Modelling Parallel and Distributed Virtual Reality Systems for Performance Analysis and Comparison

Thesis

Submitted in fulfilment of the
requirements for the Degree of
DOCTOR OF PHILOSOPHY
of Rhodes University

by

Shaun Douglas Bangay

November 1996

Abstract

Most Virtual Reality systems employ some form of parallel processing, making use of multiple processors which are often distributed over large areas geographically, and which communicate via various forms of message passing. The approaches to parallel decomposition differ for each system, as do the performance implications of each approach. Previous comparisons have only identified and categorized the different approaches. None have examined the performance issues involved in the different parallel decompositions. Performance measurement for a Virtual Reality system differs from that of other parallel systems in that some measure of the delays involved with the interaction of the separate components is required, in addition to the measure of the throughput of the system. Existing performance analysis approaches are typically not well suited to providing both these measures.

This thesis describes the development of a performance analysis technique that is able to provide measures of both interaction latency and cycle time for a model of a Virtual Reality system. This technique allows performance measures to be generated as symbolic expressions describing the relationships between the delays in the model. It automatically generates constraint regions, specifying the values of the system parameters for which performance characteristics change.

The performance analysis technique shows strong agreement with values measured from implementation of three common decomposition strategies on two message passing architectures.

The technique is successfully applied to a range of parallel decomposition strategies found in Parallel and Distributed Virtual Reality systems. For each system, the primary decomposition techniques are isolated and analysed to determine their performance characteristics. This analysis allows a comparison of the various decomposition techniques, and in many cases reveals trends in their behaviour that would have gone unnoticed with alternative analysis techniques.

The work described in this thesis supports the Performance Analysis and Comparison of Parallel and Distributed Virtual Reality systems. In addition it acts as a reference, describing the performance characteristics of decomposition strategies used in Virtual Reality systems.

Acknowledgements

This work would not have been possible without the resources of the Internet, and the assistance of members of the Internet community. Many people have provided software, information and advice which has contributed greatly toward the content of this thesis. Discussions with members of the virtual reality community across the world, both via e-mail, and through public forums have been an important source of information. Thanks go to Professor Miguel Menasche who not only provided software, but also went to the extra effort of providing translations into English.

I received a great deal of valuable advice on the preparation of this document from Professor Patrick Terry, whose meticulous attention to detail and many suggestions have improved the content tremendously.

My thanks go to the members of the RhoVeR development team of the VRSIG: Greg Watkins, James Gain, Kevan Watkins and Luis Casanueva. Their enthusiastic attitude toward the subject, and willingness to engage in discussion provided an enjoyable working environment.

My supervisors, Professors Peter Clayton and David Sewry, have provided valuable insight and constructive criticism.

Finally, to Lindsey, who survived through this.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many virtual reality systems are implemented on a variety of specialized and non-specialized hardware. Most make use of dedicated equipment to improve the efficiency of particular components of the system. Graphical output is produced on machines with hardware rendering capabilities, for example. Others are built on advanced architectures intended for parallel processing. The goal of this thesis is to identify the components of virtual reality systems and assess the suitability of various parallel decomposition strategies for implementation on a range of distributed architectures. Thus the design of a distributed virtual reality system can be done, knowing the performance characteristics of potential decomposition strategies. A new performance analysis approach is developed to attain this goal. The results of analysis of the decomposition strategies found in many parallel and distributed virtual reality systems are presented to provide a reference to system designers. The analysis approach allows the performance of new techniques, and of modifications of existing ones, to be incorporated into the results presented in this document.

There are two components to this document. The first describes the performance analysis technique; its origins, enhancements and applications, for those interested in applying it to new problems. The second component concentrates on the application of the technique, showing both the results of analysis of the decomposition strategies found in virtual reality systems, and an illustration of the ways in which the analysis technique can be applied. The two components together provide a guide to the designer of virtual reality systems, allowing the choice of a system design that offers the best performance on the target architecture.

1.1 Background

1.1.1 Virtual Reality

Virtual Reality (VR) can be viewed for the purposes of the discussion in this document as the process of creating a computer generated environment into which one or more users can be immersed. Immersion is normally achieved by generating a three dimensional view of the modelled environment, and displaying this to the users via a stereoscopic viewing device such as a Head Mounted Display (HMD). Interaction with the system is achieved by measurement of head and

hand movements via transducers attached to an HMD and to a glove. Many other variations in the equipment are possible; the devices described are those most commonly in use at present.

Virtual Reality has its roots back in the early 1960s, when Ivan Sutherland was creating *Sketchpad* and inspiring the field of Computer Graphics. At about the same time (1965-1968), he was also developing the first Head Mounted Display [Kal93]. It was a number of years before the computing power was available to provide the rendering performance needed for virtual reality systems, and it was in the early 1990s that the concept became widely popular. The latter may have been partly due to the books of science fiction writers such as William Gibson [Gib84], to whom the coining of the term *cyberspace* has been attributed.

The tendency to standardize on glove and HMD as common virtual reality equipment may have been the influence of the company VPL, who marketed what was, at one stage, a very popular glove and HMD combination. Amongst their software offerings was RB2 (Reality built for two), a multi-user virtual reality system which was also multi-platform. Different machines were used to implement different aspects of the system such as graphical rendering and world design. This tendency is noticeable amongst the other virtual reality systems of that vintage, where researchers commandeered all available equipment in an attempt to meet the enormous computational requirements of creating reality.

Thus virtual reality systems have been implemented on parallel and distributed architectures right from their fairly recent infancy. This thesis is intended to categorize the approaches used to date, to provide a mechanism for comparing them and to provide a foundation for the development and evaluation of new approaches.

1.1.2 Performance analysis and comparison in virtual reality systems

The distinction between parallel and distributed systems is determined by the degree of coupling between processors. Distributed systems consist of autonomous computers, connected by a communication network [Sin91]. Communication is solely by message passing. While many virtual reality systems fall into the category of distributed systems, some also make use of tightly coupled processors. The latter parallel systems are more commonly used for the purpose of improving system performance. Distributed virtual reality systems often occur because of the need to create multi-user virtual reality systems, where separate components must be combined into a single virtual world.

Performance issues have been considered for these parallel and distributed virtual reality systems, but usually only at a qualitative level and always in isolation (see section 2.2). Many different techniques have been developed - almost every system has its own unique variations on the parallel decomposition strategies used to distribute the components of the system. With systems running on a range of different architectures, numerical performance results do not provide a meaningful comparison mechanism.

A suitable comparison format is required. Once this exists, the different decomposition strategies in virtual reality systems must each be analysed to produce results in this format. The results of this analysis can then be used to identify the best decomposition strategy for a particular architecture and application.

1.2 Document Layout

The order in which topics are covered in this document are as indicated by the outline below.

The selection of a performance prediction technique involves identifying the required area of applicability, finding a technique matching that area and evaluating its effectiveness. Parallel and distributed virtual reality systems are examined and the parallel decomposition strategies employed in each are identified. The features of importance during this examination are those that are responsible for the interaction between the distributed components, particularly those features which differ between the various systems. The different performance analysis techniques are then evaluated, based on their suitability for performing the required analysis on the constructs present in these systems. None of the analysis techniques meet the requirements that fulfil the goals of this work. A new performance prediction technique is developed based on existing approaches, and this development process is described in detail.

An example of a common component of a virtual reality system, a collision detection algorithm, is selected and used to verify the analysis technique. The collision detection algorithm is decomposed using common strategies identified during the analysis of virtual reality systems. Implementations of each of these are modelled, and the results compared against their performance measured in practice.

The analysis of virtual reality systems is then performed on a component basis. The input, output and world modelling components are examined first, before the performance issues in complete systems are addressed. The results of this analysis provide a reference to the performance characteristics of the decomposition strategies found in parallel and distributed virtual reality systems.

1.3 Overview

- Chapter 2 describes parallel and distributed virtual reality systems, and identifies the key components of these systems.
- Chapter 3 describes a number of possible candidate techniques for performance analysis and discusses their merits with respect to the analysis and comparison of virtual reality systems. An introduction to constructs (Petri Nets and Data Flow Graphs) used in some of these techniques is given in Appendix A.
- Chapter 4 introduces the Analytical Simulation approach used for the performance analysis and comparison of virtual reality systems, and discusses some implementation considerations for this approach. A description of the tool that implements this approach is described in Appendix B, while the semantics of the modelling language are specified in Appendix C.
- Chapter 5 devises a distributed collision detection algorithm and demonstrates the ways in which it can be implemented using common decomposition techniques found in distributed virtual reality systems.

- Chapter 6 provides a detailed analysis of the client-server decomposition of the collision detection algorithm, using Analytical Simulation.
- Chapter 7 describes enhancements to the Analytical Simulation technique that overcome limitations which complicate the analysis in Chapter 6.
- Chapter 8 completes the analysis of the collision detection algorithms and tests the predictions against the performance of actual implementations.
- Chapter 9 decides on the strategy to employ in the analysis of virtual reality systems and their components.
- Chapter 10 examines the performance implications of parallel processing within the input components of virtual reality systems.
- Chapter 11 investigates graphical output, in particular the pipeline construct used for parallel rendering.
- Chapter 12 examines parallel decomposition strategies used for world simulation.
- Chapter 13 considers the performance of complete systems, combined from the components described in Chapters 10, 11 and 12.
- Chapter 14 summarizes the results obtained throughout the thesis and lists the contributions that it makes.
- Appendix A provides an introduction to Petri Nets and Data Flow Graphs.
- Appendix B describes the statements used to specify models for Analytical Simulation, and presents an example of the use of the Analytical Simulation tool.
- Appendix C describes the process of translating the models used for Analytical Simulation into CCS. This translation both specifies the semantics of the modelling constructs, and allows the formal proof techniques of CCS to be applied to these models.

Chapter 2

Decomposition strategies in virtual reality systems

Building efficient parallel virtual reality systems requires the ability to identify the different strategies for decomposing a system into components and to compare the various ways in which these components can be connected. The purpose of this work is to identify the best components and connection strategies for a particular application and architecture. Achieving this result requires a method of comparing components and strategies. Selection of a comparison tool requires that the nature of the candidates for comparison be understood.

The criteria for judging and the possible candidates for consideration are described in this chapter. Once the possible configurations have been identified, a suitable paradigm must be selected for finding the required performance values for a particular parallel decomposition strategy. The choice of comparison tool is the subject of Chapter 3.

2.1 Performance issues in virtual reality systems

The performance issues considered in this chapter are those that can be affected by increased computational power and greater communication bandwidth. Issues relating to more accurate physical simulation or human-computer interface issues are not relevant to this discussion, unless they are influenced by these performance factors.

It is well agreed that performance of virtual reality systems can be characterized by two metrics: throughput and lag [Hub93b] [Gos94] [Sty96] [Wlo95] [Ams95]. Throughput gives a measure of the rate at which the system performs an action. It is often measured for graphical output and is otherwise referred to as the frame rate, or using the reciprocal, as the cycle time. The lag, also often called the latency, is a measure of the time taken for the effect of an event to be noticed. Applied to a complete virtual reality system, the interaction latency is the difference in time between input being supplied (user performing an action) and the corresponding output (the effect of the action being rendered).

Several different effects have been described as contributing to lag in a virtual reality system [Wlo95]:

- *User input device lag* - time taken by the input device to report a reading.
- *Application-dependent processing lag* - time taken for computation using the input value.
- *Rendering lag* - time taken for rendering and display on the screen.
- *Synchronization lag* - time spent blocking by processors waiting for others to accept data.
- *Frame-rate-induced lag* - lag while the current frame remains on the screen, until the next frame replaces it.

The overall latency in the system is comprised of combinations of these. While variations in the throughput may affect the lag (frame rate affects frame-rate-induced lag), and vice versa, neither value can be derived from the other.

The term *latency* is used in the remainder of this document to indicate lag values. The measure *cycle time* is also used as the indicator of throughput, since it is conveniently based in the time domain as are all the performance measurement approaches. In both cases a decrease in value represents an improvement in performance.

2.2 Taxonomies of parallel and distributed virtual reality systems

Recently some attempts have been made to provide a systematic way to categorize the alternative decompositions of different parallel virtual reality systems. Previous work by the author includes a survey of many parallel and distributed virtual reality systems [Ban93]. Many authors have begun to acknowledge the existence of alternative approaches when describing their systems [Wan95] [Sha93] [Sin95].

A taxonomy of networked virtual environments is proposed in [Mac95b]. The areas used to categorize virtual reality systems are communication issues, synchronization of the view of the virtual world for different users, and data and process issues. The communication issues cover topics such as bandwidth, distribution schemes, latency and reliability. The comparisons tend to be dominated by issues of widely dispersed high speed networks, support for large numbers of users and considerations of broadcasting and multicasting. Coincidentally these are characteristics of NPSNET over DIS (see section 2.3.4), a system on which the author of the taxonomy has worked extensively. The data issues consider the nature of the database representing the virtual world and the manner in which the data is made available to all participants in the virtual world. This

is an area that receives a great deal of attention throughout this thesis. The different approaches identified are:

- *Replicated database*: Copied to all processors with updates broadcast periodically.
- *Centralized database*: Kept on a central machine, to which updates and requests for information must be sent.
- *Distributed database*: Replicated database but guaranteed to be synchronized at all times.
- *Distributed databases*: Database partitioned across a number of servers.

Processes in most of the systems are categorized as homogeneous, in that the same type of process runs everywhere. The emphasis is on scripting languages which allow different behaviours for these processes, and which allow process migration.

While describing the development of a distributed virtual reality system, [Ams95] identifies two communication infrastructures, a star connecting every component to every other, and a centralized model where clusters of components communicate through a central process. Access to information is obtained either by collecting it from a predetermined point, actively requesting it, or by registering to have regular updates sent.

In a tutorial dealing with the creation of virtual environments [Gos94], the communication models identified are those of the centralized database and replicated database. Mention is also made of the use of broadcasting, multicasting and unicasting as alternative approaches to implementing an update mechanism for replicated databases. Other issues which need to be implemented in a virtual reality system include collision detection, physical modelling and a way in which objects in the world are controlled.

A discussion of distributed virtual reality [Sty96] interleaves the issues required to produce realism with those related to structure and communication. Again the influence of DIS means that the aspects considered are tailored for widely distributed systems with large numbers of users. Under these circumstances, communication strategies requiring absolute synchronization compare poorly against broadcast and multicast approaches. The latter use less frequent communication and a system of dead-reckoning to provide the intermediate values. Communication issues are of great significance for these systems as emphasized in [Roe95] where multicast zones are discussed to limit portions of the communication traffic to only those objects that can make use of it.

2.3 Components of parallel and distributed virtual reality systems

The remainder of this chapter provides a survey of parallel and distributed virtual reality systems and investigates the components and distribution strategies which affect performance. The descriptions cover the evolution of these systems from their humble beginnings to the most recent versions available.

Diversity decreases as successful approaches are identified and favoured. A record of the techniques discarded in the process remains valuable, to prevent mistakes recurring, and to provide

alternatives should the architectural constraints evolve. The different stages in the development of these systems are described to allow the trends in their evolution to be seen.

The issues of relevance in the following sections are the parallel decomposition and communication strategies employed in virtual reality systems. These are related to the structural decomposition of the system (the way in which the simulation of the virtual world is broken down) and the architectural and network constraints. The different structural decompositions are summarized in Table 2.1. The taxonomies in section 2.2 identify some possible areas in which the systems differ, in particular the ways in which data is distributed and in which the control of the various components of the system is synchronized. The way in which some of the primary functions of the virtual reality system are decomposed across separate processors must also be considered. These functions include tasks such as physical modelling, collision detection and inter-object control and interaction.

The parallel decomposition strategies already mentioned in section 2.2 include a centralized database (using a client-server model for access), distributed databases and replicated databases (using either strict synchronization or periodic updates through broadcast or multicast mechanisms). Processes have been running independently, without much consideration for interaction except through the database. Different strategies for the parallel decomposition of this control exist, such as the presence of a master process that distributes the work. A summary of the parallel decomposition strategies present in parallel and distributed virtual reality systems is given in Table 2.2. The different data distribution approaches are represented diagrammatically in Figure 2.2.

2.3.1 The VROS and RhoVeR

The Virtual Reality Operating System (VROS) resulted from a project investigating the development of a virtual reality system specifically for a parallel architecture. The implementation environment is a cluster of Transputers, communicating via message passing. The design decisions are documented in detail in [Ban94a]. The following brief description summarizes the parallel decomposition strategies.

The virtual universe is divided into worlds, each representing a particular scenario. Each world contains a number of objects, which have various attributes including their position and orientation in space. Some of the objects represent the humans interacting with the system and are referred to as users. A virtual reality application normally consists of one or more worlds containing objects interacting toward a particular goal. An example of an application would be a walkthrough consisting of a world containing a building object and a number of furniture objects.

The VROS contains various device drivers for supplying data to the users (output device drivers) and reading data from them (input device drivers). The section of most interest for this discussion is the virtual world simulator, usually called the kernel.

The kernel is capable of supporting multiple worlds simultaneously, and allowing multiple users to enter each world.

To utilize the parallel architecture effectively, each world is represented as a data structure consisting of the attributes of the objects in that world. The objects, which can be located on

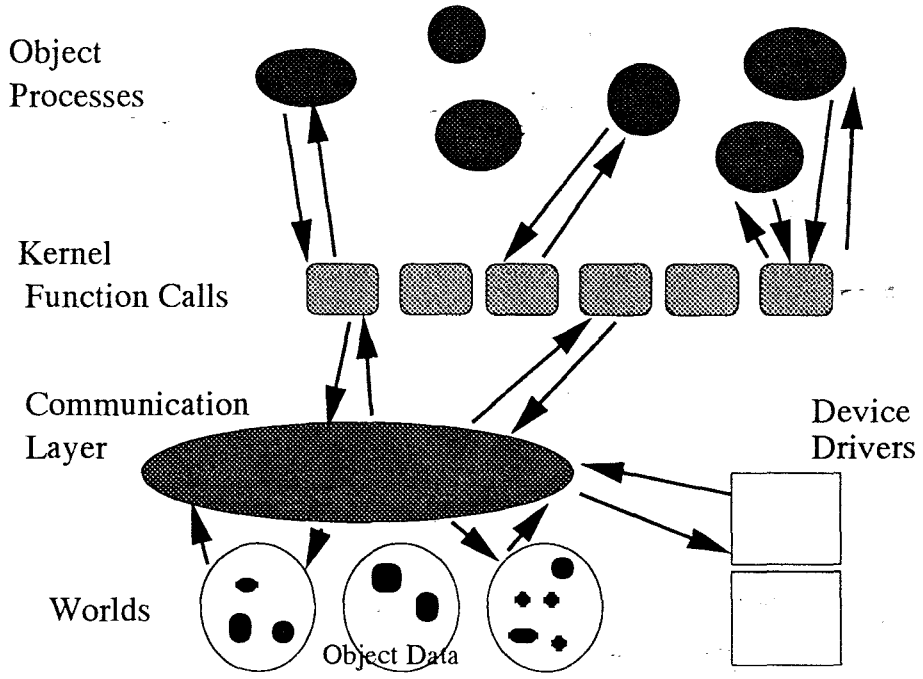

Figure 2.1: Components of the VROS

separate processors, must query the world for all data relating to them and the other objects in the world. This approach is implemented as a number of processes which can run on any of the processors in the Transputer cluster. The world processes simply act as servers, supplying data to each object and updating their databases on request. In this way these processes can keep all the data consistent. The object processes control the actions of the objects in the world and implement the laws of that world. In addition, the object processes belonging to users may invoke the device drivers for additional control information. A communication layer exists for routing messages between processes, masking the details of the physical architecture. This decomposition is represented visually in Figure 2.1.

Application programmers supply the control routines for each object. They are provided with a set of routines for manipulating aspects of the virtual worlds. The underlying architecture is invisible at this level. Routines exist for reading and updating the attributes of the current object. Similar routines exist for reading and updating attributes of the other objects, although more restrictions exist in this case. Objects can only control others if they are classed as owners of these objects. No direct communication between objects is currently supported by the kernel. Object processes may signal to each other by changing their attributes, or through communication routines which must be provided by the application programmer. Various other functions exist to provide useful facilities such as transferring objects from one world to another.

Development on VROS has ceased. Instead the experience acquired from this system and the ideas used in other systems have been used to design and create a new system intended to facilitate research into a number of areas in virtual reality. This includes the performance analysis and comparison described in this document. This system, named RhoVeR, was implemented as

a group effort to facilitate the work of the virtual reality research group at Rhodes University. Since RhoVeR is intended to allow various approaches to implementing distributed virtual reality systems to be tested, it is not limited to any single distribution approach. A description of the RhoVeR system may be found in [Ban96].

A limited selection of the various parallel decomposition approaches are implemented as a standard part of the RhoVeR system, to allow its use for other virtual reality related work. Processes are allocated to each object, to each device and to a number of control modules.

Control information is distributed by events which trigger responses from each process. Control events originate from timer processes, and other processes which implement functions such as collision detection and physical simulation.

Data distribution is accomplished through three mechanisms offering different tradeoffs in generality versus ease of use. The standard world representation is stored in a Virtual Shared Memory (VSM) block, a replicated database. On a single processor, the VSM is implemented as actual shared memory. VSM Manager processes are responsible for transmitting and receiving updates to synchronize the database. The values in the database represent the most recent values received. There is no guarantee that every copy of the database will be identical at every point in time.

For larger, less frequently changed values, a ShapeData structure exists which caches copies of data belonging to other processes. An update flag in the VSM causes the data to be reloaded if it is changed at the source.

For general communication that does not fit into either of the other categories, a message passing service is available for direct point-to-point communication.

Collision detection and physical simulation are performed by additional processes which monitor the state of the world through the VSM. Control events are then sent to the objects to enforce the correct behaviour. Control of objects and resolution of contention issues are controlled through a hierarchy of ownership. Ultimately, the process currently owning a particular object is responsible for any change in this state.

2.3.2 AVIARY

The Advanced Interfaces Group at the University of Manchester is working on the development of a general framework for advanced interfaces, which they have named AVIARY [Wes92] [Sno93]. This system is intended to support a broad range of Virtual Reality environments.

The AVIARY system runs on Transputers and SUN workstations. The communications system is the module most affected by different architectures. Versions are implemented for Transputer networks and SUNs connected by Ethernet. Graphics are produced by a hardware renderer.

AVIARY uses a structural decomposition of the virtual reality system into worlds and objects. These are represented by data structures, in contrast to other systems which implement them as processes in their own right. Worlds are collections of attributes (e.g. mass) and laws (e.g. gravitation), objects are known as *entities* and may be controlled by processes called *demons*. Control of objects can be placed into the hands of abstract *applications* which are distinct processes that manipulate the objects in the world. Many applications can exist in a single world, controlling

the various objects. Other systems, such as the VROS, create this sort of coherent behaviour by combining the efforts of the processes associated with each object and world.

While objects are permitted to be bound to processes which control their behaviour, an extra form of control is present from users or applications. A user under AVIARY combines characteristics of objects (in that it has a visible manifestation), and of applications (in that it is subject to control beyond that of the physical laws of the world). The user is modelled in the same way as an application. Other virtual reality systems which do not support the application concept must either make a special case for the user, or use the object representation with links made to input and output devices, as is done in RhoVeR.

AVIARY is segmented into processes that can run in parallel. A communication system similar to that used in the VROS is present to allow communication between processes. The processes in the system consist of:

- Input processes.
- Output processes.
- A Virtual Environment Manager.
- Environment Database that provides spatial management such as collision detection.
- Object Servers.
- Applications to control users or manipulate the virtual environment.

Objects and applications run separately as independent processes. Control is essentially distributed, although collision detection is treated as a special case [Sno94b]. The environment manager receives position updates from each object, and sends out messages to those that have collided. AVIARY implements an extension to this: once an environment manager is significantly loaded, it splits into separate processes which continue the collision detection computation in parallel.

Another special feature of AVIARY is the presence of object servers which allow one process to support more than one object. This decreases resource usage for processes with small computational requirements, and allows for the possibility of dynamic load balancing.

With AVIARY, the various processes communicate extensively. Data is kept in a series of distributed databases. Each object keeps the data relevant to it, and updates are transmitted when changes occur. Updates are limited to those processes which have expressed interest in the data belonging to a particular object.

The problem of supporting the range of features necessary to implement any reality is addressed in depth. The solution implemented is to provide a basic world that may be customized to the purpose required. This results in a conflict between the need to provide assistance to the application writer and to allow sufficient generality. To overcome this, the set of all possible worlds is structured as a hierarchy. The top of the hierarchy contains all possible worlds. Further down these laws are refined. For example, some worlds may have gravity, while others do not. This information may also be used to restrict the types of objects that may be moved from one

world to another. Consistency is maintained by making sure that the object is capable of obeying the laws of the new world. A system of portals is used to link different worlds.

A strength of the AVIARY design lies in the ability to implement physical laws without excessive involvement on the part of the application writer. The object oriented nature of the system, with the use of inheritance to control attributes for different worlds, is well suited to the design of a support environment for implementing virtual worlds.

2.3.3 Cyberterm

This system is intended to implement a single virtual world, a cyberspace that allows multiple users to share a common virtual area [Sno92]. The single world is distributed over a number of workstations with each machine acting as a server for a sector of the world. The system was initially implemented for PCs and SUNs connected by modem. Graphics are produced by rendering libraries such as VOGLE and REND386. Later versions have concentrated on the PC architecture, and on using Internet protocols, although still at modem speeds.

The positions of objects are kept by the server database. When an object enters a sector the object makes a local copy of this data. Velocity information is used to update the position of other objects, and updates are periodically issued when another object changes direction. This is appropriate where communication is over long distances and over limited bandwidth connections.

Each processor runs a server and possibly a client. Movement from one sector of the world to another requires the local client to connect to a different server.

The initial approach used in Cyberterm [Sno92] was to create a single space in which each processor acted as a server for a smaller portion. The latest releases of the product are fitting in more with the approach used in other virtual reality systems, where each server runs its own world connected to others via portal objects. This is effectively a cosmetic change; the underlying distribution techniques need not be affected by this.

Cyberterm is designed for use over modem lines, the slowest of communication media (amongst the distributed virtual reality systems at least). As such it concentrates extensively on minimal communication.

Data distribution uses the client-server model, with multiple servers, one for each area of cyberspace (or world). Clients connecting to remote servers route the information through their local server.

Control of objects is both via the server and the client. The server is the authoritative source of an object's action, while the client runs a dead-reckoning routine in an attempt to provide an accurate representation of an object's behaviour. Updates are sent from server to client when the behaviour of an object changes. The protocol also specifies periodic updates that can be sent from server to clients.

Access to the resources of an area of cyberspace (or a world) is controlled by the server process. The servers must issue permission for various actions, such as movement. Private areas of space can be created where rules decided on by the owner are enforced. Clients must explicitly request permission to create objects, or to relocate to specific areas. Such requests may be relayed to other clients who have control over the resource under contention.

An interesting notion in Cyberterm is that of an agent [Sno94a]. This is an object that can be launched by a client together with a controlling script which is executed by the server. Thus complex transactions can be limited to the server containing the data, and only final results need be communicated between server and client.

Collision detection is a recent addition. Two alternative approaches are suggested. The first uses a specialized agent which watches for collisions, the other requires that collision detection code be included in the server.

2.3.4 SIMNET, DIS and NPSNET

Distributed Interactive Simulation (DIS) and its predecessor SIMNET are standards for distributed interactive simulations [Loc93] [You95]. They are specifically intended for battlefield simulations, but are starting to be reworked for other applications, such as air traffic control simulation. The simulations may involve thousands of objects and take place over a wide area network.

Communication occurs over a relatively low bandwidth (in relation to the size of the simulation) medium, such as Ethernet. Each host machine controls its own vehicle and keeps track of others by dead-reckoning. Each host keeps track of its own dead-reckoned position and when this differs significantly from its actual position, it transmits an update to all other hosts.

A number of applications implement the DIS protocols, such as VR-Link [Mor95], Close Combat Tactical Trainer (CCTT) [Mas95], PARADISE [Sin96] and the popular NPS Networked Vehicle Simulator (NPSNET) [Mac95c]. When dealing with large simulations, NPSNET attempts to partition the space according to the relationships between the various participants [Mac95a]. Some of the disadvantages of the DIS protocol are the large number of other objects requiring updates and the need for all participants to maintain complete terrain databases. This leads to the need to limit the area of interest, and so communication between entities in the virtual world is limited to those entities which have some relationship with each other. In practice this is implemented with a multicast group, which allows a message to be transmitted by one process and received by multiple entities. Logically entities are related by their spatial, functional (e.g. involved in common simulated radio traffic) or temporal (update requirements) properties. A static decomposition into multicast groups is recommended in [Sri95] to overcome problems in identifying multicast groups for sending and listening.

The constraints given above strongly influence the manner in which distributed virtual worlds can be implemented. The use of low bandwidth communication, powerful processors and techniques such as dead-reckoning and broadcasting facilitate the use of replicated world databases. Each object controls its own behaviour, and the local processor animates all objects according to the latest motion update. The spatial partitioning of the world for multicast groups facilitates collision detection and physical modelling, which each processor performs independently on its area of interest. Object interaction, for example one object shooting another, involves sending a detonation message to a group. Members of the group are responsible for reporting any effect of the explosion.

2.3.5 DIVE

DIVE (Distributed Interactive Virtual Environment) is a loosely coupled heterogeneous distributed virtual reality system based on UNIX and running over local and wide-area networks using Internet protocols [And93a][And93b]. It provides shared memory over a network and controls the sending of signals to processes.

A world consists of a set of objects and various parameters. Objects are capable of moving from one world to another. The transfer is triggered when they intersect special objects designated as gateways. Under DIVE the world is maintained as a distributed database. Each process has its own copy of the structure, managed by the ISIS toolkit [Bir90]. Functions are provided to allow the updating of entries in each copy for all the processes in the world. The update requires locking all copies of the database before sending changes through. If all processes leave a world, the database is discarded.

An event handling system is present in DIVE allowing processes to register for certain types of event. The process can be notified when objects are created, removed, changed, or when interaction between a user and an object occurs. A timer event allows certain tasks, such as object movement, to be called periodically. Objects may be given primitive behaviour by specifying a state machine which performs certain actions on various events. A limited number of actions are possible, including moving, sending signals, and changing appearance.

The DIVE system consists of a set of processes, each capable of manipulating the world and its objects. These processes include visualizer processes that allow users to interact with the world, and application processes that operate on objects or introduce applications in the virtual world.

A number of high level tools are available for creating applications in DIVE. These functions support selection and grasping of objects. A vehicles module exists which uses the user's actions to control the virtual environment.

Recent developments on DIVE have been directed at extending the system to handle more users, over a widely distributed network.

Interaction between objects is controlled by aura, focus and nimbus [Ben94]. Aura defines a sub-space around an object within which it can interact with others. The other concepts define boundaries that govern the degree of mutual awareness of other objects. These concepts are implemented in DIVE using sub-objects to set the boundary limits, and collision detection is used to determine an object's awareness of others.

Collision checking is enabled by setting a flag for each object. If enabled, collision checking occurs when the object moves. The computation is performed by the process moving the object. When a collision occurs, a signal is distributed to all processes in the world.

The ISIS toolkit provides facilities for fault-tolerant distributed databases, amongst other things. As used in DIVE it uses a multicast protocol to distribute changes and set locks. All nodes in the system are guaranteed to have seen the same sequence of events, which, while good for system integrity, provides some limits on scalability. A limit of about ten peers is given as an upper bound for a DIVE system.

The most recent version of DIVE uses SID [Hag95], an alternative reliable multicast package. This package should make it possible to supply a distribution layer to hundreds of participants.

Requests for data are sent asynchronously, and a callback mechanism is used to notify processes of the arrival of data. Replies are generated by the nearest host, identified by packet timing. A token bucket mechanism is used to limit the back-to-back transmission rate. A simple dead-reckoning module is also present.

2.3.6 Division

The ProVision system is one of the early virtual reality systems produced by a Bristol based company, Division. It is a virtual reality server that connects to a number of host machines [Pou91]. The system was based on T425 and T805 Transputers, but has since been ported to a range of platforms. Various support software is available, including the Distributed Virtual Environment System (dVS).

This system provides real time control and distributed event handling. All activities and environment handling under dVS are performed by processes called actors. Sharing of data between the actors is controlled by dVS. The functionality of dVS is closer to that of an operating system than a system for the modelling of virtual worlds.

Parcels of data can be shared between various actors. Each actor makes a local copy of the data. In order for one actor to update the data, it must send an update request to a special actor, the director, which then propagates the update to other actors holding that data. Updating can be done in exclusive mode, which ensures that all actor processes have consistent copies at one time. The alternative is general mode which is faster, but actors separated by low bandwidth connections may experience delay in receiving the update.

In order to cope with real time constraints, each actor can maintain its own local time. When communicating, the director compares the different times of each actor and adjusts them so that they are in step. This is useful in synchronizing different hardware devices that operate at different speeds.

A renderer process called Paz converts a high level scene description to the polygon equivalent. Calls to Paz can be made to alter the position, motion and illumination of the objects.

The offerings from Division have evolved substantially from their initial ProVision system. The current system consists of two levels: dVS, the VR operating system which supports distributed virtual worlds, and dVISE , a high level package for the construction of virtual worlds. Being a commercial company there is a limit to the detail with which the description of their implementations is given. The description given in this section is pieced together from a number of publications describing various components of the system [Pou91] [Ghe95] [Div96].

The dVS VR operating system has been ported to a number of platforms. Interoperability is possible between any of the machines running the system. The complexity of the system has grown over its lifetime and it uses many techniques found in other systems. It supports a hybridization of different distribution approaches.

The basic system required on each machine is the *runtime*, which consist of a number of runtime actors controlling various aspects of the virtual reality system. These actors act as servers for application actors. The latter are external to the runtime, and are clients for the services provided by the runtime actors. The standard runtime actors provide graphical and audio output, control

input devices, perform collision detection and physical simulation, and create a representation of the virtual body of a user. All actors run as independent processes which can be located on one machine, or distributed across networked machines.

Communication actors known as agents control communication between machines. They are responsible for the byte ordering issues involved when different architectures communicate. Communication uses TCP/IP, allowing the system to be distributed over the Internet.

Distribution of updates and notification of changes occurs through the passing of events. Actors must register interest in the events which concern them. The event driven approach removes the need to work in a sequential manner. Processing is triggered by the arrival of events.

Dead-reckoning and position prediction are used in dVS to reduce latency.

Collision detection employs one of the runtime actors to watch for collisions between objects. Collision detection can be performed at various levels of accuracy, from simple bounding box checks to comparisons at a polygon level, depending on available processing power. Events are sent out to the objects involved when a collision occurs.

Data and control distribution can be implemented using event passing. Facilities remain for handling the parcels of data, the shared data elements which are visible to both application and runtime actors. Implemented as either a distributed database or a replicated database, updating is controlled by the master process (director) which ensures the consistency of the database.

2.3.7 Minimal Reality (MR) Toolkit

The MR toolkit is a library of functions for supporting the development of virtual reality interfaces [Gre92]. It provides support for a number of peripheral devices used for virtual reality. It also provides facilities for distributing the virtual reality over multiple workstations. The MR Toolkit assumes that different hardware is used to satisfy the different requirements of each process, and so concentrates on parallelism. Data sharing is via simulated shared memory on a message passing architecture.

The toolkit consists of three levels of functions. The first level contains the device support functions. These are implemented as a client-server pair, with the server continuously polling the device so that the client can have access to the most recent value without delay. The server also performs low-level processing of the data such as filtering.

The second level converts the data from the devices into a convenient form for the application programmer.

The third level of functions provides services for the application programmer. These include functions for the maintenance of distributed data structures.

The processes in an MR application can have three roles. One must be a master to control the application and start the other processes. There can be a number of slave processes that are used to produce graphical output. There may also be a number of computational processes that receive input from the master and return results to it. Data sharing is done by keeping local copies of the data with each process. The data structures must be periodically synchronized to ensure that all processes have the correct values. The application programmer is responsible for specifying when this update occurs. This differs from other virtual reality systems that use distributed shared

memory constructs. Other systems remove the need for the application programmer to attend to such details.

Communication is possible between separate MR applications. The master processes of each application can send device and application-specific data to other master processes. Slave processes must communicate via the master.

The MR toolkit can thus be operated in two ways. Firstly, a system with a single master process and some slave and computation processes can run as a distributed virtual reality application [Gre95]. Separate processes are usually not employed to represent individual objects in the system. Rather a compute process or the master provides the simulation, and the master (with a slave for stereoscopic views) does the rendering.

The other mode of operation uses the peers package [Sha95] in which master processes communicate. An example application using the peers package is provided with the system, and implements a simple virtual handball demonstration. Control of the simulation is passed between master processes, according to the user that holds the ball. The collision detection facilities are also provided by the application program. Collisions can occur between the ball, the static walls and bricks and the hand of the owner. These are all local to one process so no distributed computation occurs. Physical simulation is limited to ball movement and collision detection which, as mentioned previously, is performed by the master process that owns the ball.

The Environment Manager (EM) [Wan95] is a high level tool for constructing MR Toolkit applications. It provides two approaches to resolving contention for shared data structures. In the first approach, only one process owns the simulation at a time. Only that one process is allowed to change shared variables. The other case allows sharing of access to shared variables, either read only, or writable. Writeable variables could cause inconsistencies if there are delays in distributing changes. A token passing mechanism is used to prevent this. Global updates of the variable are only made when a token is held. In this way, the variable is subjected to the same sequence of updates on every processor.

The facilities offered by the MR toolkit are sufficiently general to allow variations on communication strategies. The applications for a single system tend to limit parallelism in the world modelling components, performing all the computation in the master process. The slave process is used to assist in rendering, while devices are also separate and sending data. The shared memory services are useful but tend to be prescriptive when designing an application. The peer package is better suited to parallel world modelling and more control is available in determining the manner in which the distribution is implemented.

2.3.8 Multiverse

Multiverse is a multi-user X-Windows based Virtual Reality system [Gra93]. The system runs on a UNIX platform and is based on a client-server model. It consists of servers that model the virtual world, and clients that are used for user interfaces. Each client and each server is a separate process, and each may run on a different machine.

Multiverse models each object as a data structure with an associated control process. A Multiverse server provides a single world containing all the objects.

The clients consist of a single program that performs roughly equivalent functions to the input and output device drivers in other virtual reality systems. The clients are generic, and independent of the world being modelled by the server. They consist of a loop which renders the view of the world and then sends any input from the user back to the server.

A server process manages the world and its objects. The main functions of managing a virtual world are taken care of transparently; the application writer is required to supply only a few functions. These are mostly trivial, the one of interest being the *animateWorld* function that defines the nature of the world. It is called from the main server loop and is usually used to move the objects in the world. Since all processes run on a single machine, there is no need for data sharing.

The objects may have special code to control their movement. Objects interact with each other and with the world using an event handling mechanism. These events include MOVE_EVENT that should cause the object to move, COLLISION_NOTIFY_EVENT for when objects have collided and TERM_NOTIFY_EVENT for when an object ceases to exist. The control routine of the object is executed as part of the thread of the server process. The object control routine is generally invoked when an event occurs which affects that object.

Simulation of the world in Multiverse uses a single thread of execution, as opposed to the multiple processes under the other parallel and distributed virtual reality systems that have been discussed. However, the machines that would support Multiverse typically contain a single processor, and so creating more processes would be redundant.

2.3.9 VR-386

VR-386 [Sta94] is a virtual reality system for the PC which is descended from Rend386, a polygon rendering library for 386 and 486 based systems with VGA displays [Sta92]. The design and implementation strongly reflect the need for efficiency when rendering views of worlds. Unlike the other systems, it runs as a single process.

VR-386 represents a world as a structure containing all the visible objects in that world. It is intended to be capable of supporting multiple worlds and to allow switching between these worlds.

The objects in Rend386 could have several representations corresponding to different levels of detail. Figures constructed of a hierarchy of objects can also be defined. Objects are then stored relative to the parent object in the hierarchy. For example, in a human figure the arms and legs may be made children of the torso object. VR-386 goes further by adding a degree of animation and automatic updating for parts of a figure. Objects move when the parent object moves, with additional effects from the joints linking them.

VR-386 provides for extensive control of input and output, and also includes many functions for manipulating virtual worlds.

2.3.10 The Virtual Environment Operating Shell (Veos)

Veos is an environment for creating distributed applications for Unix [Coc92] [Coc93]. It is designed for prototyping distributed virtual reality applications.

The processes required to implement a virtual environment are known as entities and can be distributed across a number of Unix workstations. A data type known as the *grouple* is used as the standard data structure. The grouple is an extension to the tuple used in the Linda programming paradigm [Car89]. Grouples consist of nested tuples. An interpreted Lisp system is used as the programming interface to Veos.

Each Veos entity consists of a distinct Unix process that controls interpretation of the task written in Lisp. Each entity has associated grouplespaces, for which pattern matching facilities are provided. Asynchronous message passing of grouples between entities is supported.

The initial system was intended for prototyping and performance issues were not considered important. The use of interpreted Lisp makes the system flexible and easy to use. It also allows evaluation of program stubs passed as messages.

Even though the grouplespaces may suggest use of shared memory, inter-process communication still involves message passing.

Over its lifetime, the VEOS system has undergone significant development. While still based around the variant of the Linda tuple space, the *grouplespace*, recent work [Bri93] has focussed on performance improvement.

Each entity is capable of direct communication with others. Communication between nodes makes use of direct, asynchronous message passing. Communication between entities can use this direct message passing, or it can make use of the pattern directed databases, the *grouplespace*.

Entities are provided with functions that specify each aspect of their interaction with their environment. *Perceive* functions determine what portion of the environment is accessible, *react* functions specify reactions to changes and *persist* functions specify behaviour. Entities may encapsulate others, as such providing a global environment for communication and data sharing amongst the enclosed entities.

Each entity maintains several standard local databases with specific access control. A *boundary* partition contains data for sharing with other entities, and an *external* partition keeps information about the other entities that are within perception range. The VEOS system is responsible for distributing changes from each entity's *boundary* partition to its sibling's *external* partitions at the end of each entity cycle. When changes occur faster than the system can make updates, only the most recent values are sent. Intermediate values are lost. An *internal* partition contains data from the *boundary* partitions of encapsulated entities. An *Interact* process within the entity gathers changes from the *boundary* partitions of encapsulated entities, and propagates them to the *internal* partition and to the *external* partition of each enclosed entity.

The entities contain *React* processes for collision detection and other environmental changes. These are triggered by updates to the *boundary* and *external* partitions. *Persist* processes within the entity implement the non-environmentally dependent functionality of the entity.

The use of a replicated database occurs in other systems. In DIVE, the database is locked before updates are made, making it possible to keep all copies identical. In VEOS, with the ability to drop updates, it becomes possible to update the database rapidly, but at a cost of keeping accurate copies. Much of the use of the database in virtual reality systems relies on it reflecting the most recent state of the system, rather than having it in agreement with other processors in the system, as reflected in the implementation of the Virtual Shared Memory in RhoVeR.

2.3.11 Decomposition strategies used in other systems

There are a number of other virtual reality systems available using distributed processing to greater or lesser degrees. With these systems there is relatively little documentation available as to the ways in which they are decomposed. This section gives such details as are available relating to the parallel decomposition strategies employed in these systems.

- Networked Virtual Reality (NVR) [Ber94] consists of a toolkit of functions for providing multi-user virtual reality applications networked using TCP/IP. Data distribution uses a replicated database, but with a client-server variation. All communication is routed through the server, which also maintains a copy of the database. The server copy is used to initialize the database for any new entrants into the virtual world. No access control is enforced, although a system of locks on objects is provided. This is controlled by the server.

- Project Isaac [Van93] is a distributed physical modelling system. Separate processes are used for the simulation of each object. Each process maintains a copy of the complete database, but is only able to modify the portion relating to its object.

- CALVIN is a virtual reality system intended for collaborative design for architectural applications [Lei96]. It runs in the CAVE virtual environment, a room with translucent walls onto which stereoscopic images are rear-projected. The database of objects is implemented as a centralized server, with each participant in the virtual world corresponding to a client.

- BrickNet [Sin95] is another system that makes use of the client-server approach. Each client runs its own virtual world, which can contain private and shared objects. Shared objects can occur in more than one world, and updates to the status of these are propagated through the server. Object behaviours and state information can be transferred via the servers. The servers also manage access control. Updates are sent asynchronously, although facilities are available to synchronize all clients that share an object.

