp_space;

  mul(compiler_tmp,B,C);
  A.operator=(compiler_tmp);
  compiler_tmp.Matrix::~Matrix();
}
```

For code such as:

```
{
  Matrix A,B;
  Matrix C(mul(A,B));
}
```

where no compiler generated temporary is required, the transformed code is:

```
{
  Matrix A,B;
  char C_space[ sizeof(Matrix) ];
  Matrix& C = *new(Matrix*)C_space;
  mul1(C,A,B);
  C.Matrix::~Matrix();
}
```

The transformed `mul()` function, named `mul1()` above, returns its return value still to simplify the transformation of more complex expressions.

Consider the expression:

```
{
  Matrix A,B,C,D;
  D = mul(A,mul(B,C));
}
```

The locations of the `mul()` function calls (in the form of the transformed function `mul1()`) are left as is, the extra parameters are passed, and an appropriate number of compiler temporaries are constructed before the expression, and the generated temporaries destructed after the expression.

The above expression becomes:

```
{
  Matrix A,B,C,D;

  char ctmp1_space[ sizeof(Matrix) ];
  Matrix& ctmp1 = *(Matrix*)ctmp1_space;

  char ctmp1_space[ sizeof(Matrix) ];
  Matrix& ctmp2 = *(Matrix*)ctmp2_space;

  D = mul1(ctmp2,A,mul1(ctmp1,B,C));