- A distributed virtual environment developed at the Vienna University of Technology [Sch96] makes extensive use of a client-server decomposition for distribution of data and control of actors in the virtual world. A number of servers are involved in the system, each responsible for a portion of the virtual world. A range of specialized protocols for client-server interaction is used to implement the facilities required in a virtual environment such as simulation, interaction, data transfer and connection management.

- The Virtual Environment Vehicle Interface (VEVI) is intended for teleoperation of remote robot vehicles [Pig95] [Pig96]. It allows a number of geographically dispersed participants to monitor and operate the robot. Data is transferred using message passing between processes, in a stateless manner that does not leave the system vulnerable to failure of one component.

- Some communication strategies for distributed virtual reality systems are proposed in [Dem96]. Objects are classed as static, inactive or active, according to the time at which they last moved. One strategy requires that the renderer provide a time interval when requesting data from objects. Objects only reply if they have moved during this interval. An alternative

approach requires that objects update the database of the renderer after changing position. These strategies assume that only a single viewing process requires the position data of the other objects.

- The spatial model for interaction is implemented in a distributed virtual reality system [Gre94]. This model provides the notion of auras which describe which objects can be perceived at any time (see section 2.3.5). A collision manager monitors objects in the world, and notifies each object when its aura overlaps with that of another. Peer-to-peer communication between processes then allows further transfer of information. In this way each object only has to consider others within its perceptual range.

- The VETTNet system [Nyg94] provides for data sharing by using a blackboard. This is a centralized database server that clients can write data to and read data from. Current applications use only one blackboard, although a number may be used if the data is amenable to partitioning.

- A Shared Virtual Environment (SHAVE) based on the Linda tuple space paradigm (see section 2.3.10) is described in [Ams95]. As was initially done with VEOS, protestations are made as to how performance issues are not important when prototyping a system. Information can be obtained in two ways. If the required data is in the tuple space then it can be simply fetched. Otherwise a request is placed in tuple space for another process to respond to, also through tuple space. Tuple space communication is found to limit performance, and facilities for direct peer-to-peer communication are included. Extensions to the system are proposed in the form of extra processes to provide physical simulation and object behaviours.

- The GreenSpace project [Man95] has developed *GSnet* as the network communication layer. Citing issues of scalability as reasons to avoid client-server decompositions, *GSnet* uses multicasting to maintain a replicated database. Only the portions of the database relevant to the local objects are cached.

- A distributed virtual reality system is described in [Cod92] applied to a two-user physical simulation. The virtual world consists of a set of cooperating clients and servers communicating asynchronously. A Dialog Manager acts as a client, receiving data from the device servers and sending it to application and output devices. Events can be passed between multiple Dialog Managers to create a multi-user virtual world. The physical simulation (a flexible modelling simulator including gravity, friction and collision detection producing *Rubber Rocks*) is implemented as an extra server in the system, communicating with the Dialog Managers.

- ExploreNet [Hug95] implements a distributed system using the DIS approach of replicating the simulation on each node, and using updates to keep them synchronized. Communication in practice is directed through an ExploreNet server, which rebroadcasts messages to the other nodes in the world (using peer-to-peer connections). The server also orders the messages, ensuring a consistent world view amongst all participating nodes.

- The VLNET system [Pan95] uses a completely connected communication topology to distribute data amongst the users in the world. Updated information about each user is transmitted asynchronously to all other users. The environment is initially determined by one user, but is replicated at each node in the system. The concepts of aura and nimbus (see section 2.3.5) are used to limit the transmission of updates to nearby users only.

- The Waterloo Virtual Environment System (WAVES) [Kaz93] addresses the problems of creating large distributed systems without having to broadcast updates to every node. Communication between processes is managed by a number of Message Managers. These can be provided with filters to identify a class of messages that a particular process wants to receive. Data is normally transmitted via the Message Managers, but direct peer-to-peer connections can be set up where the managers deem appropriate, such as for high traffic communication.

- The Virtual Objects Interacting Dynamically (VOID) shell [OCo95] is being developed as part of the MOONLIGHT project on interactive virtual worlds. Three distribution techniques are supported in VOID. A client-server model provides for simple communication. A replicated architecture duplicates the simulation on each processor, and input data from each user is repeated to every machine. This requires that the sequence of input actions be identical on every machine. A zoned approach supports large scale applications, where entities are limited to a small set of zones, and only receive events relating to these zones.

- A number of proposals have been made regarding the Virtual Reality Markup Language (VRML) specification. These are oriented toward providing a virtual reality interface to the World Wide Web (WWW) and allowing multi-user participation in worlds distributed across the world.

  A proposal for an addressing system for these worlds [Pes94] extends the client-server approach used in the WWW still further. Each client contributes toward the population of the virtual world. Each client needs to know which other clients inhabit the neighbouring space. Cyberspace servers are able to provide this information, using an unspecified mechanism.

  A multi-user extension, made to the initial VRML specification, to allow the creation of a Virtual Society is described in [Hon96]. Virtual Society servers are separated from the conventional WWW servers, and are capable of running applications in the virtual world. Clients receive updates of object states from the server. Scripts can also be downloaded from server to client to decrease server computation.

  Two incremental changes to the current distribution mechanisms are proposed in [Bro95]. The initial change uses the current client-server system, but allows the server to relay updates of information about the other users to each client. Dead-reckoning is suggested to limit traffic. Further enhancements require the use of multicasting, to eliminate the server as a bottleneck. Access control methods suggested are either centralized, through the server, or decentralized, with control for an object residing with the object.

  The VRML 2.0 specification [VRM96] does not provide for any multi-user interaction or distributed computation. It improves on the capabilities of previous specifications, which only

allow a static scene description, by allowing scripts to be associated with objects. These scripts are triggered by events in the virtual world, such as movement of the user or the passing of time, and can be used to animate objects. While many users can access the initial description of the world on the server, all processing is performed on the client machine independently of every other client.

2.4 Summary

Table 2.1 summarizes the structural decomposition of each system. The criteria used to judge the systems are:

Architecture:	Hardware used by the system
Level of Support:	Support for virtual reality applications in terms of basic structures included in the system. This may also apply to higher level support libraries that are included in the system.
Complexity:	Support for interaction between more than one user and the ability to represent more than one virtual world in a single instance of the program.
Structural Decomposition:	A description of the components used to implement the virtual reality.
Communication strategy:	Manner in which communication between the components in the system is implemented.

The parallel decomposition strategies employed in virtual reality systems are summarized in Table 2.2. The different areas in which alternate decomposition strategies are employed are:

Database distribution:	The manner in which data is shared between the processes in the system.
Control distribution:	The manner in which tasks are coordinated amongst the different processes.
Collision detection:	The decomposition strategy used to implement collision detection between the objects in the virtual world.
Physical simulation:	The manner in which the simulation of the "reality" is coordinated.
Access control:	The mechanism used to resolve contention for objects in the virtual world.

An overview of the different database distribution approaches is shown in Figure 2.2, together with an indication of the direction in which the systems are evolving.

	Architecture	Level of Support	Complexity	Structural Decomposition	Communication strategy
AVIARY	Transputer Clusters and SUN networks	Object User World Application	Multiple worlds Multiple users	Parallel processes for objects, input, output, management and applications	Message passing (point to point)
Cyberterm	PCs and SUNs connected by modem	Object User World Application	Multiple worlds Multiple users	Processes for clients and world servers	Message passing (point to point)
NPSNET DIS SIMNET	Workstations connected by Ethernet	Object User World	One world Multiple users	Multiple user processes interacting	Message passing (multicasting)
DIVE	Networked workstations	Object User World Application	Multiple worlds Multiple users	Application and visualizer processes	Message passing (point to point) (multicast)
Division	Networked workstations and Transputer clusters	Object User World Application	Multiple worlds Multiple users	Actors which act as applications and which control system components	Message passing (point to point)
MR Toolkit	Networked workstations	Object User World Application	Multiple worlds Multiple users	Master, slave and computational processes	Message passing (point to point)
Multiverse	Networked workstations	Object User World Application	One world Multiple users	Server process simulating the world and user (client) processes	Message passing (point to point)
RhoVeR	Networked workstations	Object User World	Multiple worlds Multiple users	Processes for objects, worlds and for system control	Message passing (point to point)
VEOS	Networked workstations	—	—	Entity processes and grouple space	Message passing (point to point)
VR-386	Single PC	Object User World Application	Multiple worlds One user	Single component	—
VROS	Transputer clusters	Object User World	Multiple worlds Multiple users	Processes for objects, worlds, input and output devices	Message passing (point to point)

Table 2.1: Structural decomposition strategies in virtual reality systems

	Database Distribution	Control Distribution	Collision Detection	Physical Simulation	Access Control
AVIARY	Distributed databases	Control from applications	Single process	World attributes and application control	—
Cyberterm	Centralized databases (Client-Server)	Control by server, with dead-reckoning	On server	—	By server
NPSNET DIS SIMNET	Replicated databases	Control by object	Distributed amongst objects	Distributed amongst objects	Resolved by the object
DIVE	Distributed database (synchronized)	Control by applications	Distributed amongst applications	Distributed amongst applications	—
Division	Replicated databases	Control by application actors	Single process	Distributed amongst applications	—
MR Toolkit	Replicated databases, Distributed databases (Message Passing)	Control by master, token passing	Round robin sharing	Round robin sharing	Token passing
Multiverse	Data in single process	Single process	Single process	Single process	—
RhoVeR	Replicated database, Distributed databases (ShapeData, Message Passing)	Control by object processes	Single process	—	By object via virtual shared memory
VEOS	(Centralized) Grouplespace, Distributed databases (Message Passing)	—	Distributed amongst objects	—	—
VR-386	Single process	Single process	Single process	Single process	Single process
VROS	Centralized databases (Client-Server)	Control by object processes	Distributed amongst objects	Distributed amongst objects	By object

Table 2.2: Parallel decomposition strategies in virtual reality systems

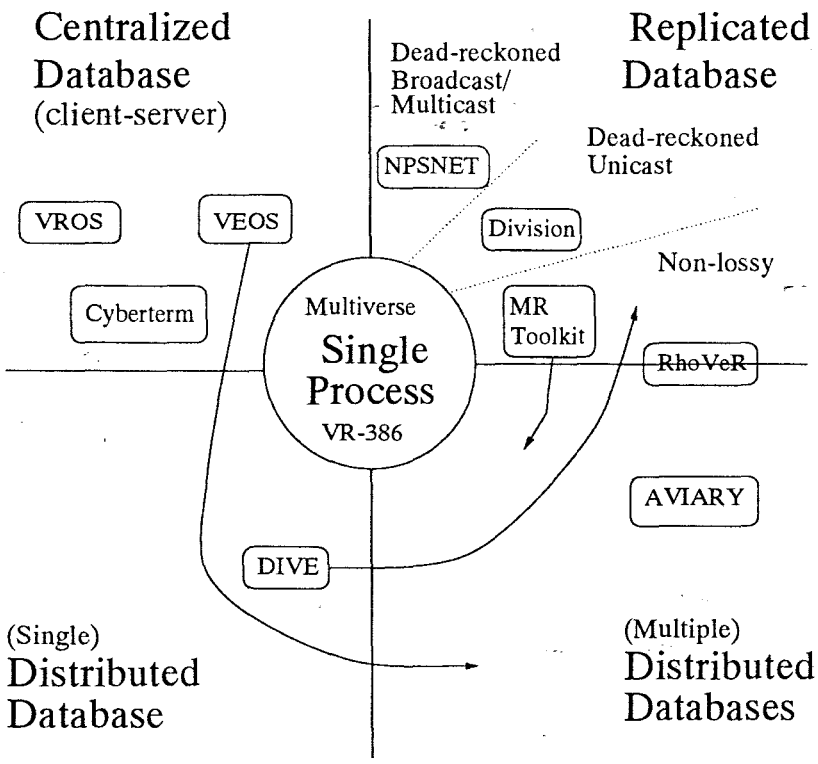

Figure 2.2: Overview of the database distribution approaches

2.5 Conclusion

Examination of a variety of parallel and distributed virtual reality systems reveals a number of features common to all systems. These are implemented in different ways.

The systems identify the concept of an object, with objects grouped into worlds. The objects are generally controlled in some way so as to respond in a realistic manner to the other objects and to the nature of the world. Some of the systems described have the means to enforce this control on objects. Control of the objects uses two approaches; either each object has its own controlling process, or the system supports application processes which can manipulate groups of objects.

It is especially noticeable that almost all systems make use of some degree of parallel processing. The extent of this varies with the implementation architecture. As a result some form of data sharing is necessary. The ways in which this is done are very much dependent on the bandwidth available for communication. Systems with slow links may use prediction to estimate the position of other objects, while faster communication allows data to be shared whenever necessary.

The problems of the distributed architecture are not limited to data distribution alone. Synchronization and coordination of the components of the system must also be performed. Apart from the interactions manifested visibly in the virtual world, such as contention for objects, parallel processing is also used in underlying levels such as collision detection and simulation of the physical mechanics of the virtual world.

Various techniques recur in these systems. The client-server system used in the VROS is used

in Multiverse, Cyberterm and in the device support layer of the MR Toolkit. A replicated memory strategy is used in the MR Toolkit, the dVS system and in DIVE. The master-slave system used in the MR Toolkit has a similar construct in the Virtual Environment Manager of AVIARY. The dead-reckoning approach crucial to the use of DIS occurs in other systems as well.

These virtual reality systems distribute their data and control in a number of different ways. The most common approaches to distributing data are:

- Keeping data local to a process with that process and distributing it to other processes when needed (Distributed databases).
- Keeping data in a central server process which distributes it to clients when necessary (Centralized database).
- Keeping all data with all processes and keeping it consistent by broadcasting updates (Replicated database).

The most common approaches to distributing control are:

- Each object is associated with a process which controls only that object.
- Objects are controlled by events from a number of processes, where each process implements a particular application. Each application process can send events to a number of objects.

The two most popular approaches are the client-server approach (due to its ease of implementation), and the replicated database (because of its ability to scale to large networks by introducing broadcasting and dead-reckoning).

These are examples of the techniques which are categorized, benchmarked and extended in the following chapters.

Chapter 3

Performance analysis of parallel systems

Performance analysis of parallel systems is a well established field with a number of well documented approaches. Analysis of parallel and distributed virtual reality systems requires specific functionality from a performance analysis technique. Related work in performance analysis is described in this chapter. The most likely candidates for use with virtual reality systems are applied to a simple client-server system to assess their suitability.

The goal of this chapter is to find a method that allows a comparison of the decomposition strategies used in the virtual reality systems presented in Chapter 2. This comparison must make use of the measures associated with virtual reality systems, namely latency and cycle time. These metrics are defined in section 2.1. The methods that satisfy most, or all of these criteria are described here in detail. Other related work in performance analysis is described more briefly toward the end of this chapter.

3.1 Requirements of a performance analysis technique

The ideal analysis technique for comparing decomposition strategies should possess a number of characteristics, making it feasible for use during the design stage of a distributed virtual reality system. It should be simple to use. There should be a close correlation between the approach being tested and the model of the approach used for analysis, and the relationship must be easy to see. The analysis should be sufficiently mechanical that no great expertise be required. Instead the effort should go into interpreting the results and modifying the design. The analysis should be quick, so as not to discourage testing of alternative models. It must be possible to use the results of the analysis to provide a clear comparison of the different approaches being modelled.

In summary, the desired technique must:

- Provide measures of latency and cycle time.
- Provide an analysis of the decomposition strategies used for virtual reality systems.
- Produce results suitable for comparison purposes.
- Perform analysis rapidly.
- Use models that are easy to create.
- Produce results that are easy to interpret.

The different methods examined in detail in this chapter are used for the analysis of a simple client-server system with two clients. The client-server approach is used as a decomposition strategy in many virtual reality systems, such as Cyberterm (see section 2.3.3) and the VROS (see section 2.3.1).

3.2 Naïve Analytic Modelling methods

In the case of simple parallel programs, various performance figures can be derived from analysis of the program code itself, as illustrated in [Li95], or from a simple model of the system [Dem96] [And91]. An example of an instance of this sort of analysis is shown below for a simple client-server system.

The following code fragment shows the form of a simple client-server system.

```
PROCESS client(i)
    WHILE (running ())
        SEND (server, REQUEST)
        RECEIVE (server, REPLY)
        Perform some computation on the results

PROCESS server
    WHILE (running ())
        RECEIVE (client(i), REQUEST) for some i
        Process this request
        SEND (client[i], REPLY)
```

A specific instance of this system is selected for trial solution by the different analysis techniques examined in this chapter. The case with two clients and a single server is chosen. The time to perform sequential computation in the client is modelled by the variable X, and the time required for the server to generate the response to a request is modelled by the variable Y. Communication is assumed to be blocking (both send and receiving process must wait until the other is able to take part in the communication), and takes a period C to complete. Thus the complete system can be described by the following outline:

```
process client(1)
    repeat
        send request to server
        receive reply from server
        process for period X
    forever

process client(2)
    repeat
        send request to server
        receive reply from server
        process for period X
    forever

process server
    receive request from client(i)
    process for period Y
    send reply to client(i)
```

The symmetry of the system suggests that all clients must perform equally in the stable state. The performance factor of interest is the time taken between the start of a client cycle and the end. Assuming that input and output occurs at these points respectively, this gives both cycle time (time taken for a single cycle) and latency (time between input and output) for the client process. The analysis of this system looks at two situations:

1. The server is saturated:
   In this case the server never blocks waiting for a client process to send a request as there is always an incoming request waiting. The cycle time of the server (SCT) can easily be seen to be $C+Y+C$. The cycle time of the client (CCT) is $\delta+C+Y+C+X$, where δ is the period the client blocks waiting for the server. In one iteration of a client, the server must have serviced every client. Thus CCT= $n\times$SCT, where n is the number of clients. Thus, for this case:

$$\begin{aligned} 2(C+Y+C) &= \delta+C+Y+C+X \\ &\geq C+Y+C+X \\ 2C+Y &\geq X \end{aligned}$$

   To derive the value of CCT:

$$\begin{aligned} 2(C+Y+C) &= \delta+C+Y+C+X \\ \delta &= 2C+Y-X \\ CCT &= 2C+Y-X+C+Y+C+X \\ &= 4C+2Y \end{aligned}$$

2. The server is not saturated:

   In this case, it is the client that does not block, and which determines the cycle time. The cycle time for the client is $C+Y+C+X$ and that for the server is $\delta+C+Y+C$, where δ represents that time spent blocking by the server. This situation occurs when:

$$\begin{aligned} 2(\delta+C+Y+C) &= C+Y+C+X \\ 2(C+Y+C) &\leq C+Y+C+X \\ 2C+Y &\leq X \end{aligned}$$

This analysis can be extended to the case of n clients fairly easily, by using the relationship CCT $= n \times$ SCT to calculate the inequalities as above.

The limitations of this approach are the difficulty in deriving a solution for an arbitrary approach, and the number of simplifying assumptions required to enable a solution to be found. The derivation given above is suitable only to simple client-server systems. Introduction of a variation, such as an additional server, would require that all analysis be repeated. In addition, as the model gets more complicated, the amount of effort required to identify the behaviour increases substantially. In effect, naïve analytic modelling requires that a new performance analysis technique be created for every problem. This tends to prohibit casual use.

The advantage is that the symbolic form of the results is not limited to one specific set of variable values, and can often be used to understand the relationship between the variables involved in the model. The symbolic nature of the output allows the exact dependency of the performance on a particular variable to be seen. In addition it supports performance comparison in that trends in the results, rather than just sampled numerical values, can be compared. This is particularly relevant for comparison across platforms, where the values of system parameters may vary.

Analytic solutions are suitable for use with virtual reality systems if a systematic analysis procedure can be developed that produces the required metrics.

3.3 Simulation methods

Performance estimates can obviously be obtained by implementing a chosen model. Alternatively, a subset of the model can be implemented in a simulation environment.

An interesting approach using a simulation technique is the orbit model [Wlo95] used to evaluate latencies resulting from synchronization lag (see section 2.1). A process must block when passing data on if the next stage is busy with computation. The length of time spent blocking affects processes further on. Simulating this process produces strings of possible lags at each stage. Cycles of these values occur after a number of items have passed through the system, producing *orbits* whose values represent statistically relevant latencies. The number of times the value orbits is a measure of its significance.

A model of the client-server system used in a simple simulator is presented below. The model is specified in a C-like language. Included in the description are details of the variable values for which simulation is performed (C from 1 to 25 in steps of 5, X and Y from 1 to 25 in steps of 1). The simulation is run for 10000 time units for each case. The time values for this simulation

are discrete and constant, although it is easy to perform simulation using random variables with a specific distribution. A point is designated for the measurement of latency and cycle time.

```
void client (int pid)
{
    Message message;
    while (running ())
    {
        send (server, 1, 1, NULL, 0);
        message = receive (0);
        process (x);
        if (pid == 1)
        {
            addlatency (syslat, message);
            addrate (sysrat, message);
        }
        dropmessage (message);
    }
}
void server (int pid)
{
    Message message;
    int sourcepid;
    while (running ())
    {
        message = receive (0);
        sourcepid = message->sourceid;
        process (y);
        resend (message, client, sourcepid, 1, NULL, 0);
    }
}
void startsim ()
{
    syslat = latency ("System Latency");
    sysrat = rate ("Frame rate");
    num = 2;
```

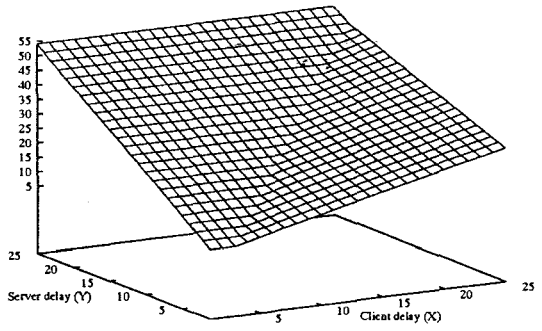

Figure 3.1: Results of the simulation for C=1

```
    runsim ("C", &c, 1, 25, 5);
    runsim ("X", &x, 1, 25, 1);
    runsim ("Y", &y, 1, 25, 1);
}

void init ()
{
    int i;
    for (i = 1; i <= num; i++)
    {
        startup (client, i);
    }
    startup (server, 1);
    setdefaultchannel (1, 0, c);
    stopat (100001);
}
```

The values for cycle time and latency are almost equivalent. A small difference between the two arises from a one cycle transient at startup. The effect of this is diminished by running the simulation for a number of cycles and using the average result. The results are shown in Figures 3.1 and 3.2, giving the results for all simulated values of X and Y, for two values of C. The results agree with the analytic values obtained in section 3.2, but are limited to the values explicitly simulated.

Simulation is sufficiently general to be applied to all the parallel decomposition strategies used in virtual reality systems. In addition it can be used to generate values of the metrics which characterize virtual reality systems.

This approach can allow experimentation with different parameters for the system components. The output of the simulation applies only to the particular parameters used to drive the simula-

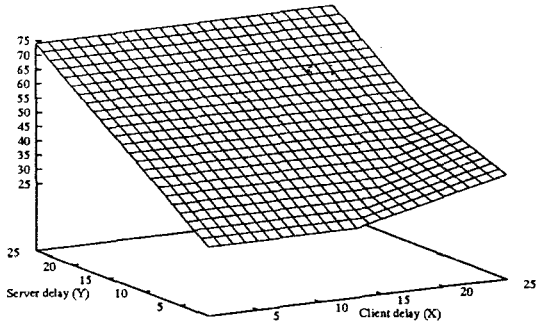

Figure 3.2: Results of the simulation for C=6

tion. It does not, however, provide an indication that the chosen values cover areas of "interesting" behaviour. The results are applicable only to variable values used in the simulation and extrapolation to other situations is limited. The suitability of simulation to performance comparison is thus limited.

Simulations are normally run only for a limited period, simulating a finite number of instructions. All of the possible interactions between the components of the model being simulated may not occur before the simulation ends. Behaviour of the model that does not manifest during this period does not affect the results. On the other hand, the effect of startup transients and other spurious behaviour is incorporated into the results. This influences the accuracy of the results produced by simulation.

3.4 Abstract Modelling

Another alternative to the approaches already described is to transform the model into an abstract form which has a well-established analysis method. The approaches examined are Petri Nets and Data Flow Diagrams.

3.4.1 Petri Nets

An introduction to Petri Nets and their workings is given in Appendix A.

By extending the basic model of the Petri Net to include timed transitions, it becomes possible to analyse the performance of systems. The analysis of Stochastic Petri Nets is based on queueing theory and so depends heavily on statistical distributions and firing probabilities. These are often not easily derived from the description of the system being modelled. The solution of a Stochastic Petri Net model gives the probabilities of being found in various states. These can be used to determine values such as utilization and throughput. These measures are not easily translated into the particular values that are required in virtual reality systems, namely cycle times and latency.

The results produced by analysis of Petri Nets usually relate to certain properties of a system, for example, absence of dead lock and boundedness.

The difficulty of solving a Petri Net increases dramatically as the complexity of the model increases. This tends to make analytical solutions impractical for all but the simplest cases [Bal91].

An alternative method of solution for Petri Nets is simulation, which does provide the type of output required. In this case there is a trade off between the overhead of creating the Petri Net model versus the use of simulation on a more conventional model. In this case it may be simpler to choose the model representation that most closely resembles the system to be implemented. The difficulty of translating to the Petri Net formalism must be weighed against the large formal background of the field, and the well established methods for solving Petri Nets.

An example of the solution of a simple client-server system with two clients and a single server using Petri Nets is given below. This example provides an analysis of the same system previously examined in section 3.2. Communication delays are ignored.

A Petri Net implementing this algorithm is shown in Figure 3.3, where the places and transitions correspond to the following states and actions:

P1 (P3) Client1 (2) about to ask for data

P2 Server free

P4 (P6) Client1 (2) waiting for response to request

P5 Server busy

P7 (P8) Client1 (2) has data and can perform some computation

T1 (T2) Client1 (2) sends a request

T3 (T4) Client1 (2) receives a reply

T5 (T6) Client1 (2) performs sequential computation

To obtain useful results from this model, it is assumed that the server takes period Y to respond to a request, and each client spends a period X performing some sequential computation. This corresponds to transitions T3 and T4 having a firing time of Y and T5 and T6 having a firing time of X. For the purposes of this example, it is assumed that the firing times are random variables with an exponential distribution. It is possible to assume other distributions, but this leads to considerable increase in the complexity of the analysis.

The first step in the analysis of the net is to draw up the reachability graph. This shows the relationships between the markings that can be reached from the given initial marking, diagramming the state space of the model. These markings are shown in Table 3.1. The markings are written as the number of tokens in each of places 1 to 8.

Each state can be classified as either vanishing or tangible. The tangible markings are those having only timed transitions (T3, T4, T5, T6) enabled, whereas the vanishing transitions are capable of firing instantaneously. The reachability graph is shown on the left of Figure 3.4.

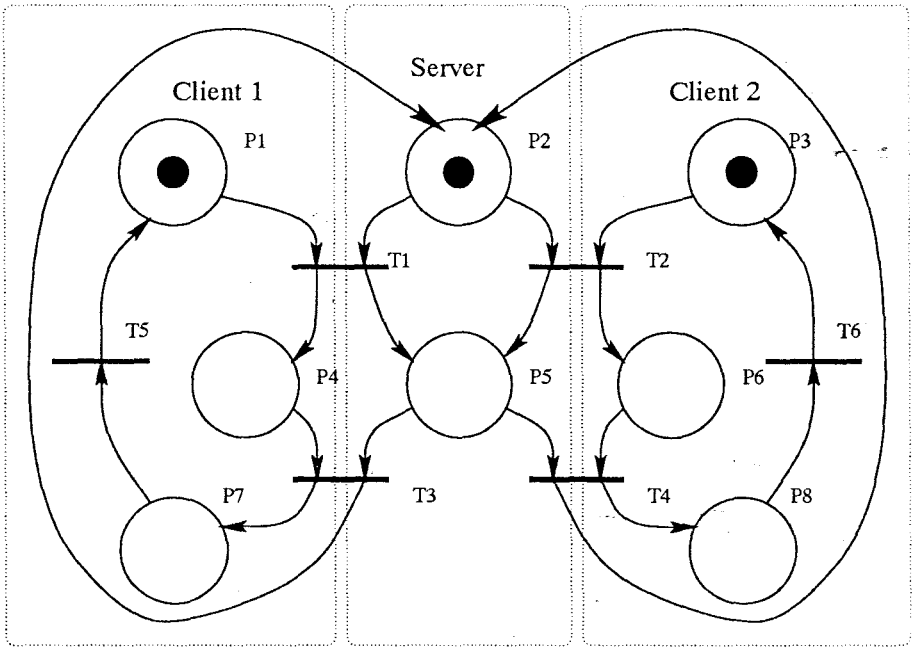

Figure 3.3: Petri Net for the client-server system

State	Marking	Enabled Transitions	Type
1	1.1.1.0.0.0.0.0	T1(4) T2(5)	Vanishing
2	0.1.1.0.0.0.1.0	T2(4) T5(1)	Vanishing
3	1.1.0.0.0.0.0.1	T1(8) T6(1)	Vanishing
4	0.0.1.1.1.0.0.0	T3(2)	Tangible
5	1.0.0.0.1.1.0.0	T4(3)	Tangible
6	0.0.0.0.1.1.1.0	T4(7) T5(5)	Tangible
7	0.1.0.0.0.0.1.1	T5(3) T6(2)	Tangible
8	0.0.0.1.1.0.0.1	T3(7) T6(4)	Tangible

Table 3.1: Possible states of the Petri Net

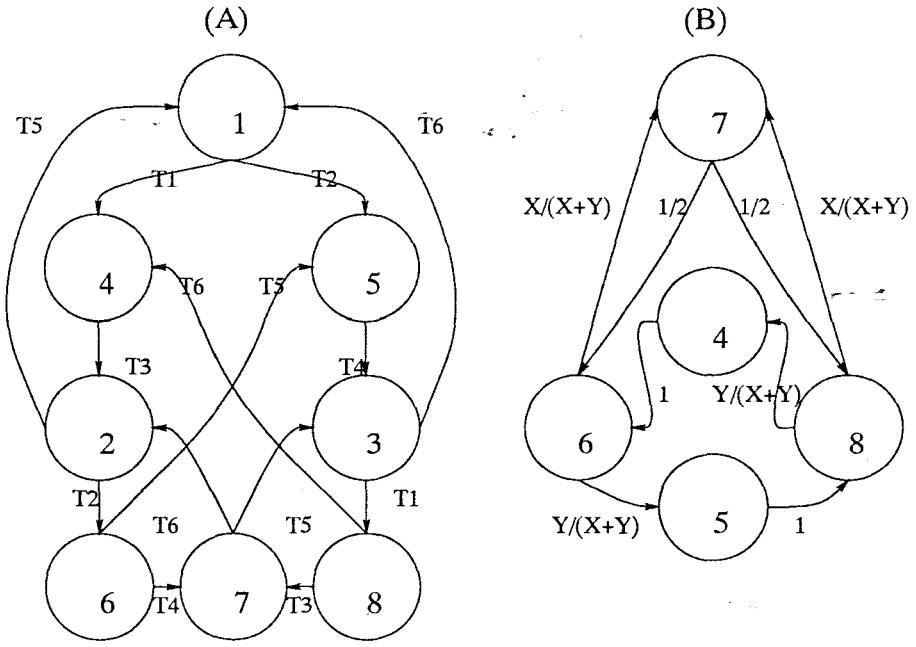

Figure 3.4: Reachability graphs of the client-server system

The solution technique requires solving the continuous time Markov chain for the model. This requires first solving the discrete parameter Markov chain embedded at the transition points of the continuous time version. Simply put, a Markov chain consists of a series of states and transition probabilities, where the probabilities at any state are independent of history. This memoryless property requires the exponential distribution mentioned earlier in this section. Discrete Markov chains can be specified by the matrix of transition probabilities.

The transition probability matrix for the embedded discrete parameter Markov chain is given in Table 3.2. The transition probabilities are easily derived. In the case where there is only one path from one state to the next, or only one immediate transition, the probability of that path being taken is one. Where there are two timed transitions with equal delay, the probability is $\frac{1}{2}$ for each path. In the case where there is a transition T_X takes period X to fire and a T_Y takes period Y to fire, the probability of T_X firing first is calculated by:

T_X fires at frequency $\frac{1}{X}$

T_Y fires at frequency $\frac{1}{Y}$

Since the exponential distribution corresponds to memoryless transition probabilities, the probability of T_X firing first is:

$$P(T_X) = \frac{\frac{1}{X}}{\frac{1}{X} + \frac{1}{Y}} = \frac{Y}{X+Y}$$

	1	2	3	4	5	6	7	8
1	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	0
5	0	0	1	0	0	0	0	0
6	0	0	0	0	$\frac{Y}{X+Y}$	0	$\frac{X}{X+Y}$	0
7	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0
8	0	0	0	$\frac{Y}{X+Y}$	0	0	$\frac{X}{X+Y}$	0

Table 3.2: Transition probability matrix

	4	5	6	7	8
4	0	0	1	0	0
5	0	0	0	0	1
6	0	$\frac{Y}{X+Y}$	0	$\frac{X}{X+Y}$	0
7	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$
8	$\frac{Y}{X+Y}$	0	0	$\frac{X}{X+Y}$	0

Table 3.3: Compacted transition probability matrix

To simplify calculations it is possible to compact the embedded Markov chain to remove vanishing states. This process is described in [Kan92] on pages 440 - 442. The required chain is given by:

$$A = Q'_{TT} + Q'_{TV}\left[I - Q'_{VV}\right]^{-1} Q'_{VT}$$

where Q'_{VV} is the portion of the original transition matrix translating vanishing states to vanishing states, Q'_{VT} is the portion translating vanishing states to tangible states, Q'_{TV} is the portion of the matrix translating tangible states to vanishing states and Q'_{TT} is the portion translating tangible states to tangible states.

Performing this compaction results in the transition probability matrix in Table 3.3 which can be represented by the reachability graph on the right of Figure 3.4.

The states of the compacted chain may be interpreted as described in Table 3.4.

The stationary distribution of the embedded chain can be found by solving the balance equation

State	Interpretation
4	Client 2 waiting for server, Client 1 using server
5	Client 1 waiting for server, Client 2 using server
6	Client 2 using server, Client 1 performing computation
7	Client 1 and Client 2 performing computation
8	Client 1 using server, Client 2 performing computation

Table 3.4: Interpretation of the states in the compacted Markov chain

$D' = D' \cdot Q$. Here $D' = [d'_4, d'_5, d'_6, d'_7, d'_8]$ is the required stationary distribution and Q is the transition probability matrix given above.

The solution, subject to the constraint that $d'_4 + d'_5 + d'_6 + d'_7 + d'_8 = 1$, yields the following:

$$d'_4 = \frac{Y}{4(X+Y)}$$
$$d'_5 = \frac{Y}{4(X+Y)}$$
$$d'_6 = \frac{1}{4}$$
$$d'_7 = \frac{X}{2(X+Y)}$$
$$d'_8 = \frac{1}{4}$$

The solution to the original Markov chain may be found by [Kan92]:

Let T be the set of tangible states, let i denote some tangible state, let H_i be the set of timed transitions enabled in this state and let $r_j(i)$ be the mean firing rate of transition t_j in state i.

$$\sigma_i = \frac{1}{\sum_{j \in H_i} r_j(i)}$$

$$d_i = \frac{d'_i \sigma_i}{\sum_{k \in T} d'_k \sigma_k}$$

Thus:

$$\begin{aligned}
\sigma_4 &= Y \\
\sigma_5 &= Y \\
\sigma_6 &= \frac{XY}{X+Y} \\
\sigma_7 &= \frac{X}{2} \\
\sigma_8 &= \frac{XY}{X+Y}
\end{aligned}$$

$$\sum d'_i \sigma_i = \frac{2Y^2 + 2XY + X^2}{4(X+Y)}$$

$$\begin{aligned}
d_4 &= \frac{Y^2}{2Y^2 + 2XY + X^2} \\
d_5 &= \frac{Y^2}{2Y^2 + 2XY + X^2} \\
d_6 &= \frac{XY}{2Y^2 + 2XY + X^2} \\
d_7 &= \frac{X^2}{2Y^2 + 2XY + X^2} \\
d_8 &= \frac{XY}{2Y^2 + 2XY + X^2}
\end{aligned}$$

The tuple $D = [d_4, d_5, d_6, d_7, d_8]$ represents the state probabilities for the original Markov chain. This is used to produce performance figures by using the probabilities to calculate a weighted sum of the resource usage in each state.

Some examples of this solution used to calculate occupancy of client and server follow (see the interpretation of each state given in Table 3.4):

- $X = 0, D = [\frac{1}{2}, \frac{1}{2}, 0, 0, 0]$ With no time required to perform computation, the clients have equal share of the server, and spend as much time waiting for the server as using it.
- $X = Y, D = [\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}]$ Clients spend $\frac{2}{5}$ of their time using the server, $\frac{2}{5}$ performing sequential computation and $\frac{1}{5}$ waiting for the server.
- $Y = 0, D = [0, 0, 0, 1, 0]$ Since the server never blocks, clients spend all their time in sequential computation.

Analytical solutions for Petri Nets are possible, although the complexity grows rapidly with the size of the model; most large models tend to be solved numerically [Ibe93] [Has93]. This requires substitution of values for the operating constraints, limiting the generality of the solution.

Analytical solution of Petri Nets is not well suited to deriving latency values for virtual reality systems. By their nature, the places represent states of system components. This paradigm does not translate easily into the need to time data flow through a portion of the system. Tracing messages is better suited to some form of simulation. Queueing theory based approaches are better able to produce measures based on resource utilization such as throughput, and response time.

The translation from a system specification to a Petri Net model involves a paradigm shift from considering event sequences to considering state transitions. Creating the models involves a degree of expertise in the formalism. Verification of the behaviour of the model against the specification is non-trivial. Case studies using Petri Nets tend to concentrate on models of a single system or approach for analysis purposes, rather than performing comparative studies.

Petri Nets have the advantage that they are easily applied to modelling parallel and distributed systems and that a large formal background has been built up with many case studies illustrating their use. The field is highly fragmented, with numerous variations on the basic Petri Net model, each created for use in a specific application area.

3.4.2 Data Flow Diagrams

This section describes the work of [Som93] which reports on a system to calculate execution time and iteration period of programs on a data flow architecture. An introductory description of data flow concepts is given in Appendix A. This approach deals with real-time systems where performance measurement concentrates on the time taken for a single iteration, since programs usually do not terminate.

The measures used are iteration execution time (the time between arrival of input and the corresponding output, which is equivalent to latency) and iteration period (the time between successive outputs which is equivalent to cycle time).

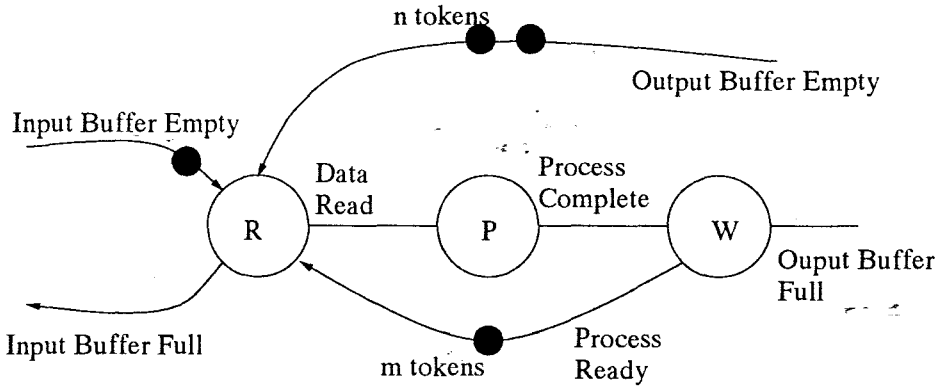

Figure 3.5: NMG model

The Algorithm to Architecture Mapping Model (ATAMM) developed in [Som93] describes a strategy to map algorithms onto multiprocessor architectures in such a way that it is possible to predict system time performance as well as processor requirements. Starting with a standard Data Flow Graph (DFG) representation, the model develops an Algorithm Marked Graph (AMG), with nodes representing algorithm tasks and arcs for communication paths, and with source and sink nodes for data differentiated.

A Node Marked Graph (NMG) is a data flow graph representing the execution process in a processor. This graph displays flow of control and data for the three processes of input, computation and output. The notable difference from the firing rules of a normal DFG is the requirement that a processor be free before any input can occur. An example NMG is shown in Figure 3.5. The initial marking of m tokens on the *Process Ready* arc allows for instantiation on up to m processors. For static architectures m is set to one. The n tokens in the Output Buffer Empty allows limited repetition of the execution of the NMG before the output is consumed.

These two graphs are combined into the Computational Marked Graph (CMG) by a process of:

- Copying source and sink nodes in the AMG to become source and sink in the CMG.
- Replacing nodes in the AMG that correspond to algorithm tasks with NMGs.
- Replacing arcs with arc pairs, one forward for data flow, the other backward for control flow.

To execute this enhanced data flow graph, which now explicitly contains all necessary control and data flow, the ATAMM model includes a queue of processors and a graph manager. When a read node in the CMG is enabled, the graph manager assigns the algorithm task to the processor at the head of the queue. After the task is complete, the processor is returned to the tail of the queue. The graph manager also controls parallel access to shared resources. This processor assignment strategy allows processors to be easily added and removed from the processor queue.

Given a CMG with a specific initial marking, there are greatest lower bounds imposed by the graph on the execution time and iteration period. With sufficient processors, performance at those bounds may be achieved, while performance below these bounds is not possible.

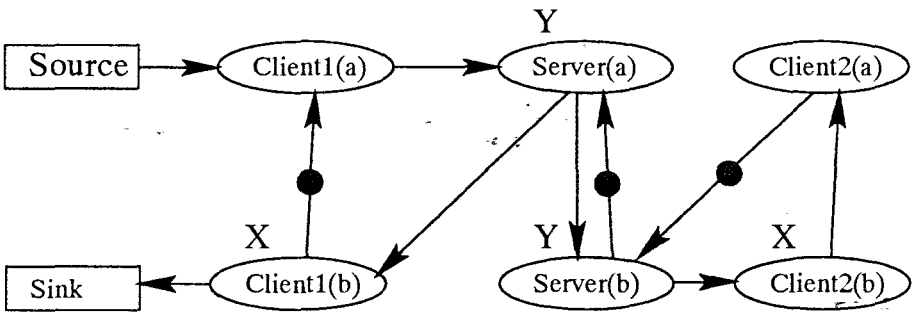

Figure 3.6: DFG for the client-server system

The greatest lower bound on execution time is determined by the longest directed path from input source to output sink. The greatest lower bound on iteration period is the largest time per token of all directed loops. A directed path is a connected, alternating sequence of nodes and arcs directed from source to sink, and with each node occurring at most once. A directed loop is a directed path with the same initial and terminal node. If the critical loop occurs within an NMG, then the lowest iteration period can be reduced by adding more tokens, equivalent to allowing the number of instantiations of that node to increase.

It is also possible to calculate the processor requirements to achieve a given performance level. This uses Single Graph Play (SGP) and Total Graph Play (TGP) diagrams. These concepts inspired the development of the Analytical Simulation technique described in Chapter 4.

The SGP diagram displays execution as a function of time, assuming infinite processors. Node activity is indicated by a solid line for the intervals during which it occurs. Activity of different processes is separated vertically on the same horizontal time axis. This diagram easily shows the maximum number of processors required, by taking the maximum number active at any one time. This value is the minimum required to achieve the minimum execution time.

The TGP diagrams resembles the SGP, but displays execution when the graph is executed with a period P. This is created from the SGP, by dividing it into time intervals of width P, and stacking each of the intervals. Algorithm iterations are numbered. The maximum number of processors required to execute the graph repetitively with period P can be found from the largest number of executing nodes in any time interval.

Applying this method to a simple client-server model with two clients produced a number of problems. The Data Flow Graph representation might not correspond exactly with the models produced in previous sections. Each task, corresponding to a node of the graph, does not have to be executed on the same processor during each cycle. This could have implications for the interpretation of the server process as residing on the processor containing its database. A Data Flow Graph of the model is given in Figure 3.6. In this case, the two server processes are constrained to execute sequentially, and so can be limited to a single processor without affecting the results of the analysis.

The result of interest is the performance of a single client. Its operation is affected by the other client. The graph cannot be decomposed into a history independent system since performance changes with history. Cycles are thus introduced which complicate analysis.

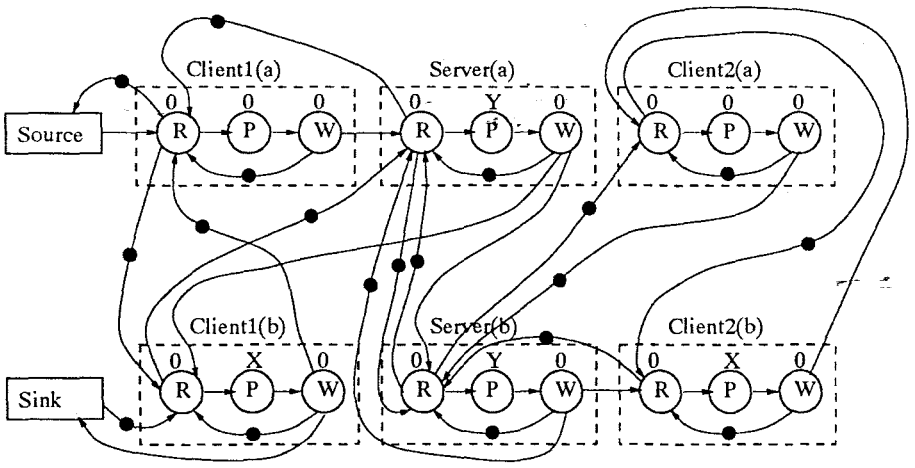

Figure 3.7: CMG for the client-server system

Creating the CMG as required produces the graph shown in Figure 3.7. The greatest lower bound on execution time is given by the longest directed path from source to sink. Since no node may appear twice on a directed path, this is the path from source through *Client1(a)*, *Server(a)* and *Client1(b)* which has a path length of $X + Y$. Iteration period is given by the largest time per token of all directed loops. The two largest directed loops are

- *Server(a)*, *Server(b)*, with length $2Y$ and one token.
- *Server(a)*, *Client1(b)*, *Client1(a)* with length $X + Y$ and one token (or its image using *Client2*).

This gives iteration period as $2Y$, if $Y > X$, and $X + Y$ otherwise. The execution time is correct for the first iteration but for further iterations the blocking created by the second client must be taken into account. The mechanism used for calculating execution time applies only to acyclic graphs [Koh75]. The initial value is just a transient when $Y > X$, and so extra work must be done to determine the steady state value.

The steady state value for latency can be determined using some results for scheduling of Data Flow Graphs [Mur94]. An Acyclic Precedence Graph (APG) is constructed from the Data Flow Graph by removing all arcs containing tokens. The APG gives the intra-iteration precedences between the nodes. A J-unfolded precedence graph is then constructed, by replicating the APG J times and adding arcs from node u in the i^{th} to node v in the j^{th} copy $(i < j)$ if there is an arc from u to v in the original Data Flow Graph containing $j - i$ tokens. The APG for the client-server example is shown in Figure 3.8, a 3-unfolded precedence graph is given in Figure 3.9.

The critical path is given by the path with the largest weight in the J-unfolded graph. It can easily be seen in this example that critical path in the J-unfolded graph from the initial source node to the final sink node is:

$X \geq Y$: $\qquad X + Y + J(X + Y)$

$Y \geq X$: $\qquad X + Y + J(2Y)$

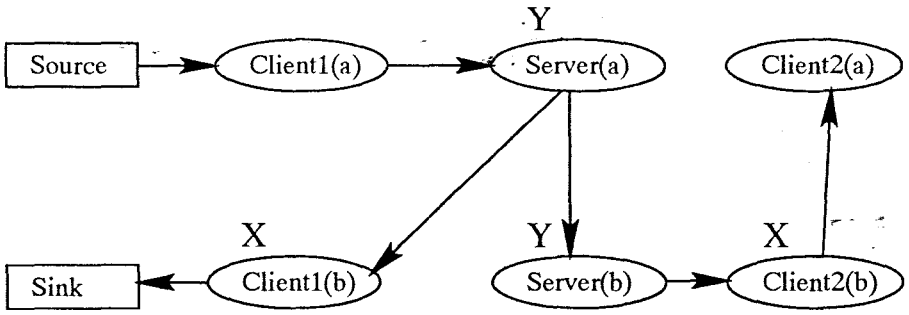

Figure 3.8: APG for the client-server system

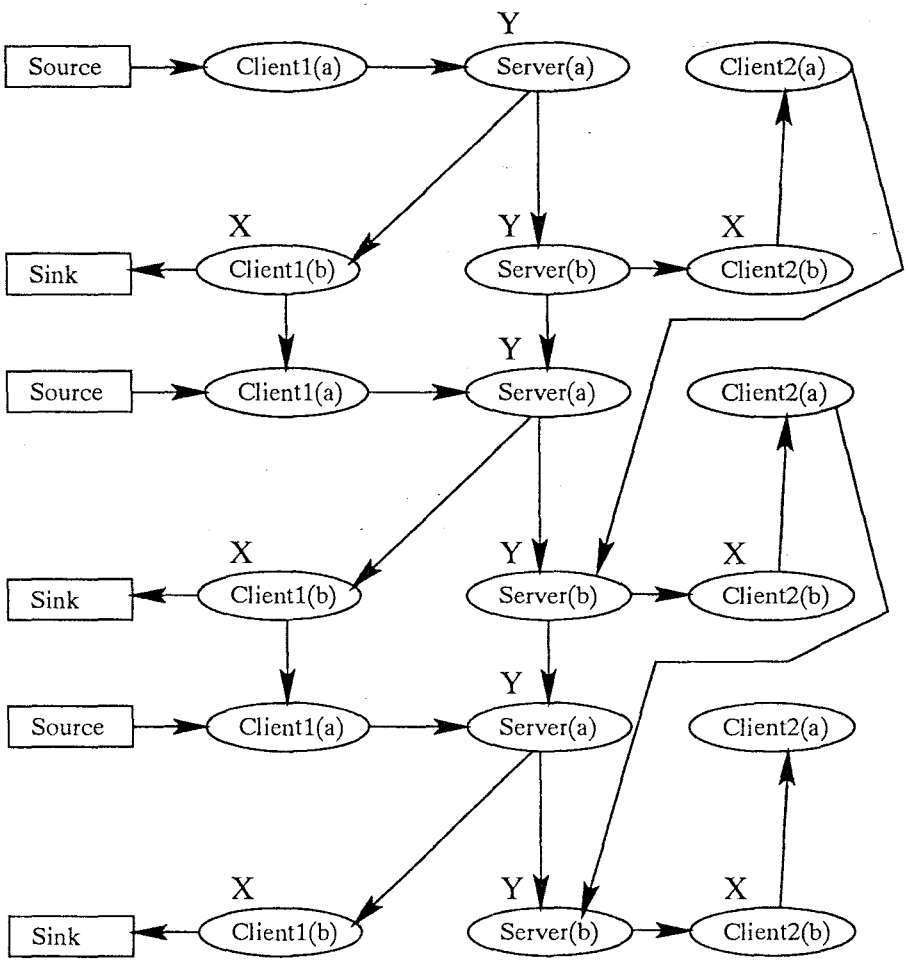

Figure 3.9: 3-unfolded precedence graph for the client-server system

The required steady state latencies can easily be derived from this result, as the coefficients of J.

This approach provides a clear indication of the optimal execution points and makes it possible to predict changes in performance given changes in the number of processors. It also finds the optimal operation point for a set number of processors. The approach is also well suited to automation. At present systems are being solved numerically, although it appears that there is scope for analytic solutions.

The chief drawback is that it is limited to data flow architectures. However, many of the ideas are used as a basis for the Analytical Simulation approach introduced in Chapter 4.

3.5 System Specific Performance Prediction

The performance analysis techniques presented so far in this chapter are suitable for the analysis of any parallel system. This section describes other work related to performance analysis of parallel systems which is more limited in its area of application. This includes techniques that are confined to a particular architecture, or limited in the form of the results that are generated.

- An approach to performance prediction for a specific class of parallel systems is described in [Mak90]. The parallel computation must be expressed as a series-parallel acyclic precedence graph. Resources are modelled as service centres in a queueing network model. The predicted system completion time is found by iteratively calculating the residence time for each task, and using these to reduce the precedence graph until the values converge within a set limit. This approach is limited to programs that can be written as the required graph. Results are numerical, and do not provide the values required in virtual reality systems.

- In a study of performance improvements in Fortran-D [Men95] an attempt is made to predict application execution time as a function of the problem size (N) and the number of processors (P). The Fortran-D compiler generates Single-Program-Multiple-Data (SPMD) code from high level directives in the Fortran code. The approach in this study involves expressing the execution time for all possible SPMD events as symbolic expressions of N and P. Constants in the expression are evaluated by measuring the performance of the system for given values of N and P. On the two example programs tested, the error in prediction varied from about 2% to 20% depending on the level of detail with which the message passing is modelled. This approach requires the assistance of the compiler producing the code, both for generating the expressions for each parallel event generated and for adding the instrumentation necessary for measuring the values of the constants. While producing analytic output, the variables that may be used are limited.

- Another paper [Adv94] also investigates improvements in the performance of programs compiled by the Fortran-D compiler. A model is developed to predict the time required for pipelined computations as a function of pipeline granularity. Using additional values dependent on the machine architecture, the model is able to predict the granularity value to minimize execution time. The model is reasonably complex, has to be created by hand, relies

on a number of simplifying assumptions and is only valid for pipelined computation. It is suggested that the compiler be responsible for implementing the model, instantiating the variables and finding the optimal granularity. In addition it is expected that the compiler should detect the violations in the assumptions and either use a corrected model or inform the user of the problem. Performance monitoring code would be automatically added by the compiler. The performance of pipelined systems is examined in Chapter 11 of this document.

- The research by [Fah93] investigates ways to measure performance of a number of program transformations and data distributions of a parallelizing compiler, and automatically select the best one. Only a single profile run is used to determine branching probabilities and other variables in data and control flow. The parameters derived from the profiling information are adapted when calculating the effects of the compiler's program transformations. This investigation also concentrates on providing useful performance information. Parameters relating to three performance aspects are examined: work load, communication overhead and data locality. These values provide greater insight into the performance of the program than a single runtime value. Based on the program parameters, the compiler can apply program transformations to program segments based on each different performance drawback. The prediction tool can be trained to order the priority of each transformation based on the target architecture. The accuracy of the predictions is usually within 5%, and in some rare cases is off by 10 to 20%.
  This work [Fah93] also contains a comprehensive classification of related work in the performance prediction field. That section also describes work done in using benchmarking for performance prediction. The performance of a set of code-fragments is measured on a given architecture. A parallel program is then analysed to detect the presence of these code-fragments, and runtime is estimated by combining the results of the previous measurements. The practical problems with implementing this approach are found to be considerable.

- The effects of porting programs to different architectures are examined by analysis of program stability in [Men93] and [Men94]. This approach uses traces of program events, the events being used are computation, message sending and message receiving. Stability is measured by determining the degree of change in the events in the trace when a disturbance is introduced. These disturbances are produced by adding time delays to portions of the program. The degree of change is found by converting each program trace into a graph and measuring the degree of overlap, found by the size of the largest isomorphic subgraph. The problems introduced by a non-deterministic receive are discussed. Analysis is restricted to "stable" programs which do not contain this construct. Non-deterministic constructs are common in virtual reality systems (and are found in the client-server model examined in this chapter), and need to be modelled.
  Prediction of performance on a new architecture is done by scaling the times taken for computation and message passing, according to the relative performance values of the two machines. An extra constraint is added to ensure that messages do not arrive before they are sent. The prediction error for the example given in [Men93] is 16%. For the examples in [Men94], the errors are under 19%, and usually range from 5% to 15%.

- Trace files can be used to generate performance indices which can provide insights into the nature of the parallel processing that is taking place [Sar94]. The indices are computed from instrumented programs run directly on the target architecture. Here performance indices are also associated with the various data structures. This makes it possible to identify the structures which most affect the performance of the system. A range of performance indices are used, with the intention being to isolate the areas in which bottlenecks are occurring.

- Another example of the use of the compiler for generating performance information is found in [Mal94]. A data-parallel compiler pC++, which generates parallel code from high level additions to C++ code, is constructed with performance analysis issues as an integral part. Profiling code is included when programs are compiled. Event tracing is also possible with this system. Shared memory and distributed memory architectures are supported.

- An alternative approach to performance measurement that concentrates on the improvements not being obtained is the lost cycle analysis as described in [Cro93]. A metric, the lost cycles, is defined to represent the overhead in the execution of a parallel program. A set of categories into which lost cycles can fall is chosen which is complete, orthogonal and meaningful for analysis purposes. One such set is described, although it is mentioned that it may need to be extended for more varied situations. Monitoring code is inserted into a program to determine the lost cycles for each category. For a given program, analytical models are constructed to predict the overhead for each category as a function of data size and number of processors. Constants are found by measuring the performance of the program in a few configurations. This model is then used to predict performance for other configurations of processors and data size. The models are constructed by hand, and vary for different architectures. Average error for a prediction of the performance of a 2D FFT is 12.5%.

The goals of the research presented here differ in a number of ways from those required for virtual reality systems. The values measured in the above studies concentrate on finding values for run-time and performance improvement through the use of parallelism. On the other hand, virtual reality systems require values for latency and cycle time, as mentioned previously. Virtual reality systems generally consist of cyclic processes, which do not terminate; this makes values such as run-time redundant.

Many of the studies mentioned concentrate on finding certain magic numbers which define the system and a model which uses these numbers. The models are usually developed by hand, and tailored to each system. This requires considerable ingenuity and effort. A consistent approach is required for different architectures if performance comparison is to be performed. Often the emphasis is on finding the best algorithm for a specific architecture, or on predicting performance when the configuration of the components of the architecture is changed. In this thesis there is greater emphasis on examining the effects of different architectures on a given algorithm.

Many approaches require that the system be implemented and provide performance metrics to optimize it once it is running. Usually these metrics require data from the running system.

Many studies refer to shared memory or SIMD architectures whereas this thesis concentrates on message passing, MIMD architectures as used by the virtual reality systems in Chapter 2.

3.6 Conclusion

This chapter reviews a number of approaches to performance prediction and analysis in parallel message passing systems. These approaches are evaluated with respect to their suitability for use with virtual reality systems.

Virtual reality systems have two important measures associated with them; latency and cycle time. The approaches have to be able to generate values for these measures. The evaluation also takes into account a number of other desirable characteristics:

- Ease of translation from system specification to model.
- Suitability of output to performance comparison.
- Ability to handle constructs found in virtual reality systems.
- Ability to scale to larger problems.

Some approaches are considered in detail:

- Analytic modelling approaches give the correct metrics, and provide results suitable for performance comparison. There is no general analytic approach that can be applied to an arbitrary architecture.
- Simulation produces the correct metrics and is suitable for arbitrary architectures. The results are limited to the values used for the simulation, and so are not suitable for comparison across architectures, or for general performance analysis.
- Petri Nets have the advantage of a substantial theoretical base, and of having been used for a number of years on a wide range of performance analysis problems. They are also able to model stochastic events. The construction of the Petri Net model is, however, not trivial. Analysis for all but the simplest models requires the use of simulation. The form of the results does not correspond well with that required for performance comparison in virtual reality systems.
- It is possible to produce the required values from a model using Data Flow Graphs. The models require some effort to produce, and certain constraints such as scheduling of processes to processors can introduce extra complications.

A number of other approaches to performance prediction in parallel systems are briefly surveyed. None of the performance analysis approaches examined in this chapter are suitable for use with parallel and distributed virtual reality systems.

Chapter 4

Performance analysis using Analytical Simulation

The performance analysis techniques examined in the previous chapter are not suitable for use with virtual reality systems. This chapter presents an approach that satisfies all the criteria for the analysis and comparison of the performance of parallel and distributed virtual reality systems. In addition, the approach presented is suitable for the analysis of a number of other classes of system.

4.1 Analytical Simulation

A technique combining aspects of existing methodologies has been devised by the author. It overcomes the limitations of the other methodologies for modelling parallel virtual reality systems. The limitation of simulation is that it is not general enough to cover results that are not explicitly simulated. Analytic performance modelling on the other hand is either ad hoc, does not provide the desired performance measures, or does not support the required architectural features.

This approach is introduced by way of the simple example used in Chapter 3 (see section 3.2). Consider the client-server system with two clients which each send a request to the server, get a reply, and process it for period X. The server, on the other hand, picks up a request, takes period Y to service it, and returns a reply. When using this example to illustrate the performance analysis techniques in the previous chapter the result depended on the relative sizes of X and Y.

Tracing the execution of the program produces the process activity versus time diagram shown in Figure 4.1. The times at which processing occurs are shown as solid blocks. During the remainder of the time the processes are blocked, waiting for communication with other processes.

In this case the clients are identical, so the one which is first serviced by the server is labelled C1 and the second C2. The diagram shows that the second synchronization between C1 and the server occurs when both are ready to communicate, which occurs at:

From C1's point of view: $Y + X$

From the server's point of view: $Y + Y$

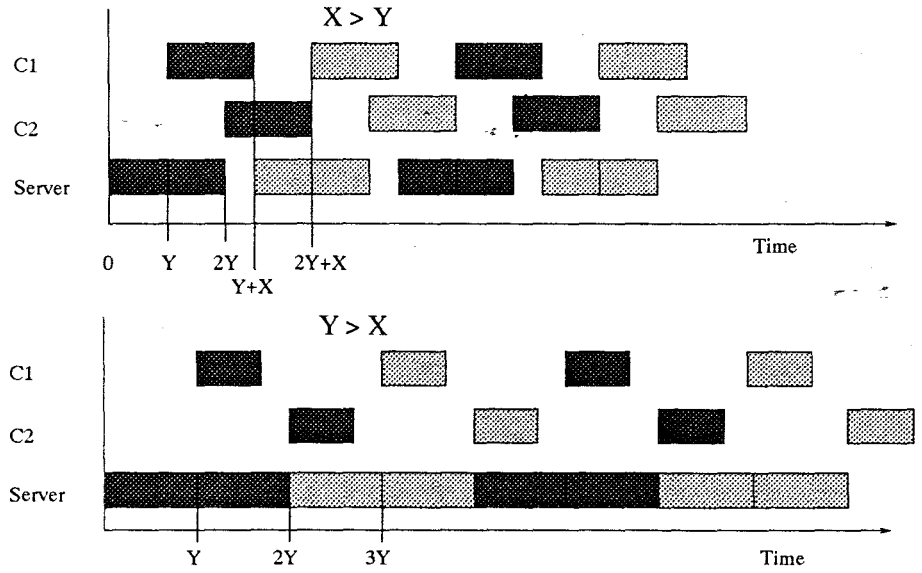

Figure 4.1: Process activity versus time diagram for the client-server system

Clearly this time depends on the relative sizes of X and Y. If X is larger then the client delays, otherwise the server is the delaying factor.

A trace of the times at which C1 finishes its processing is given below:

For $X \geq Y$: $X+Y$ $2X+2Y$ $3X+3Y$ $4X+4Y$ ... $n(X+Y)$

For $Y \geq X$: $X+Y$ $X+3Y$ $X+5Y$ $X+7Y$ ... $2nY+X-Y$

The latency and cycle times for C1 can now be easily calculated. In this case, it is assumed that input to the client process arrives immediately before the client issues a request to the server, and output occurs straight after client processing (X) is complete. Since latency is the period from input to corresponding output, latency and cycle time are the same in this example, and are given by the time between two successive cycles:

For $X \geq Y$: Latency = Cycle time = $X+Y$

For $Y \geq X$: Latency = Cycle time = $2Y$

This technique is illustrated for a more complex model, in which latency and cycle time are different, before the general algorithm is presented.

Consider a common parallel processing topology, the pipeline. The one modelled has three nodes, and consists of the following processes:

```
process 1

    do forever
        measurement point:  Latency 1
        consume input
        process for a period of duration A
        send data to 2
```

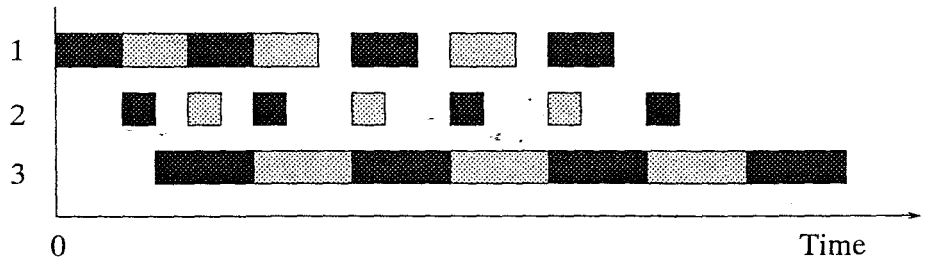

Figure 4.2: Process activity versus time diagram for the pipeline system

```
process 2

    do forever

        receive data from 1
        process for a period of duration B
        send data to 3


process 3

    do forever

        receive data from 2
        process for a period of duration C
        produce result
        measurement point:  Latency 2 / Output
```

It is assumed that input data is always immediately available and that results can be delivered without any delay. The latency in this system is the time taken for any specific datum to be transformed from input data to a result. The cycle time is the time taken per result.

As with the previous example, a process activity versus time diagram shows several decisions that need to be made regarding the relative sizes of A, B and C in order to draw the diagram. The possible diagrams that may be drawn are slightly more numerous in this example, in fact there are infinitely many of them. The relationships between the variables, which define situations in which different behaviour occurs, are given below:

- $A \geq B, A \geq C$
- $A \geq B, C \geq A$ and $B + kC \leq (k+1)A \leq (k+2)A \leq B + (k+1)C \qquad k = 0 \ldots \infty$
- $B \geq A, B \geq C$
- $B \geq A, C \geq B$

An example of the second case is illustrated with $A = 2B$, $C = 3B$ in Figure 4.2. An interesting feature of this diagram in particular is the manner in which latency becomes constant after the second iteration. The complex inequality arises from a transient which works its way out of the system after a certain number of iterations, dependent on the values of A, B and C.

By marking certain points in the program (*Latency 1* and *Latency 2 / Output*) and by calculating the time at which they occur, it is possible to find values for latency and cycle time. The values are given below for cycle n:

$A \geq B, A \geq C$:

$$\begin{aligned} &\textit{Latency 1}: && (n-1)A \\ &\textit{Latency 2/Output}: && nA+B+C \end{aligned}$$

$A \geq B, C \geq A$ and $B + kC \leq (k+1)A \leq (k+2)A \leq B + (k+1)C$, $k = 0 \ldots \infty$:

$$\begin{aligned} &\textit{Latency 1}: && \begin{cases} (n-1)A & \text{if } n \leq k+2 \\ A+B+(n-3)C & \text{otherwise} \end{cases} \\ &\textit{Latency 2/Output}: && A+B+nC \end{aligned}$$

$B \geq A, B \geq C$:

$$\begin{aligned} &\textit{Latency 1}: && \begin{cases} 0 & \text{if } n = 1 \\ A+(n-2)B & \text{otherwise} \end{cases} \\ &\textit{Latency 2/Output}: && A+nB+C \end{aligned}$$

$B \geq A, C \geq B$:

$$\begin{aligned} &\textit{Latency 1}: && \begin{cases} 0 & \text{if } n = 1 \\ A & \text{if } n = 2 \\ A+B+(n-3)C & \text{otherwise} \end{cases} \\ &\textit{Latency 2/Output}: && A+B+nC \end{aligned}$$

Latency is the difference between corresponding execution times at the two measurement points. Cycle time is given by the coefficient of n at the output point. Thus in the steady state :

$A \geq B, A \geq C$:

$$\begin{aligned} &\text{Latency:} && A+B+C \\ &\text{Cycle Time:} && A \end{aligned}$$

$A \geq B, C \geq A$ and $B + kC \leq (k+1)A \leq (k+2)A \leq B + (k+1)C$, $k = 0 \ldots \infty$:

$$\begin{aligned} &\text{Latency:} && \begin{cases} B+nC-(n-2)A & \text{if } n \leq k+2 \\ 3C & \text{otherwise} \end{cases} \\ &\text{Cycle Time:} && C \end{aligned}$$

$B \geq A, B \geq C$:

$$\text{Latency:} \quad \begin{cases} A+B+C & \text{if } n=1 \\ 2B+C & \text{otherwise} \end{cases}$$

Cycle Time: B

$B \geq A, C \geq B$:

$$\text{Latency:} \quad \begin{cases} A+B+C & \text{if } n=1 \\ B+2C & \text{if } n=2 \\ 3C & \text{otherwise} \end{cases}$$

Cycle Time: C

An algorithm for deriving the sets of inequalities associated with the Analytical Simulation technique, as well as the sequence of time values at various points, is presented below. This algorithm performs essentially the same process as illustrated in the previous examples. A process activity versus time diagram is constructed by simulating the program. The synchronization points are analysed to determine the time of synchronization. This analysis may result in various constraints being placed on the variables, to guide further simulation.

```
REPEAT
    Simulate processes keeping track of the running time of each
    process
    When two processes need to synchronize, compare their running
    times
    IF the relation between times cannot be determined
    THEN
        FOR all possible relationships between the times
            Assume that relationship holds and simulate with
            that assumption
    ELSE
        Synchronization occurs at the later of the two times
UNTIL sufficient data has been accumulated
```

Several requirements of the algorithm limit the variety of systems upon which analysis can be performed. The last line assumes the system is going to settle into a stable state after some point. In the present form this requires a human controlled cutoff, to prevent it degenerating into solving the halting problem. Some systems may never reach a point at which all possible relationships have been enumerated. Such a system is illustrated in the pipeline example earlier. In this case, however, the complete set of constraints can easily be extrapolated from a smaller sample.

4.2 Implementation of Analytical Simulation

The implementation of the Analytical Simulation algorithm used for the analyses in this document allows for the simulation of parallel processes communicating using message passing. Other architectures may also use the algorithm. This is superfluous in this case, since all the systems modelled fall into the message passing category. The simulation language consists of a few simple commands:

send :- send a message to another process.

receive :- receive a message from another process.

think :- perform sequential processing for a specified time.

These commands determine the architecture dependence. Other suitable constructs would be needed for modelling other architectures such as shared memory machines. The system is assumed to consist of a number of processes each consisting of an infinite loop with these commands in the body of the loop. A complete description of the simulation language and the use of the Analytical Simulation process in practice is given in Appendix B. The language shares constructs found in various process algebras such as CCS and CSP. These algebras are used for describing communicating, concurrently executing systems [Lee94]. The process of translation into one of these formal specification languages is shown in Appendix C. This allows easy access to the formal proof techniques available under such calculi.

Synchronization occurs when two processes attempt to communicate. The run times of each of the processes is a linear expression consisting of the sum of a number of *think* times and possibly communication times. Deciding which time is the later involves comparing the two expressions and deciding which is greater. These comparisons are often made in the presence of assumptions about the interrelationships of other expressions involving the same variables.

Making the comparisons turns out to be a reasonably complex problem. The previously mentioned problem of finding a suitable point to stop the simulation can be solved comparatively easily. In practice it is usually possible to identify the tendencies of the system by eye after each process has run for a limited number of cycles.

4.2.1 Comparison of run times

The method for comparing linear expressions is presented below:

$$\text{Let } A = \sum_{i=1}^{n} e_i X_i$$

$$\text{Let } B = \sum_{i=1}^{n} f_i X_i$$

The problem is to determine if $A \leq B$, $A \geq B$ or no known relationship exists, given assumptions

of the form:

$$\sum_{i=1}^{n} g_{ij} X_i \geq \sum_{i=1}^{n} h_{ij} X_i \qquad j = 1 \ldots m$$

$$X_i \geq 0 \qquad i = 1 \ldots n$$

By assigning $a_i = e_i - f_i$ and $b_{ij} = g_{ij} - h_{ij}$ this problem can be restated more simply as: Determine if $\sum_{i=1}^{n} a_i X_i \geq 0, \leq 0$ or if no known relationship exists, given:

$$\sum_{i=1}^{n} b_{ij} X_i \geq 0 \qquad j = 1 \ldots m$$

$$X_i \geq 0 \qquad i = 1 \ldots n$$

The inequality $\sum_{i=1}^{n} a_i X_i \geq 0$ holds if $\sum_{i=1}^{n} a_i X_i$ is greater than a linear combination of the assumptions combined using only positive coefficients. The case $\sum_{i=1}^{n} a_i X_i \leq 0$ can be reduced to the previous case if rewritten as $\sum_{i=1}^{n} -a_i X_i \geq 0$. Thus it is necessary only to attempt to solve the first case.

A simple transformation reduces the problem to one with a known solution. Attempting to find the desired linear combination produces an expression of the form shown in equation 4.1, where $w_i \geq 0$ and $v_i \geq 0$. The w_i are the required coefficients and the v_i are slack variables to enforce the inequality.

$$[X_1..X_n] \left(\begin{bmatrix} 1 & 0 & .. & 0 \\ 0 & 1 & .. & 0 \\ .. & .. & .. & .. \\ 0 & 0 & .. & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ .. \\ v_n \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & .. & b_{1m} \\ b_{21} & b_{22} & .. & b_{2m} \\ .. & .. & .. & .. \\ b_{n1} & b_{n2} & .. & b_{nm} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_n \end{bmatrix} \right) = [X_1..X_n] \begin{bmatrix} a_1 \\ a_2 \\ .. \\ a_n \end{bmatrix} \tag{4.1}$$

Solving for the v_i, and using the requirement that each $v_i \geq 0$, produces equation 4.2. This is the standard form of the constraints in a linear programming problem [Str76]. The complete solution to the linear programming problem is not required however, it suffices to find a single point in the feasible region. The existence of such a point is equivalent to having a positive coefficient linear combination of the assumptions which satisfies the desired inequality.

$$\begin{bmatrix} b_{11} & b_{12} & .. & b_{1m} \\ b_{21} & b_{22} & .. & b_{2m} \\ .. & .. & .. & .. \\ b_{n1} & b_{n2} & .. & b_{nm} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_m \end{bmatrix} \leq \begin{bmatrix} a_1 \\ a_2 \\ .. \\ a_n \end{bmatrix} \tag{4.2}$$

Finding this single point uses the Two-Phase Method described in [Kol80]. This method consists of introducing two sets of slack variables and applying the simplex method to remove one set. The success of the simplex method is equivalent to the existence of a solution.

If a relationship between A and B can be found then the next step in the simulation of the model is well defined. If no relationship exists, then an additional constraint is introduced, specifying

the relationship between A and B. Since two possibilities exist ($A \geq B$, or $B \geq A$), each must be introduced in turn and both branches simulated. Resolving non-determinism can cause a number of constraints to be introduced at one point, splitting the simulation path in more than two ways.

4.2.2 Non-determinism

The presence of non-determinism in a system when using the Analytical Simulation algorithm can lead to cases where one set of restrictions on the variables can produce different performance values when calculating cycle times and latencies. Non-determinism can arise, for example, when a process receives messages from a number of other processes at the same time, all of which are valid under the current *receive* command. In that case the receiving process is usually expected to choose one in a non-deterministic manner. Non-determinism can severely affect the usefulness of Analytical Simulation, and ways are required to deal with it.

To speed up the analysis, the simulation can be instructed to ignore the non-determinism, and make the choice in some consistent manner. This limits the analysis to only one of the possible execution paths, and gives an approximation to the true behaviour. This and other problems are discussed in greater detail later when exploring this method with various example problems (see sections 6.1.2 and 6.1.4). Modification of the Analytical Simulation technique to enhance its ability to deal with non-deterministic constructs is described in Chapter 7.

4.3 Conclusion

This chapter introduces the Analytical Simulation approach to performance analysis. This approach satisfies the three main requirements of a performance analysis and comparison tool for parallel and distributed virtual reality systems in that:

- It provides for the measurement of latency and cycle time.
- It can be applied to arbitrary decomposition strategies, including those used for virtual reality systems.
- It provides symbolic results suitable for comparison purposes.

In addition, the implementation of the algorithm allows rapid analysis of models, which can easily be created, and which closely resemble the structure of the system being modelled. Results include constraint regions which specify the variable values for which each result holds, simplifying interpretation of the results. The Analytical Simulation approach is the most successful in fulfilling the requirements for modelling parallel and distributed virtual reality systems.

The next few chapters (Chapters 5 to 8) describe improvements to the Analytical Simulation approach which enhance its ability to deal with virtual reality systems. These chapters also verify the results of analysis by comparing them against values measured from implementations of the systems being modelled.

Chapter 5

Collision detection

A test case is required to verify the suitability of the Analytical Simulation approach to performance analysis of virtual reality systems. This chapter selects a common component of virtual reality systems, and develops parallel algorithms for this component using the different decomposition strategies associated with such systems.

The problem selected for analysis is one that occurs in a number of virtual reality systems and which is solved in a variety of ways (see Chapter 2). It involves the detection of collisions between objects in a virtual world. The example used in this chapter is that of point particles colliding within a closed container.

It is particularly instructive to examine this problem, since it requires that each particle is aware of every other particle. Thus it requires that particle data is distributed to every other particle. Implementation in a distributed system requires use of the database distribution approaches used in virtual reality systems. Coordination of the computation requires use of a control distribution approach, as discussed in Chapter 2.

The chapter starts with an examination of the algorithms used in existing systems before describing the development of an appropriate test case.

5.1 Previous implementations of collision detection

The level at which collision detection is examined is limited to the selection of the objects to be checked for collision, and the acquisition of the positions and other relevant information regarding the affected particles which is required to do this. The problem of computing points of intersection, given a boundary representation of the object, is beyond the scope of this document. Solutions to the latter problem are well documented elsewhere, for example [Coh94] [Gar94] [Van93]. A more detailed survey of collision detection techniques can be found in [Hub95a]. This section is intended only as background for the algorithms developed in this chapter.

Approximate solutions to collision detection, such as are used in [Ban93], involve sampling the object positions at discrete intervals and checking for collisions during the snapshot. Provided that the sampling interval is sufficiently small, most collisions are successfully detected. More detailed discussion of the problems involved in implementing reliable collision detection is given

in [Hub93].

Performing pairwise comparisons between all pairs of objects results in an $O(N^2)$ algorithm. Spatial subdivision approaches divide the virtual environment into smaller volumes, and collision detection need only be performed between objects sharing these volumes [Zyd93]. Collision detection algorithms often use a number of successive levels of refinement. The topmost levels are fast but may imagine collisions where none in fact exist; the lower levels are more accurate, but slower. The upper levels typically use bounding box comparisons, checking for collisions between simple shapes such as spheres or cubes surrounding the object.

Some restrictions are imposed on the creation of parallel versions of collision detection algorithms for use within a virtual reality environment. The algorithm must fit into a larger system without imposing constraints on the distribution of the other components. Other operations overlap with collision detection and the system may be implemented more efficiently if these are combined with the collision detection. This limits the use of sophisticated distribution mechanisms using special properties of the problem. Such considerations apply to the use of approaches such as [Yas92], whose distributed N-Body algorithm uses a tree decomposition. Branch nodes in the tree contain centre of mass information for simplifying calculation of forces. Integration with a collision detection algorithm would require that additional information be added and may necessitate changes to the distribution mechanism. Approaches such as that used in [Par92], where a SIMD machine is used for calculating the N-body simulation, are also not feasible. Use of this architecture requires that all objects be functionally identical, and so no specialized processing can be included with some of the object processes. The parallel decompositions considered here are limited to coordinating control of the system, and to distributing data about every object, to every other object.

Discussions amongst the developers of the VRML specifications[1] suggested three approaches to implementing distributed collision detection [War95]. A single process could perform all the collision detection for the entire system, or every process could perform this collision detection, each duplicating the work of the others. The first approach would simplify data distribution, the second would simplify distribution of the results. The third approach suggested would be to distribute the computation across all the processors in the system.

The third approach offers the most comprehensive test of an analysis technique, and so the approaches to distribution of collision detection that are described assume that the problem is decomposed across multiple processors.

5.2 A sequential algorithm for collision detection

The goal of this section is to develop a collision detection algorithm that can be used as a fair test case for a performance analysis technique. The required solution must function correctly and be amenable to implementation in a distributed environment. It must also use constructs found in distributed virtual reality systems, thus making the results of the evaluation of the performance

[1]VRML: Virtual Reality Modelling Language, a project to combine Virtual Reality with the World Wide Web (see section 2.3.11)

analysis tool applicable to these virtual reality systems. Generality, rather than efficiency, is emphasized, although some optimization strategies are described where appropriate.

A simple form of the collision detection problem is examined, which satisfies the above requirements. The problem addressed is that of a single virtual world containing N virtual gas molecules enclosed in a virtual box. The particles do nothing other than move about within the box, colliding occasionally with the walls and with each other.

The calculation of trajectories, points of collision, and changes in motion after collision is not considered. The only matter of interest is to ensure that the simulation proceeds realistically, assuming that functions to perform these calculations are supplied.

It might be expected that a routine along the lines of:

```
while (simulation_running)
    for i = 1 to N
        move_particle(i)
    for i = 1 to N
        for j = i + 1 to N
            if colliding(i, j) then
              change_direction(i)
              change_direction(j)
```

would perform the required task. However, it turns out on closer examination of this algorithm that there are a number of flaws [Hub93].

The problems result from the discrete time nature of the simulation. As the simulation advances, the molecules move from one point to another without truly touching all intervening points. Thus a situation where *molecule1* moves from A to B and *molecule2* moves from B to A during the same time interval results in the two molecules passing through each other without a collision being detected. This problem may be addressed by extending the *colliding* predicate to check for collisions at all intervening points between two successive positions.

However, there is yet another problem that this additional check does not solve. This is the case where a molecule would collide with more than one other molecule during a simulation step. The *colliding* predicate cannot be expected to realize that a molecule, as a result of a collision, might change direction and strike another.

A possible solution to this problem is to reinvoke the pair of nested loops until no more collisions occur during the current simulation step. This should catch all secondary collisions.

At this point, a brief digression is necessary to provide a view of the overall intention of this section. The algorithm developed should be capable of being successfully used in a parallel virtual reality system. The simulation required in a virtual reality system need not always be accurate; in many cases, speed is a far more important consideration. There are a number of cases of faster but less correct algorithms being used in virtual reality systems, for example, the Painter's algorithm used in hidden surface removal. Hence the approximate solution derived above should not be discarded immediately. The other consideration is that the algorithm should parallelize easily,

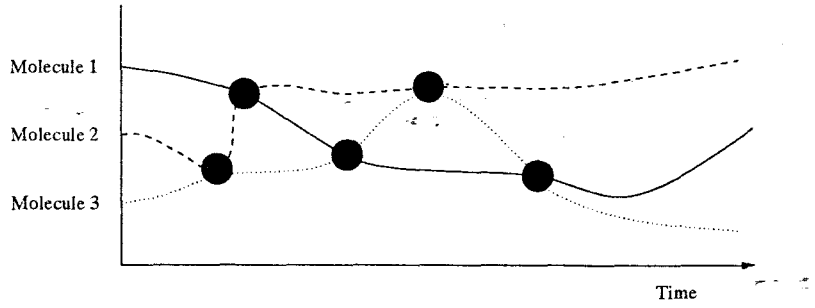

Figure 5.1: Example of a trace

and hence the last adjustment above, which requires repeated access to a shared resource, may not be feasible.

The other alternative is to drop the requirement of simulation steps which are independent of the collision detection process. Instead let the steps be governed by the requirements of the problem.

Let the *trace* of a run of the simulation be a graph, where a node represents a point at which a collision occurred between particles and the arcs represent movement of the particles. The nodes can all be ordered on the time at which the collision occurred, as illustrated in Figure 5.1. The next node on the trace can be determined while at the previous node, irrespective of the time difference between them. Thus the collision detection algorithm can be rewritten as:

```
while (simulation_running)
    for i = 1 to N
        for j = i + 1 to N
            let t be the minimum time until a collision
            between i and j
    for i = 1 to N
        move_particle(i) to its position at time t
        if particle[i] is involved in a collision at time t
        then
            change_direction_of_particle(i)
```

This algorithm for collision detection can be used in virtual reality systems, and is used as the basis for developing distributed collision detection routines for testing performance prediction techniques. Additional enhancements are still possible, such as the volume subdivision step mentioned in the previous section. Another optimization is to cache collision points, since they do not need to be recalculated until at least one of the molecules involved changes trajectory.

5.3 Collision detection on parallel processors

Virtual reality implementations are often constructed from whatever components are available. In some cases, specialized components are added to enhance the performance of some aspect. Rarely

are these systems integrated to a state where shared memory is available. In general, as may be seen in the systems reviewed in Chapter 2, parallel virtual reality systems communicate using message passing, often over relatively low bandwidth links. Thus all the parallel decomposition attempted here assumes a message passing architecture, with consideration paid to minimizing communication.

The approach when parallelizing the collision detection algorithm is to assume that each object resides on a different processor. The alternative of a single application process performing collision detection in parallel with the rest of the system does not look much different from the sequential algorithm.

5.3.1 Collision detection using direct Message Passing

Generally a virtual reality system provides some operating system level facilities, which attempt to enforce a particular communication paradigm. These limit the possible communication options to make parallel programming easier and less susceptible to errors. These functions are usually implemented with constructs which perform message passing directly between processes. This section describes a distributed collision detection algorithm where all processes are identical, communicating via the direct message passing calls.

The outline of the algorithm for the direct message passing approach is as follows:

```
PROCESS molecule[k]

    { process corresponding to a single molecule }
    WHILE (running ())
        FOR (i = 1; i <= N; i++)
            SEND (molecule[i], DATA)
              { send own position to all others }
        FOR (i = 1; i <= N; i++)
            otherpositions = RECEIVE (DATA)
              { get position of other molecules }
        find first collision between this molecule and all
        other molecules and the walls
        FOR (i = 1; i <= N; i++)
            SEND (molecule[i], time_first_collision)
              { send best guess at first collision }
        FOR (i = 1; i <= N; i++)
            temp = RECEIVE ()
            IF (temp < smallest_received) THEN
              smallest_received <- temp
                { combine guesses to get earliest
                collision }
```

```
move molecule along trajectory by an amount
corresponding to the smallest time received
IF the molecule is involved in a collision at this
point THEN
    alter_trajectory
```

To simplify the algorithm, it is assumed that messages do not block when they are sent, but queue at the destination point. This implementation has the disadvantage that all molecules must resynchronize after each collision, even those that are not involved and which are not affected by the outcome of the collision. Other approaches such as that described in section 5.3.3 do not have this limitation, and so the amount of computation required can be reduced for that case.

5.3.2 Collision detection using a Client-Server approach

The message passing approach associates the data for each object with the corresponding object process, a distributed databases approach. For all simulation cycles, each molecule must be polled by every other molecule in order to obtain up to date position information, i.e. $O(N^2)$ communication. This may be reduced by creating a central repository for the position data, a centralized database, at the cost of requiring each molecule to transmit updates whenever changes are made.

A server process is created which acts as the central repository for the position data for every molecule in the system. With this particular problem it is also used to coordinate finding the time of the next collision, as well as for synchronizing the various processes.

The outline of the algorithm for the client-server approach is as follows:

```
PROCESS client[k]
    { process corresponding to a molecule }
    WHILE (running ())
        SEND (server, GETOTHER)
            { send a request for the position database }
        allposition = RECEIVE (OTHERDATA)
            { get the database }
        find first collision between this molecule and all
        other molecules and the walls
        SEND (server, BESTTIME)
            { send details of first collision }
        besttime = RECEIVE (BESTDATA)
            { get overall value for first collision }
```

```
            move molecule along trajectory by an amount
            corresponding to the smallest time received
            IF the molecule is involved in a collision at this
            point
               THEN alter_trajectory
            SEND (server, UPDATEPOS)
               { update the database }
            ack = RECEIVE (UPDATEOK)

PROCESS server
    { coordinates the world database }
    WHILE (running ())
        req = RECEIVE (ANYTHING) from client[i]
        SWITCH on request type
            CASE GETOTHER:
              SEND (client[i], OTHERDATA);
                { send position database }
            CASE BEST:
              IF collision time from client[i] is earliest
              to date THEN
                update earliest to data value
              IF all clients have sent their best values
              THEN
                FOR (i = 1; i <= N; i++)
                  SEND (client[i], besttodate)
                    { send time of first collision }
            CASE UPDATEPOS:
              update entry for client[i]
              IF all clients have updated THEN
                FOR (i = 1; i <= N; i++)
                  SEND (client[i], UPDATEOK)
                    { synchronize clients at this step }
```

Since all the collision data gets combined in the end, and due to the fact that the collision predicate is transitive, this algorithm can easily be enhanced to halve the number of comparisons required. Thus the time that A collides with B is the same as the time that B collides with A.