  ctmp2.Matrix::~Matrix();
  ctmp1.Matrix::~Matrix();
}
```

The above transformations allow compiler temporaries to be expressed explicitly. This is necessary for the subsequent transformations. The relationships between the transformations is described in Chapter 5.

# Bibliography


[1] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting Heterogenous Parallelism on a Multithreaded Multiprocessor. Technical report, Tera Computer Company, 1992.

[2] Gregory Andrews. *Concurrent Programming Principles and Practice*. Benjamin / Cummings Publishing, 1991.

[3] Colin Atkinson, Stephen Goldack, Andrea Di Maio, and Rami Bayan. Object–oriented Concurrency and Distribution in DRAGOON. *Journal of Object–oriented Programming*, 4(1):11–18, 1991.

[4] M Atkinson. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

[5] Adam Back and Steve Turner. Portability and Parallelism with Lightweight p4. In *BCS PPSG Conference on General Purpose Parallel Computing*, 1993.

[6] Adam Back and Steve Turner. Time–stamp Generation for Optimistic Parallel Computing. In *Proceedings of the 28th Annual Simulation Symposium, Pheonix, AZ*. IEEE Press, April 1995.

[7] Adam Back and Steve Turner. Using Optimistic Execution Techniques as a Parallelisation Tool for General Purpose Computing. In *Lecture Notes in Computer Science*, volume 919, pages 21–26. Springer–Verlag, 1995.

[8] Adam Back and Steve Turner. Transformations for the Optimistic Parallel Execution of Object–oriented Programs. In *Proceedings of the Parallel Object–oriented Methods and Applications Conference (POOMA 96)*, 1996.

[9] D F Bacon and R E Strom. Optimistic Parallelization of Communicating Sequential Processes. *ACM SIGPLAN Notices*, 26(7):155–166, 1991.

[10] David Bacon, Susan Graham, and Oliver Sharp. Compiler Transformations for High–performance Computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, November 1993.

[11] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. On Modelling Consistent Checkpoints and the Domino Effect in Distributed Systems. Technical report, INRIA, June 1995.

[12] A Beguelin, J Dongarra, A Geist, R Manchek, and V Sunderam. A User's Guide to PVM: Parallel Virtual Machine. Technical report, Oak Ridge National Laboratory, September 1991.

[13] Steve Bellenot. Global Virtual Time Algorithms. *Proceedings of the Multiconference on Distributed Simulation*, 22(1):122–127, 1990.

[14] Guy Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1988.

[15] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An Object–oriented Toolkit and Class

Library for Building Fortran and C++ Restructuring Tools. *Object Oriented Numerics*, 1994.

[16] Luc Bomans and Rolf Hempel. The Argonne/GMD Macros in FORTRAN for Portable Parallel Programming and their Implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[17] R Butler and E Lusk. *User's Guide to the p4 Parallel Programming System*, 1992.

[18] R Butler and E Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. Technical report, Argonne National Laboratory, 1993.

[19] K M Chandy and J Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions of Software Engineering*, S.E.5(5):440–452, 1979.

[20] K Mani Chandy and Carl Kesselman. Compositional C++: Compositional Parallel Programming. Technical report, California Institute of Technology, 1992.

[21] Arunodaya Chatterjee. Futures — A Mechanism for Concurrency Among Objects. *Proceedings: Supercomputing*, pages 562–567, 1989.

[22] Arunodaya Chatterjee, Arjun Khanna, and Ying Hung. ES–KIT: An Object–oriented Distributed System. *Concurrency: Practice and Experience*, 3(6):525–539, December 1991.

[23] Lyndon Clarke and Greg Wilson. Tiny: An Efficient Routing Harness for the Inmos Transputer. *Concurrency: Practice and Experience*, 3(3):221–245, 1991.

[24] Crispin Cowan. Optimistic Replication in HOPE. In *In Proceedings of the CAS Conference*, pages 269–282, 1992.

[25] Crispin Cowan. Optimistic Programming in PVM. In *In Proceedings of the 2nd PVM User's Group Meeting*, Oak Ridge, TN, USA, May 1994.

[26] Sajal K Das and Falguri Sarkar. A Hypercube Algorithm for GVT Computation and its Application in Optimistic Parallel Simulation. In *Proceedings of the 28th Annual Simulation Symposium*, pages 51–60, April 1995.

[27] M Debbage, M B Hill, D Nicole, and A Sturges. The Virtual Channel Router. *Transputer Communications*, 1(1):1–16, 1993.

[28] Margaret A Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison–Wesley, 1990.

[29] John B Evans. *Structures of Discrete Event Simulation*. Ellis Horwood, 1988.

[30] M J Flynn. Some Computer Organisations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[31] Richard M Fujimoto. The Virtual Time Machine. *SPAA (Symposium on Parallel Algorithms and Architectures)*, pages 199–208, 1989.

[32] Richard M Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[33] Adele Goldberg and David Robson. *Smalltalk 80: The Language and its Implementation*. Addison–Wesley, May 1983.

[34] A Grimshaw and J Liu. Mentat — An Object–oriented Macro Data Flow System. *OOPSLA Conference on Object–oriented Programming Systems, Languages and Applications*, 22(12):35–47, 1987.

[35] J Gurd, C Kirkham, and I Watson. The Manchester Prototype Dataflow Machine. *Communications of the ACM*, 28(1):35–52, January 1985.

[36] Andrew Hipperson. T9000 Virtual Link Performance: A Comparison of Short Message Performance on T805s and T9000s. Technical report, PARSYS Limited, England, 1993.

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

[38] C A R Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[39] C A R Hoare, editor. *OCCAM 2 Reference Manual*. Prentice Hall International, 1988.

[40] HP FORTRAN Forum. High Performance Fortran Language Specification, Version 1.0. High Performancd Fortran Forum, May 1993.

[41] W Hwu, S Mahlke, W Chen, P Chang, N Warter, R Bringmann, R Ouellette, R Hank, T Kiyohara, G Haab, J Holm, and D Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993.

[42] Inmos. *Transputer Instruction Set: A Compiler Writers Guide*. Cambridge University Press, 1988.

[43] Inmos. *Transputer Reference Manual*. Prentice Hall, 1988.

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

[45] ISO. Information Technology — Programming Languages — FORTRAN Reference No. ISO/IEC 1539:1991(E). International Standards Organisation, 1991.

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

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

[48] Leslie Lamport. The Parallel Execution of DO Loops. In *Proceedings of the ACM*, volume 17, pages 83–93, 1974.

[49] Ted G Lewis. *Introduction to Parallel Computing*. Prentice Hall, 1992.

[50] Yi–Bing Lin, Bruno Priess, Wayne Loucks, and Edward Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, pages 3–10, 1993.

[51] Brian Lings and Lou Foster. A Temporal Model for Active Object–oriented Databases. Technical report, Exeter University, R284, 1993.

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

[53] Bertrand Meyer. *Object–oriented Software Construction*. Prentice Hall International, 1988.

[54] Bertrand Meyer. Systematic Concurrent Object–oriented Programming. *Communications of the ACM*, 36(9):56–80, September 1993.

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

[56] James G Mills and R Alasdair Bruce. CHIMP Version 1.0 Interface. Technical report, University of Edinburgh, May 1992.

[57] M Minsky. *Computation: Finite and Infinite Machines*. Prentice–Hall, 1967.

[58] MPI Forum. MPI: A Message–Passing Interface Standard. Message Passing Interface Forum, May 1994.

[59] OMG. CORBA. Technical report, The Object Management Group, 1994.

[60] Perihelion. *The Helios Parallel Operating System*. Prentice Hall, 1991.

[61] Runping Qi and Alan Wagner. Programming for Data Parallelism using C++. In *Proceedings of NATUG2*. IOS Press, October 1989.

[62] Michel Raynal. About Logical Clocks for Distributed Systems. *ACM Operating Systems Review*, 26(1):41–48, 1992.

[63] P Reiher and D Jefferson. Virtual Time Based Dynamic Load Management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation*, 7(2):91–120, 1990.

[64] Ronald Rivest. The RC5 Encryption Algorithm. Technical report, MIT Laboratory for Computer Science, 1994.

[65] Robert Ronngren. *Performance Issues in Discrete Event Simulation*. PhD thesis, Royal Institute of Technology, Stockholm, 1995.

[66] John Sargeant. United Functions and Objects: An Overview. Technical report, University of Manchester, Department of Computer Science, 1993.

[67] V Sarkar and J Hennessey. Partitioning Parallel Programs for Macro Dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 202–211, 1986.

[68] Jeff Steinman. Discrete–event Simulation and the Event Horizon. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 39–49, 1994.

[69] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, Second edition, 1991.

[70] Alexander Tomlinson and Vijay Garg. An Algorithm for Minimally Latent Global Virtual Time. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, pages 35–42, 1993.

[71] Steve Turner and Adam Back. A T9000 Implementation of the p4 Parallel Programming Model. In *Proceedings of the North American Transputer User Group*, 1994.

[72] Steve Turner and Adam Back. General Purpose Optimistic Parallel Computing. In *Proceedings of 7th PARSYS User Group Meeting*, 1994.

[73] University of Kent. *T9000 Systems Workshop*, 1992.

[74] L G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[75] S Winter, N Kalantery, and D Wilson. From BSP to a Virtual von Neumann Machine. In *Parallel Processing Specialist Group*, pages 92–99, 1993.

[76] S Winter, N Kalantery, and D Wilson. The Virtual von Neumann Machine Project at the University of Westminister. In *Proceedings of the Parsys Ltd User Group*, 1994.

[77] Klaus Wolf, Andrea Holtz, and Monika Lang. High Performance C++. In *Lecture Notes in Computer Science: High Performance Computing and Networking Conference Proceedings*, pages 542–547. Springer–Verlag, May 1995.

[78] H P Zima and B Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.