5.3.3 Collision detection using a Master-Slave approach

Many virtual reality systems use an event handling approach to control object processes, with a master process issuing events to a number of slaves. This simplifies control of the objects by

other processes. In some cases, a separate process is used for collision detection. This process then sends events to the object processes to indicate when a collision will occur. The routine below shows the adaptation of the collision detection algorithm to an event driven system. The object processes are also used for some of the collision detection calculations, thereby increasing the extent to which parallelism can be used.

The outline of the algorithm for the event handling approach is shown below:

```
PROCESS slave[k]

    { process corresponding to a molecule }
    WHILE (running ())
        req = RECEIVE (ANYTHING);
        SWITCH on request type
            CASE REQDATA :
              SEND (master, HEREDATA)
                { send position of this molecule }
            CASE FINDYOURBEST :
              find first collision between this molecule
              and all other molecules and the walls
              SEND (master, MYBEST)
            CASE RUNUNTIL :
              move molecule along trajectory by an amount
              corresponding to the value just received
              IF the molecule is involved in a collision at
              this point THEN
                alter_trajectory

PROCESS master

    { process in charge of collision detection }
    WHILE (running ())
        FOR (i = 1; i <= N; i++)
            SEND (slave[i], REQDATA);
        FOR (i = 1; i <= N; i++)
            req = RECEIVE (HEREDATA);
            add data to a local database
        FOR (i = 1; i <= N; i++)
            SEND (slave[i], FINDYOURBEST);
              { get each slave to do some of the work }
```

```
FOR (i = 1; i <= N; i++)
   req = RECEIVE (MYBEST);
     { get each molecule's first collision }
Calculate time of first collision
FOR (i = 1; i <= N; i++)
   SEND (slave[i], RUNUNTIL);
     { let molecules run until first collision }
```

This approach is easily modified to accumulate the times of the various collisions and to only update those that change. It is easy to see that only those particles involved in a collision can affect subsequent collisions. This modification is not as natural in the other systems where control is distributed, as opposed to the centralized control present in this variation.

5.4 Conclusion

Collision detection is chosen as a problem suitable for testing the capabilities of the Analytical Simulation approach to performance analysis. A sequential algorithm for the detection of collisions between point particles is described. This is parallelized in three different ways, using approaches found in distributed virtual reality systems.

The approaches used are:

- A client-server approach for data distribution, control remaining with the objects.
- A message passing approach maintaining data and control locally with each object process, and specialized communication for interaction.
- A master-slave approach keeping data for each particle with the corresponding particle process, but using central control from a master process.

Analysis of the client-server model is described in Chapter 6. Some enhancements to Analytical Simulation resulting from this are given in Chapter 7. The remainder of the analysis for the models described in this chapter, followed by a comparison with the results measured from implementation, is given in Chapter 8.

Chapter 6

Analysis of client-server collision detection

Chapter 5 provided a solution to the collision detection problem and examined the parallelization of this solution. Some of the methods of parallel decomposition used in virtual reality systems were used to select strategies for the parallel decomposition of the collision detection algorithm. Amongst the strategies used was the client-server approach for distribution of data and control information.

Evaluation of the performance analysis techniques in Chapter 3 used a simple client-server system as a test case. This chapter extends the analysis to a realistic model. The limitations uncovered during this analysis are used to improve the Analytical Simulation approach (Chapter 7) before analysis of the other collision detection algorithms is performed (Chapter 8).

This chapter investigates the use of Analytical Simulation on successive refinements of a client-server system until the complete client-server decomposition of the collision detection algorithm is reached.

6.1 Analysis of client-server models

Analysis of the client-server decomposition strategy starts with the analysis of a simplified version. Additional complexity is added incrementally, and the effect of these increments can then be seen in relative isolation. The enhancements are incorporated into the collision detection algorithm in section 6.2.

6.1.1 Deterministic, simple communication

A simplified client-server model is used as an example when introducing Analytical Simulation. This consists of two clients, each requesting data from the server and then processing that data for an interval of X. The server accepts requests from each client in turn and returns a response after an interval of Y.

This model can be extended to N clients using the following algorithm:

```
process client[i] (where i = 1 ...N)
    repeat forever
        send request for data to the server
        receive the reply from the server
        process the reply for period X
    end repeat

process server
    repeat forever
        for i = 1 to N
            receive request from client[i]
            prepare the reply, taking period Y
            send the reply to client[i]
        end for
    end repeat
```

This corresponds to the following model for the Analytical Simulation tool:

```
generate markerend1 -
define N 5

replicate N
    process client#
        send server request
        receive server reply
        think X
        report markerend#
endreplicate

process server
    replicate N
        receive client# request
        think Y
        send client# reply
    endreplicate
```

The *generate* statement causes the tool to generate the cycle time of *client1*. The variable N defines the number of clients in the model. The tool, run for 100 iterations of the server process,

produces the following results, for the model with five clients:

```
Trace 1:  Program with assumptions:
Example:  X = 1000 Y = 200
1 X       >       4 Y
Change in markerend1    1 Y + 1 X (100)

Trace 2:  Program with assumptions:
Example:  Y = 1000 X = 1000
4 Y       >       1 X
Change in markerend1    5 Y (100)
```

The constraint regions are indicated, as well as the values for each region. The values in parentheses indicate the length of the sequence containing that particular result. Repeating the analysis with different values of N produces the following results:

$X \geq (N-1)Y$: Latency = Cycle Time = $X + Y$

$(N-1)Y \geq X$: Latency = Cycle Time = NY

Further options for increasing the complexity of the model are to have a differing processing time for each of the clients, to introduce non-deterministic selection by the server, and to increase the complexity of the communication to approximate that of the desired algorithm more closely.

The latter two options are more relevant to the problem that is being solved, so these are examined first.

6.1.2 Non-deterministic, simple communication

Non-determinism can easily be introduced by changing the action of the server. At present it communicates with each client in turn, waiting for each client, regardless of whether others are already available for communication. This can be modified so that the server accepts requests from the first client capable of communication. Should more than one client satisfy this condition, then one must be selected in a non-deterministic fashion.

The modified server code in the model for the Analytical Simulation tool is as follows:

```
process server

    receive ANYONE request
    think Y
    send ANYONE reply
```

The receive statement contains a variable name as the name of the source process. This will be instantiated to the name of the first process to communicate, or each of the names in turn if there is more than one. The value of the same variable is used as the name of the destination process to which the reply must be returned.

As might be expected from the symmetry of the program, the expected results are not altered by introducing this change. However, it does have a significant effect on the complexity of the

Analytical Simulation. For a situation with N clients, the server must select one of these in a non-deterministic fashion. After the first cycle, there are still $N-1$ requests at the head of the request queue. The total number of permutations is quickly seen to be related to the factorial of N. The difference between the permutations only affects the relative order in which the processes run. No change occurs in the latency values or in the cycle times.

The analysis tool faithfully explores all $N!$ paths, producing much the same results as before. The lengths of the sequences of results are shorter, the new server process cycles once for every client request now, rather than every set of N requests as before.

As mentioned in section 4.2.2, the number of cases can be reduced in this case by instructing the simulation to solve the non-deterministic choice in a predefined and consistent manner. This does not cause loss of information in this simple case.

6.1.3 Deterministic, complex communication

Increasing the complexity of the communication involves increasing the number of queries to the server in each cycle of the client. As may be seen from the algorithm presented in section 5.3.2, the client requests two items of data from the server, before performing some calculation and then sending an update to the server. The client processes are thus modelled as:

```
process client

    repeat forever

        send server request_for_data
        receive server some_data
        send server request_for_data
        receive server some_data
        process for a period of duration X
        send server request_for_data (update)
        receive server some_data (acknowledgement)
```

The server, as before, accepts a single request from each client in turn, processes for period Y and returns a reply.

The results have the same form as those given in section 6.1.1:

$X \geq (N-1)Y$: Latency = Cycle Time = $X + (2N+1)Y$

$(N-1)Y \geq X$: Latency = Cycle Time = $3NY$

The actual algorithm being modelled has some extra complexity in the server in that in some cases it waits for data from all clients before responding to each of the requests. The server code

modified to support this approach has the following outline:

```
process server
    repeat forever
        receive client1 request_for_data
        process for a period of duration Y
        send client1 some_data
        receive client2 request_for_data
        process for a period of duration Y
        send client2 some_data
            ...for all N clients

        receive client1 request_for_summary
        receive client2 request_for_summary
            ...for all N clients
        process for a period of duration N * Y
        send client1 some_data
        send client2 some_data
            ...for all N clients

        receive client1 request_for_summary
        receive client2 request_for_summary
            ...for all N clients
        process for a period of duration N * Y
        send client1 some_data
        send client2 some_data
            ...for all N clients
```

The second and third requests from each client to the server, for each cycle of the client, request summaries rather than position data.

The model for the Analytical Simulation tool is shown below:

```
replicate N
    process client#
        send server requestfordata
        receive server reply
        send server requestforsummary
        receive server reply
        think X
        send server requestforsummary
        receive server reply
        report markerend#
endreplicate
```

```
process server
    replicate N
        receive client# requestfordata
        think Y
        send client# reply
    endreplicate
    replicate 2
        replicate N
            receive client## requestforsummary
        endreplicate
        replicate N
            think Y
        endreplicate
        replicate N
            send client## reply
        endreplicate
    endreplicate
```

The effect of the synchronization is to restrict the order in which events may occur, so that only one path of execution is possible, regardless of the relative values of X and Y. The processing in each of the clients is forced to occur in parallel, and is unable to overlap with any processing in the server. Applying the Analytical Simulation technique to this system yields a single report, giving the value of $3NY + X$ as the cycle time and latency.

6.1.4 Non-deterministic, complex communication

The next logical step to complete the sequence of analyses is to combine the complex communication sequence in the client with a non-deterministic server process. As illustrated in section 6.1.2, this server process accepts a request in a non-deterministic fashion, processes it for a period of duration Y and returns a response.

The model for analysis using Analytical Simulation is shown below:

```
replicate N
    process client#
        send server request
        receive server reply
        send server request
        receive server reply
        think X
```

```
            send server request
            receive server reply
            report markerend#
    endreplicate

    process server

        receive ANYONE request
        think Y
        send ANYONE reply
```

The version of the system with only two clients produces four cases. The case where $X \leq Y$ is well behaved and the performance settles down rapidly to a value of $6Y$. When $Y \leq X \leq 2Y$, latency also stabilizes on $6Y$, but an unstable initial value of $5Y + X$ may prevail for as long as chance dictates. Chance, in this case, determines the way in which the non-determinism is resolved. The other two cases depend on whether X is greater or less than $3Y$. In both cases they settle down to steady state latencies of $6Y$ for $2Y \leq X \leq 3Y$ and $3Y + X$ for $X \geq 3Y$. However there is again a transient phase where the latency may be $5Y + X$ or $4Y + X$ for both cases. These transients are unstable and once a decision is made that eliminates a particular transient, then the system moves into a state in which that transient never recurs. The analysis explores 548 paths within the 100 cycles of the server process.

This initial unstable state would be missed, or would deceive a simulator.

The complexity is greatly increased when adding a third client. For some values of X and Y, the latency does not settle down to a steady, stable value. Instead it assumes a number of values, corresponding to different orders in which each of the clients interact with the server. In addition the number of permutations is sufficiently large as to prove daunting when it comes to analysis. At this point it becomes worthwhile to examine the requirements of the analysis and find some appropriate method of summarizing the results without adversely affecting the analysis. A number of alternatives present themselves.

Firstly, the range of values can be expressed in some condensed form. An average value and a standard deviation, or possibly the minimum and maximum values encountered would suffice. An alternative is to remove the non-determinism by preselecting the order in which decisions are made, as discussed in section 4.2.2. This is more applicable in this situation where the clients are identical and symmetry exists.

The second approach has the advantage of reducing the simulation search space, but can pose certain problems. Firstly, it does not reveal all details of the behaviour of the system. It can also result in certain relationships between the variables in the system being ignored.

The range of latency values assumed for the different variable values is listed below, with the

result obtained by removing the non-determinism given in parenthesis:

$X > nY$, for $n > 8$	$(-)$	$3Y + X, 4Y + X, 5Y + X, 6Y + X$
$X > 8Y$	$(3Y + X)$	$3Y + X, 4Y + X, 5Y + X, 6Y + X$
$8Y > X > 7Y$	$(3Y + X)$	$3Y + X, 4Y + X, 5Y + X, 6Y + X$
$7Y > X > 6Y$	$(4Y + X)$	$3Y + X, 4Y + X, 5Y + X, 6Y + X$
$6Y > X > 5Y$	$(4Y + X)$	$9Y, 4Y + X, 5Y + X, 6Y + X$
$5Y > X > 4Y$	$(10Y, 9Y, 8Y$ (in cycles))	$8Y, 9Y, 5Y + X, 10Y, 6Y + X$
$4Y > X > 3Y$	$(9Y)$	$9Y, 6Y + X$
$3Y > X > 2Y$	$(9Y)$	$9Y$
$2Y > X > Y$	$(9Y)$	$9Y$
$Y > X$	$(9Y)$	$9Y$

The values given above are taken after the initial transient has died away, which is usually after three cycles. The analysis does not add any further constraints on the relative values of the variables for the deterministic case once $X > 8Y$. These restrictions are found for the non-deterministic case but do not yield any additional values for the cycle time of the system. Transient values are not given above. The simulation for the results above was performed for 100 cycles of the server process. The deterministic version is analysed rapidly, and the values given above are easily extracted. The non-deterministic case takes the better part of a week for the analysis, on a dual processor Sparc Server-10, and produces 60 Mbytes of data (approximately 100000 paths) that require further analysis by hand. Transient values have to be removed and the results summarized. Examples of the results are shown below. The constraints marked with the "###" are those that do not form boundaries of the constraint region, and which can be ignored.

```
Trace 51277:  Program with assumptions:
Example:  X = 1000 Y = 181.818
1 X       >       2 Y     ###
1 X       >       3 Y     ###
1 X       >       4 Y     ###
1 X       >       5 Y
6 Y       >       1 X
Change in markerend1    6 Y + 1 X (2)
Change in markerend1    7 Y + 1 X (1)
Change in markerend1    5 Y + 1 X (1)
Change in markerend1    4 Y + 1 X (2)
Change in markerend1    5 Y + 1 X (1)
Change in markerend1    4 Y + 1 X (1)
Change in markerend1    9 Y (1)
...
Trace 51305:  Program with assumptions:
Example:  X = 1000 Y = 222.222
1 X       >       2 Y     ###
1 X       >       3 Y     ###
```

```
1 X      >      4 Y
5 Y      >      1 X
Change in markerend1    6 Y + 1 X (2)
Change in markerend1    7 Y + 1 X (1)
Change in markerend1    5 Y + 1 X (1)
Change in markerend1    9 Y (1)
Change in markerend1    10 Y (1)
Change in markerend1    9 Y (1)
Change in markerend1    8 Y (1)
Change in markerend1    9 Y (1)
```

It is interesting to compare the results obtained by using a conventional simulation tool on the same model. The simulation has X ranging from 1 to 50 and Y ranging from 1 to 25. For the two molecule case, the results are found to conform to:

$X < Y$: $6Y$

$X > Y$: $5Y + X$

In the three molecule case, the latency/cycle time is:

$X < 5Y$: $9Y$

$X > 5Y$: $4Y + X$

Ironically, the results are far more complex for the simple server than in the case with a more complex one that imposes more restrictions on order of execution. However, it is useful to explore the limitations of the Analytical Simulation technique that are emphasized by this situation.

A significant problem with Analytical Simulation is that it traverses the tree of possible program states to a particular depth. This can cause a number of complications. The state space may be very large, particularly if non-determinism is present. The results calculated for a given set of constraints may differ according to the order in which events occur in a particular program trace. Also the depth to which the tree is explored may not be sufficient to define the behaviour of the system adequately.

Another disadvantage is that the effects of simple variations on the program structure cannot be easily compared, since these variations create a new and different system. In the examples given here this means that the effects of adding more clients must be extrapolated from trends visible in the results of analyses of systems with smaller numbers of clients.

No further analysis of this particular system is carried out using this version of Analytical Simulation due to the complexity of the analysis. This issue is addressed in Chapter 7, when the state space extensions to the Analytical Simulation algorithm are discussed. The next case to be considered is the complete model of the client-server collision detection algorithm.

6.2 Collision detection using a client-server decomposition

This section combines the refinements of the client-server model presented in this chapter, and applies them to the client-server decomposition of the collision detection algorithm, given in section 5.3.2.

The complexity of the analysis in the previous section is reduced in this model because the additional complexity in the model imposes more constraints on the order in which the various processes may interact. This fact suggests a useful by-product of the complexity explosion of the previous section: after a minimum latency/cycle time value is found, impose restrictions on the system so that this order of events is the only one that may occur.

The model used to implement the collision detection algorithm is shown below and may be compared with that presented in section 5.3.2.

```
process client

    send server request_for_data
    receive server some_data
    send server request_for_summary
    receive server some_data
    process for a period of duration X
    send server request_for_summary (update)
    receive server some_data (acknowledgement)

process server

    receive any_client[A] any_request
    case any_request of
       request_for_data :
          process for a period of duration Y
          send any_client[A] some_data
       request_for_summary :
          receive any_client[B] request_for_summary
            ...until had a request from all N clients
          process for a period of duration N * Y
          send any_client[A] some_data
          send any_client[B] some_data
            ...for all N clients in the order in which
            requests were received
```

In this model the array *any_client* represents the processes from which messages are received. Its members are instantiated during the receive operation. This is the source of the non-determinism in this model. The variable *any_request* is set to the type of message at the same time.

Comparing this model to the original algorithm, the variable X represents the time taken to move a molecule and perform any other necessary processing. The time taken to calculate the

point of collision is ignored, although this could easily be included by addition of another *process* statement. It is excluded to provide consistency with the previous discussion. Moving the position of the existing processing section of the client does not alter the final result. The variable Y could represent the time taken for the server to access its database.

The form of this model for use with the Analytical Simulation tool is as follows:

```
replicate N
    process client#
        send server requestfordata
        receive server reply
        send server requestforsummary
        receive server reply
        think X
        send server requestforsummary
        receive server reply
        report markerend#
endreplicate
process server
    init
        assign COUNT 0
    endinit
    receive ANYONE REQUEST
    if REQUEST == requestfordata
        think Y
        send ANYONE reply
    endif
    if REQUEST == requestforsummary
        if COUNT != N
            assign COUNT [COUNT+1]
        endif
        if COUNT == N
            replicate N
              think Y
            endreplicate
```

```
            replicate N
              send client# reply
            endreplicate
            assign COUNT 0
          endif
        endif
```

The results from this model are interesting in that there are no constraints on the relative sizes of X and Y. The results are numerous, due to the non-determinacy, but are all identical. The tool shows a desire to generate about 17×10^9 solutions for the two client case, limited to 100 cycles of the server. These result from the possible alternate execution paths that exist because of the different non-deterministic choices that can be made. The identical results can be explained by examining a trace of events. The restrictions imposed by synchronization through *requestforsummary* prevent client and server from running concurrently. The only processes that run at the same time are the various clients. Non-deterministic message passing occurs between client and server, but this must be completed for every client before any client can enter its processing stage.

The value of latency/cycle time for the N client system may be easily guessed and is: $3NY+X$. This is consistent with results obtained using standard simulation techniques.

6.3 Conclusion

The client-server approach to parallel collision detection is successfully analysed using the Analytical Simulation technique. Different aspects of the model are examined to determine their influence on the overall performance of the model, and to test the analysis tool.

The tool performs well and delivers the required results. Problems are caused by the number of paths resulting from non-deterministic choices. Substantial human intervention is required to simplify the vast quantities of results that are produced. These problems are addressed in an extension to the original technique described in Chapter 7.

Chapter 7

State space extensions to Analytical Simulation

This chapter describes the state space extensions to Analytical Simulation that are applicable to periodic systems. It begins by describing the advantages of state space analysis, before explaining the method in more detail. Once the description of Analytical Simulation is complete, its areas of application, limitations and advantages are discussed in section 7.3.

The Analytical Simulation approach as described previously suffers from two significant limitations:

- Non-determinism causes the possible simulation paths to increase, often exponentially. This makes thorough analysis of the results time-consuming, and increases the computing resources required to perform the analysis.
- The simulation is only performed for a limited number of steps. Any characteristics of the model that are not present in this portion of the execution trace are ignored.

The systems being modelled in this thesis, virtual reality systems as well as real-time systems in general, share a common characteristic. They are all intended to run indefinitely, and thus repeat the same sequence of events periodically. Results are produced for every cycle of the program, but at no point does the system reach a state at which all computation is complete. The values of input data, and those of the output results are ignored for the purposes of performance analysis. In such systems all data undergoes the same sequence of transformations, regardless of its value. This periodic nature means that the states of the program recur and the state space graph of the program is cyclic. Conventional simulation as used in Analytical Simulation often explores states that have been examined before. It is much more efficient to keep track of the state space graph of the program and to use this to prevent duplication.

Non-determinism creates states with multiple outgoing arcs to each of the states resulting from resolving the non-determinism in all of the possible manners. If these nodes do not occur in cycles then this behaviour is only transient, and can be identified as such. If the node occurs in a cycle, then it can be identified as a recurring state. Finding all paths in a cycle may be expensive but

only has to be done once for the cycle, not an arbitrary number of times as is being done with standard Analytical Simulation.

Since the complete state space can be explored, all aspects of the model are examined. Traversing state space allows automatic detection of recurring states so there is no need for guessing at a limit to the interesting activity as before.

Before examining the methods for analysis of state space graphs, a working definition of the state of a parallel program is given.

7.1 The state of a parallel program

The state of a conventional sequential program consisting of a sequence of instructions can be specified by providing values for the program counter and all variables defined in the program. These variables include the registers and stack used for executing the program.

The requirement of any definition of a program state is that given any particular state it must be possible to determine the successor state uniquely. The successor state is found by executing a single program statement. This requirement must be satisfied when selecting a specification for the state of a parallel program.

If one considers a number of sequential programs running simultaneously, a parallel program in which no interaction occurs between processes, then the state of the parallel program can be given as a tuple, containing the states of the individual processes. As long as a successor state is chosen using some consistent rule (e.g. advance every process by one statement) then it is uniquely determined by its predecessor. Once synchronization constructs are introduced, however, this is no longer sufficient. Consider a message passing situation where two processes are both executing a statement which sends a message to a third process. To suitably imitate reality, it would be best if the message sent first arrives first (assuming equal transmission times). At present there is no information included in the state definition to allow this. Adding a field giving the local time of each process would provide this.

In the case when non-deterministic communication can occur, the successor state must be chosen non-deterministically and is not uniquely determined. This is acceptable (since it occurs in real programs) and must be modelled.

As explained previously, the periodic nature of the programs produces cyclic graphs of the program execution. Adding in a field giving the absolute time for each process would prevent this, since time is monotonically increasing. Instead relative values are used. One process is used as a reference and is set as the origin of the time axis in each state. The times of the other processes are given relative to the reference. More than one level of referencing can be used if all the processes do not have equal cycle times. This could reduce the number of unnecessary duplicate states.

A simple successor function executes one statement of one of the processes. Where processes must synchronize, each involved process must wait until all are ready to communicate. Other successor functions are possible, some of which may allow uninteresting states to be ignored. For example, three successive states in which the same variable is incremented can be replaced by one state incrementing the variable by three times the amount. Such a modification does not affect the subsequent analysis of the state space graph.

The analysis of the state space graph examines paths in the graph. Interesting points occur where state markers (start and end points of paths) and non-deterministic communication (points where paths split) are found. The state markers are statements to identify useful states, such as when a process is in the first statement of its cycle. One relatively useful successor function eliminates unneeded states by executing statements in other processes until it is impossible to avoid executing the marker or performing non-deterministic communication.

7.2 The exploration of state space

The model is simulated, generating a description of each successive state. When a state is repeated, then that branch of the simulation can terminate, knowing that the sequence of states that follows has already been examined. Branching in the sequence of states occurs when non-deterministic communication occurs and there is more than one valid successor state. The subset of the state space traversed while simulating the model can be represented as a directed, cyclic graph.

The process is illustrated with the model given below.

```
process client1

    report marker1
    send server req
    receive server rep
    think X1
    report marker2

process client2

    send server req
    receive server rep
    think X2

process server

    declare ANY
    receive ANY req
    think Y
    send [ANY] rep
    assign ANY undef
```

This model examines the first client-server variation suggested in section 6.1.1, where the clients spend different amounts of time processing. The *declare* statement in the *server* process causes an extra dimension to be added to the state space to represent the values of the variable *ANY*. The *assign* statement at the end of the server process sets the value of the variable back to a default value, to reduce the number of states that will be explored. For purposes of illustration, and to

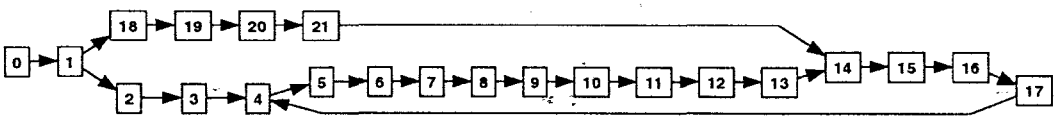

Figure 7.1: State space graph for the client-server model

limit the size of the state space, the range of the variables is constrained to the region:

- $X1 \geq Y + X2$
- $2Y + X2 \geq X1$
- $2X2 \geq X1$
- $Y + X1 \geq 2X2$

The state space graph for this model, under the given constraints is shown in Figure 7.1. State 2 corresponds to the second client getting the first message to the server, state 18 occurs when the first client gets the initial message to the server. Marker 1 occurs in states 0, 6, 12 and 20, while states 5, 11 and 19 contain marker 2.

The values that must be extracted from the graph are cycle times and latencies. Large amounts of data are available in the graph, relating to these values. It is possible to generate all possible sequences of the values that the model may produce. In order to provide a manageable amount of information, only steady state values are extracted. Often only extreme steady state values are given.

Cycle times can be found from the time it takes for a state corresponding to the execution of a marker function to recur. Given that a particular marker is executed in states A and B, cycle times corresponding to the time taken to go from states A to A, A to B, B to A, and B to B may be found.

Latency is slightly more complicated to calculate. Latency can be found by calculating the time taken to go from a state where a first marker occurs, to a corresponding state where a second marker occurs. Latency measures the time taken for information to move from one state to another. Thus the notion of corresponding markers requires that there be a message sent from the process with the first marker, after the marker is executed, and before it is executed again. This message must arrive at the process with the second marker. Time measurement ceases with the first execution of the second marker after the message arrives.

This description of latency does not cover the situation where there is more than one message that qualifies. In the case where the times for all messages are equal then this approach is sufficient. When the times differ, for example when the frequency of the first marker differs from that of the second, then further decisions need to be made. Choosing the messages on which the information is transmitted would require further effort on the part of the user. At present, all messages are traced, unless explicit constructs are used in the model to limit search paths.

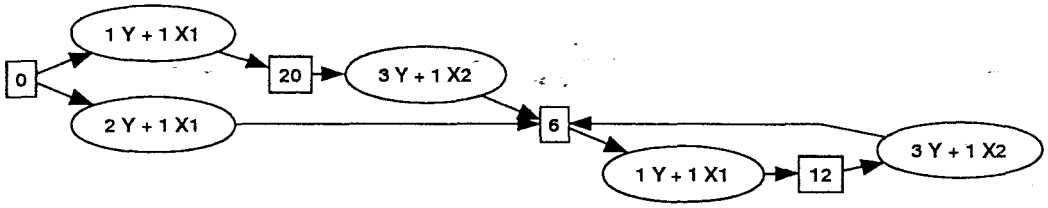

Figure 7.2: Graph extracted for cycle time analysis

The analysis of the state space graph to determine cycle time and latency can be simplified and unified once the relevant information is extracted. Only those states in which the required markers occur are extracted, together with the time interval between these states. This can be represented as a graph with the edges labelled by the time values. Cycles in this graph contain recurring states, these states can occur infinitely often in the execution of the model. Minimum and maximum latency/cycle time can be extracted by locating minimal/maximal cycles respectively.

The simplified graph can be easily extracted for cycle time analysis by searching for paths between marked states in the original state space graph. The corresponding time value is that which elapses in the process containing the marker along each path. The graph extracted for the client-server example is shown in Figure 7.2.

Generating this graph for latency is slightly more complex. Start states are those containing the initial marker while stop states contain the final marker. The times for messages to move from start states to stop states are required. This is done by tracing messages from start to stop states. A pointer to the current location of the message is kept while searching for paths between start and stop states. This is initialized to point to the process containing the marker in a start state, and is set to the destination process whenever a message is traced. The path is only complete when a stop state is reached with the pointer set at the process containing the final marker.

Calculating the values to assign to the edges in the latency graph is complicated by the possibility of messages changing processes. The time fields in the state definition are used to offset the time interval between two states by the difference in time values of the source and destination processes. When no change in process occurs, this reduces to the value used to calculate edge values for cycle time analysis.

The simplified graph does not contain the stop states, but joins the start states. Successive latency values occur according to the sequence of states containing the start marker. The states that may follow another are influenced by the value linking them. To obtain this value a message is traced through the reachability graph of the program. The path followed by this message may pass several branches in the graph. Each time it selects one route it is making a choice as to the programs behaviour at that point. Successor states that are linked by the value produced from this message are required to subscribe to the same behaviour. Thus these states must occur on, or follow an extension of, the path of the message.

In some cases no message arrives at the destination process. In this case the nodes are joined

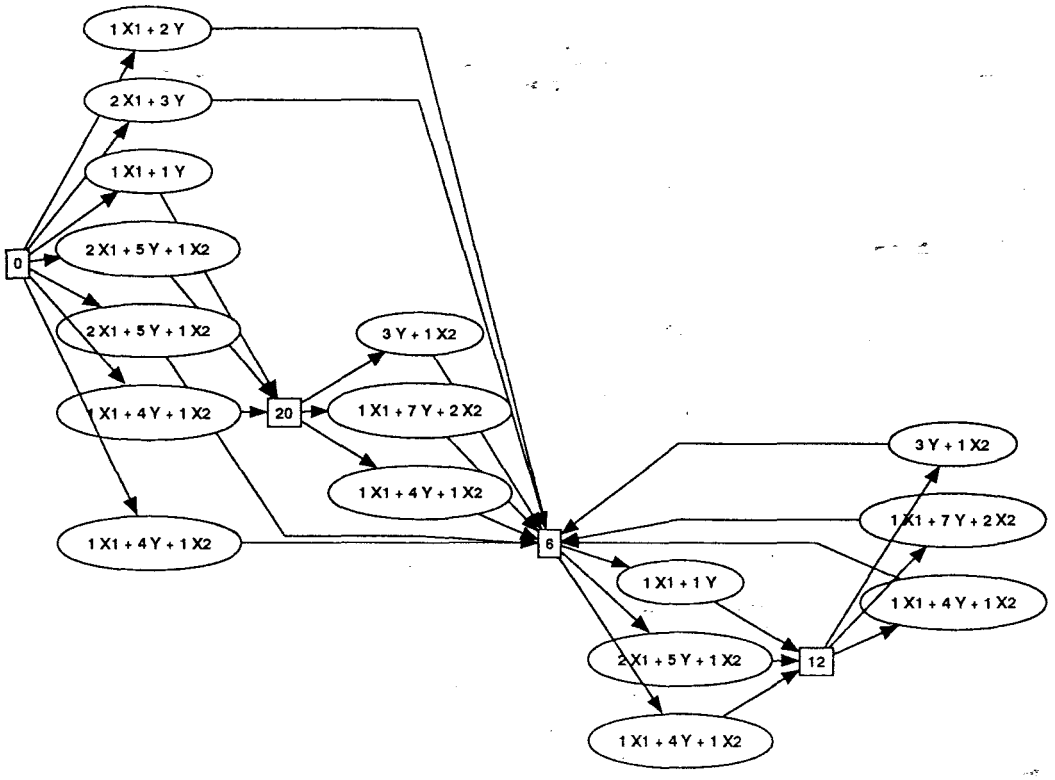

Figure 7.3: Graph extracted for latency analysis

by a NULL edge. This edge contains no value for latency.

The graph extracted to find the latency for the client-server example is shown in Figure 7.3. The effect of tracing every message can be clearly seen, when compared to the graph used to find cycle times. An example of one of the additional paths would be that from *client1* to the *server*, then from the *server* to *client2* (assuming the reply from the *server* is not traced, but instead execution is followed around the *server* loop until a request from *client2* arrives), around the loop in *client2*, back to the *server* and then back to *client1*. Obviously, in this example, this is not the path that should be used to calculate latency; in other cases such a message path might well be valid. The shortest path gives the fastest way to transmit messages between the two markers and almost always gives the minimum latency. The exception is when modelling control flow with message passing. In these cases special care must be taken to disable tracing of paths representing flow of control.

7.3 Use of Analytical Simulation

7.3.1 Relationship to other work

The initial version of the Analytical Simulation approach was inspired by work on data flow graphs. This approach, discussed in section 3.4.2, introduces the process activity versus time diagram upon which Analytical Simulation is based. In addition, analysis of data flow graphs provides the basis of a system capable of measuring both cycle times and latencies.

Analytical Simulation is obviously based on analytic modelling (section 3.2) and simulation (section 3.3) and combines the flexibility of simulation with the advantages of the analytic solution.

Analytical Simulation is extended to include state space analysis in Chapter 7, which improves its ability to handle non-determinism in finite time. State space analysis is an integral part of the analysis of Petri Nets (see section 3.4.1).

The areas in which Analytical Simulation stands out is in the generation of analytical solutions to the models, and in the automatic generation of constraints, defining the regions in which a solution holds. Analytical solutions using other approaches are limited to very simple models. Numerical simulation is usually used for larger models.

Analytical solution techniques are also used for performance prediction in compilers for superscalar architectures [Wan94]. Cost of code fragments is estimated and represented symbolically, as functions of the unknowns in the control structures. This allows estimation of the values of these unknowns to be delayed as long as possible. Comparison of the performance measures symbolically is discussed. For expressions given as polynomials of a single variable, the difference is taken, and the roots identified. The intervals between these roots identifies the range of variable values for which one expression is greater than the other. The sign of the difference in a particular interval indicates which expression is greater. This can be compared to the approach described in section 4.2.1 for linear expressions with multiple variables.

The use of analysis of traces bears a resemblance to work on CHITRA94 [Abr94] which performs analysis on collections of traces resulting from program execution. Analysis produces a stochastic model which fits these traces to a specified level of accuracy. The characteristics of the random variables contained in the model can be derived. In this way an analytic solution can be produced. Traces can be produced by simulation, rather than from system implementation. Filtering is performed on the traces to reduce the state space of the model that is produced.

The nature of the timing constructs with the Analytical Simulation approach bears some resemblance to work done on Time Petri Nets. Stochastic timing is considered inappropriate for modelling large, real-time systems by van der Aalst [Aal94], especially when real-time deadlines need to be considered. Instead he proposes the Interval Timed Coloured Petri Net model which assigns minimum and maximum deterministic firing times to the tokens in the Petri Net. A similar approach [Ber91] uses Time Petri Nets which have minimum and maximum times assigned to the firing of the transitions. Analysis in both cases involves generating a reachability graph of state classes, where a state class consists of a marking of the Petri Net, together with a range of firing times. The state class extension is necessary to produce a finite reachability graph. This bears some relation to the constraint generation of Analytical Simulation, since manipulation of the

ranges of firing times also involves manipulating sets of inequalities. The minimum and maximum values for the Petri Nets are pre-specified and not automatically generated. Analysis is also simplified by the use of numerical values for the limits, as opposed to the symbolic manipulation of Analytical Simulation.

7.3.2 Area of application

The development of the Analytical Simulation approach has produced a number of enhancements to the analysis process, which tend to limit the area of applicability of the approach. Ultimately, the approach is intended for use on message passing virtual reality systems, and is capable of that in all its manifestations. In its least sophisticated versions, Analytical Simulation is applicable to many other areas.

The original algorithm, described in section 4.1, is applicable to any architecture and model. At this stage, the only requirement is the ability to simulate the program, and to determine the length of the execution path for each process when synchronization occurs. With this very general approach there is no indication of when the analysis should terminate.

The next refinement, discussed when considering the implementation details, is to limit the simulation to models of message passing architectures. This relies on the fact that all virtual reality systems surveyed use message passing as their communication method. Having made this decision, the modelling language can be specified and the simulation engine can be implemented.

Analytical Simulation is then refined to allow for automatic termination of the analysis and to permit finite analysis in the presence of non-deterministic constructs. This state space analysis requires that the region of state space that can be reached by the model (reachability graph) is finite. For real-time systems which do not terminate, this requirement means that the model must be periodic. This is true for all models of virtual reality systems considered in this document.

7.3.3 Limitations

7.3.3.1 Discontinuities in the results

On occasion results quoted from Analytical Simulation analysis for different constraint regions may show discontinuities in the region of overlap. These may often be associated with the presence of transients in the boundary region, which are not explicitly mentioned.

The presence of transients usually requires the existence of specific relationships between the variables in the system being modelled. The length of the transient is also affected by the relative values of these variables. Transients can arise, for example, when a process blocks waiting for a second process. The length of time the first process blocks may decrease during each cycle of the system. At some point the first process does not block any longer, possibly forcing the second process to block instead, and a sudden change in the performance parameters results. As the transient behaviour stops a change is seen in system performance. When viewing only the steady state values of the performance values, this is seen as a discontinuity as one moves from one constraint region to another.

7.3.3.2 Constraints on the variables

The Analytical Simulation technique differs from many other performance modelling approaches in that it is not stochastic. The presence of random variables would make the simulation step impossible. This technique requires that two expressions be compared; this is not possible should the expressions contain random variables.

The automatic generation of constraint regions allows a degree of freedom in the variable values, since the same expression for the performance value applies throughout a particular region. Rather than having to specify the range or distribution of variables values beforehand, the Analytical Simulation approach allows the nature of the model to define these. Variations in the values of the variables within a constraint region do not affect the nature of the program behaviour.

7.3.3.3 Limitations on the systems being modelled

Analysis, especially that using state space constructs, requires the presence of repeated values. Thus the system being modelled should be periodic. This is usually the case in systems that run continuously, especially virtual reality systems.

The complexity of the analysis, and the time required to perform it, grow rapidly with the size of the model, especially if there are many points at which a non-deterministic choice is possible. Analysis may have to be limited to relatively small numbers of processes; however these are usually enough to exhibit the characteristic behaviour of the system.

7.3.3.4 Interpretation of the results

The interpretation of the results can be complicated, though possibly less so than for many other techniques. Since there is no explicit dependence in the model on the number of processes, this dependency needs to be extracted by repeating the analysis for different numbers of simulated processes, and then deducing the dependency. The complexity of the output often benefits from human intervention. The order in which constraints are introduced is not always the most economical, especially since that step is independent of the nature of the values being derived.

7.3.4 Advantages

Given that the Analytical Simulation approach has some limitations, it also has several features that are not found in other performance analysis and prediction tools.

7.3.4.1 Best at the analysis of virtual reality systems

It is intended for use on virtual reality systems and so provides the two measures critical to determining the abilities of such a system. The cycle time value is found in a number of other analysis techniques. Very few techniques produce a value for latency, and these are not suitable for use with virtual reality systems. This approach has the advantage that much of the work required to produce the one value is also useful for producing the other.

7.3.4.2 Performance comparison

Producing output as a symbolic expression, given as a function of the delays in the system, is another feature specific to this approach. While some other approaches can give similar output, the number of variables which can be used are usually limited, and predefined by the paradigm. With Analytical Simulation, any processing or communication delay can be represented by a variable, and the effect of that variable can be clearly seen in the result of any analysis.

The symbolic nature of the output means that Analytical Simulation is particularly suitable for performance comparison. Rather than performing comparisons based on the performance on a particular architecture, the relative complexities of the different approaches can be compared.

The automatic generation of constraints is found only in Analytical Simulation. In this way, the manner in which variations in the system parameters affect the performance parameters can easily be examined.

7.3.4.3 Automatic generation of constraints

The inability to use random variables is not necessarily a significant limitation. A number of other performance analysis measures have the same limitations (for example those based on Data Flow Graphs). For systems where there is not much dependence on random events such as real-time DSP applications, and to a certain extent virtual reality applications, processing load is almost constant, often within fine tolerances. Other performance prediction approaches, Petri Nets in particular, are not well suited to the use of deterministic firing times. Markovian analysis of Petri Net models with deterministic firing times is very difficult [Kan92].

The automatic generation of constraints allows a degree of freedom for the variable values. Variables can take on any value within the limits given by a set of constraints without changing the behaviour of the system. The only effect on performance is the result of reevaluating the performance expression corresponding to that constraint region.

7.3.4.4 Support for non-determinism

An advantage to this approach is the ability to handle non-deterministic constructs and still produce useful results. Other approaches [Men93], examining program traces, have difficulty with non-deterministic receives.

An advantage of non-determinism is that it allows choices concerning the behaviour of the system to be delayed until run-time. This simplifies the programmers task in that ordering of events does not become crucial, but can lead to the system taking a less than optimal execution path. In some cases the optimal path is not stable; once the system leaves that path it will never return. Analysis of systems containing non-determinism can detect these cases, and the model can be adjusted to eliminate the poor choices.

The origin of the Analytical Simulation approach means that it is easy to produce a trace of process activity and communication, allowing the sequence of events corresponding to optimal execution paths to be easily extracted. The close correspondence between Analytical Simulation models and the system from which they are derived makes it relatively simple to translate between

the two. Alterations in the model to improve performance can readily be carried across to the original system.

7.3.4.5 Automatic transient analysis

Transient analysis is straightforward using Analytical Simulation. The most common transient arising is that from system startup, and the effect of this is clearly visible once latency and cycle times are extracted from the state space graph.

A given model can easily be modified to introduce perturbations in variable values. Extracting the performance values immediately after such an event can be done with ease, and both the value and length of the transient can be seen. The automatic generation of the constraints in the variables associated with a model, implicit in the Analytical Simulation approach, also provides some revealing information about transients in the system that may occur only for certain variable values (see section 6.1.4).

7.3.4.6 Proof of program properties

Another useful side effect of Analytical Simulation with automatic constraint generation is the ability to determine certain properties of the model, such as the absence of deadlock. During simulation, all possible states of the program are visited; checks for deadlock and other properties of the program can easily be included. Checking for deadlock is in fact obligatory, since that condition interferes with the periodic nature of the system. This ability is shared by other performance analysis approaches which enumerate the state space of their models, such as Petri Nets.

7.4 Conclusion

The state space extensions to the Analytical Simulation approach overcome the two major difficulties in the practical application of the technique. The performance implications of non-deterministic constructs can be completely assessed. Complete analysis of the model is possible, without the need to choose cut-off points. This results in improved performance as duplication of effort is eliminated.

Section 7.3 describes the relationship of Analytical Simulation to other performance analysis approaches. It gives details of the nature of the systems to which Analytical Simulation is best suited. The strengths and weaknesses of Analytical Simulation are discussed. Its limitations do not affect its applicability to the desired category of systems, while its advantages make it well suited to the desired form of analysis. Analytical Simulation is ideal for the performance analysis and comparison of parallel and distributed virtual reality systems.

Chapter 8

Verification of Analytical Simulation

An approach suitable for the analysis of parallel virtual reality systems was developed and refined in Chapters 4 to 7. This chapter describes its application to the decomposition techniques used in the collision detection algorithms, which were introduced in Chapter 5. This starts with the analysis of the message passing and master-slave variations of the distributed collision detection algorithm. The client-server approach to collision detection has been examined in detail in Chapter 6. A comparative evaluation is also made of each technique in this chapter.

The results obtained from this analysis are compared with values measured from implementations of each algorithm. The implementation is carried out on two architectures: a Transputer cluster using synchronous communication with a well-connected network, and a group of workstations connected by Ethernet, a shared asynchronous medium.

The decomposition strategies used in implementing the collision detection algorithm are commonly used in virtual reality systems. Successful analysis and validation thus holds implications for the use of the analysis technique with virtual reality systems. In addition, some confidence in the validity of the Analytical Simulation approach can be gained, before applying it in more varied situations.

8.1 Message Passing paradigm

This approach is more common with hand written applications. Each process communicates using direct, explicit message passing calls.

A simplified outline of the algorithm given earlier in section 5.3.1 is presented below:

```
while (running ())

    Send message to every other process (own position)
    Receive message from every other process (their position)
    Calculate time of first collision for this molecule
```

```
Send message to every other process (local best time)
Receive message from every other process (their best time)
Move molecule
```

This model assumes non-blocking communication, allowing processes to send and receive simultaneously. To stay in line with the blocking calls of the other approaches, the model is altered slightly. A second process is added to buffer the incoming communication. Each processor is assumed to contain two processes, a molecule process and its accompanying buffer process. The updated molecule appears as follows:

```
Buffer process:
    while running ()
        Receive start message from local molecule
        Receive data from all other molecules
        Send receive_complete to local molecule

Molecule process:
    while (running ())
        Send start message to local buffer
        Send message to every other process
        Receive receive_complete from local buffer
        Calculate time of first collision for this molecule
        Send start message to local buffer
        Send message to every other process
        Receive receive_complete from local buffer
        Move molecule
```

Having developed the model, we now need to identify the areas where sequential computation occurs. As in the example in the previous section, the time taken to move the molecule, and perform any other manipulations to it, is denoted by the variable X. In that example the variable Y was used to denote the time it takes for the server to obtain the required data, and to package it for transmission. In this approach it can reasonably be modelled by a processing interval before firing off the batch of sends to each of the other molecules.

The model used for the Analytical Simulation tool is as follows:

```
replicate N
    process molecule#buffer
        receive molecule# start
        replicate [N-1]
            receive ANY data
        endreplicate
        send molecule# finished
```

```
process molecule#
    replicate 2
        think Y
        send molecule#buffer start
        replicate [N-1]
          send molecule[###+((###+1)>#)]buffer data
        endreplicate
        receive molecule#buffer finished
    endreplicate
    think X
    report moleculeend#
endreplicate
```

Analysis of this model gives a latency/cycle time value of $2Y + X$, irrespective of the number of molecules involved. This result assumes that each molecule occupies a separate processor and that communication takes no time at all. This latter assumption affects the implementation of the model significantly. Sending off messages to the other molecule processes can be done naïvely by assigning a number to every process and simply sending in numeric order. When the program starts, almost all of the processes attempt to send a message to the buffer process of the first molecule. The zero communication costs allow this load imbalance to pass unnoticed.

Before examining the effects of communication costs, it is beneficial to decide on the manner in which communication is modelled. In [Men95] the times taken for the send and receive message passing operations are modelled as linear functions of the length of the message. However the time for the receive operation in this case includes the time spent waiting for the message to arrive. This factor is automatically included when using Analytical Simulation due to the nature of the implementation of the simulation. In [Adv94], communication time, as well as the time for the processor to send and receive data, is modelled as a linear function of the message size. In the model at present, two types of message are being transferred. For each of these messages, the length is fixed and so the costs are constant. For simplicity, the two types of message are assumed to have equal length and so total overhead for a single message is modelled by the single variable C. The sending process blocks from the time it executes the *send* until a period C after the receiving process has executed the corresponding *receive* command. The receiving process is occupied with receiving the message for period C after synchronization with the sending process. The extra level of detail involved in modelling time taken for the processor to execute the send and receive commands can be implemented, if necessary, by including appropriate delays before *send* commands and after *receive* commands.

Repeating the analysis of the model with the naïve message passing order now that communication costs are considered yields the following results: In the three processor case, the latency/cycle time is $2Y + X + 6C$. Occasional values of $2Y + X + 7C$ are seen, immediately followed by $2Y + X + 5C$, keeping the average value unchanged. The four processor case is even more complex, due to the greater potential for non-determinism. Steady state values in this case can range between $2Y + X + 8C$ and $2Y + X + \frac{21}{2}C$.

The minimal values occur when the amount of time spent waiting for communication to occur is minimized. The destination of the messages sent is fixed, but the receiving processors may select the source in such a way that blocking in future communication is reduced. For the first send, $N-1$ processors are attempting to communicate with the first node and $N-2$ must end up waiting. This wastes at least $N-2$ communication periods which can be used to improve performance.

Optimal performance values are $2C+2Y+X$, when $N=2$, and $2NC+2Y+X$ for values of $N>2$.

A more reasonable approach is to have each molecule start off by sending a message to its (numeric) successor, and send to successive successors thereafter. This involves replacing the line in the molecule process that is used to transmit data to the other processes with:

```
send molecule[((#+###)%N)+1]buffer data C
```

This approach yields immediate improvement. Latency/cycle time is now reduced to $2(N-1)C+2Y+X$, where only the communication causes dependency on the number of processors.

The advantages of non-determinism are few. It simplifies the implementation of the code, leaving difficult performance considerations to be resolved by the system at runtime. It allows programs to respond to changes in the requirements of the various processes without explicitly planning for these in advance. The disadvantages on the other hand are serious. Non-determinism complicates the analysis of programs by exploding the possible execution paths. In a number of the examples examined already it is possible to get trapped into a less efficient mode of execution. In future discussions, only the optimal solutions are presented.

8.2 Client-Server paradigm

The client-server model is analysed extensively in the preceding sections. The algorithm is introduced in section 5.3.2 and analysed in section 6.2. The final result given is $3NY+X$ as the latency/cycle time. This result assumes instantaneous communication.

Adding in an extra value C, for the communication overhead yields the following result for latency/cycle time:

$(N-1)C \geq X$: $\quad 6NC+3NY$

$X \geq (N-1)C$: $\quad (5N+1)C+3NY+X$

Analysis of the analytical version of the performance of the system can be valuable in its own right. The dependence on the variable Y (time spent in the server) in all the results indicates that the system is always dependent on this variable. Thus any decrease in Y improves performance. The variable X on the other hand, can be made sufficiently small such that the performance of the system does not depend on it. There is a limit beyond which performance improvements must result from increased bandwidth or faster server access.

8.3 Master-Slave paradigm

The algorithm for the master-slave implementation of the collision detection algorithm (see section 5.3.3) can also be modelled easily. The slave process is the more interesting and is outlined below. The model of the master process is almost identical to the algorithm.

```
Slave Process:

    receive message from master
    case message of
        reqdata :
            process for a period of duration Y (look up data)
            send heredata to master
        findyourbest :
            (assume this is very fast)
            send mybest to master
        rununtil :
            process for a period of duration X (move
            molecule)
```

The communication time is represented by the variable C.
The model used for the Analytical Simulation of this system is shown below:

```
replicate N

    process slave#
        report start_slave#
        receive master COMMAND
        if COMMAND == reqdata
            think Y
            send master heredata C
        endif
        if COMMAND == findyourbest
            send master mybest C
        endif
        if COMMAND == rununtil
            think X
        endif

endreplicate
```

```
process master
    report startmaster
    replicate N
        send slave# reqdata C
    endreplicate
    replicate N
        receive slave# heredata
    endreplicate
    replicate N
        send slave# findyourbest C
    endreplicate
    replicate N
        receive slave# mybest
    endreplicate
    replicate N
        send slave# rununtil C
    endreplicate
```

Finding a point to attach markers for the measurement of latency and cycle time is less obvious in this case. The slave cycles three times to carry out the equivalent of a full collision detection cycle in one of the other two approaches. The alternatives involve either measuring a slave and adding the results of three cycles, or measuring the cycle time of the master process. The former option is chosen in this case, since it provides more information, allowing greater insight into the system.

The results are shown below:

$Y \geq (N-1)C \geq X$:

$(N+1)C+Y-X$	*Reqdata*
$2NC$	*Findyourbest*
$NC+X$	*Rununtil*
$(4N+1)C+Y$	Total

$Y \geq (N-1)C, X \geq (N-1)C$:

$2C+Y$	*Reqdata*
$2NC$	*Findyourbest*
$NC+X$	*Rununtil*
$(3N+2)C+Y+X$	Total

$(N-1)C \geq Y, (N-1)C \geq X$:

$2NC - X$	*Reqdata*
$2NC$	*Findyourbest*
$NC + X$	*Rununtil*
$5NC$	Total

$X \geq (N-1)C \geq Y$:

$(N+1)C$	*Reqdata*
$2NC$	*Findyourbest*
$NC + X$	*Rununtil*
$(4N+1)C + X$	Total

8.4 Modelling on a MIMD machine

The preceding analyses have all dealt with a model of the various approaches to implementing distributed collision detection. To confirm that the model is a fair reflection of reality it should be compared with the performance of an actual implementation.

This section discusses some of the results obtained by implementing the three collision detection algorithms on a cluster of Transputers. Each Transputer is an independent processor which can exchange data via a communication link with one of its four neighbours. Communication is synchronous, with both processes blocking while transfer of data occurs.

For the purposes of this experiment the processors are arranged in a completely connected network. Given four links, it could be expected that five Transputers could be connected this way. One link from one of the processors is required for communication with the outside world, limiting maximum processors to four when complete connectivity is required.

8.4.1 Message Passing on Transputers

The message passing model initially tested is the unoptimized one described in section 8.1. The processing overhead modelled by the variable Y is extremely small compared to the other values and is left out of the model. The cycle time predicted is thus $2NC+X$, for $N > 2$, $2(N-1)C+X$ otherwise.

The implementation of the algorithm also includes an artificially produced delay to increase the value of X, otherwise that to would be too small to produce a noticeable effect. The value of X was measured as part of the experiment and came out at about 5ms. An attempt was made to find a value for C, using a variety of small test programs. A message size of 1000 bytes is used for all communication. This value fluctuated so wildly according to the nature of the communication that it was decided to calculate it from one of the measured cycle times using the predicted formula.

The different values for C during the tests result from the nature of the communication protocol on the Transputer links. Each byte sent is acknowledged by a few bits sent in the opposite direction.

N	X/[ms]	C/[ms]	Cycle time/[ms] Practice	Cycle time/[ms] Theory	% Theory/Practice
1	5.06	1.02	5.13	5.06	98.6
2	5.06	1.02	7.09	7.09	100 (Fixed)
3	5.06	1.02	11.08	11.15	100.6
4	5.06	1.02	13.11	13.18	100.5

Table 8.1: Performance of the MIMD message passing system

N	X/[ms]	C/[ms]	Cycle time/[ms] Practice	Cycle time/[ms] Theory	% Theory/Practice
1	5.16	0.92	5.12	5.16	100.8
2	5.16	0.92	7.14	7.01	98.1
3	5.16	0.92	8.71	8.88	101.6
4	5.16	0.92	10.74	10.69	99.5

Table 8.2: Performance of the optimized MIMD message passing system

Consequently programs which use extensive bidirectional communication such as this one find a different link speed to those which use one way traffic (such as the other two algorithms).

In general, it may be expected that the predicted results are always less than those achieved in practice since other delays are left out of the model. This is less noticeable when calculating the variables from the model's predictions since these extra delays get added to the variable value, increasing its value beyond what it may really be in practice. The value of C calculated in this case is about 1ms.

Table 8.1 shows the values for different numbers of processors and provides a comparison of real and predicted values.

If the communication is optimized as discussed previously, then the model predicts cycle times of $2(N-1)C+X$, for all values of N. Results from an implementation of this model are shown in Table 8.2. Since the relationship between cycle time and N is a linear one, a linear regression analysis is used to find the values of X and C in this case.

The predicted values from the model agree well with the values measured from the implementation.

8.4.2 Client-Server on Transputers

The cycle time for the client-server model is found to be $(5N+1)C+3NY+X$ for $X \geq (N-1)C$, and since $X > 5C$ in the implementation, this value applies in all situations described here. The value of C is calculated from the measured cycle time in the single client case which has a predicted time of $6C+3Y+X$, where Y is insignificant compared to the other variables. This value is lower than those calculated for the message passing case, since the links are only being used in one direction at any time. The results are shown in Table 8.3.

Once again the results are in agreement.

N	X/[ms]	C/[ms]	Cycle time/[ms] Practice	Cycle time/[ms] Theory	% Theory / Practice
1	5.08	0.88	10.35	10.35	100 (Fixed)
2	5.07	0.88	14.67	14.75	100.5
3	5.08	0.88	19.53	19.15	98.1
4	5.03	0.88	23.84	23.50	98.6

Table 8.3: Performance of the MIMD client-server system

N	X/[ms]	C/[ms]	Cycle time/[ms] Practice	Cycle time/[ms] Theory	% Theory / Practice
1	5.00	0.88	9.37	9.37	100 (Fixed)
2	5.01	0.88	12.76	12.88	101.0
3	5.01	0.88	16.20	16.39	101.2
4	5.01	0.88	20.08	19.89	99.1

Table 8.4: Performance of the MIMD master-slave system

8.4.3 Master-Slave on Transputers

The master-slave model in section 8.3 yields a cycle time of $(4N + 1)C + X$ for the situation in which the implementation is tested, namely $X > 5C > Y = 0$. Results are given in Table 8.4.

Once again the results agree.

To justify the above statement and the others like it given in the previous few sections, it may be worth describing some of the sources of error and their approximate magnitude. The system is relatively homogeneous, all processors are identical and should all communicate at the same speed. Measurements made on a random sample reveal differences of up to 15% in a few cases. This may justify attempting to find an average value for C as is done above. Other overheads, such as extra processing not accounted for in the measurement of X, may become more noticeable as the number of processors increases. Measurement of the largest of these, the intersection calculations for the collision detection, showed this overhead to be about 1% of the measured value of X.

8.5 Modelling with Ethernet

Modelling the collision detection algorithms on a network of PC's connected using Ethernet involves several changes to the model. Ethernet is a shared medium, and only one process is permitted to communicate at a time. All processors are connected using the same cable. This is modelled by introducing an extra process to represent the medium. Other processes must first request permission from the medium before performing interprocessor communication. The medium ensures that only one process communicates at a time.

The models below assume that only the processes in the system being modelled have access to the network cable. The effect of other systems sharing the cable is ignored. The models do not implement other Ethernet protocols such as packet collision detection and random backoff.

Monitoring of cable transmission shows that these effects occur extremely infrequently when the medium is used for single, synchronized systems such as those modelled.

The other difference over the MIMD model is that communication is non-blocking. Buffer processes are included to simulate this aspect. The nature of the models, in that every message requires an acknowledgement, means that the size of the buffers can be easily predicted in advance.

8.5.1 Client-Server on Ethernet

This section examines the client-server model on Ethernet, examining successive refinements to the model until a reasonable degree of accuracy is achieved. The first step is to model Ethernet simply as having non-simultaneous communication which is non blocking.

The client-server model is such that each message requires an answer, so each client needs to be able to buffer only one message, and the server needs to buffer at most N. The buffering process includes receiving messages and transferring them to the destination process in the correct order. The simple model outlined below also uses the buffer processes to model the transfer of the message on the Ethernet cable, handling contention for the medium and modelling communication overheads.

```
process clisen[i] (N processes)

    receive client[i] MESSAGE
    send medium wantmedium
    think C
    send medium givemedium
    send server MESSAGE

process serreq[i] (N processes)

    receive server MESSAGE
    send medium wantmedium
    think C
    send medium givemedium
    send client[MESSAGE] somedata

process client[i] (N processes)

    send clisen[i] reqdata
    receive SERVER somedata
    send clisen[i] reqsummary
    receive SERVER somedata
    think X
    send clisen[i] reqsummary
    receive SERVER somedata
```

```
process server

    receive CLIENT MESSAGE
    if MESSAGE == reqdata
        think Y
        send serreq[COUNT] CLIENT
    if MESSAGE == reqsummary
        assign COUNT [COUNT+1]
        if COUNT == [N+1]
            think N*Y
            send serreq[i] i (N times)

process medium

    receive SOMEONE wantmedium
    receive [SOMEONE] givemedium
```

Comparing the predictions for this model with the results from an actual implementation yields poor results. The parameters for the implementation are measured at 5ms for X, 0.02ms for Y and 2.9ms for C (A packet size of 1000 bytes is used). For between 1 and 5 molecules practice differs from theory by up to 50%. Examining the real system shows substantial overheads in sending and receiving data, as opposed to simply transmitting the information across the physical cable. At the specified bandwidth of 10Mbits/s, transmission should take only about 1ms. The extra 1.9ms measured by measuring the round time for a message to pass from one machine to another and then back again includes extra time required to get the message through the hardware and a rather extensive array of network drivers.

A portion of the model, modified to take these overheads into account is shown below:

```
process clisen[i]

    receive client[i] MESSAGE
    think S1
    send medium wantmedium
    think C
    send medium givemedium
    think S2
    send server MESSAGE

process serreq[i]

    receive server MESSAGE
    think S1
    send medium wantmedium
```

```
think C
send medium givemedium
think S2
send client[MESSAGE] somedata
```

The values $S1$ and $S2$ represent the overheads of sending and receiving messages from the network respectively. This model only requires the sum of $S1$ and $S2$; the relative sizes of these two variables are not needed at this point.

Even this model is not much more accurate. Examination of packet transmission for a variety of different communication patterns reveal further interesting behaviour introduced by some underlying network software. This software contains some interesting buffering mechanisms, which complicate the interpacket transmission times. Since these network drivers are documented only in expensive places, a number of experiments were performed to increase the accuracy of the model.

Firstly, the relative sizes of $S1$ and $S2$ are needed. These were measured using an oscilloscope connected directly to the Ethernet. Two machines were programmed to bounce a packet between them, as described above. The sending and receiving machines produce a pulse to mark the limits of the $S1$ and $S2$ periods. These pulses were monitored on a second channel of the oscilloscope. This gave the values of $S1$ and $S2$ as 0.9ms and 1.0ms respectively.

A second experiment examined a simple client-server system in which the server would receive a request from each client until every client had sent one, and then would send replies to each as quickly as possible. The results from examining the traffic patterns give rise to a number of interesting refinements to the model. The behaviour of the send operation depends on the time of the last communication. If the network driver is still occupied in sending the last message, the new message is placed straight into a buffer and the sending process can continue immediately. If the driver is idle, the sending process is required to block for a period $S1$ before the message is placed on the wire and the sender allowed to continue. A buffered message cannot be sent as soon as the wire is idle again, instead it has to wait an additional period $S1$.

This asymmetric communication complicates the model. Fortunately this is limited to the server process, since the client processes never send two messages in quick succession. The enhanced model is shown below.

```
replicate N

    process serrec#

        receive clisen# MESSAGE
        think S2
        send server MESSAGE

    process clisen#

        receive client# MESSAGE
        send medium wantmedium
        send medium givemedium
        send serrec# MESSAGE
```

```
        process cliserrec#
            receive comm somedata
            think S2
            send client# somedata
    endreplicate

    process buffersend
        send bufferlist get
        receive bufferlist DEST
        send numsending bufinc
        receive numsending ok
        think S1
        send comm DEST

    process bufferlist
        receive CLIENT MESSAGE
        if MESSAGE == get
            if LENGTH == 0
                assign READY [READY+1]
            endif
            if LENGTH != 0
                send buffersend BUF[HEAD]
                assign HEAD [((HEAD+1))%(N-1)]
                assign LENGTH [LENGTH-1]
            endif
        endif
        if MESSAGE != get
            if READY == 0
                assign BUF[TAIL] MESSAGE
                assign TAIL [((TAIL+1))%(N-1)]
                assign LENGTH [LENGTH+1]
            endif
            if READY != 0
                assign READY [READY-1]
                send buffersend MESSAGE
            endif
        endif
```

```
process numsending

    receive CLIENT MESSAGE
    if MESSAGE == get
        send [CLIENT] [COUNT]
    endif
    if MESSAGE == inc
        assign COUNT [COUNT+1]
    endif
    if MESSAGE == bufinc
        if COUNT != 0
            assign ISWAIT 1
        endif
        if COUNT == 0
            assign COUNT 1
            send buffersend ok
        endif
    endif
    if MESSAGE == dec
        if ISWAIT == 0
            assign COUNT [COUNT-1]
        endif
        if [ISWAIT] != 0
            send buffersend ok
            assign ISWAIT 0
        endif
    endif

process comm

    receive SOMEONE DEST
    send medium wantmedium
    send medium givemedium
    send [DEST] somedata
    send numsending dec

process odi

    receive server DEST
    send numsending get
    receive numsending SENDING
```

```
        if [SENDING] == 0
            send numsending inc
            think S1
            send comm DEST
            send server sent
        endif
        if [SENDING] != 0
            send bufferlist DEST
            send server sent
        endif

    replicate [N]
        process client#
            report start_client#
            send clisen# reqdata S1
            receive cliserrec# somedata
            send clisen# reqsummary S1
            receive cliserrec# somedata
            think X
            send clisen# reqsummary S1
            receive cliserrec# somedata
    endreplicate

    process server
        receive CLIENT MESSAGE
        if MESSAGE == reqdata
            think Y
            send odi cli[CLIENT]
            receive odi sent
        endif
        if MESSAGE == reqsummary
            assign COUNT [COUNT+1]
            if COUNT == [N+1]
                replicate N
                  think Y
                  send odi cliserrec#
```

```
            receive odi sent
          endreplicate
          assign COUNT 1
        endif
      endif

process medium

      receive SOMEONE wantmedium
      think C
      receive [SOMEONE] givemedium
```

The model works as follows: Messages sent from the clients are directed to the corresponding *clisen* processes which model the network drivers. The communication between these is modelled as blocking for period $S1$, the expected delay for an idle network driver. This driver process negotiates with the medium (Ethernet cable) for a free gap in which to send its packet, after which it passes the packet to one of the buffers for incoming packets on the server.

Communication in the other direction is slightly more complicated. Messages sent from the *server* process go to the *odi* process which models the network driver. This checks to see if a global variable indicating the communication state is set. This is modelled by the process *numsending*. If the driver is not idle it buffers the message and allows the server to continue immediately. If it is idle, it delays for period $S1$ before allowing the server to continue. At the same time it passes the message to the process that models sending the packet over the cable.

This process, *comm*, also waits for a gap in the traffic on the wire before taking control, sending the packet and releasing control. After the packet is sent, process *comm* sends a message to the *numsending* process to indicate that the system is no longer engaged in the transmission of a packet. This action triggers the process *buffersend* which feeds packets out of the buffer, if the buffer is not empty.

This process, which extracts messages one at a time from the buffer, is activated when the first message arrives in the buffer. It sets up a callback with the *numsending* process which allows it to continue as soon as the previous packet is transmitted.

The *bufferlist* process maintains the queue of messages in the buffer.

A comparison between the measured and predicted results from this model is shown in Table 8.5. The values of the variables are shown in Table 8.6.

The theoretical values are less than those measured as may be expected, since the model ignores various small overheads. The accuracy is less than for the Transputer model. This may be attributed to lower precision in the measurement of the variables; the accuracy of measurement is estimated at about 5%. There is also greater inhomogeneity in the machines involved. Variations in the values measured by the machines (X and Y) are about 10%.

Having shown that the Analytical Simulation technique can successfully model this environment (albeit at the cost of an extremely complex model) for future modelling the network stack is replaced with one that is simpler to model. This network driver does no buffering and so simplifies

N	Cycle time/[ms] Practice	Theoretical value	Cycle time/[ms] Theory	% Theory / Practice
1	22.7	$6S1 + 6C + 6S2 + 3Y + X$	22.1	97.7
2	26.6	$8S1 + 8C + 6S2 + 3Y + X$	25.8	97.0
3	31.9	$9S1 + 10C + 7S2 + 4Y + X$	29.6	92.6
4	38.1	$12S1 + 12C + 8S2 + 4Y + X$	35.2	92.5
5	44.7	$14S1 + 15C + 9S2 + 4Y + X$	40.8	91.2

Table 8.5: Performance of the buffered Ethernet client-server system

X	5.00 ms
Y	0.02 ms
$S1$	0.92 ms
C	0.92 ms
$S2$	1.00 ms

Table 8.6: Variable values for the buffered Ethernet client-server model

the model, at the expense of limiting the number of processes that can be used before dropped packets become a problem.

The model for this simple network stack is very similar to the one initially proposed to cater for the sending and receiving overheads. The major difference is an addition to the receive stage of the server to prevent the $S2$ stages running in parallel. This is not needed on the client side because only one packet is received at a time. The model is shown below.

```
process getrec

    receive SOURCE want
    receive [SOURCE] give

replicate N

    process serrec#

        receive clisen# MESSAGE
        send getrec want
        think S2
        send getrec give
        send server MESSAGE

    process clisen#

        receive client# MESSAGE
        think S1
        send medium wantmedium
        send medium givemedium
        send serrec# MESSAGE
```

```
    process cliserrec#
        receive sersen somedata
        think S2
        send client# somedata
endreplicate

process sersen
    receive server MESSAGE
    think S1
    send medium wantmedium
    send medium givemedium
    send [MESSAGE] somedata

replicate [N]
    process client#
        report start_client#
        send clisen# reqdata
        receive cliserrec# somedata
        send clisen# reqsummary
        receive cliserrec# somedata
        think X
        send clisen# reqsummary
        receive cliserrec# somedata
endreplicate

process server
    init
        assign COUNT 1
    endinit
    receive CLIENT MESSAGE
    if MESSAGE == reqdata
        think Y
        send sersen cli[CLIENT]
    endif
    if MESSAGE  == reqsummary
        assign COUNT [COUNT+1]
        if COUNT == [N+1]
            replicate N
              think Y
```

N	Cycle time/[ms] Practice	Theoretical value	Cycle time/[ms] Theory	% Theory / Practice
1	20.7	$6S1 + 6C + 6S2 + 3Y + X$	20.4	100.1
2	24.1	$8S1 + 8C + 6S2 + 3Y + X$	23.6	97.8
3	28.7	$9S1 + 10C + 7S2 + 4Y + X$	27.1	94.4
4	33.9	$10S1 + 15C + 7S2 + 2Y + X$	32.3	95.1
5	40.2	$12S1 + 18C + 8S2 + 2Y + X$	37.3	92.8

Table 8.7: Performance of the standard Ethernet client-server system

X	5.10 ms
Y	0.02 ms
$S1$	0.70 ms
C	0.91 ms
$S2$	0.93 ms

Table 8.8: Variable values for the standard Ethernet client-server model

```
            send sersen cliserrec#
          endreplicate
          assign COUNT 1
        endif
    endif

process medium

    receive SOMEONE wantmedium
    think C
    receive [SOMEONE] givemedium
```

The results for this model are shown in Table 8.7, with variable values given in Table 8.8.

8.5.2 Message Passing on Ethernet

The message passing model is also based on the simple Ethernet communication principles introduced in section 8.5.1. The major difference is that every process receives a number of messages in a short space of time. Thus the protection system used in the client-server model to prevent parallel receives is extended to each of the processes in this model. The relevant portions are shown below.

```
replicate N

    process clientr#

        receive client# start
        replicate [N-1]
```

```
            receive mol#recfrom[##+((##+1)>#)] somedata
        endreplicate
        send client# allhere

    process client#
        replicate 2
            send clientr# start
            replicate [N-1]
              send mol#sen mol[###+((###+1)>#)]recfrom#
            endreplicate
            receive clientr# allhere
        endreplicate
        think X

    process mol#sen
        receive client# MESSAGE
        think S1
        send medium getmedium
        send medium givemedium
        send [MESSAGE] data

    process get#rec
        receive FROM wait
        receive [FROM] give

    replicate N
        process mol#recfrom##
            receive mol##sen data
            send get#rec wait
            think S2
            send get#rec give
            send clientr# somedata
    endreplicate
endreplicate
```

Since the value for Y in this case is insignificant compared to the other variables, it is dropped from the model. The results for this approach are shown in Table 8.9. The variable values are the same as those given for the client-server approach in Table 8.8.

8.5.3 Master-Slave on Ethernet

The master-slave model resembles the client-server version quite closely. Protection against parallel receiving is implemented only for the master. Since sending for the master is implemented (for

N	Cycle time/[ms] Practice	Theoretical value	Cycle time/[ms] Theory	% Theory / Practice
1	5.3	X	5.1	95.6
2	10.9	$2S1 + 2C + 2S2 + X$	10.2	93.3
3	16.3	$S1 + 10C + S2 + X$	15.8	97.4
4	26.6	$S1 + 18C + 2S2 + X$	24.0	90.5

Table 8.9: Performance of the standard Ethernet message passing system

convenience) as a number of parallel processes in this model, protection against parallel sending is also included. Portions of the model are shown below.

```
process getrec

    receive SOURCE want
    receive [SOURCE] give

process getsen

    receive SOURCE want
    receive [SOURCE] give

replicate N

    process masrec#
        receive slasen# MESSAGE
        send getrec want
        think S2
        send getrec give
        send master MESSAGE

    process massen#
        receive master MESSAGE
        send getsen want
        think S1
        send medium wantmedium
        send medium givemedium
        send slarec# [MESSAGE]
        send getsen give

    process slave#
        receive slarec# COMMAND
        if COMMAND == reqdata
            send slasen# heredata
        endif
```

N	Cycle time/[ms] Practice	Theoretical value	Cycle time/[ms] Theory	% Theory / Practice
1	16.1	$4S1 + 4C + 4S2 + X$	15.9	98.7
2	19.2	$6S1 + 6C + 4S2 + X$	19.1	99.6
3	23.0	$6S1 + 8C + 6S2 + X$	22.8	99.1
4	27.1	$10S1 + 16C + 6S2$	27.1	100.2

Table 8.10: Performance of the standard Ethernet master-slave system

```
        if COMMAND == findyourbest
            send slasen# mybest
        endif
        if COMMAND == rununtil
            think X
        endif
    endreplicate

    process master
        replicate N
            send massen# reqdata
        endreplicate
        replicate N
            receive masrec# heredata
        endreplicate
        replicate N
            send massen# findyourbest
        endreplicate
        replicate N
            receive masrec# mybest
        endreplicate
        replicate N
            send massen# rununtil
        endreplicate
```

The results for this model are shown in Table 8.10. The value of X in this case is 5.7 ms, the other values are as given in Table 8.8.

8.6 Comparison of models and architectures

The symbolic values for the performance of the different models on the two architectures are summarized in Table 8.11. The expressions shown are those that apply to the operating conditions

Architecture	Decomposition strategy	Predicted Cycle Time
Transputer	Client-Server	$(5N+1)C+X$
Transputer	Message Passing	$2(N-1)C+X$
Transputer	Master-Slave	$(4N+1)C+X$
Ethernet	Client-Server	$6S1+6C+6S2+3Y+X$ $8S1+8C+6S2+3Y+X$ $9S1+10C+7S2+4Y+X$ $10S1+15C+7S2+2Y+X$ $12S1+18C+8S2+2Y+X$
Ethernet	Message Passing	X $2S1+2C+2S2+X$ $S1+10C+S2+X$ $S1+18C+2S2+X$
Ethernet	Master-Slave	$4S1+4C+4S2+X$ $6S1+6C+4S2+X$ $6S1+8C+6S2+X$ $10S1+16C+6S2$

Table 8.11: Summary of the predictions for all architectures and paradigms

under which the implementations run.

Examining the different strategies on a particular architecture shows that the message passing approach is best suited to the Transputer architecture, with its well connected network. The dependence on a central service by the other two approaches limits the extent to which they can take advantage of the communication facilities. The opposite situation occurs on the shared medium where the client-server and message passing approaches scale better with number of processors than the message passing approach.

In all but one case, there is a constant dependence on processing time. The exception is the last sample of the master-slave paradigm implemented under Ethernet. At this point communication bottlenecks result in the critical path being determined by communication time alone.

Comparison of the results for a particular strategy on both architectures yields additional insight into its communication requirements. The client-server and master-slave approaches have lower per-process communication costs on the shared medium. These approaches are better suited to cooperating in a shared environment, and interleaving their processing and communication without performance cost to the other processes. Message passing, on the other hand, makes efficient use of the Transputer network. It passes more messages than the other two approaches, which affects performance noticeably when it has to use a shared medium.

8.7 Conclusion

This chapter describes the analysis of all three parallel collision detection approaches. This includes a comparison between the results of the modelling and results measured from implementation of the algorithms on two parallel architectures.

The performance of the implementation on Transputer hardware compares extremely well with that predicted from the modelling tool. The accuracy is slightly less when working on a distributed system of PC's connected with Ethernet. In both cases the results are within the bounds of experimental error resulting from inaccuracies in measurements.

The symbolic form of the results of the Analytical Simulation allows comparison across architectures and between different decomposition strategies. It also permits identification of the variables on the critical path.

In its current form Analytical Simulation is capable of accurate analysis of the constructs found in parallel and distributed virtual reality systems. In addition, the results allow comparison between:

- The performance of different approaches on the same architecture.
- The effects of a variety of architectures on one approach.

8.8 Future work

The chapters leading up to this point have dealt extensively with the development of the Analytical Simulation approach to performance prediction. A number of enhancements are described, of which the state space extensions are the most significant.

There are still a number of areas in which enhancements can be made to improve the speed of the analysis and to increase the immediate value of the results.

At present the system tends to produce more constraints than are strictly necessary given the results required. A number of constraint regions can correspond to the same values for the metrics. Often these result from the order in which the constraints are created, and the combined region can be defined by a smaller set of inequalities. It should be possible to remove or collapse constraints, certainly in a postprocessing stage, but possibly while performing the analysis. This latter option would also prevent duplication of effort during the analysis.

The number of states used in the state space analysis is at present an upper bound on the number strictly required. It may be possible to remove some components of the state space vector without affecting the final result. This would result in improvements in both the time and the space complexity of the analysis tool.

Chapter 9

Performance analysis of virtual reality systems

Parallel decomposition strategies used in parallel and distributed virtual reality systems were examined in Chapter 2. Performance analysis techniques were surveyed in Chapter 3 and an approach that is capable of generating the metrics required for the characterization of virtual reality systems was developed in Chapter 4. The Analytical Simulation approach was refined, and tested in Chapters 5 to 8. It is suitable for use with the constructs found in the various decomposition strategies.

The remainder of this document is concerned with the application of Analytical Simulation to virtual reality systems. The purpose of the following chapters is twofold:

- To provide a reference which describes the performance characteristics of the parallel decomposition strategies used in virtual reality systems.
- To explore the use of Analytical Simulation, and to identify the ways in which it can enhance the understanding of the model under analysis.

This chapter describes the strategy employed for the analysis, and provides a summary of the ways in which Analytical Simulation can be employed.

9.1 Components of virtual reality systems

Chapter 2 identified parallel decomposition strategies used in virtual reality systems. The software structure of any virtual reality system can be decomposed into three subsystems which perform the functions of receiving input, modelling the world and producing the output respectively [Pra93] [Ban93] [Pin96].

As seen previously in Chapter 6, understanding the complexities of the performance characteristics of a system is made easier by simplifying the object under analysis. Each subsystem is examined in isolation, after which the components are combined into complete systems.

In multi-user virtual reality systems there are likely to be a number of input and output devices. These can usually operate independently, since they operate on data specific to a particular user and device. Thus no communication occurs between the various input/output devices. Sharing of data and coordination of effort between nodes of the virtual reality system is the domain of the world modelling subsystem. The effects of the various parallel decomposition strategies described in Chapter 2 are examined with respect to the world modelling components.

The use of parallelism within the input and output subsystem is not unknown, but in light of the discussion above it is limited to parallelism within a single instance of the corresponding subsystem. Thus the following strategy is applied when examining the subsystems of virtual reality systems: Analysis of input and output concentrates on the parallelism within the subsystem; analysis of the world modelling component examines the effects of the parallel decomposition strategies described in Chapter 2.

9.2 Techniques used in Analytical Simulation

As may be expected for any complex system, a complete analysis of the performance characteristics can produce a substantial amount of information. This is time-consuming to generate and may produce a lot more information than is required to judge the behaviour of the model. There are ways to decrease both the time and effort required to extract only the salient features. Many of these are used in the following sections, and are pointed out as they occur. A brief summary of these principles is given below:

- Simulate completely for small numbers of processes, and sample at higher numbers to confirm trends.
- Identify optimal deterministic paths, and limit further analysis to the new, deterministic model (see section 12.3.7).
- Random variables can be simulated using multi-state processes (see section 11.2), or by using variables to represent the probabilities of different paths (see section 11.4).
- Limit the use of variables to the most significant delays, or where detail is required in the analysis (see section 11.5).
- Simulate each decomposition strategy independently first, before considering their combined behaviour (see section 12.3.2).
- Model only the visible characteristics of complex systems (as in section 12.3.4).
- Simplify non-critical portions of the model (see section 12.3.5).
- Tabulate (section 12.3.5) and graph (section 12.3.8) performance results to identify patterns and to evaluate behavioural trends.
- Variables can be removed from the model, simplifying analysis without reducing the complexity of the model, by expressing one variable in terms of others (see section 13.4).

9.3 Conclusion

The input, output and world modelling components of virtual reality systems are to be examined separately at first, after which analysis of systems constructed from these components is to be performed. The input and output subsystem are to be examined with respect to parallelism within a single unit. The effect of multiple nodes interacting is to be examined with respect to the world modelling component.

The Analytical Simulation approach can be deployed in a number of different ways. Some strategies which speed up analysis by extracting only the required results are described.

Chapter 10

Analysis of the input subsystem

Input in current virtual reality systems is usually limited to devices such as some form of glove for measuring hand configuration, and trackers for locating the gloves and the head mounted displays. These and other more esoteric input devices can usually be viewed as sources of discrete events. Each event provides an update on the status of the device at the next instant in time. The actual data input is not relevant to the performance analysis. The factors of importance are the sampling frequency and the latency of the input data.

Many input devices used in virtual reality systems have a measure of sophistication in that they are capable of converting physical measurements into digital form on their own. Some also have the facility to perform some initial processing on the data, providing readings at regular intervals and in the most appropriate format. This justifies modelling an input device as a separate processor, and examining the performance effects resulting from this component of the virtual reality system.

Performance enhancements for the input components of virtual reality systems concentrate on look-ahead algorithms, where the values associated with future events can be predicted. In this way the latency, being the time between the actual event and the corresponding response, can be cut down.

This chapter explores the performance implications of the input subsystem, examining different ways to read the data from the device, and considering the analysis of look-ahead.

10.1 Polled and interrupt-driven input

Many input devices can be read in two ways [Pol93] [Pra93]. The simplest is by using polling in which the device is read when a data value is required. Alternatively, the device can be set up to generate values continuously, so that the most recent value is always available. The computer is usually interrupted to receive each data value, hence this technique is referred to as interrupt-driven.

A model of the polled approach to data input is shown below:

Variable	Interpretation
C	Communication time between input device and system processor
K	Communication time between system processors
X	Time between data produced by the input device

```
process device

    receive input req
    report create
    think X
    send input rep C

process input

    receive useinput request
    send device req C
    receive device rep
    send useinput data K

process useinput

    report startask
    send input request K
    receive input data
    report endask
```

The model contains three processes. The first reproduces the actions occurring on the input device. Upon receiving a request for a value from the transducer, this process proceeds to interrogate the device, taking period X to do so. This is then passed to the input process of the virtual reality system itself. Some other component of the system needs to use the value obtained, and this is modelled by the third process. Communication time from the input processor to the attached device is modelled by the variable C, communication time between processors in the virtual reality system is represented by variable K.

Two intervals are of interest in this model, the total time taken to fetch a value (the interval from *startask* to *endask*), and the "freshness" of the value (from *create* to *endask*). The latter indicates how well the value corresponds with the current situation in the real world.

For this model, the time taken to fetch the input value is $2K + 2C + X$, while the freshness of the value is given by $K + C + X$.

The alternative approach which continuously reads the device corresponds to the model below:

Variable	Interpretation
C	Communication time between input device and system processor
K	Communication time between system processors
X	Time between data produced by the input device

```
process device

    report create
    think X
    send input message C

process input

    receive ANY message
    if ANY == useinput
        send useinput data K
    endif
    assign ANY undef

process useinput

    report startask
    send input message K
    receive input data
    report endask
```

The input device produces new readings at intervals of X. These are passed onto the input process which supplies the latest value upon request. The total time to obtain a value with this model, and the "freshness" are as given below:

$2nK \geq X \geq 2(n-1)K$: $(n = 0 \ldots \infty)$

$$\begin{aligned} \text{Fetch time} &= 2K + \tfrac{1}{n}C \\ \text{Freshness} &= 2(n+1)K + C \end{aligned}$$

The overall time taken to obtain a value has decreased over that for the polled approach, while the freshness of the value has increased. This latter effect can be attributed to the need for the input process to attend to the communication from the rest of the system, delaying its response to the input device. The "freshness" value is an average one, particularly where X is large compared to K. In this case re-reading the input is still returning the same value which is ageing steadily. Another interesting point is the sudden discontinuity at points where $X = 2nK$, caused by using the old value for an additional $2K$ cycles. If the input device is capable of supplying values sufficiently rapidly ($X < K$), then these effects are not present.

The interrupt-driven approach provides faster turn-around times for including an input value into the virtual reality system. For a device capable of producing values rapidly ($X < K$), the "freshness" of the value is larger for the interrupt-driven approach due to the inability of the input driver to service the interrupt while communicating with the rest of the system. Should the architecture not be suited to performing communication and servicing external devices simultaneously then the polled approach yields values with lower latency.

10.2 Look-ahead during input

Look-ahead using a predictive filter is studied in a number of cases related to virtual reality systems. A number of different predictors, in particular Kalman filters and predictors based on Grey system theory, are examined in [Wu95]. Predictors based on polynomial extrapolation and Kalman filters are analysed in the frequency domain in [Azu95].

Analytical simulation is not well suited to examining the effects of prediction. Predictive filters give an advance estimate of the values expected from the input devices. While this has the effect of reducing latency, the value predicted may be inaccurate. The effects of these errors in the prediction influences the quality of the virtual reality experience. This quality of experience can only be judged subjectively, and so a quantitative analysis is not given.

10.3 Conclusion

Relatively little analysis is performed relating to the input subsystem. The performance implications of polled versus interrupt driven input are described. Both approaches have their merits, the choice depends on the target architecture. Look-ahead affects the quality of the system and is not amenable to quantitative analysis. Further effects, involving the combination of the input subsystem with the other components of virtual reality systems, are examined in Chapter 13.

Chapter 11

Analysis of the output subsystem

The graphical output of most virtual reality systems is produced using graphics workstations which contain hardware implementations of the rendering functions. However these functions are still combined using a variation of the standard graphics pipeline [Seg94]. Thus this chapter concentrates on modelling the graphics pipeline. Effects of parallelism on portions of the pipeline are examined. Finally models of some pipelines occurring in virtual reality systems are investigated.

Output to other devices, such as force feedback joysticks, is also found in some virtual reality systems. The use of parallelism in these components is limited, and so these devices are not covered in this chapter.

11.1 A model of the graphical pipeline

A model of a simple graphical pipeline is shown below. The pipeline used is composed of three stages, representing transformation, hidden surface removal and rendering. Each stage is assumed to take a certain length of time to complete, represented by T, H and R respectively, before passing its results to the next stage. Processes to model both the supply of data to the pipeline and receipt of data after rendering are included. These are included to enable the effects of the processes outside the pipeline to be modelled, and to facilitate some of the parallel modifications described later. It is assumed that all processes run on separate processors and that all communication is blocking.

Variable	Interpretation
C	Communication time between the stages of the pipeline
D	Time for the slowest stage ($max(H, R, T)$)
H	Time spent on hidden-surface removal for each item
R	Time spent on rendering each item
T	Time spent on transforming each item

```
process source

    report marker1
    send transform input C

process transform

    receive source input
    think T
    send hiddsurf m1 C

process hiddsurf

    receive transform m1
    think H
    send render m2 C

process render

    receive hiddsurf m2
    think R
    send sink output C

process sink

    receive render output
    report marker2
```

Latency is measured from *marker1* to *marker2*. The cycle time is determined from the period between recurrences of *marker2*, which is related to the frequency at which data is output.

Analysis of this system yields the following results:

$T \geq H, T \geq R$:

Cycle time =	$2C + T$
Latency =	$5C + 2T + H + R$

$T \geq H, R \geq T$:

Cycle time =	$2C + R$
Latency =	$5C + 4R$[1]

[1]The system with these constraints is subject to an initial transient value whose duration depends on the relative values of T, H and R. This eventually reaches the stable state shown. See the previous example on pipelines in section 4.1 for more detail.

$H \geq T$, $H \geq R$:

Cycle time =	$2C + H$
Latency =	$5C + 3H + R$

$H \geq T$, $R \geq H$:

Cycle time =	$2C + R$
Latency =	$5C + 4R$

The results for this model differ only slightly from the previous pipeline model given in the initial example of Analytical Simulation (see section 4.1). These differences can be attributed to the source and sink stages, and to the communication delay.

The results agree with intuitive expectations. The cycle time is dominated by the slowest stage. The value for the cycle time is the time for this stage plus the time to get the data in and to send it out. Latency may be slightly less obvious: before the slowest stage, messages block at each stage for a time equal to the time of the slowest stage. After this stage there is no further bottleneck and data passes through as fast as possible, limited only by the processing time of each stage.

It is interesting to note that the source process contributes to the latency, even though it produces messages as fast as possible, and has no explicit delay. The next message is ready as soon as the transmission of the previous one is complete. The source process then blocks for a period equal to the delay of the slowest process while trying to send. By adding an extra delay in the source routine, latency can actually be reduced without affecting cycle time.

Since latency in this model is measured from the time that data is generated (usually immediately before it is sent) until it arrives at its destination, delaying the time of generation decreases latency. Changing the model to implement this is done by sending a message back to the source from the next stage, once that stage has finished processing and is ready to receive the next message. The source can then produce the next message and send it, knowing that the destination is ready and that it will not block on the send operation. Latency in the model is reduced by $C + D$, where D is the time taken on the slowest stage.

Comparing this to other work on lag in virtual reality systems [Wlo95], the effect being noted is synchronization lag, resulting from processes blocking while waiting to pass data on to others. The solution proposed by Wloka involves delaying processes so that one finishes just as the next starts. This requires the ability to determine the exact time taken, so that computation can be started at the correct point. Given that communication delays are not being modelled by Wloka, it is simpler to adopt the feedback approach suggested in this section.

This result has significant implications for virtual reality systems. If the system is sending redraw requests to the graphics pipeline at a rate faster than the rate at which the pipeline can render, then better performance can be achieved by either dropping certain frames, or by performing extra work before the state of the world for the next frame is sent to the renderer.

Adding the feedback to the end of the pipeline sets the latency to a constant value of $4C + T + H + R$ for all values of the variables, but has the side effect of increasing the cycle time to $5C + T + H + R$.

11.2 The use of buffering in the pipeline

When processes take variable amounts of time to process data, the steady flow of data is interrupted and processes must block while waiting to send and receive data. This wastes processor time and lowers the efficiency of the parallel system.

One approach to maintaining a continuous flow of data through a pipeline is to add buffers between the output and input of two adjacent stages. This allows the rest of the pipeline to continue to perform useful work, while the blocking affects only the buffer. It increases the chance that there is data waiting whenever a process finishes its current cycle and needs to fetch more.

There some difficulty in modelling this with Analytical Simulation, since the analysis tool does not provide for random processing times in the model. Stochastic modelling tools such as Petri Nets might allow this but such tools do not easily provide analytical values for latency, the important variable with this modification.

The system is modelled using a tristate process to introduce an extra occasional delay in one of the stages of the pipeline. This process cycles through three delay values, two small and one large. The effect of placing the slow stage and the buffer in different sections of the pipeline is explored this way. The use of a multi-state process is not too dissimilar from reality. Virtual reality scenes often contain a mixture of complex and simple objects whose relative processing times in different portions of the pipeline differ.

Rendering is typically slower than the other stages and more sensitive to the complexity of the object. The effect of placing a buffer before this last stage of the pipeline is examined below.

The model that is used in this case is a simple variation of the pipeline seen in section 11.1. For simplicity, each process cycles with a period D. The final stage delays for $3D$ once every three cycles. The modified final stage is shown below:

Variable	Interpretation
C	Communication time between the stages of the pipeline
D	Delay in each stage of the pipeline (except for the tristate stage)

```
process render

    receive hiddsurf m2
    think D
    send sink output C
    receive hiddsurf m2
    think D
    think D
    think D
    send sink output C
    receive hiddsurf m2
    think D
    send sink output C
```

Analysis of this system gives a cycle time of $2C + \frac{5}{3}D$ and a latency of $5C + \frac{20}{3}D$.

A buffer process can now be added between the hidden surface removal and rendering stages. This process is shown below. A communication cost B is associated with extracting information from the buffer.

Variable	Interpretation
B	Time required for communication with the buffer
C	Communication time between the stages of the pipeline
D	Delay in each stage of the pipeline (except for the tristate stage)

```
process hiddsurf

    receive transform m1
    think D
    send buffer1 m2 C

process buffer1

    receive hiddsurf m2
    send render m2 B

process render

    receive buffer1 m2
    ...
```

The performance of this model is given below:

$B \geq C$:

$$\text{Cycle Time} = C + \tfrac{5}{3}D + B \qquad (>)$$
$$\text{Latency} = 2C + \tfrac{25}{3}D + 4B \qquad (>)$$

$C \geq B$:

$C \geq 2D + B$:

$$\text{Cycle Time} = 2C + D \qquad (<)$$
$$\text{Latency} = 5C + \tfrac{14}{3}D + B \qquad (><)$$

$C \geq B$:

$2D + B \geq C$:

$2C \geq 2D + 2B$:

$$\text{Cycle Time} = 2C + D \qquad (<)$$
$$\text{Latency} = \tfrac{14}{3}C + \tfrac{16}{3}D + \tfrac{4}{3}B \qquad (><)$$

$C \geq B$:

$2D + 2B \geq 2C$:

$3C \geq 2D + 3B$:

$$\text{Cycle Time} = 2C + D \qquad (<)$$
$$\text{Latency} = 4C + 6D + 2B \qquad (><)$$

$C \geq B$:

$2D + 3B \geq 3C$:

$$\text{Cycle Time} = C + \tfrac{5}{3}D + B \qquad (<)$$
$$\text{Latency} = C + \tfrac{25}{3}D + 5B \qquad (><)$$

The symbols in parentheses after the values indicate whether the value is less (<) or greater (>) than the value for the system without buffering. It can be seen immediately that the case $B \geq C$ gives worse performance in the buffered situation. Thus buffering is not effective unless communication with the buffer is faster than communication with other processors. This has implications for the practical implementation. Since the buffer does not have a substantial processing overhead, it can be added to a processor containing a process that communicates with the buffer. Communication can then occur using a faster mechanism, such as shared memory.

The cases where $B < C$ show improvement in the cycle time. The effect on the latency can be less beneficial, as may be expected since the length of the pipeline is being increased. Depending on the relative values of the variables, latency may either increase or decrease. Examples of each possibility are shown below:

For the case:

$C \geq B$:

$2D + 3B \geq 3C$:

Unbuffered latency is given by $5C + \frac{20}{3}D$, and buffered latency is given by $C + \frac{25}{3}D + 5B$. For the variable values:

B	D	Unbuffered latency	Buffered latency
0	$\frac{3}{2}C$	$15C$	$\frac{27}{2}C$
C	0	$5C$	$6C$

For this case, a decrease in latency when buffering is added occurs when $4C > \frac{5}{3}D + 5B$.

Adding a second buffer routine also causes increases in cycle time and latency for the case when $B > C$. For the other case, $C > B$, cycle time is reduced over the unbuffered case. No improvement in the cycle time occurs relative to the version with a single buffer. For some of the latency values corresponding to a set of constraints on the variables it is no longer possible to achieve a latency value smaller than the unbuffered version, by varying the values of the variables within the constraints.

The buffering effects are intended to smooth out delays around the slow stage. The next results describe the effects of placing the slow stage earlier in the pipeline, and placing the buffer before, and/or after it. The hidden surface removal stage is replaced with the tristate process. The transformation and rendering stages are the simple versions with only a single processing delay of duration D. The results are as follows, where it is now assumed that $C > B$:

Constraint region	Cycle Time	Latency
No buffering		
—	$2C + \frac{5}{3}D$	$5C + 6D$
Buffer before slow stage		
$2D + B \geq C$	$\frac{5}{3}C + \frac{5}{3}D + \frac{1}{3}B$	$\frac{17}{3}C + \frac{23}{3}D + \frac{1}{3}B$
$C \geq 2D + B$	$2C + D$	$5C + 6D + B$
Buffer after slow stage		
—	$2C + \frac{5}{3}D$	$5C + 6D + B$
Buffer both before and after slow stage		
$C \geq 2D + B$	$2C + D$	$5C + \frac{14}{3}D + 2B$
$2D + B \geq C$ $2C \geq 2D + 2B$	$2C + D$	$\frac{14}{3}C + \frac{16}{3}D + \frac{7}{3}B$
$2D + 2B \geq 2C$ $3C \geq 2D + 3B$	$2C + D$	$4C + 6D + 3B$
$2D + 3B \geq 3C$	$C + \frac{5}{3}D + B$	$2C + \frac{23}{3}D + 5B$

Adding buffers before the hidden surface removal stage results in improvement in cycle time. Latency is increased. The greatest improvement in cycle time with least cost in latency occurs for relatively large values of C, where communication is slowing the system and the buffer smoothes the flow of information.

It is thus possible to improve the performance of the graphics pipeline with an uneven load by adding buffering. Cycle time in particular can be decreased. In some cases latency can be improved as well, in others there is a tradeoff between improvements in the cycle time against increases in the latency. Care has to be taken with the implementation to ensure that the operating conditions do not cause a reduction in performance instead.

11.3 Parallelism and the pipeline

There are additional ways to exploit parallelism in a pipeline, other than by just adding additional stages [Ban94b]. A section of the pipeline can be rewritten to run in parallel, with the intention to lower the time taken by that particular section. Another option is to replicate the pipelines and to have a number of pipelines running in parallel.

The first model divides the final stage of the pipeline into N smaller sections which run in parallel. The time taken for each of these sections should be about $\frac{1}{N}$ of the time taken for the original stage. In practice it is probably slightly greater than this, but should not exceed the time required for the original if any gain is to be expected. The relevant portions of this model are

shown below:

Variable	Interpretation
C	Communication time between the stages of the pipeline
H	Time spent on hidden-surface removal for each item
N	Number of rendering stages in use
P	Time spent on rendering each item in each of the parallel renderers
T	Time spent on transforming each item

```
process hiddsurf
    receive transform m1
    think H
    replicate N
        send render# m2 C
    endreplicate

replicate N
    process render#
        report rend#
        receive hiddsurf m2
        think P
        send sink output C
endreplicate
process sink
    replicate N
        receive RENDER# output
    endreplicate
report marker2
```

The latency and cycle time measured from this model are given below:

$(N-1)C+H \geq T$, $P \geq (N-1)C+H$:

$$\text{Cycle Time} = \quad 2C+P$$

$$\text{Latency} = \quad 5C+4P$$

$(N-1)C+H \geq T$, $(N-1)C+H \geq P$:

$$\text{Cycle Time} = \quad (N+1)C+H$$

$$\text{Latency} = \quad (3N+2)C+3H+P$$

$T \geq (N-1)C + H, T \geq P$:

Cycle Time = $2C + T$

Latency = $(N+4)C + 2T + H + P$

$T \geq (N-1)C + H, P \geq T$:

Cycle Time = $2C + P$

Latency = $5C + 4P$ (Transient present in this case)

The behaviour of this approach agrees with what could be extrapolated from the previous results for parallel pipelines. Cycle time is dominated by the slowest stage in all but one case. Latency follows the same lines as before, except for a large communication component. Ignoring this for the moment, the latency has the same relative dependency on the other three variables, with P taking the place of R. Since $P \cong \frac{R}{N}$, this gives an improvement in both the cycle time and the latency.

The substantial dependency on the communication is a result of the manner in which this aspect of the model is designed. The model assumes that each process is only able to send one message at a time. This would apply to situations where not all processors are directly interconnected, or where hardware limitations require that one message be completed before the next can be sent. The large communication component in the second case above is due to the substantial bottleneck produced by the combination of a slow second stage and the extra communication. The first stage is ready to send again almost immediately and has to wait almost a whole cycle before being able to pass data on to the next stage. This accounts for one third of the communication factor in the latency, the sequential communication into and out of the final stages is responsible for the rest. This is also a case where a delay in the source process would actually decrease latency.

A model of a system with sufficient connectivity to allow parallel communication is shown below:

Variable	Interpretation
C	Communication time between the stages of the pipeline
H	Time spent on hidden-surface removal for each item
L	Communication time between intra-processor processes
N	Number of rendering stages in use
P	Time spent on rendering each item in each of the parallel renderers
T	Time spent on transforming each item

```
process hiddsurf

    receive transform m1
    think H
    replicate N
        send renders# m2 L
    endreplicate
```

```
replicate N
    process renders#
        receive hiddsurf m2
        send render# m2 C

    process render#
        receive renders# m2
        think P
        send renderr# output C

    process renderr#
        receive render# output
        send sink output L
endreplicate

process sink
    replicate N
        receive RENDER# output
    endreplicate
    report marker2
```

The variable L is introduced to model the time required to send a message to or from one of the extra processes. These processes are likely to be on the same processor as the process using them for concurrent communication. Thus, this value can be expected to be small, certainly less than C. Analysis of this model gives the following:

$NL + H \geq C + T$, $C + P \geq NL + H$:

Cycle Time =	$2C + P$
Latency =	$7C + 5P$

$NL + H \geq C + T$, $NL + H \geq C + P$:

Cycle Time =	$C + NL + H$
Latency =	$3C + 3H + (3N + 1)L + P$

$C + T \geq NL + H$, $T \geq P$:

Cycle Time =	$2C + T$
Latency =	$5C + 2T + H + (N + 1)L + P$

$C + T \geq NL + H$, $P \geq T$:

Cycle Time =	$2C + P$
Latency =	$7C + 5P$

This approach does not automatically yield the improvement that could be expected from using a better connected network. The cycle times are either unchanged, or improve. Latency on the other hand, improves in two cases (assuming $L < C$), but increases in the other two cases, where the parallel rendering time P dominates. In this latter case, an increase of $2C + P$ is found. The additional factor P can be expected, since an extra stage is added before the bottleneck process. The extra $2C$ results from earlier stages in the pipeline having to block while holding data, thereby increasing latency. Once again, feedback can reduce this latency without cost to the cycle time. A feedback message from the first stage to the source is capable of reducing latency to the values achieved with the previous model. Using feedback from the second stage results in a improvement in latency for large values of P. The latency in this case is $4C + 3P$.

Parallelizing additional stages of the pipeline yields similar results. The variable representing the time for the stage is replaced by the variable representing the time for one of the parallel components. Cycle times improve when this change occurs to the slowest stage. Latency can be improved when the affected stage occurs at or after the slowest stage.

Two points are again reinforced by these results. The length of the pipeline before the slowest process should be kept as short as possible, since the latency in these stages is equal to the processing time of the slowest process. This may make it feasible to combine some of the stages, provided the total time for the combined stage does not exceed that of the slowest stage. The results also indicate that having each stage accept data as soon as possible is not always best when considering performance issues. Having processes sitting idle can yield better performance than having a process waiting to send data on to the next stage. Both these issues affect latency independently of cycle time.

The manner in which the relative performance of different models differs according to the constraints on the variables suggests that modelling could also be useful after implementation of a system. Some enhancements, for example the feedback used above, are more relevant to certain variable values, and could be applied once these values are measured for a particular system.

The preceding discussion has examined the use of further parallelism within the pipeline. The effect of combining a number of pipelines in parallel is examined with the next model.

The following specification defines the model of parallel pipelines. The source process feeds data into each of the N pipelines in turn. The results are removed at the other end of the pipeline in the same order.

Variable	Interpretation
C	Communication time between the stages of the pipeline
H	Time spent on hidden-surface removal for each item
N	Number of pipelines in use
R	Time spent on rendering each item
T	Time spent on transforming each item

```
process source
    replicate N
        report markers
        send transform# input C
    endreplicate

replicate N
    process transform#
        receive source input
        think T
        send hiddsurf# m1 C

    process hiddsurf#
        receive transform# m1
        think H
        send render# m2 C

    process render#
        receive hiddsurf# m2
        think R
        send sink output C
endreplicate

process sink
    replicate N
        receive render# output
        report markerf
    endreplicate
```

The results of using Analytical Simulation on this model differ, as may be expected, according to which stage of the pipeline takes the longest. The results are shown below:

Dominant Variable	Cycle Time	Latency
T	$\frac{2C+T}{N}$	$\frac{(3N+2)C+(N+2)T+NH+NR}{N}$
H	$\frac{2C+H}{N}$	$\frac{(3N+2)C+(2N+1)H+NR}{N}$
R	$\frac{2C+R}{N}$	$\frac{(3N+2)C+(3N+1)R}{N}$
$(N-2)C$, for $N>2$	C	$4C+T+H+R$

The most obvious benefit from this approach is the perfectly linear decrease in cycle time with the number of processors, provided communication is fast enough. At some point, however $(N-2)C$ outstrips each of T, H and R and performance reaches a constant level, independent of number of processors. The equally obvious limitation is the near constant value of the latency which decreases by marginal amounts as N is increased (and while $(N-2)C$ does not dominate).

11.4 Pipelines in practice : Zero latency rendering

The problem of latency in rendering stages of virtual reality systems is addressed by some clever modifications to the hardware of the display controller [Reg92] [Reg93]. When the scene is rendered, it is drawn in a manner independent of orientation. In effect the scene is drawn on a sphere surrounding the viewer rather than just on a rectangular viewport. Extracting the portion containing the area that the viewer is looking at and displaying this on the screen is the responsibility of the display controller. This requires extensive modification to the address recalculation hardware to allow this transformation to occur quickly enough for screen display directly from the spherically rendered image.

Thus changes in orientation can be made to take effect immediately (or on the next refresh cycle if a stable image is required). Changes in orientation (rotation) can be introduced directly at the end of the graphics pipeline. As mentioned in [Reg93]:

> "...it becomes possible to bind rotational latency to the refresh cycle period which tends to be short, fixed in length and independent of scene complexity."

This modification is of use only for changes in orientation. Translation of the scene still requires re-rendering. Since a greater area needs to be rendered, it can be expected that latency on translation is increased. An effort is made to reduce this by providing image overlaying functionality with the modified address recalculation hardware. Objects at different depths may be rendered on different display memories. These are automatically combined during the refresh cycle. The images corresponding to objects further away could be re-rendered less frequently.

The model of a pipeline with zero latency abilities is shown below. Since the zero latency enhancement is limited to rotation, the normal pipeline is maintained for cases when translation is required as well. As done previously, random selection of rotation/translation has to be emulated by preselecting some periodic mixture of the two. This model produces a sequence of N simulated rotations followed by M translations.

Variable	Interpretation
C	Communication time between the stages of the pipeline
H	Time spent on hidden-surface removal for each item
M	Length of each sequence of translations
N	Length of each sequence of rotations
R	Time spent on rendering each item
T	Time spent on transforming each item

```
process source
    replicate N
        report marker1
        send transform nolat C
    endreplicate
```

```
    replicate M
        report marker1
        send transform input C
    endreplicate

process transform
    receive source X
    if X == nolat
        send hiddsurf nolat
    endif
    if X != nolat
        think T
        send hiddsurf m1 C
    endif

process hiddsurf
    receive transform Y
    if Y == nolat
        send render nolat
    endif
    if Y != nolat
        think H
        send render m2 C
    endif

process render
    receive hiddsurf Z
    if Z == nolat
        send sink output
    endif
    if Z != nolat
        think R
        send sink output C
    endif

process sink
    receive render output
    report marker2
```

The simulated rotations in the above model are sent through the pipeline in the same way as the translations, but with fewer delays. The zero latency stage is modelled this way, rather than having source send directly to sink so as to preserve the order of events. It would not be realistic to have a rotation that is performed after a translation show up on the display before the effect of that translation. The overall delay is still the same, a rotation message is still delayed for a period C before reaching the final stage.

Analysis of this model produces the following predictions for cycle time and latency:

$T \geq R, T \geq H$:

$$\text{Cycle Time} = \begin{cases} \frac{(N+2M)C+MT+H}{N+M} & \text{if } N = 1 \\ \frac{(N+2M)C+MT+H+R}{N+M} & \text{if } N > 1 \end{cases}$$

$$\text{Latency} = \begin{cases} \frac{(N+5M+1)C+2MT+(M+2)H+(M+1)R}{N+M} & \text{if } N = 1 \\ \frac{(N+5M+1)C+2MT+(M+2)H+(M+3)R}{N+M} & \text{if } N > 1 \end{cases}$$

$H \geq R, H \geq T$:

$$\text{Cycle Time} = \begin{cases} \frac{(N+2M)C+MH+R}{N+M} & \text{if } R \geq T \text{ and } N = 1 \\ \frac{(N+2M)C+MH+T}{N+M} & \text{if } T \geq R \text{ and } N = 1 \\ \frac{(N+2M)C+MH+T+R}{N+M} & \text{otherwise} \end{cases}$$

$$\text{Latency} = \begin{cases} \frac{(N+5M+1)C+3MH+(M+3)R}{N+M} & \text{if } R \geq T \text{ and } N = 1 \\ \frac{(N+5M+1)C+3MH+(M+1)R+2T}{N+M} & \text{if } T \geq R \text{ and } N = 1 \\ \frac{(N+5M+1)C+3MH+(M+3)R+2T}{N+M} & \text{otherwise} \end{cases}$$

$R \geq H, R \geq T$:

$$\text{Cycle Time} = \begin{cases} \frac{(N+2M)C+H+MR}{N+M} & \text{if } N = 1 \\ \frac{(N+2M)C+H+T+MR}{N+M} & \text{if } N > 1 \end{cases}$$

$$\text{Latency} = \begin{cases} \frac{(N+5M+1)C+3H+4MR}{N+M} & \text{if } H \geq T \text{ and } N = 1 \\ \frac{(N+5M+1)C+3H+4MR+2T}{N+M} & \text{if } H \geq T \text{ and } N > 1 \\ \frac{(N+5M+1)C+(4+K)H+(4M+\frac{K^2+K}{2})R-\frac{(K+1)(K+2)}{2}T}{N+M} & \text{if } A(K) \geq T \geq B(K) \text{ and } N = 1 \\ \frac{(N+5M+1)C+(4+K)H+(4M+\frac{K^2+K}{2})R+(2-\frac{(K+1)(K+2)}{2})T}{N+M} & \text{if } A(K) \geq T \geq B(K) \text{ and } N > 1 \\ \frac{(N+5M+1)C+(M+2)H+\frac{M^2+5M+2}{2}R-\frac{M^2-M}{2}T}{N+M} & \text{if } T \geq \frac{H+(M-2)R}{M-1} \text{ and } N = 1 \\ \frac{(N+5M+1)C+(M+2)H+\frac{M^2+5M+2}{2}R-\frac{M^2-M-4}{2}T}{N+M} & \text{if } T \geq \frac{H+(M-2)R}{M-1} \text{ and } N > 1 \end{cases}$$

where: $A(K) = \frac{H+(K+1)R}{K+2}$, $B(K) = \frac{H+KR}{K+1}$, $K = 0 \ldots M-3$

To extract useful information from these results, the values for various extremes of the parameters N and M are considered.

The case where both N and M are small corresponds to a case where both translations and rotations occur, but neither occurs alone in significant quantity. Taking $N = 1$, $M = 1$ as an example, the corresponding performance values are:

$$\text{Cycle time} = \frac{3C+H+\begin{cases} T & \text{if } T \geq R \\ R & \text{if } R \geq T \end{cases}}{2}$$

$$\text{Latency} = \frac{7C+3H+2R+\begin{cases} 2T & \text{if } T \geq R \\ 2R & \text{if } R \geq T \end{cases}}{2}$$

This is a slight improvement on the results achieved with a standard pipeline, as should be expected. The improvement is limited by the need for the pipeline to complete processing on the translation calculations before the zero latency rotation message can get to the sink process. Having this message waiting builds up the latency for the message.

Increasing the number of successive rotations should eliminate most of this waiting and produce improvements in latency. The results for $N \gg M$ are approximately as follows:

$$\text{Cycle Time} \cong C + \frac{D(M)}{N}$$

$$\text{Latency} \cong C + \frac{D(M)}{N}$$

where $D(M)$ is independent of N. Improvements in both cycle time and latency are expected as N increases and effect of the zero latency modifications dominates. The lower limit of C in this case corresponds to the cost of the rotation message to get from source to sink while bypassing most of the pipeline. Thus if use of the virtual reality system consists mostly of looking around while stationary then the zero latency modifications result in substantial performance improvements.

Examining the alternative case, where $M \gg N$, the performance measurements approximate:

$$\text{Cycle time} \cong 2C + D$$

$$\text{Latency} \cong 5C + 4D$$

where D depends only on the pipeline delays; T, H and R; and is independent of both M and N. Thus where the zero latency enhancement is used infrequently, performance approximates that of a normal pipeline, as should be expected. Given that the hardware modifications to implement zero latency rendering are relatively expensive, they are not viable unless required for an application where extensive use will be made of them.

11.5 Pipelines in practice : PixelFlow

PixelFlow is the latest of the PixelPlanes series of graphics supercomputers developed at the University of North Carolina. The architecture of the PixelFlow system is described in [Mol92],

together with the use of image composition to achieve linear performance improvement as the number of renderers increases.

Image composition in a parallel rendering system involves dividing the primitives amongst each of the rendering processors. Each renderer then produces a rasterized version of its primitives in a local frame buffer. All of the frame buffers are then combined, or composed, at a central frame buffer which is then used to drive the display. Transferring the frame buffers to the composition stage involves moving large amounts of data, which in turn requires a high speed network. The required bandwidth is independent of the graphical primitives, and depends only on screen resolution and frame rate. This permits performance to scale linearly with the number of rendering processors.

The graphics pipeline in the PixelFlow machine is implemented as follows: The host computer passes each renderer a specific set of primitives to render. Each renderer then performs some standard geometric transformations on its primitives and sorts them into 128x128 pixel regions on the screen. Each renderer contains a rasterizer which contains an array of 128x128 pixel processors which computes the pixel values for the region. When all renderers have completed rasterization, the region is passed to the compositor. The composed image is then passed to one of the shader processors where the pixel colours are computed. This process is represented graphically in Figure 11.1.

Deferred shading is used which involves storing surface normal information with each pixel until it reaches the shading stage. In this way each pixel has to be shaded exactly once. Since the amount of information associated with each pixel is greater, a greater bandwidth for the image composition network is required. On the other hand, shading becomes independent of image complexity, and depends only on the shading model.

The PixelFlow system needs to transfer about 30Gbits/second when using deferred shading and calculating 5 samples for anti-aliasing purposes to achieve a frame rate of 30 frames/second. This is achieved with a network bus of 256 wires, each operating at 132MHz.

The estimated performance quoted for this system is:

$$T_{frame} = \sum_{i=1}^{N_{regions}} max(T_{rend_i}, T_{comp}, \frac{T_{shade}}{N_{shaders}}) \tag{11.1}$$

where T_{frame} is the time for one frame, T_{rend_i} is the time for rendering the *ith* region, T_{comp} is the compositing time for a region and T_{shade} is the time to shade a region.

The remainder of this section describes the use of Analytical Simulation to obtain performance values for this model. This provides both a validation of the Analytical Simulation approach, when comparing the results to those shown in equation 11.1, and demonstrates the abilities of the approach to extend the analysis to include latency measurements.

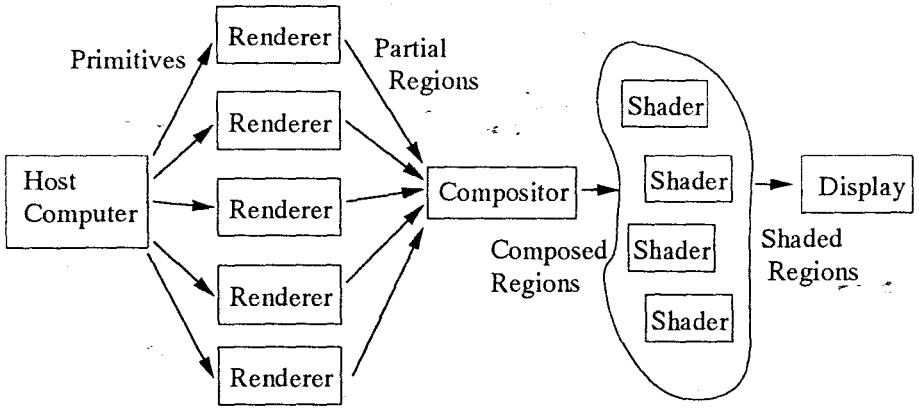

Figure 11.1: PixelFlow layout

The version intended for use with the Analytical Simulation tool appears as follows:

Variable	Interpretation
C	Communication time between renderers and compositor
N_{region}	Number of 128x128 regions per image
N_{render}	Number of rendering processes
$N_{shaders}$	Number of shading processes
R, T_{rend_i}	Time spent on rendering for one region
T, T_{comp}	Time to compose the images
S, T_{shade}	Time to shade one region

```
process source

    report marker1
    replicate NREGION
        replicate NRENDER
            send render## input
        endreplicate
    endreplicate

replicate NRENDER

    process render#
        receive source input
        think R
        send composit m1 C

endreplicate
```

```
process composit
    replicate NSHADE
        replicate NRENDER
            receive render## m1
        endreplicate
        think T
        send shading# m2
    endreplicate

replicate NSHADE
    process shading#
        receive composit m2
        think S
        send sink output
endreplicate

process sink
    replicate NSHADE
        replicate NREGION
            receive
            shading[((((#-1)*NREGION)+(##-1))%NSHADE)+1]
            output
        endreplicate
        report marker2
    endreplicate
```

This pipeline is similar to models used previously when examining parallelization of portions of the graphics pipeline. The variables R, T and S represent rendering time for one region (T_{rend_i}), time to compose the images (T_{comp}), and shading time for one region (T_{shade}). The principle difference is the allocation of the shaders. This is modelled so as to assign the first available shader to an incoming region in need of shading. Since shading time is constant, this is the shader that has been shading the longest. Thus allocating shaders on a round robin basis simulates this requirement adequately.

The variable N_{render} represents the number of rendering processes, N_{region} represents the number of 128x128 regions per image, and $N_{shaders}$ is the number of shading processes.

The results from the analysis of this model are as follows:

$(N_{render}-1)C+T_{comp} \geq T_{rend_i}$, $T_{shade} \geq N_{shaders}(N_{render}C+T_{comp})$

Cycle Time = $\frac{N_{region}}{N_{shaders}}T_{shade}$

Latency =

$$\begin{cases} \frac{N_{shaders}+2+N_{region}}{N_{shaders}}T_{shade}-N_{render}C & \text{if } HCF(N_{region},N_{shaders}) \leq 2 \\ \frac{N_{shaders}+N_{region}}{N_{shaders}}T_{shade}+N_{render}C+2T_{comp} & \text{otherwise} \end{cases}$$ [2]

$(N_{render}-1)C+T_{comp} \geq T_{rend_i}$, $T_{shade} \leq N_{shaders}(N_{render}C+T_{comp})$

Cycle Time = $N_{region}(N_{render}C+T_{comp})$

Latency = $N_{render}(1+N_{region})C+(2+N_{region})T_{comp}+T_{shade}$

$(N_{render}-1)C+T_{comp} \leq T_{rend_i}$, $T_{shade} \geq N_{shaders}(C+T_{rend_i})$

Cycle Time = $\frac{N_{region}}{N_{shaders}}T_{shade}$

Latency =

$$\begin{cases} \frac{N_{shaders}+2+N_{region}}{N_{shaders}}T_{shade}-N_{render}C & \text{if } HCF(N_{region},N_{shaders}) \leq 2 \\ \frac{N_{shaders}+N_{region}}{N_{shaders}}T_{shade}+2T_{rend_i}-(N_{render}-2)C & \text{otherwise} \end{cases}$$

$(N_{render}-1)C+T_{comp} \leq T_{rend_i}$, $T_{shade} \leq N_{shaders}(C+T_{rend_i})$

Cycle Time = $N_{region}(C+T_{rend_i})$

Latency = $(1+N_{region})(C+T_{rend_i})+T_{comp}+T_{shade}$

The results from this analysis agree with the predictions made by the PixelFlow researchers in equation 11.1. When rendering time is larger than composition and shading times, then cycle time is proportional to this value. Similarly for the compositing time and the shading time, the latter in inverse proportion to the number of shaders. In all these cases, the cycle time is also proportional to the number of regions being rendered. This replaces the summation in equation 11.1.

This approach also provides insight into the effect of the network used for the image composition and into the latency of the system.

The model above assumes that only one message can be received at the composition stage at a time. Communication time is only modelled for this section of the network, since the amount of data transferred in this section is likely to be considerably higher than at any other point.

An interesting point is the dependence of the latency on the highest common factor of N_{region} and $N_{shaders}$ when T_{shade} dominates. Data flow is smoother when the number of regions in the screen shares a common factor with the number of shaders, producing a lower latency. When this is not the case, often the final few regions have to block at the composition stage waiting for access to a shader. The requirement for a common factor greater than two is related to the

[2] HCF = highest common factor

effective length of the pipeline, which determines how much blocking can occur before backing traffic up all the way back to the source.

For: $(N_{render} - 1)C + T_{comp} \geq T_{rend_i}$, $T_{shade} \geq N_{shaders}(N_{render}C + T_{comp})$:

$$\begin{aligned}
\text{Latency (without} &= \frac{N_{shaders} + 2 + N_{region}}{N_{shaders}} T_{shade} - N_{render}C \\
\text{common factor)} &= \frac{N_{shaders} + N_{region}}{N_{shaders}} T_{shade} + 2\frac{T_{shade}}{N_{shaders}} - N_{render}C \\
&\geq \frac{N_{shaders} + N_{region}}{N_{shaders}} T_{shade} + 2N_{render}C + 2T_{comp} - N_{render}C \\
&\geq \frac{N_{shaders} + N_{region}}{N_{shaders}} T_{shade} + N_{render}C + 2T_{comp} \\
&\geq \text{Latency (with common factor)}
\end{aligned}$$

For: $(N_{render} - 1)C + T_{comp} \leq T_{rend_i}$, $T_{shade} \geq N_{shaders}(C + T_{rend_i})$:

$$\begin{aligned}
\text{Latency (without} &= \frac{N_{shaders} + 2 + N_{region}}{N_{shaders}} T_{shade} - N_{render}C \\
\text{common factor)} &\geq \frac{N_{shaders} + N_{region}}{N_{shaders}} T_{shade} + 2C + 2T_{rend_i} - N_{render}C \\
&\geq \frac{N_{shaders} + N_{region}}{N_{shaders}} T_{shade} + 2T_{rend_i} - (N_{render} - 2)C \\
&\geq \text{Latency (with common factor)}
\end{aligned}$$

11.6 Conclusion

This chapter examines the effects of different pipeline variations on cycle time and latency. Latency is a value that is not used frequently for performance analysis and a number of interesting effects are uncovered during the analysis process.

Synchronization delay results in ageing of values, increasing their latency while not accomplishing any significant work. Delaying production of values, using the delay time to improve the result or adding feedback to the pipeline, improves the latency without adversely affecting cycle times.

The slowest stage in a pipeline has a significant effect in the overall performance and is an ideal candidate for further parallel decomposition. Synchronization delay causes all processes before the slowest process in the pipeline to inherit its delay as their contribution to the latency. Ideally the slowest process should be placed at the front of the pipeline where possible, or the number of stages before the slowest process should be minimized.

Buffering can be introduced into the pipeline to smooth flow where processing times at the nodes are irregular. Buffering produces improved cycle times, but can increase latency due to the extra stage. In some cases, operating conditions do exist where an improvement in latency can be attained.

The use of additional parallelism within the pipeline can result in improvements in both cycle time and latency. The effects of the communication network need to be carefully considered in this case. Replication of the pipeline produces substantial improvement in the cycle time as the number

of pipelines increase. Very little change occurs in the latency value. Performance improvement is only obtained until the communication time dominates the other variables.

Use of Analytical Simulation on pipelines found in virtual reality systems provides results that agree with values quoted for those systems. Analytical simulation is able to improve on the results, examining the effects of factors such as communication times, and producing values for metrics such as latency. The values obtained indicate the change in performance that can be expected for an additional investment in hardware.

Chapter 12

Analysis of the world modelling components

The manner in which a world is modelled on a virtual reality system is very much dependent on the nature of the world, and on the requirements on the model. Nevertheless an examination of distributed virtual reality systems (see Chapter 2) reveals a number of common constructs which are implemented in a number of standard ways.

The decomposition into parallel components usually results in a process for every object in the system. Other aspects of the system, for example collision detection, may be implemented using additional processes. These may also be implemented in parallel within the object processes. Some systems make use of application processes where process decomposition is according to the tasks in the system, rather than the structural components. To encompass these variations, the models in this chapter assume the presence of *object processes* which each control the behaviour of a set of one or more objects.

The data representing the world is distributed to the object processes in a variety of manners. Each object can be responsible for the data pertaining to itself and must store it locally to the object process, data can be stored externally possibly at some central point allowing easy access to the complete database, or the database may be replicated across all participating processors.

In practice the divisions are not so clear cut in existing systems. Variations of these approaches may be implemented and this blurs the distinctions. In addition, some systems combine a number of approaches in an attempt to benefit from the positive features of each approach, or because of the constraints of implementation environment. Other aspects of the virtual reality system such as access control or physical simulation may be implemented using a different approach to that used for data distribution.

In an attempt to isolate the effects of the different approaches to providing distributed control and to distributing data, this chapter begins by presenting simple models of the most common decomposition approaches. Analysis of these models provides some insight into the merits of the corresponding approach. These models are examined on a range of network topologies. Since these models are very general, models of specialized decomposition strategies in virtual reality

systems are then presented. These models examine enhancements or alternative approaches which are limited to a specific application or architecture.

12.1 Control distribution methods

This section presents a number of methods for controlling objects in a parallel virtual reality system. This control may be required to dictate the behaviour of the object in the virtual world, or it may coordinate a distributed version of an algorithm necessary for the simulation of the virtual world, such as collision detection or Newtonian mechanics.

A number of simplifying assumptions are made. Each object in the system is assumed to participate in a common task which can be subdivided without overhead into any number of identical components which are run in parallel. The variable X represents the processing time for this task in one of the components, and the variable Y corresponds to time taken to do some housekeeping (for example calculating changes of position) by each object. The only interprocess communication required is that needed to share the results of the common task. The input data is assumed to be up to date and freely available to every process.

12.1.1 Decentralized control

This approach distributes control amongst all of the object processes taking part in solving the problem. Each object process performs a predefined sequence of events, uninfluenced by the remainder of the system. This decomposition could be implemented equally well on a SIMD architecture.

The basic model is as follows:

```
replicate N

    process objectr#
        replicate [N-1]
            receive SOMEWHERE data
        endreplicate
        send object# ok

    process object#
        think X
        replicate [N-1]
            send objectr[(((N-1)+#+##)%N)+1] data C
        endreplicate
        receive objectr# ok
        think Y
        report marker#

endreplicate
```

The model is similar to those used previously for the message passing decomposition of the collision detection algorithm (see section 5.3.1). Each object performs part of the shared computation for period X, sends the result to all other processes and then performs local computation for a further period Y. In this model, messages are sent sequentially, but ordered such that no two messages arrive at the destination simultaneously. The model assumes there are sufficient communication channels to allow this. This model would fit a network with a star topology, where a central switched hub can connect pairs of processors together for the duration of their communication. Thus only one message can be sent and only one received per process at any point in time.

Since there is no interaction in this model, latency is ignored. Only cycle time is measured for the object processes. The cycle time for this model is $X + (N-1)C + Y$ as may be expected, since the X and Y stages are separated by the step in which $N-1$ messages are sent sequentially by each process.

The decentralized decomposition approach is best suited to systems that are able to send messages in parallel across a well connected network. A model which implements this enhancement is shown below:

```
replicate N
    replicate N
        process objectr#x##
            receive objects##x# data
            send object# ok

        process objects#x##
            receive object# data
            send objectr##x# data C
    endreplicate

    process object#
        think X
        replicate [N-1]
            send objects#x[(((N-1)+#+##)%N)+1] data
        endreplicate
        replicate [N-1]
            receive objectr#x[##+((##+1)>#)] ok
        endreplicate
        think Y
        report marker#
endreplicate
```

Extra send and receive processes are added to model the parallel communication. This model assumes that there is no contention for communication channels between any two processors.

Analysis of this model gives the value $X + C + Y$ as the cycle time. This is independent of the number of object processes involved.

Taking the approach to the opposite extreme, a model using a shared communication medium such as Ethernet is shown below.

```
process medium

    receive ANY get
    receive [ANY] give

replicate N

    process objectr#
        replicate [N-1]
            receive SOMEWHERE data
        endreplicate
        send object# ok

    process object#
        think X
        replicate [N-1]
            send medium get
            send objectr[(((N-1)+#+##)%N)+1] data C
            send medium give
        endreplicate
        receive objectr# ok
        think Y
        report marker#

endreplicate
```

This model is non-deterministic and performance varies according to the manner in which this non-determinism is resolved. The results quoted below are for an optimal deterministic variation of the model. In this model the order in which processes may access the communication medium is predetermined and fixed. An optimal ordering for this model corresponds to that given by creating a grid of the numbers from 1 to N, repeated for $N - 1$ rows. Traversing this array from left to right along the upward diagonal gives the order in which process communication should occur. For example, with four processes, the grid is:

1↗	2↗	3↗	4↗
1↗	2↗	3↗	4↗
1↗	2↗	3↗	4↗

and the communication sequence is 1 1 2 1 2 3 2 3 4 3 4 4.

The cycle time for this model is given by:

$PC \geq X + Y$: $\quad N(N-1)C$

$X + Y \geq PC$: $\quad X + QC + Y$

where $\quad P = \begin{cases} (\frac{N}{2})^2 & \text{if } N \text{ is even} \\ (\frac{N-1}{2})^2 + \frac{N-1}{2} - 1 & \text{otherwise} \end{cases}$

$$Q = \begin{cases} \frac{3N^2}{4} - N + 1 & \text{if } N \text{ is even} \\ \frac{3(N-1)^2}{4} + \frac{(N+1)}{2} & \text{otherwise} \end{cases}$$

12.1.2 Central control

This approach uses an additional process to gather the results of the distributed computation and redistribute the combined value, as illustrated in the master-slave version of collision detection described in section 5.3.3. Centralized control makes it easier to vary the number of computational processes, since they are not aware of each other. The model for a star network is given below:

```
process masterr
    replicate N
        receive ANOBJECT data
    endreplicate
    send master ok

process master
    receive masterr ok
        replicate N
            send object# combineddata C
        endreplicate

replicate N
    process object#
        think X
        send masterr data C
        receive master combineddata
        think Y
        report marker#
endreplicate
```

The cycle time for this system is $X + (N+1)C + Y$.

The corresponding model, with additional processes to emulate a totally connected network,

is as follows:

```
replicate N
    process masterr#
        receive object# data
        send master ok

    process masters#
        receive master data
        send object# combineddata C
endreplicate

process master
    replicate N
        receive ANY ok
    endreplicate
    replicate N
        send masters# data
    endreplicate

replicate N
    process object#
        think X
        send masterr# data C
        receive masters# combineddata
        think Y
        report marker#
endreplicate
```

The cycle time is $X + 2C + Y$, independent of the number of processors. Communication overhead with this technique is higher than that for the less centralized approach given previously. The central controller acts as a bottleneck in the system.

The model for this approach implemented on a shared medium is shown below.

```
process medium
    receive ANY get
    receive [ANY] give
```

```
process masterr
    replicate N
        receive ANOBJECT data
    endreplicate
    send master ok

process master
    receive masterr ok
    replicate N
        send medium get
        send object# combineddata C
        send medium give
    endreplicate

replicate N
    process object#
        think X
        send medium get
        send masterr data C
        send medium give
        receive master combineddata
        think Y
        report marker#
endreplicate
```

The cycle time for this model is of the form:

$X + Y \geq (N-1)C$: $\quad X + (N+1)C + Y$

$(N-1)C \geq X + Y$: $\quad 2NC$

with an interesting variation for lower relative values of $X+Y$ versus C. Let $M = 1 \ldots (N-4)$, then for $(M+1)C \geq X + Y \geq MC$, the cycle time is given by $X + (2N - M)C + Y$.

Another interesting fact about this approach when modelled on Ethernet is that there is not the N^2 dependence on communication that occurs with the decentralized approach. While performance using central control is worse than that using distributed control for the networks with better connectivity, central control actually performs better when using a shared communication medium. The overall amount of communication has decreased with this approach.

The totally connected network used previously is not actually required with this approach. It would suffice to have only the processor containing the controller connected to every other processor.

12.1.3 Central service provider

A single process is used to perform the calculation previously distributed over all the object processes. This approach appears to remove any advantage given by the presence of parallel processes. However this may be appropriate when a task requiring access to combined resources must be performed, especially if it cannot be easily distributed.

The model showing control flow using a star network topology is:

```
process master
    replicate N
        think X
    endreplicate
    replicate N
        send object# combineddata C
    endreplicate

replicate N
    process object#
        receive master combineddata
        think Y
        report marker#
endreplicate
```

A delay of NX is incorporated into the central service provider, since X is used to represent the duration of a portion of the task on each of the N object processors used previously. Each object spends time Y on local housekeeping tasks after receiving word that the central task is complete.

The cycle time of this model is:

$Y \geq (N-1)C + NX$: $C + Y$

$(N-1)C + NX \geq Y$: $NC + NX$

Expanding this model to a totally connected system has the expected benefit of producing a constant communication time. The cycle time is:

$NX \geq C + Y$: NX

$C + Y \geq NX$: $C + Y$

	Totally connected	Star topology	Bus (Ethernet)
Decentralized	1	N	N^2
Central control	1	N	N
Central server	1	N, 1	N, 1

Table 12.1: Control distribution dependency on N by C

and this corresponds to the model given below:

```
replicate N
    process masters#
        receive master go
        send object# combineddata C
endreplicate
process master
    replicate N
        think X
    endreplicate
    replicate N
        send masters# go
    endreplicate

replicate N
    process object#
        receive masters# combineddata
        think Y
        report marker#
endreplicate
```

Since only one process is communicating, the results for the system running on Ethernet are the same as for the star topology.

12.1.4 Summary

Table 12.1 shows the relationship between the number of object processes (N) and the communication time (C). The relationship between N and component processing time (X) is shown in Table 12.2. The value shown is the highest power of N in a term involving each variable (C and X) in the expression for cycle time.

The decentralized approach is well suited to networks of processors that are capable of carrying the relatively large numbers of messages that are involved. As network limitations start to become

	Totally connected	Star topology	Bus (Ethernet)
Decentralized	1	1	1
Central control	1	1	1
Central server	N	N	N

Table 12.2: Control distribution dependency on N by X

significant, solutions which may have been less attractive start to perform significantly better, particularly if transmitting a combined result has a similar cost to transmitting each of the original values. In cases where communication is expensive and time required for the combined task is small, it becomes worthwhile to abandon any attempt at parallelism, and to use a central service provider.

12.2 Data distribution methods

The approaches to data distribution are examined in a similar manner to those for control distribution. The assumptions in this case are that each process needs no externally supplied control and that the variable Y represents the time required for housekeeping during each cycle of the object process. Only the time to fetch data is of interest, so the processing delay in the object, previously represented by X, is ignored.

There are a number of enhancements to the data distribution approaches which are not particularly amenable to modelling. These involve the use of techniques such as dead-reckoning and updating only selected areas of databases. These often produce only approximations of the results achieved by more rigorous methods, although they may be sufficient for the purposes required. These variations are analysed in section 12.3, together with more complex combinations of these skeletons.

12.2.1 Distributed databases

In this scenario each object contains the data relating to itself. All objects in the system must explicitly request data from all of the others to build up a complete picture of the world.

The model for this approach is almost identical to the model of decentralized control (see section 12.1.1), once the delay for X is removed. The results for that analysis apply with $X = 0$.

12.2.2 Centralized database

The client-server approach represents a move toward a centralized database. In section 12.1.2 this was successful in improving the performance of the control skeletons under conditions of limited connectivity. The objects are now required to both fetch data from the central server and to update the server when changing the information. The model for a completely connected network

is shown below.

```
process server
    receive REQ SOURCE
    think Y
    if SOURCE != update
        send SOURCE snd
    endif

replicate N
    process serverr#
        receive client# MESSAGE
        if MESSAGE == request
            send server servers#
        endif
        if MESSAGE != request
            send server update
        endif

    process servers#
        receive server snd
        send client# data C

    process client#
        send serverr# request C
        receive servers# data
        send serverr# update C
        report marker#

endreplicate
```

Analysis gives the performance of this model (for N clients) as :

$Y \geq \frac{C}{N-2}$:	$2NY$
$\frac{C}{N-2} \geq Y \geq \frac{C}{N-1}$:	$(2N-1)Y + C$
$\frac{C}{N-1} \geq Y \geq \frac{C}{N}$:	$(N+1)Y + 2C$
$\frac{C}{N} \geq Y$:	$3C + Y$

This solution holds for values of $N > 2$. Similar results hold for smaller values of N, as shown

below:

For $N = 1$:

$Y \geq C$:	$2C + 2Y$
$C \geq Y$:	$3C + Y$

For $N = 2$:

$Y \geq C$:	$4Y$
$C \geq Y \geq \frac{C}{2}$:	$2C + 3Y$
$\frac{C}{2} \geq Y$:	$3C + Y$

Thus, when server access times dominate, the performance depends on the number of clients. For slow network communication, the performance is constant and independent of the number of clients.

The model for the star network follows:

```
process serverrec

    receive ANY MESSAGE
    if MESSAGE == update
        send server update
    endif
    if MESSAGE != update
        send server ANY
    endif

process sendrepl

    receive server REPL
    send [REPL] data C

process server

    receive serverrec MESSAGE
    think Y
    if MESSAGE != update
        send sendrepl MESSAGE
    endif
```

```
replicate N
    process client#
        send serverrec request C
        receive sendrepl data
        send serverrec update C
        report marker#
endreplicate
```

As might be expected, the cycle time for this model shows a linear dependence on N when communication dominates. The cycle time is given by (for $N > 1$):

$Y \geq C$: $2NY$

$C \geq Y$: $2NC$

The results for $N = 1$ are the same as for the completely connected network, as might be expected.

Finally the Ethernet version:

```
process medium
    receive ANY get
    receive [ANY] give

replicate N
    process forclient#
        receive client# MESSAGE
        send server MESSAGE

    process serversendforclient#
        receive server go
        send medium get
        send client# data C
        send medium give
endreplicate

process server
    receive SOURCE MESSAGE
    think Y
    if MESSAGE == request
        send serversend[SOURCE] go
    endif
```

	Totally connected	Star topology	Bus (Ethernet)
Distributed databases	1	N	N^2
Client-server	1	N	N

Table 12.3: Data distribution dependency on N by C, for large C

```
replicate N
    process client#
        send medium get
        send forclient# request C
        send medium give
        receive serversendforclient# data
        send medium get
        send forclient# update C
        send medium give
        report marker#
endreplicate
```

The cycle time is (for $N > 2$):

$Y \geq \frac{3C}{2}$: $2NY$

$\frac{3C}{2} \geq Y$: $3NC$

The results for $N = 1$ are as for the other two models in this section, while for $N = 2$ they are:

$Y \geq \frac{3C}{2}$: $4Y$

$\frac{3C}{2} \geq Y \geq C$: $5C + Y$

$\frac{3C}{2} \geq Y$: $6C$

The dependence on N is linear, with respect to both communication and server access times.

12.2.3 Summary

The relationships between the two decomposition strategies examined in this section are summarized in Tables 12.3 and 12.4. They show the dependency on N by each variable in the system when that variable dominates. The two variables concerned are the communication time (C) and the server response time (Y).

As with the control distribution strategies (section 12.1), the totally distributed approach performs best with a well connected network. Simpler networks benefit from the trade-offs offered by the client-server approach. Communication costs may be higher in some cases for the client-server approach which transmits large amounts of data in single messages. Lower communication

	Totally connected	Star topology	Bus (Ethernet)
Distributed databases	1	1	1
Client-server	N	N	N

Table 12.4: Data distribution dependency on N by Y, for large Y

overheads, and the possibility of selective data transmission for some applications, may make this approach an attractive alternative.

The next section examines specialized decomposition techniques used in virtual reality systems, relating to the manner in which objects are controlled and data is distributed. These techniques also influence the manner in which collision detection, physical modelling and access control is implemented. The performance of each approach is analysed in order to identify the merits of that approach.

12.3 Techniques employed in virtual reality systems

The previous sections examine performance aspects of some common techniques for distributing control and data. The remainder of this chapter investigates the performance of decomposition techniques which are specializations of these approaches, or that are specific to particular architectures.

Results of the performance analyses are summarized at the end of this section to facilitate comparison.

12.3.1 Dead-reckoning

Dead-reckoning is extremely useful in distributed virtual reality systems, being used for data distribution in systems such as DIS (NPSNET), DIVE and applications of the MR Toolkit.

In dead-reckoning, low communication bandwidths are compensated for with increased processing by each entity in the system. Instead of just transmitting position information, additional information, such as the direction of movement and speed, is included. This information is used by each entity to continuously update its estimate of the position of the others. Fewer updates can be sent, since intermediate values can be "dead-reckoned". Objects calculate their own dead-reckoned position and issue updates when this diverges substantially from their actual position.

The model below implements a simple form of dead-reckoning. Assuming N objects, each which takes time X to recalculate the position of one object, the model issues an update every K cycles. The only restriction on communication is a delay of C on the part of the sending object. Each object has a receiving process which, one can imagine, caches the latest position of the various objects for use with the dead-reckoning calculation.

Variable	Interpretation
C	Time required for communication
K	Number of cycles between updates
N	Number of objects in the system
X	Time for sequential processing in each object process

```
replicate N
    process receiver#
        receive FROM THING
        if FROM == object#
            send object# latest
        endif

    process object#
        init
            assign COUNT 0
        endinit
        report mark#
        send receiver# whatsup
        receive receiver# latest
        replicate N
            think X
        endreplicate
        assign COUNT [COUNT+1]
        if COUNT == K
            replicate [N-1]
              send receiver[##+((##+1)>#)] update
              think C
            endreplicate
            assign COUNT 0
        endif
endreplicate
```

Measuring latency is difficult in this case because it is not well defined. The time between changes in the position of the objects is the same as the cycle time in this case, but these positions are not completely accurate, due to the guesswork inherent in the dead-reckoning. The average cycle time is $NX + \frac{N-1}{K}C$, the communication overhead benefits from the less frequent updates, but a price is paid in a higher computational overhead. This approach would be suited to situations in which communication bandwidth is more expensive than processor time.

The model above assumes a star topology. Dead-reckoning is well known for its use in NPSNET, which uses large networks with relatively limited bandwidth. The model below examines dead-reckoning in a more restricted environment, using a shared communication medium.

Variable	Interpretation
C	Time required for communication
K	Number of cycles between updates
N	Number of object processes in the system
X	Time for sequential processing in each object process

```
process medium

    receive ANY get
    receive [ANY] give

replicate N

    process receiver#
        receive FROM THING
        if FROM == object#
            send object# latest
        endif

    process object#
        init
            assign COUNT 0
        endinit
        report mark#
        send receiver# whatsup
        receive receiver# latest
        replicate N
            think X
        endreplicate
        assign COUNT [COUNT+1]
        if COUNT == K
            replicate [N-1]
              send medium get
              think C
```

```
            send receiver[##+((##+1)>#)] update
            send medium give
          endreplicate
          assign COUNT 0
        endif
    endreplicate
```

The cycle time for this model shows a higher order dependence on the communication time. The values are given below:

$NX \geq \frac{(N-1)^2}{K}C$: $\quad NX + \frac{N-1}{K}C$

$\frac{(N-1)^2}{K}C \geq NX$: $\quad \frac{N(N-1)}{K}C$

As long as N is relatively small and the update interval is sufficiently large, the cycle time scales linearly with the number of processors. As N increases, the factor $\frac{(N-1)^2}{K}C$ will dominate at some point, causing an N^2 dependence in the cycle time. The only way to counteract this is to increase the interval between updates, which may affect the quality of the simulation.

Dead-reckoning improves the cycle time. It allows a linear dependence on the number of processes in the system for shared media, provided the number of processors does not grow beyond a certain threshold.

12.3.2 Token passing

This approach addresses the problems of distributed control and contention for shared resources within virtual reality systems. Control of shared databases and the resolution of disputes becomes the responsibility of the process holding the token.

The use of token passing is illustrated in an application written for the MR Toolkit. This application allows people at different physical locations to play a game of handball. A number of bricks, which disappear on contact with the ball, are placed at the front of the virtual court. The colour of the ball corresponds to the player whose turn it is to hit the ball. Each player takes it in turn to hit the ball and have it reflect off the front and side walls and strike the bricks. Each MR master process corresponds to a user, complete with HMD, dataglove and trackers. The ball is used as a token to transfer responsibility for movement and collision detection between master processes.

The user whose turn it is to hit the ball is the owner of the ball. The corresponding master process simulates the ball motion and performs the collision detection to cause reflection off walls and hand. After deflection off the hand, ownership is passed on to the next player.

The access control in this application is very simple and is a function of the application, in that no specialized functions provided by the system for this purpose are used. The round robin passing of control of the object in this situation is only suitable in the case of a single shared resource being shared equally amongst all the contending processes.

A simple model implementing the round robin approach to control of the system is shown below. Each master process needs to perform a certain amount of work, modelled by the delay of

Y. Each master process also takes it in turn to control other objects for N cycles which takes an additional time X. The time taken to pass the token on to the next process is represented by C. One process is charged with creating and inserting the token into the system. The model assumes M master processes in the system.

Variable	Interpretation
C	Time required to communicate the token
M	The number of processes being modelled
N	The number of sequential cycles for which the token is held
X	Time taken for additional work while holding the token
Y	Time taken for standard processing during each cycle
Z	Potential extra work that can be done while not holding the token

```
replicate M
    process master#
        init
            if # == M
              send master1 token
            endif
        endinit
        report marker#
        receive master[((((#+M)-2)%M)+1)] token
        replicate N
            think X
            think Y
        endreplicate
        send master[(#%M)+1] token C
        replicate [N*(M-1)]
            think Y
            // think Z
        endreplicate
endreplicate
```

As can be expected, the system is dominated by the process carrying the extra load and this limits the average performance of each process. The average cycle time is $\frac{1}{N}C + X + Y$. The processes that are not carrying any extra load can run ahead but must block eventually when they receive the token. This blocking period could be usefully used, either by splitting the task modelled by X into smaller ones which could be run in parallel, or by performing additional work on the free processors. The amount of extra work that can be included when not holding the token can be found by including the variable Z into the above model, to soak up the extra cycles.

As long as $\frac{M-2}{N}C + (M-1)X \geq (M-1)Z$, the average performance of the system is unaffected. When Z dominates, the average cycle time becomes $\frac{1}{(M-1)N}C + Z + Y$. This could be viewed as a situation in which the process holding the token performs less work than the others. This has application where interactive performance is required of only one of the processes at a time.

The model above makes use of synchronous communication, where processes block once their work is complete and they are waiting for the token. This would apply where all processes have to run the same number of cycles. This is applicable to distributed access control using token passing where the extra work for the process is probably quite small and the major overhead arises from the communication costs in transferring the token. For control distribution, an asynchronous model is required that will allow processes to run ahead of the one holding the token and performing the extra work. Such a model is presented below:

Variable	Interpretation
C	Time required to communicate the token by sending process
M	The number of processes being modelled
N	The number of sequential cycles for which the token is held
Y	Time taken for standard processing during each cycle

```
replicate M
    process token#
        init
            if # == 1
              assign TOK 1
            endif
            if # != 1
              assign TOK 0
            endif
        endinit
        receive SOMEONE COMMAND
        if COMMAND == token
            assign TOK 1
        endif
        if COMMAND == passon
            send token[(#%M)+1] token
            assign TOK 0
        endif
        if COMMAND == read
            send SOMEONE TOK
        endif
```

```
process master#
    init
        assign COUNT 0
    endinit
    report markerstart#
    sendnotrace token# read
    receive token# TOK
    if TOK == 1
        think Y
        think Y
        assign COUNT [(COUNT+1)%N]
        if COUNT == 0
          sendnotrace token# passon
          think C
          assign COUNT 0
        endif
    endif
    if TOK == 0
        think Y
    endif
    report markerend#
endreplicate
```

To simplify the analysis, the extra work when holding the token is represented as an extra delay of period Y. Communication time is modelled by a delay of period C in the source process.

The cycle time of this model is given by:

$(K+1)Y \geq C \geq KY$: $\frac{(2MN+M-1-K)Y+C}{2MN+M-N-1-K}$

(where $K = 0 \ldots 2MN + M - 2N - 2$)

$C \geq (2MN + M - 2N - 1)Y$: $2Y + \frac{C}{N}$

Unless communication is extremely slow, the average performance indicates that the load is successfully being balanced, particularly as the number of processors increases. Transient analysis shows instantaneous values of the higher load occurring when the process is holding the token.

The major reason for the token passing mechanism in the MR Toolkit's handball application is to provide interactive performance to the owner of the ball, even if equivalent performance cannot be guaranteed elsewhere. This effect is not obvious from the results presented so far. The models look at average performance which hides any localized performance enhancements.

The reason for using token passing to distribute collision detection and physical simulation is that the time required to communicate these results from other processes has an impact on the

interactive performance. For a token passing mechanism to be considered, the communication time would have to be much larger than the time required to compute the results locally. The model below represents the handball application found in the MR Toolkit. It uses asynchronous communication with dead reckoning - some information is updated every five cycles, while other information is only updated every ten cycles. Information is updated only by the process holding the token.

Variable	Interpretation
C	Time required to communicate the token
M	The number of processes being modelled
N	The number of sequential cycles for which the token is held
Y	Time taken for standard processing during each cycle

```
generate zero1 -

replicate M

    process recmaster#
        init
            assign GOTUPDATE no
        endinit
        receive ANYWHERE ANYTHING
        if ANYTHING == update
            assign GOTUPDATE yes
        endif
        if ANYTHING == latest
            if GOTUPDATE == yes
              send ANYWHERE update
            endif
            if GOTUPDATE == no
              send ANYWHERE noval
            endif
            assign GOTUPDATE no
        endif

    process havetoken#
        init
            assign TOK 0
            assign TIME 0
        endinit
        receive SOMEONE COMMAND
```

```
    if COMMAND == update
       assign TOK [1-TOK]
       assign TIME 0
    endif
    if COMMAND == read
       send SOMEONE TOK
       if TOK == 1
         assign TIME [TIME+1]
       endif
       send SOMEONE TIME
    endif

process mastermedium#
    receive master# DESTIN
    think C
    send DESTIN update

process master#
    init
       assign COUNT 0
       if # == M
         send havetoken1 update
       endif
    endinit
    report marker#
    send havetoken# read
    receive havetoken# TOK
    receive havetoken# TIME
    if TOK == 1
       think Y
       think Y
       assign COUNT [(COUNT+1)%10]
       if [COUNT%5] == 1
         replicate [M-1]
           send mastermedium#
           recmaster[##+((##+1)>#)]
         endreplicate
       endif
```

```
            if [COUNT%10] == 1
              replicate [M-1]
                send mastermedium#
                recmaster[##+((##+1)>#)]
              endreplicate
            endif
            if TIME == N
              send havetoken# update
              send mastermedium# havetoken[(#%M)+1]
            endif
            report zero#
          endif
          if TOK == 0
            send recmaster# latest
            receive recmaster# VALUE
            if VALUE == update
              report zero#
            endif
            think Y
          endif
    endreplicate
```

This model allows processes transmitting messages across process boundaries (tokens and updates) to do so asynchronously by passing the message to a buffer process. This simulates the delay in transmitting across the medium after allowing the original process to continue. The buffer can only hold one message at a time, after which the transmitting process blocks. This is considered sufficient for the analysis. More detail would require additional assumptions about the communication medium which the system is using.

This model also assumes that the time taken to simulate ball motion is approximately equivalent to the time required to animate the user. The processing time with the token is $2Y$, or Y without the token. The performance measure of interest is the time between the updates of the ball position that occur at points marked by report markers in the model.

The number of periodic actions within the system (of period 5, 10, $N \times M$) makes solving the model a complex process and the resulting solution excessively convoluted. For the purposes of the performance analysis, samples of the solution under various conditions are sufficient to illustrate the various behavioural characteristics.

A typical cycle for two processes ($M = 2$) and $N = 10$ with $Y \cong 5C$ is as follows:

$$2Y \text{ (9 times)} \qquad 13Y \qquad 6Y \qquad 10Y \qquad 10Y + C$$

The delay between updates is well controlled when the token is held, shown by the sequence of $2Y$s. Performance drops off by a far greater amount when the token is passed on, resulting in an average of about $11Y$ between updates. The length of the cycle without the token lasts for a period of $39Y + C$ which seems to imply an asymmetry in the system (since this is well over half of the total token cycle). It turns out that the difference is made up of a period in which nobody holds the token but in which it is still being transferred. The slower the communication, the more noticeable this phenomena becomes. Since the processes are cycling at a constant rate, the time between updates increases if the token is passed frequently. The factor of C in the cycle time above occurs because of the double update every 10 cycles which overflows the communication buffer. With a large enough buffer, time between updates would be dependent only on Y.

Removing the dead-reckoning operations and transmitting updates every cycle produces the following cycle of values for the operating conditions used previously:

$5Y$ (9 times) $8Y$ C (8 times) $2C$ $13Y$

The cycle time while holding the token has risen due to the blocking resulting from attempting to communicate. The average time between updates has risen, from about $4.8Y$ when using dead reckoning, to $5.8Y$ using only the token passing mechanism. The inter-update time when not holding the token has dropped to about $6.5Y$.

The value of token passing for ensuring interactive performance levels on the process holding the token in the presence of slow networks is demonstrated with these models. A measure of dead-reckoning is beneficial, since off-processor traffic must not saturate the communication links.

12.3.3 Broadcasting

Combined with dead-reckoning is the notion of broadcasting. Updates are transmitted in a single message, which is received by all other entities in the system.

Broadcasting tends to require a dedicated network, since it is considered to be an undesirable communication technique when sharing a communication medium with other applications. It impacts on the performance of other machines attached to the same network, by forcing them to process packets that are not intended for them. It also sends messages to every node on the network that may only be applicable to a smaller subset.

The model below implements broadcasting on Ethernet. Each process has some sequential processing time, X, before sending an update to all other processes. A message from every other process is then required before the cycle is repeated.

Variable	Interpretation
C	Time required for communication
N	Number of object processes in the system
X	Time for sequential processing in each object process

```
process medium

    receive ANY get
    receive [ANY] give
```

```
replicate N
    replicate [N-1]
        process object[##+((##+1)>#)]from#
            receive broadcast# update
            send object[##+((##+1)>#)] update
    endreplicate

    process broadcast#
        receive object# update
        send medium get
        think C
        send medium give
        replicate [N-1]
            send object[##+((##+1)>#)]from# update
        endreplicate

    process object#
        report mark#
        think X
        send broadcast# update
        replicate [N-1]
            receive INCOMING update
        endreplicate
endreplicate
```

The results show an improvement on the similar system presented under the generic control distribution models.

$C \geq X$: Cycle time $= NC$

$X \geq C$: Cycle time $= X + (N-1)C$

The dependence on C is now linear in N, rather than the N^2 relationship for the non broadcast version.

The model below combines dead-reckoning with broadcasting over Ethernet.

Variable	Interpretation
C	Time required for communication
K	Number of cycles between updates
N	Number of object processes in the system
X	Time for sequential processing in each object process

```
process medium
    receive ANY get
    receive [ANY] give

replicate N
    process receiver#
        receive FROM THING
        if FROM == object#
            send object# latest
        endif

    process object#
        init
            assign COUNT 0
        endinit
        report mark#
        send receiver# whatsup
        receive receiver# latest
        replicate N
            think X
        endreplicate
        assign COUNT [COUNT+1]
        if COUNT == K
            send medium get
            replicate [N-1]
              send receiver[##+((##+1)>#)] update
            endreplicate
            think C
            send medium give
            assign COUNT 0
        endif
endreplicate
```

The cycle times achieved with this model are:

$NX \geq \frac{N-1}{K}C$: $\quad NX + \frac{1}{K}C$

$\frac{N-1}{K}C \geq NX$: $\quad \frac{N}{K}C$

The use of broadcasting with dead-reckoning removes the upper limit for the linear dependence on N. The local processing time has increased against the values found for the broadcast model without dead-reckoning, while the communication time is reduced further by using less frequent updates.

12.3.4 Multicasting

Multicasting is a variant on broadcasting without some of its disadvantages. Multicast messages can be addressed to a group of machines. In this way a single message can be used to distribute an update to a set of processes. Multicasting is becoming increasingly popular, particularly with large systems where the many participants are distributed over large areas. Multicasting is used in such systems as NPSNET, and DIVE.

A simple model of multicasting is to treat each multicast group as a broadcast system that does not affect the other groups. This assumes that there is no overlap in any of the groups; each process communicates with only one group. In that case the results for broadcast apply, with a smaller value for the number of processes.

The effectiveness of multicasting depends on the coherence of the objects in a particular simulation, an analysis better suited to a more statistical approach. Simulations are quoted to have shown a reduction of 90% in traffic [Mac95a]. It is still beneficial to examine the use of multicasting in simple models, to gain some insight into the relationship between performance gains and resources required.

NPSNET is used over the wide area network of the Internet. As such, message propagation is not completely equivalent to any used in the models given previously in this section. The physical size of this network means that it is not feasible to attempt to use synchronous communication. The complexity of the network prevents testing for contention on the network. A more accurate model may be to treat the network as a black box into which a message disappears and reappears at some other point some time K later. To model bandwidth limits, we assume the ability to carry L messages at any time. Each process takes a time interval represented by C to send a message. This runs concurrently with K, and so we also assume $C < K$.

Given that a link is established between broadcasting and multicasting, this network model considers only the broadcasting issues and does not investigate the use of grouping.

Variable	Interpretation
C	Communication overhead in transmitting a message
K	Propagation delay for a message through the network
L	Bandwidth of the network (number of messages that can be carried)
N	Number of object processes in the system
X	Time for sequential processing in each object process

```
generate mark2 - mark1
assume K > C

replicate L

    process message#

        receive broadcast ready
        receive broadcast FROM
```

```
        think K
        send broadcast[FROM] update
endreplicate

process broadcast

    replicate L
        send message# ready
        receive SOMEONE NUMBER
        send message# NUMBER
    endreplicate

replicate N

    replicate [N-1]
        process object[##+((##+1)>#)]from#
        receive broadcast# update
        send objectreceiver[##+((##+1)>#)] update
    endreplicate

    process broadcast#
        receive MESPROC update
        replicate [N-1]
            send object[##+((##+1)>#)]from# update
        endreplicate

    process objectreceiver#
        receive ANY ANYTHING
        if ANY == object#
            send object# latest
        endif

    process object#
        report mark#
        send objectreceiver# wotyagot
        receive objectreceiver# latest
        think X
        send broadcast #
        think C
endreplicate
```

Trends in the latency between two markers in different processes are summarized in the fol-

lowing table:

	$L = 1$	$L \gg 1$
$N = 2$	$2K \geq X + C$: $5K$ $X + C \geq 2K$: $2C + 2X + K$	$K \geq L(X + C)$: $2K$ $X + C \geq K$: $3(X + C)$
$N > 2$	$NK \geq X + C$: $7K$ $X + C \geq NK$: $2C + 2X + K$	-

More detailed examination of the latency values gives the following information:

- For constant numbers of object processes:
  - Latency is largest for small values of L at constant N.
  - Drop off in latency is initially extremely rapid as L increases and slows once the bandwidth is no longer a major influence.
  - With relatively large values of $X + C$ latency initially increases as L increases due to the additional buffering offered by the greater bandwidth. Processes can place more values on the network before congestion occurs and they have to block. Latency drops rapidly when bandwidth increases and no further congestion occurs.
- For constant bandwidth:
  - Where K is the dominant variable, the latency is dependent only on K, and increases with the number of object processes.
  - Where $X + C$ is dominant, the value is dependent on $X + C$, and becomes independent of N.

Thus for large networks ($K \gg X + C$), the major factor influencing latency is the propagation delay. It is important to have sufficient bandwidth to carry the total load of the system. Very significant performance loss can result when this is not the case. Severe increase in latency occurs when several messages are queued in the network when congestion occurs. Where computation time is greater than the propagation delay, latency is unaffected by network performance issues.

While these trends may not be unexpected to anyone with an understanding of network behaviour, this analysis does provide increased detail on the dependence of latency on the variables in the system. The effects of message queueing on the latency is significant, and is not particularly easy to observe in large networks.

12.3.5 Object servers

The use of separate processes for each object in the system can require substantial resources if each process is to be housed on a separate processor. Usually a number of processes are run concurrently on a single processor. This can be inefficient since it leads to replicated code and the need to maintain separate contexts for each process. In some systems such as AVIARY and RhoVeR, there is support for a single process to run the code corresponding to a group of object

processes. This reduces system resources required, decreases the quantity of context switching and allows migration of object processes from one object server to another for load balancing.

A model used to investigate the communications implications of these object servers is shown below. It assumes the presence of M processors each with N processes. Communication time is K for off-processor communication, and C for communication between processes local to a processor. The medium model is that of Ethernet with a token passing mechanism included as an additional enhancement to reduce the model's complexity. This enforces a round robin approach to access to the medium. Local (intra-processor) communication time is assumed to be small compared to the other variables.

Variable	Interpretation
C	Time required for communication between processes on the same processor
K	Time required for communication between processors
N	Number of object processes on each processor
M	Number of processors in the system
P	Number of processes in the system ($N.M$)
X	Time for sequential processing in each object process

```
process medium
    replicate [M*N]
        receive object# get
        receive object# give
    endreplicate

replicate N
    replicate M
        process objectr[((#-1)*M)+##]
            replicate [(N*M)-1]
              receive SOMEWHERE data
            endreplicate
            send object[((#-1)*M)+##] ok

        process object[((#-1)*M)+##]
            think X
            replicate [(M*N)]
              if [(((N*M)+###)-##)%M] != 0
                send medium get
                send objectr[###] data K
                send medium give
              endif
```

$N \times M$ \ M	2	3	4	5	6
2	$2K$				
3		$5K$			
4	$6K + 2C$		$10K$		
5				$17K$	
6	$14K + 2C$	$20K + C$			$26K$
7					
8	$26K + 2C$		$42K + C$		
9		$47K + 4C$			
10	$42K + 2C$			$72K + C$	
11					
12	$62K + 2C$	$86K + 9C$	$98K + 4C$		$110K + C$

Table 12.5: Cycle times for X dominant

```
            if [(((N*M)+###)-##)%M] == 0
              if [((#-1)*M)+##] != ###
                send objectr[###] data C
              endif
            endif
          endreplicate
          receive objectr[((#-1)*M)+##] ok
          report marker[((#-1)*M)+##]
      endreplicate
  endreplicate
```

When X dominates over the inter-processor communication time (K), the results are as shown in Table 12.5 (A common term of X in each value is not shown).

When K dominates the situation is as shown in Table 12.6.

The system depends quite heavily on the inter-processor communication time, even when local computation is substantial. Examining the coefficient of K, for equal numbers of processes, it can be seen that halving the number of processors in the system does not quarter (or even halve) the communication factor, as might be naïvely expected given the normal square relationship between number of processors and communication time on a shared communication medium. The difference, of course, arises from the presence of more than one communicating process on each processor. The coefficient (for K dominating the other variables) is given by $(NM)^2 - N^2M$, or $P^2 - NP$ when phrased in terms of the number of processes ($P = NM$). The dominant term is still dependent only on P, and so the advantages of sharing processors are limited, for $N < P$.

$N \times M$ \ M	2	3	4	5	6
2	$X + 2K$				
3		$6K$			
4	$8K$		$12K$		
5				$20K$	
6	$18K$	$24K + C$			$30K$
7					
8	$32K$		$48K + C$		
9		$54K + 4C$			
10	$50K$			$80K + C$	
11					
12	$72K$	$96K + 9C$	$108K + 4C$		$120K + C$

Table 12.6: Cycle times for K dominant

12.3.6 Synchronous databases

The problem of keeping the distributed world representation consistent across all machines was addressed in an early version of DIVE.

Initially database locking was used to ensure consistency. The database was managed by the ISIS toolkit [Bir90]. Changes to objects in the database required changes to be distributed to all processes. Access control for the database was managed by distributed locks. The ISIS toolkit provides facilities for fault tolerant distributed databases, amongst other things. As used in DIVE it uses a multicast protocol to distribute changes and set locks. All nodes in the system are guaranteed to have seen the same sequence of events, which while good for system integrity, provides some limits on scalability. A limit of about ten peers is given as an upper bound for a DIVE system.

The mechanism involved in maintaining the database is modelled below. Each process spends a period X calculating updates, before distributing the change to other members of the database. It is assumed that one multicast message is enough to allow processes to choose the one that gains control of the database and sets the lock. Once a process has locked the database, it can send an update and then an unlock message. The locking is controlled in the model by the process representing the medium. Each process is allowed to have a turn. Since the system is symmetrical the actual order is not important.

Variable	Interpretation
C	Time required for communication between processors
N	Number of object processes in the system
X	Time for sequential processing in each object process

```
process medium
    replicate N
        receive application# get
        receive application# give
        replicate N
            if # != ##
              receive localdbmanager## get
              receive localdbmanager## give
            endif
        endreplicate
        receive application# get
        receive application# give
        receive application# get
        receive application# give
    endreplicate

replicate N
    process localdbmanager#
        replicate N
            if # != ##
              receive application## DATA
              think C
              if DATA == lockreq
                send medium get
                send application## lockrep C
                send medium give
                receive application## update
                receive application## lockrel
              endif
            endif
        endreplicate

    process application#
        report reports#
        think X
        // will block until lock established
        // send all others lock request
        send medium get
```

```
            replicate N
               if # != ##
                 send localdbmanager## lockreq
               endif
            endreplicate
            think C
            send medium give
            // receive reply from all others
            replicate N
               if # != ##
                 receive localdbmanager## lockrep
               endif
            endreplicate
            // send all others update
            send medium get
            replicate N
               if # != ##
                 send localdbmanager## update
               endif
            endreplicate
            think C
            send medium give
            // send all others lock release
            send medium get
            replicate N
               if # != ##
                 send localdbmanager## lockrel
               endif
            endreplicate
            think C
            send medium give
      endreplicate
```

Cycle times for this system are:

$(N^2+N-2)C \geq X: \;\; ((N+1)^2-1)C$

$(N^2+N-2)C \leq X: \;\; X+(N+2)C$

The N^2 dependence in the coefficient of C for large values of C can be traced to the messages sent by each process acknowledging the setting of the lock. These cannot be multicast. Removing

this stage from the model produces the cycle time shown below:

$3(N-1)C \geq X$: $3NC$

$X \geq 3(N-1)C$: $X+3C$

Communication in DIVE with SID uses an enhancement to multicast protocols. By taking advantage of the multicast environment, processes can watch for specific information for a short period before actually requesting it. In situations where frequent access to the same item is required, this could conceivably reduce traffic due to requests and repeated replies.

12.3.7 Computation on the server

The client-server approach for data and control distribution is employed in a number of virtual reality systems. It is usually used to store data to which each client process requires equal access. A concept introduced in Cyberterm is the ability to run programs (agents) on the server. This is an advantage when communication bandwidth is limited. Instead of transferring the data to the process, the process is relocated to the source of the data.

The idea of a client-server system where the server is capable of performing computational tasks, as well as just providing access to a shared database, is explored in the model below. The network model is that of the star topology, which may be a fair representation of a Cyberterm system interconnected via modem. The model assumes the presence of N clients requesting data only, which requires a processing time of Y on the server, and M clients running an agent on the server, which requires an additional time W on the server.

Variable	Interpretation
C	Time required for communication between processes on the same processor
N	Number of data-only clients in the system
M	Number of clients requiring processing on the server
X	Time for sequential processing in each object process
Y	Time required to fetch data on the server
W	Time required to perform requested processing on the server

```
process server

    receive CLI REQ
    if REQ == work
        think W
    endif
    think Y
    send [CLI] rep C
```

```
replicate N
    process reqclient#
        report rep#
        think X
        send server req C
        receive server rep
endreplicate

replicate M
    process workclient#
        think X
        send server work C
        receive server rep
endreplicate
```

The performance of this system is given by:

$2(M+N-1)C+(M-1)W+(M+N-1)Y \geq X$:

$$\text{Cycle Time} = \quad 2(M+N)C+MW+(M+N)Y$$

$X \geq 2(M+N-1)C+(M-1)W+(M+N-1)Y$:

$$\text{Cycle Time} = \quad X+2C+W+Y$$

The dependency on both W and Y when the server is not completely loaded (for large X) is an interesting case which benefits from further investigation. One might expect the factor of W not to affect the performance of the data-only clients, particularly if the server is not busy. However the cycle time for both clients in the steady state is in fact the value given above. Investigation of the system for $N=1$, $M=1$, shows that a smaller cycle time can in fact be experienced by the data only clients, but only during a particularly unstable transient. As long as the data-only client can get its request in ahead of a client running the agent, it can benefit from an improved performance without cost to the one running the agent. The transient is manifested only if the initial non-deterministic choice of the server is for the data-only client, and lasts only for a short period determined by the value of X.

The approach can be modified to take advantage of this transient, and to bias the system toward maintaining it. The model below shows how this can be done:

Variable	Interpretation
C	Time required for communication between processes on the same processor
N	Number of data-only clients in the system
M	Number of clients requiring processing on the server
X	Time for sequential processing in each object process
Y	Time required to fetch data on the server
W	Time required to perform requested processing on the server

```
process serverreceiver

    receive ANY THING
    if ANY != server
        if HAVE != waiting
            receive server REQ
            send server [ANY]
            send server [THING]
        endif
        if HAVE == waiting
            send server [ANY]
            send server [THING]
            assign HAVE undef
        endif
    endif
    if THING == waittill
        assign HAVE waiting
    endif
    if THING == gotany
        send server none
        send server none
    endif

process server

    send serverreceiver waittill
    receive serverreceiver CLI1
    receive serverreceiver REQ1
    if REQ1 == work
        think W
        send serverreceiver gotany
        receive serverreceiver CLI2
        receive serverreceiver REQ2
        if REQ2 == req
            think Y
            send [CLI2] rep C
        endif
        think Y
        send [CLI1] rep C
```

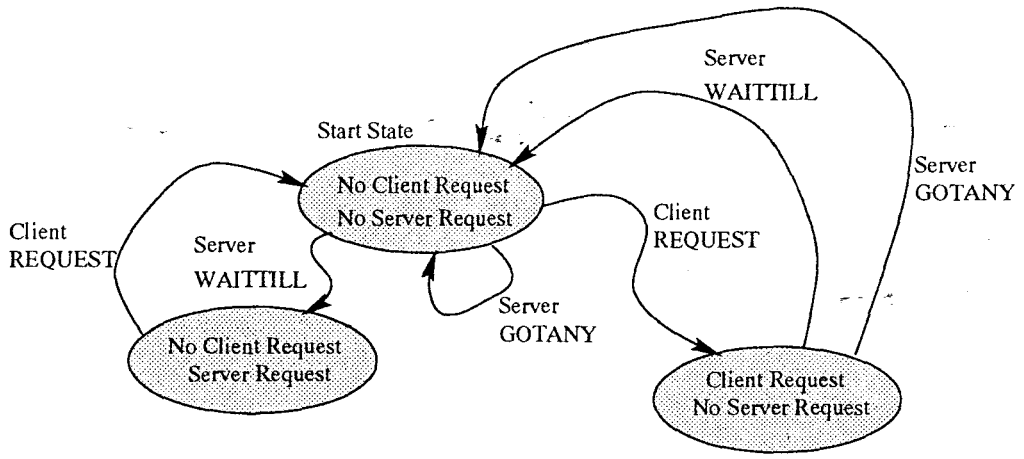

Figure 12.1: State transition diagram of the *serverreceiver* process

```
    if REQ2 == work
        think W
        think Y
        send [CLI2] rep C
    endif
endif
if REQ1 != work
    think Y
    send [CLI1] rep C
endif
```

The clients are as before, except that messages are directed to *serverreceiver* instead of *server*. The server now tries to service data-only clients before those requiring processing for agents. The *serverreceiver* routine can notify the *server* process of a data-only request that has arrived while an agent is being processed, and this is serviced before the response to the agent is returned. The *serverreceiver* process embodies the simple state transition diagram shown in Figure 12.1.

Performance of the data-only clients turns out to be independent of W in many cases, particularly when W is greater than Y. Performance of both clients benefits in many cases, as often the dependency on W only manifests once in a number of cycles. Thus a simple priority scheme as illustrated here can produce performance enhancements in portions of the system, without adversely affecting others.

12.3.8 Multiple servers

Previously, client-server systems have been treated as if there is only a single system-wide server. In light of the advantages of grouping used in multicasting, the use of multiple servers which each provide a specific set of data is suggested. Multiple servers occur in systems such as Division's dVS, and Cyberterm.

An area that has not been examined previously is the use of multiple client-server systems utilizing a shared medium. Provided clients do not interact with more than one server, a completely connected network can run multiple client-server systems at the same performance level as a single client-server system.

The basic model for this client-server system running over Ethernet is shown below, for N servers each with M clients:

Variable	Interpretation
C	Time required for communication between processes on the same processor
N	Number of servers in the system
M	Number of clients accessing one server
X	Time for sequential processing in each object process
Y	Time required to fetch data by the server

```
replicate N
    process server#
        receive ANY SOURCE
        think Y
        send mediums get
        send medium get
        send [SOURCE] rep C
        send medium give

    replicate M
        process serversender#for##
            receive server# rep
            send client##forserver# rep

        process serverreceiver#from##
            receive client##forserver# req
            send server# serversender#for##

        process client##forserver#
            think X
            send medium get
            send serverreceiver#from## req C
            send medium give
            receive serversender#for## rep
            report c##x#
    endreplicate
endreplicate
```

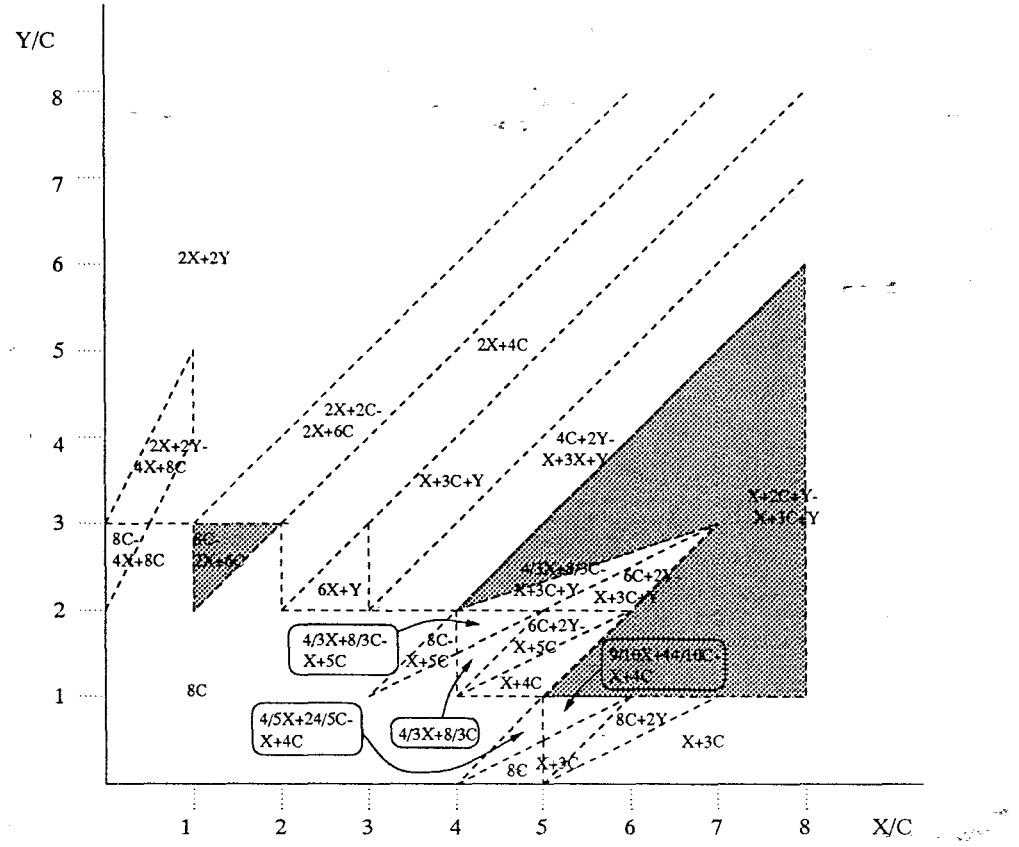

Figure 12.2: Performance of the 2 server/2 client system

```
process medium

    receive ANY get
    receive [ANY] give
```

This model explores a variation to the client-server approach described in 12.2.2. Apart from the multi-server aspect, this model also assumes that the server cannot send replies and process requests simultaneously. The update step is also ignored with this model. Difficulties arise when trying to apply an analysis tool to this model. These result from the frequent use of non-determinism to choose between processes accessing the servers and the communication medium. This creates a large state space which is time-consuming to generate and explore. The result of that analysis produces the possible range of cycle time for clients within the model.

Given that the value of interest is the minimum cycle time, it is beneficial to use the results of the analysis for relatively small systems to determine the manner in which the non-determinism should be resolved in order to produce a minimum cycle time. This can then be used to simplify analysis of larger models. The performance of the model for two servers each with two clients is shown in Figure 12.2. The values in the relative constraint regions can be seen in this graph of $\frac{Y}{C}$ versus $\frac{X}{C}$. The areas in which non-determinism affects the results are marked with the range of performance values that can be expected in the steady state.

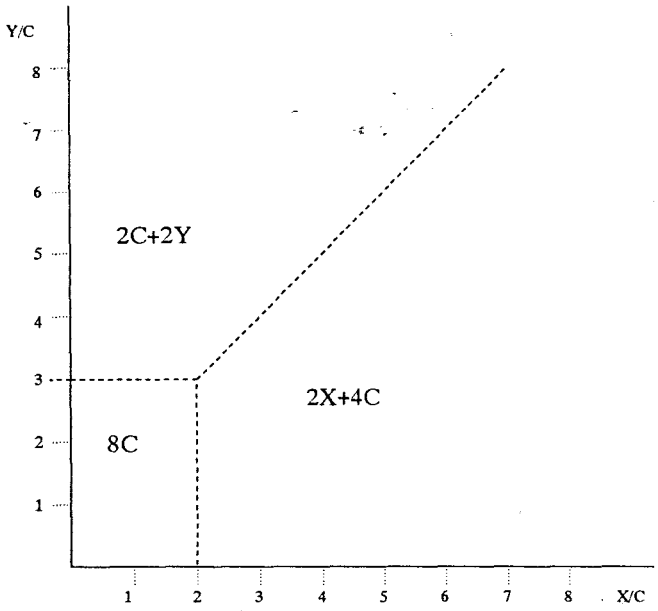

Figure 12.3: Performance of the 2 server/2 client system optimized for large Y

Two areas are selected for examination in the attempt to eliminate non-determinism and select an optimal execution path. A sample where Y and C are relatively large compared to X, and another where X is the dominant variable are shown as the shaded areas in Figure 12.2. Both require that server and client access to the communication medium be alternated to achieve optimal performance. The situation where server delay (Y) is large requires that some jobs from the clients be queued at the server, so as to keep it continually busy. The case where client delay (X) dominates requires the opposite, that jobs be serviced immediately so as to keep round trip time at a minimum. Since these requirements are mutually exclusive, the optimized algorithms need to be selected once the operating constraints are known, or alternatively, some dynamic switching needs to be performed as discussed in section 13.2.

The effect of the two different approaches on the other constraint regions is shown in Figures 12.3 and 12.4. As can be seen, they achieve optimal performance only in some regions.

These models are used for examining the performance of the multiple server system with regard to the three constraint regions where each variable is dominant, and optimal performance is achieved (the area with large values of C occurs at the origin of the diagrams).

Where C dominates: Cycle time $= 2MNC$

Where Y dominates:

$(Y \geq (2N-1)C)$

Cycle time $= M(C+Y)$

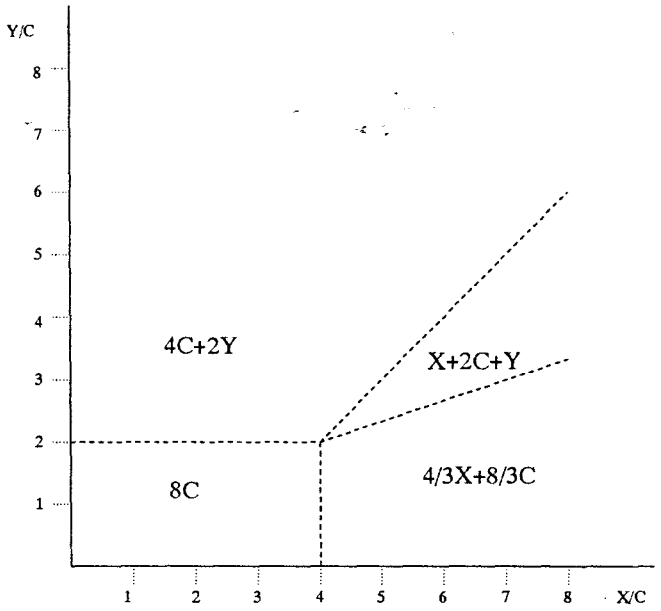

Figure 12.4: Performance of the 2 server/2 client system optimized for large X

Where X dominates:

$$(X \geq (M-1)(Y+2C),\ (MN-N+1)Y \geq (N-1)(2C+X))$$

$$\text{Cycle time} = X + 2C + Y$$

These values hold for $M > 1$, $N > 1$. The absence of a dependence of M and N (numbers of clients and servers) is some cases above can be slightly misleading. While it does imply that these can be increased arbitrarily without impairing the performance, the region in which C dominates expands with increased M and N, eventually resulting in a different expression for cycle time being applicable.

12.3.9 Asynchronous lossy databases

The use of a synchronous database was discussed in section 12.3.6, where identical copies are maintained for all processes. The locking mechanism causes performance loss. Many virtual reality applications would be content if provided with the most recent version of data regarding the state of the world; consistency between processes is not required. The VEOS system uses this principle when performing updates from *boundary* to *external* partitions. Values that are not transferred before new values are generated are dropped.

The performance considerations with this approach are examined with the following model:

Variable	Interpretation
C	Time required for communication between processes on the same processor
N	Number of object processes in the system
M	Ratio between N and C ($\frac{C}{N}$)
X	Time for sequential processing in each object process

```
replicate N
    process entity#
        report report#
        think X
        send environ# endcycle

    process statussendentity#
        init
            assign STAT ready
        endinit
        receive SOURCE how
        if SOURCE == environ#
            send environ# STAT
            if STAT == ready
              assign STAT busy
            endif
        endif
        if SOURCE != environ#
            assign STAT ready
        endif

    process sendentity#
        send statussendentity# how
        receive environ# message
        replicate N
            if # != ##
              send entityexternal## update C
            endif
        endreplicate

    process entityexternal#
        receive ANY update
        report arrive#
endreplicate

process sync
    replicate N
        receive environ# sync
    endreplicate
```

```
replicate N
    process environ#
    send sync sync
    receive entity# endcycle
    send statussendentity# how
    receive statussendentity# STAT
    if STAT == ready
        send sendentity# message
    endif
endreplicate
```

This model simulates N entities and transfers data from each entity to the external partitions of the others. The support environment is modelled by a number of processes which pick up messages and pass them on for transmission to the other processes, provided the transmitting process is free. If the environment is unable to pass the message, it is dropped. A construct of interest is the mechanism (in process *statussendentity*) to identify a process that is unable to communicate. This problem is addressed in other models with the *sendifready* command.

Each entity process has a delay of X in its cycle, and since it does not block at any point, the cycle time in all cases is exactly X. A value of greater interest is the latency of the update messages. There are variations on the latency value possible with this model, due to the dropped packets. A message may reach its destination and the latency measured for that message, or it may be dropped and the latency measured from the start of the first attempt until a message does actually reach the destination. Alternatively, an average time for a message to be passed from the source entity to the *external* partition of the destination may be found.

The time taken for a message that successfully passes through the system is $X + C$, as may be expected. This value is independent of the number of processes in the model. To measure this value, blocks must be placed in the model to prevent exploration of the control paths to the *sync* and *statusentity* processes. In addition, blocks must be added to prevent searching of paths that might traverse more that one cycle of any particular process.

Cycles of the *environ* processes can be traversed while calculating latency. This produces the time between messages being sent and their eventual arrival at the destination process. These results give the range of latencies that may be experienced for a system of this nature. The case where $X > C$ yields a constant value of $X + C$ as the maximum latency, irrespective of the number of processes. The case where C dominates, where network communication is slow compared to the delay in the cycle of each process, increases the likelihood of messages being dropped. The maximum latency found in this case is $C + ((N-1)M + 1)X$, where N is the number of processes in the model and $C = MX$, for the cases where M is an integer. The average latency usually has half the dependency on X, having been averaged out over a number of cycles. The maximum latency for fractional values of M can be found by rounding up. The system does not reach this value for every message successfully transmitted, so the coefficient of X for the average latency is lower. The variation in this coefficient between integral values of M is illustrated in Figure 12.5

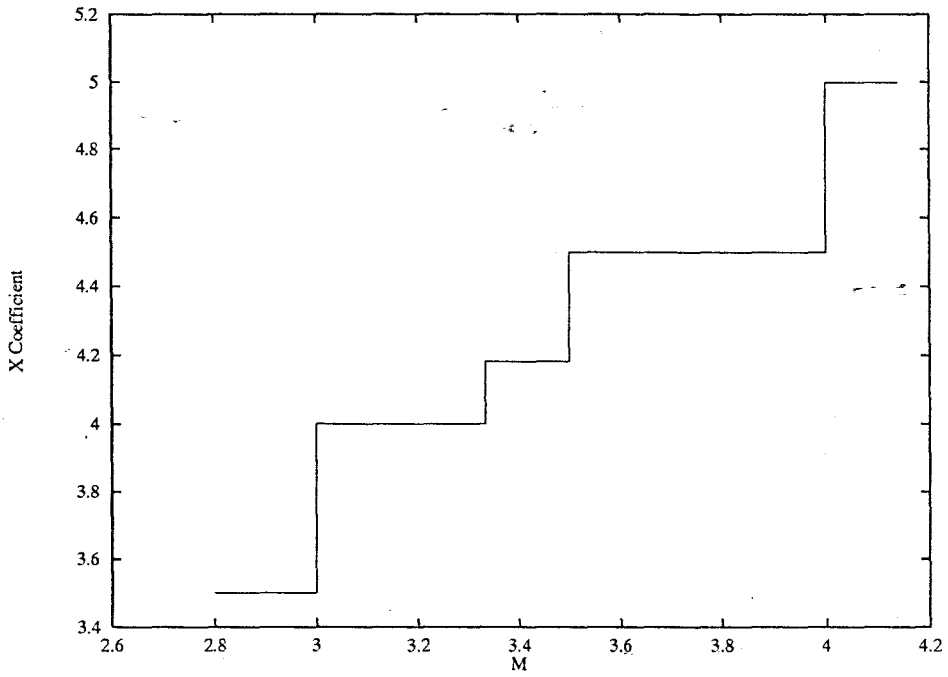

Figure 12.5: Coefficient of X for non-integral values of M

for the three processor model for values of M between 3 and 4. The stepped nature of this graph illustrates the sudden changes in performance values as the relative values of X and C change.

The peak and average latencies are thus dependent on the number of processes in the system when communication is slow.

12.3.10 Summary

The results obtained from the analyses in section 12.3 are summarized below. The performance values referred to are cycle time and latency within a single node. Usually these values are the same (see section 4.1), but the exceptions are explicitly noted.

Dead reckoning: Section 12.3.1.

Dead-reckoning uses infrequent updates and causes a decrease in the cycle time. It produces a linear dependence on the number of processors when using a shared communication medium, provided the number of processors is below a threshold level, determined by the communication time, and the interval between updates. Processing overhead in each node is increased due to the extra computation required.

Token passing: Section 12.3.2.

The effect of relocating the larger portion of the processing to the process carrying the token produces an improvement in the average cycle time, provided processors do not need to remain synchronized. Synchronous token passing can be used as an access control mechanism. Asynchronous token passing can be used as a distributed control mechanism for load balancing, and to improve interactive performance on a single

processor. The latter application benefits from the use of dead-reckoning to prevent saturation of the communication links. Movement of the token results in substantial performance degradation. Frequent movement of the token eliminates all benefits.

Broadcasting: Section 12.3.3.

Broadcasting using a distributed data model reduces communication time for a shared medium from an N^2 relationship to a linear one. Combined with dead-reckoning, communication costs are reduced still further, but introduce a dependence on N to the computation time in each node.

Multicasting: Section 12.3.4.

Multicasting uses the same principles as broadcasting, but allows selective communication between groups of processes. The effect of grouping strategies is better suited to statistical analysis. The effects of multicasting and broadcasting are examined when applied to a wide area network such as the Internet. This section introduces a fourth network model, in which bandwidth and propagation delays are independent. Using the measurement of message latency between two nodes in the network it is found that the effect of broadcasting/multicasting in a wide area network is dependent on the propagation delay, provided sufficient bandwidth is present. As bandwidth decreases, the latency increases substantially, particularly when messages are forced to queue throughout the network. If the computation time is large compared to network performance then the latency depends only on computation time.

Object servers: Section 12.3.5.

The cost of inter-processor communication does not decrease in proportion to the decrease in processors. The advantage of localized processes is offset by increased inter-processor communication.

Synchronous databases: Section 12.3.6.

Use of locks causes an N^2 dependence on the communication time (for a shared medium), even when combined with multicasting. This is due to the need for each process to acknowledge the lock.

Server computation: Section 12.3.7.

The results of the analysis indicate the presence of unstable transient behaviour with this model. The transient behaviour produces improved performance for the data-only clients without affecting that of the computation clients. The transient can be introduced into the steady state of the system through a simple alteration to the server. The change affects the priority with which service is given to each category of client.

Multiple servers: Section 12.3.8.

The coefficient of the communication component of the cycle time for a multiple server system over Ethernet is proportional to both the number of servers and the number of clients per server for large communication times. For large server times, the cycle time is dominated by the server time scaled by the number per server. For large client

computation times, there is no dependence on the numbers of clients and servers. For constant client, server and communication times, the effect of communication eventually dominates as numbers of clients and servers increase. In all cases the dependence on numbers of clients and servers is at most linear.

Asynchronous databases: Section 12.3.9.

The latency associated with messages between processes experiences some variation when the communication time dominates over the processing time in a node. Discontinuities occur in the value of the latency, suggesting the possibility of sudden changes in the performance of a system using this approach.

12.4 Conclusion

Analysis of the data and control distribution techniques identifies the dependencies on system characteristics for a range of architectures. These architectures represent the type of platforms upon which virtual reality systems are most likely to be constructed; a well connected parallel machine, a switched hub based network (star) and a shared medium such as Ethernet. Other decomposition techniques intended for specific networks are examined separately.

The two popular strategies mentioned in Chapter 2 are the centralized approaches for their simplicity of implementation, and the replicated approaches for their scalability. The results given in this section show that centralized client-server approaches scale well on all network architectures, exhibiting linear increase in cycle time/latency as the number of processes increases. The totally distributed architectures perform better with a well connected network, but have an N^2 dependency on communication on shared media. The use of broadcasting/multicasting reduces this to a linear dependency. Dead-reckoning reduces this further, but with an increase in local processor time. Dropping messages when communication load is too high can produce a constant dependency on communication time.

The factors that cannot be evaluated are the reliability of a client-server system which has a single point of failure in the server, and the quality of a large distributed simulation in which positions are estimated.

This chapter contains details of the analysis of the most common parallel decomposition strategies found in virtual reality systems, and the most frequently used variations on these. While a substantial number of strategies are described, they form only an overview of the possible configurations that are possible. An additional significant contribution of this chapter is the description of the analysis strategies used in each case, which provides an insight into the application of Analytical Simulation. This will assist the designer of a distributed virtual reality system when performing analysis of refinements of the models presented in this chapter.

Chapter 13

Analysis of complete virtual reality systems

This chapter examines the combination of the components of virtual reality systems that were modelled in Chapters 10 to 12. Different arrangements of the components and various interconnection strategies are examined. Latencies and cycle times in this chapter are representative of a complete system and provide an indication of the performance as seen by an external observer, or a person using the system.

13.1 Asymmetric rendering

This section explores two problems. Firstly, it investigates the use of the *sendifready* construct to model the constant availability of up-to-date data. Previous use of a create-and-send sequence produces synchronization delay when the send has to block (see Chapter 11). In some cases, it is desirable to model a system where the data is created immediately before the send succeeds. This can be implemented using extra communication or additional processes which complicates the model (see section 12.3.9).

The second effect that is examined is the use of the asymmetric rendering construct, as found in the MR Toolkit. Many applications in this system consist of a master process performing the computation and the rendering for one of the two views required for stereoscopic viewing. A slave process is used to produce the other view. A number of device servers provide data from the input devices. This section compares the effect of this extra effort in the master process to that resulting from the use of an additional slave process.

The device servers in the MR Toolkit continuously sample the device [Gre95] and make only the most recent values available. This is needed to minimize latency. This trick is common to many buffered and pipelined systems where it is possible to discard values should the rate of the incoming data exceed that of the outgoing information.

A model of this system, with instant input and asymmetric rendering, is shown below:

Variable	Interpretation
L	Maximum number of successive failures of the *sendifready* statement
R	Time spent on rendering per cycle
X	Time between data produced by the input device
Y	Time spent running the application per cycle

```
generate end -
generate end - start

process device

    report start
    sendifready master devicedata
    //send master devicedata
    think X

process master

    receive device devicedata
    think Y
    send slave picture
    think R
    receive slave display
    report end

process slave

    receive master picture
    think R
    send master display
```

The most interesting aspect of this model is in the device process where use is made of the *sendifready* statement. This command is intended to send a message only if the receiving process is waiting for it. This requires a non deterministic decision in that the state of the receiving process must be known to the sending process before the decision of whether to allow any communication between them can be made. Simulation of these semantics also raises issues of some complexity. Instead, it is implemented as a choice, being able to both fail and succeed. The analysis tool models both paths. The number of consecutive failures is limited to make analysis possible. The variable L represents the limit on consecutive failures. A variable X is included to prevent the device from cycling instantaneously.

Analysis of this model gives the following results:

$X \geq Y + R$:

Minimum cycle time: X

Minimum latency: $Y + R$

$(L+1)X \geq Y + R \geq X$:

Minimum cycle time: $Y + R$

Minimum latency: $Y + R$

$Y + R \geq (L+1)X$:

Minimum cycle time: $Y + R$

Minimum latency: $2Y + 2R - (L+1)X$

The case where $L = 0$, or the normal send is used (shown commented out in the model above), gives a latency of $2Y + 2R - X$ for $Y + R \geq X$. The case with data being dropped in the manner explained above would be where the *sendifready* statement is working as required, i.e. when $L \to \infty$ and $X \to 0$. If the second of the three constraints above holds, this gives a latency of $Y + R$ for the case $Y + R \geq X$. Otherwise the third constraint applies, and the minimum latency occurs if $(L+1)X \to Y + R$, again giving a latency of $Y + R$.

The model above can be compared to that given below, which uses three processes for the computation. A single master still processes for period Y, before passing the results to two slave processes for rendering. In this case there is no need to synchronize with the master at the completion of rendering.

Variable	Interpretation
L	Maximum number of successive failures of the *sendifready* statement
R	Time spent on rendering per cycle
X	Time between data produced by the input device
Y	Time spent running the application per cycle

```
process device

    report start
    sendifready master devicedata
    //send master devicedata
    think X
```

```
process master
    receive device devicedata
    think Y
    replicate 2
        send slave# picture
    endreplicate

replicate 2
    process slave#
        receive master picture
        think R
        report end#
endreplicate
```

The results for this model are given below, where it is assumed that both R and Y are greater than $(L+1)X$.

$R \geq Y$:

Cycle time:	R
Latency:	$2R + (R - (L+1)X)$

$Y \geq R$:

Cycle time:	Y
Latency:	$R + Y + (Y - (L+1)X)$

Minimum latencies occur in each case if $(L+1)X \rightarrow max(R,Y)$. The minimum latency achieved is:

$R \geq Y$:	$2R$
$Y \geq R$:	$R + Y$

The addition of the extra rendering process decreases cycle time, but only increases latency if the rendering time is greater than the time for running the application in each cycle. The effects of concurrent world modelling and rendering are examined in greater detail in section 13.2.

The usefulness of the *sendifready* statement is limited. While it does provide the desired functionality, it is difficult to use. It requires the addition of two extra variables into the system (L and X) which then have to be eliminated from the results. Any simplicity gained in the form of the model is lost in the added complexity of analysis.

13.2 Slave renderers

A system used to drive a pair of slave renderers is described in [Wat94], where two slave machines are used for rendering. These are controlled by a third master machine which also runs a solid modelling application. Input hardware is attached to the master machine. The two slave machines are running VR-386 and each renders the view for one of the eyepoints to create the stereoscopic view for a Head Mounted Display.

The virtual reality system running on this configuration must receive input from the input device (a glove), use this to make changes to the objects in the world, and transmit the updates to the two renderers. Once both renderers have completed their task, they can display the images on receipt of a synchronizing message from the master. Some latency and cycle time analyses have been performed for this system. Two variations on the sequence of events which introduce different tradeoffs in the two performance measures are described. The analysis is repeated here, using the automated approach developed in this thesis and some extensions are examined.

Variable	Interpretation
C	Time required to communicate a message between processors
N	Number of slave renderers in the system
X	Time spent running the application per cycle
Y	Time spent on rendering per cycle

```
process glove

    report source
    send master data

process master

    receive glove data
    think X
    replicate N
        send slave# draw C
    endreplicate
    replicate N
        receive slave# done
    endreplicate
    sendnotrace broadcast update C
    report cyclepoint
```

```
process broadcast
    receive master update
    replicate N
        send slave# show
    endreplicate

replicate N
    process slave#
        receive master draw
        think Y
        send master done
        receive broadcast show
        report show#
endreplicate
```

The above model, described as the linear approach, assumes the presence of N slave processes, and implements the sequence of events described previously. The model assumes that the master process can only communicate with one other process at a time, making the model applicable to both a shared communication medium such as Ethernet, or a switched hub topology. The current model assumes that each slave receives its own specific update of the state of the world, otherwise this could be implemented with the broadcast mechanism that is used to synchronize the output of the slaves.

This approach requires that processing on the master be completed before rendering on the slave can begin. An alternative approach is to run both in parallel, with the disadvantage that the view being rendered is out of date as soon as the master process finishes its calculations, but can only be changed on the next cycle. Thus a longer latency can be expected for this model. The change to the master process for the pipeline approach is shown below:

Variable	Interpretation
C	Time required to communicate a message between processors
N	Number of slave renderers in the system
X	Time spent running the application per cycle
Y	Time spent on rendering per cycle

```
process master
    replicate N
        send slave# draw C
    endreplicate
    receive glove data
    think X
```

```
replicate N
    receive slave# done
endreplicate
sendnotrace broadcast update C
report cyclepoint
```

The results for these approaches are:

Linear approach:

$$\text{Cycle time} = (N+1)C + X + Y$$
$$\text{Latency} = 2[(N+1)C + X + Y]$$

Pipeline approach:

$X \geq Y$:

$$\text{Cycle time} = (N+1)C + X$$
$$\text{Latency} = (2N+3)C + 3X$$

$Y \geq X$:

$$\text{Cycle time} = (N+1)C + Y$$
$$\text{Latency} = (2N+3)C + 3Y$$

As expected, the latency is higher for the second approach, but the cycle time has decreased. The latency is approximately twice the cycle time for both approaches, an effect that can be explained by viewing the glove and master processes as a short pipeline. If the glove process is allowed to collect data at its own pace and is only polled for the latest value when needed, the results show a substantial improvement. The relevant parts of the affected processes in the model are shown below:

Variable	Interpretation
C	Time required to communicate a message between processors
K	Time required to poll the glove device
N	Number of slave renderers in the system
X	Time spent running the application per cycle
Y	Time spent on rendering per cycle

```
process glove
    receive master givedata
    report source
    send master data
```

```
process master
    ...
    send glove givedata K
    receive glove data
    ...
```

The time required to poll the glove process is represented by the variable K. The revised results are now:

Linear approach:

$$\text{Cycle time} = (N+1)C + X + Y + K$$
$$\text{Latency} = (N+1)C + X + Y$$

Pipeline approach:

$X \geq Y$:

$$\text{Cycle time} = (N+1)C + X + K$$
$$\text{Latency} = (N+2)C + 2X$$

$Y \geq X$:

$$\text{Cycle time} = (N+1)C + Y$$
$$\text{Latency} = (N+2)C + 2Y - K$$

The presence of a negative coefficient is a welcome sight in the last case. It indicates a situation in which a delay can be increased to improve performance. The increase only improves matters for the pipeline approach with Y the dominant variable; in the other cases it causes an increase in the cycle time.

The final system implemented in [Wat94] was a mixture of the two approaches. The tradeoff between the two approaches depends on the loads on the master and slaves; these could vary depending on the nature of the solid modelling being performed. The system would monitor its performance variables and switch from one strategy to the other when doing so would result in performance improvement.

The performance values derived for this model agree with those quoted in [Wat94], once the effect of using broadcasts for updating the world is taken into account. The dependence on the number of slave processes derived above shows that additional displays can be added to the system without loss of performance, provided communication costs are low. With a communication medium capable of supporting broadcast, the communication cost becomes constant as well.

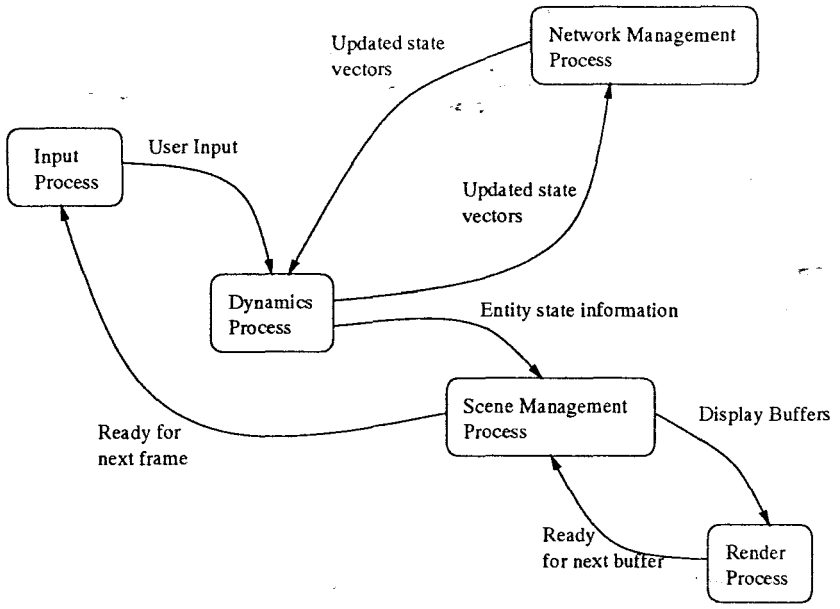

Figure 13.1: Data flow in an NPSNET node

13.3 Feedback between components

This section describes the Analytical Simulation of a model of a single node of a virtual reality system. It illustrates the power of the approach for providing simulation results which are applicable to all values of the variables in the system. This section examines the effects of feedback communication strategies between the components of the virtual reality system. The use of feedback is explored in pipelines in section 11.1.

The model examined is that of the software running on a single machine in the well known NPSNET virtual reality system [Pra93]. The relationships between the components of the system are illustrated in Figure 13.1. The interactive performance of the single node is of interest; interprocessor communication is modelled very coarsely.

The model of this system is given below:

Variable	Interpretation
C	Time between updates from all other nodes in the system
D	Time spent simulating dynamics per cycle
S	Time spent on scene management per cycle
R	Time spent on rendering per cycle

```
process input

    report input
    send dynamics data
    receive scene next
```

```
process network

    think C
    send dynamics data
    receive dynamics update

process dynamics

    receive input data
    receive network data
    think D
    send network update
    send scene state

process scene

    receive dynamics state
    think S
    receive render ready
    send render display
    send input next

process render

    send scene ready
    receive scene display
    think R
    report output
```

The variables D, S and R represent the time spent simulating the dynamics, managing the scene, and rendering each frame of graphical output, respectively. The variable C is the time between updates from the other nodes in the network. The two values of interest in a virtual reality system are the latency and the cycle time. The cycle time for this node is the interval between recurrences of the *output* report marker, the latency is the time taken for data created at the point indicated by the *input* report marker to reach the *output*.

The results for the Analytical Simulation of this model are as follows:

$S \geq D$, $D + S \geq R$:

Cycle time =	$D + S$
Latency =	$R + S + D$

$S \geq C$, $R \geq D + S$:

Cycle time =	R
Latency =	$2R$

$C \geq S, D + C \geq R$:

$$\begin{aligned} \text{Cycle time} &= D + C \\ \text{Latency} &= R + C + D \end{aligned}$$

$C \geq S, R \geq D + C$:

$$\begin{aligned} \text{Cycle time} &= R \\ \text{Latency} &= 2R \end{aligned}$$

The results are closely related to those expected for a pipeline construction. It is worth noting the use of feedback messages within this model to enable output from the input stage, and to permit transfer of data from the scene management process to the renderer. This construction is used with the pipelines in section 11.1 to reduce latency. The performance without this feedback is given by:

$S \geq D + C, S \geq R$:

$$\begin{aligned} \text{Cycle time} &= S \\ \text{Latency} &= R + 3S \end{aligned}$$

$S \geq D + C, R \geq S$:

$$\begin{aligned} \text{Cycle time} &= R \\ \text{Latency} &= 4R \end{aligned}$$

$D + C \geq S, D + C \geq R$:

$$\begin{aligned} \text{Cycle time} &= D + C \\ \text{Latency} &= R + S + 2C + 2D \end{aligned}$$

$D + C \geq S, R \geq D + C$:

$$\begin{aligned} \text{Cycle time} &= R \\ \text{Latency} &= 4R \end{aligned}$$

The latency with this version is substantially greater than that for the original.

The original model uses feedback between successive stages. The use of only a single feedback message, from the end of the output stage, all the way back to the input stage produces the following result:

$S + R \geq C$:

$$\begin{aligned} \text{Cycle time} &= D + S + R \\ \text{Latency} &= D + S + R \end{aligned}$$

$C \geq S + R$:

$$\text{Cycle time} = C + D$$
$$\text{Latency} = C + D$$

While the latency is smaller than that for the version without feedback, the cycle time has increased. This system is not making effective use of the parallel processors, and could probably be implemented on a single processor with similar results (provided network management and dynamics could overlap).

The use of feedback within pipelined structures is again demonstrated in this section when it is applied to pipelines formed from the components of virtual reality systems. Latency is substantially reduced when using the feedback mechanism, without affecting the cycle time.

13.4 Fast interaction

The traditional model of a virtual reality system requires that input be processed through the world simulation component, before its effect can be realized in the output. Much of the input to a virtual reality system is comprised of changes to the users position and orientation. This affects only the camera position, and the attitude of the user's avatar (a term often used to refer to the user's representation in the virtual world). This information would only affect the view of the scene that is rendered. Thus the input data could be fed directly into the output renderer, for rapid incorporation into the view, and only then into the simulation engine.

This approach is used in virtual reality systems developed at the University of Virginia [Gos93] [Pau95]. The separation of the rendering rate from the simulation speed is examined in the model below:

Variable	Interpretation
M	Relative amount of time spent rendering
N	Relative amount of time spent in the application
Y	Smallest quantum time interval in the model

```
process input

    report input
    send output data

process application

    send apppoll getlatest
    receive apppoll latest
```

```
        report inapp
        replicate N
            think Y
        endreplicate
        send apppoll latest

    process apppoll
        receive ANY TYPE
        if TYPE == getlatest
            send [ANY] latest
        endif

    process output
        receive input data
        send apppoll getlatest
        receive apppoll latest
        replicate M
            think Y
        endreplicate
        report picture
```

Input data is generated and passed directly to the output process. The output process polls a third process for data shared by the application which would also be required for rendering. The application, performing the simulation, also polls this process for the latest input values. This model makes no assumptions about the location of the data. It could be located with the simulation engine, as is modelled previously (in Chapter 12), or it could reside with the output process to improve access for the rendering engine. It could also be located on its own processor. The change required to the model to implement one of these strategies would be to assign a communication value to messages to and from the *apppoll* process when off-processor communication occurs.

The value of interest at present is the performance of the system with the direct connection between input and output. The model assumes that rendering takes period MY, and simulation takes period NY. The reason for this choice is explained below. As may be expected, the cycle time of the output process is MY, and the latency from input to output is $2MY$. The factor of two in the latency results from the input process blocking waiting for the output, a pipeline phenomenon described in detail in Chapter 11.

A more interesting value is the time taken for the effect of input to be incorporated in the simulation and show up on the output. A latency value is required for messages from input to output that pass through the application process. This is measured by adding an extra report marker to the application process, and measuring lengths of the two intervals from input to application and from application to output.

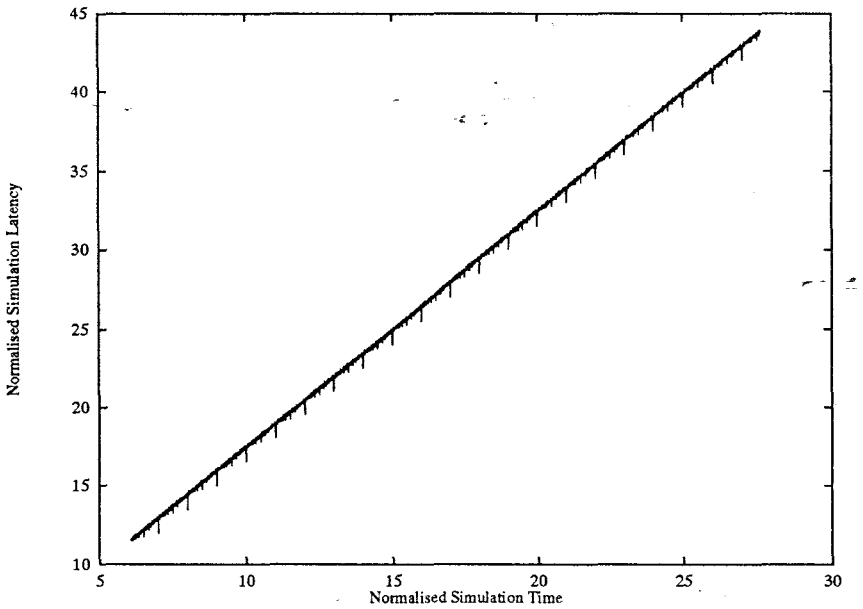

Figure 13.2: Simulation latency for large simulation times

This model has the interesting characteristic that it is not periodic if there is any variation possible in the computation times for the simulation and rendering processes. Thus state space analysis using two independent variables to model these times would fail because the constraint generation always allows some variation in the relative values of the variables. Constraint generation also produces an infinite number of constraints in an attempt to lock variables down into exact values. This behaviour results from the polling action of the application and output processes which do not have to synchronize with each other at any time.

This model can be converted into a form suitable for analysis by reducing the number of independent variables, as shown in the model above. This allows solution for any rational ratio of the two processing times. The sequential processing time in both is written as integral multiples of a common variable Y. Thus, given a simulation time of NY and a rendering time of MY, the time taken for input to pass through the simulator and affect the output is $[2.5M + 1.5N - HCF(M,N)]Y$. This function is shown graphically for a range of simulation times in Figures 13.2 and 13.3, assuming unit rendering time. This result is interesting in its seemingly chaotic behaviour. Small changes in the ratio of simulation and rendering times can produce large changes in the latency. This effect is averaged out in reality since these values are unlikely to be perfectly constant.

13.5 Combinations of components

The analysis of virtual reality systems examines the various parallel decompositions in a bottom-up manner. Initially, the performance characteristics of the three sub-components of a virtual reality system are examined, followed by an investigation into some of the effects resulting from complete systems. The advantage of doing things this way is that complexity is reduced, and the

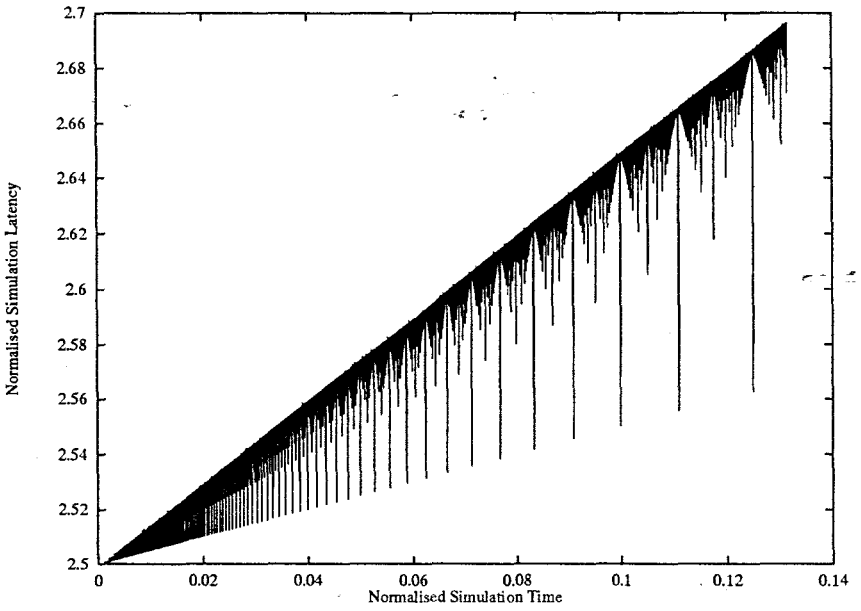

Figure 13.3: Simulation latency for large rendering times

effect of one approach is not shadowed by another.

A disadvantage is that the performance of a complete system with all its complexity cannot be examined without repeating the work that has already been performed on each subcomponent. It is desirable to have the ability to create components with known performance characteristics which can then be combined to represent a complete system. Analysis of the combined model then uses the values calculated for the constituent modules to compute the overall performance of the system.

This "black box" can be created relatively easily should latency and cycle time be equal. The following process has a latency (= cycle time) of X.

```
process blackbox

    receive input message
    think X
    send output message
```

If latency and cycle time differ then this model is not able to produce both metrics simultaneously. A more complicated piece of code can be created to overcome this limitation. Consider a system with latency L producing output after every period CT in its stable state. The following model has the same performance characteristics:

Variable	Interpretation
CT	Required cycle time of the model
L	Required latency of the model
$MAXOVERLAP$	Maximum number of messages that may be buffered

```
process blackbox
    init
        assign DUMP 0
    endinit
    send input ready
    receive input data
    send bblatency[DUMP] data
    assign DUMP [(DUMP+1)%MAXOVERLAP]
    think CT
    report notrace

replicate MAXOVERLAP
    process bblatency[#-1]
        receive blackbox data
        think L
        send bboutput data
        report notrace
endreplicate

process bboutput
    receive SOMETHING data
    send output data
    report notrace
```

An item of interest with this model is the use of the MAXOVERLAP variable which represents the number of messages that can be buffered by the system, and must be greater than the ratio $\frac{L}{CT}$ for the model to behave as required. The *blackbox* process actively requests data, to eliminate synchronization delays on the input.

There are some problems using a model such as this to represent a more complex system. Any transient behaviour is unlikely to be duplicated using a simplified model. Reproducing a system that accepts more than one input and produces more than one output requires modifications to the "black box" model. It has also not been proven that the two metrics (latency and cycle time) are sufficient to characterize a component of a system sufficiently to allow component-wise replacement as proposed, without affecting performance measurements in the remainder of the system.

Characterization of the cases where component substitution is possible, and identification of the metrics required to specify these components completely for performance analysis purposes, is the subject of future work.

13.6 Conclusion

The behaviour of complete virtual reality systems is described in this chapter.

The use of asymmetric rendering (section 13.1) produces lower latency when rendering times are high, than a model in which separate processors are used for the application and for rendering. Higher cycle times are associated with the asymmetric model. The *sendifready* statement used in the models in this section complicates the analysis.

The use of slave renderers (section 13.2) extends the analysis of separate application and rendering processes. This approach can be implemented in two different ways, which offer a trade-off between cycle time and latency. This example again illustrates the advantage of automatic analytical analysis over that which is performed manually and which is limited to particular models.

The effect of synchronization delay is again demonstrated in section 13.3 in pipelines formed by combining the components of the virtual reality systems. Feedback between successive stages is found to reduce latency without affecting the cycle times.

Immediate reduction in latency is obtained by routing input through the output stage, before processing it through the world modelling component (section 13.4). The latency for input data that is first processed by the application shows an almost chaotic dependence on the system parameters, although this effect is averaged out over time.

It is possible to construct a model with any desired cycle time and latency as shown in section 13.5. Additional formalism must be developed before a system can be completely characterized by a simplified model, for performance analysis purposes.

Parallel and distributed virtual reality systems are examined at two levels. The use of parallelism within the components of virtual reality systems is described in Chapters 10 to 12. Issues regarding the systems that can be built up from these components are discussed in this chapter.

Chapter 14

Conclusions

The original goal of this work was to provide a comparative study of the performance of components of parallel and distributed virtual reality systems, so as to allow appropriate decisions to be made when designing for a particular architecture. This has been successfully achieved. To accomplish this goal, this work has created a performance analysis technique that is ideal for application to virtual reality systems. It has also provided a reference as to the performance characteristics of the decomposition strategies found in virtual reality systems.

14.1 Analytical Simulation: Development

14.1.1 Summary of the results

Virtual Reality systems have particular requirements when considering the selection of performance metrics (see section 2.1). Existing performance analysis tools are inappropriate for determining the cycle time and latency of a range of system models in such a way that a comparison of the characteristics of each can be made (see Chapter 3). For comparison purposes, the availability of more than a single sample of these metrics is required.

The Analytical Simulation approach uses concepts derived from other tools to provide a powerful analysis technique, and includes several additional enhancements which make it suitable for performance comparison purposes. By producing output as an expression specifying the effects of each delay in the model, the results are representative of a range of the system parameters. In this form the contribution from each delay can be identified and compared to that for other models. Delays which affect the critical path are also identified in this way.

In addition, the Analytical Simulation approach automatically generates constraint regions which define the relative values of the system variables for which a given result holds. Distributions need not be specified in advance for each variable. Performance for all variable values within a region can be found by evaluating a single expression.

The performance measures produced using Analytical Simulation are cycle time and latency, the two measures essential for proper characterization of virtual reality systems.

The Analytical Simulation approach can be extended in a variety of ways, from conventional

simulation to state space analysis. The latter version is limited to periodic systems, but can provide a complete analysis of the model's performance, including any non-deterministic behaviour. All models of virtual reality systems examined are periodic, so the restriction is not a problem.

The accuracy of the approach is confirmed by comparing predicted performance with actual implementation for three parallel decomposition strategies on two parallel architectures. In all cases there is excellent correspondence between theory and practice. The form of the results allows the different strategies to be compared. In addition the effects of the different architectures on each strategy can be seen.

Analytical Simulation is well suited to the analysis of the performance of parallel and distributed virtual reality systems. It is also ideal for the comparison of different parallel decomposition strategies within these systems.

14.1.2 Contributions of the work

The following contributions are made in this portion of the thesis (Chapters 2 to 8):

- A taxonomy of the decomposition strategies used in parallel and distributed virtual reality systems is developed.
- The suitability of existing performance analysis techniques is assessed with regard to their use with virtual reality systems.
- A performance prediction approach, Analytical Simulation, is developed which is ideal for the performance analysis and comparison of parallel and distributed virtual reality systems.
- A method is devised for comparing the run times of processes, expressed as a symbolic expression, in the presence of constraints on the variables.
- Methods are described for extracting values for Cycle Time and Latency from simulation results, and from reachability graphs in the state space of a model.
- Analytical Simulation is applicable to a range of classes of systems in its different forms, and is capable of:
  - Measurement of latency and cycle time
  - Producing symbolic output, suitable for comparison purposes
  - Complete analysis for all values of the variables in the model
  - Analysis of non-deterministic constructs
  - Analysis of the transient behaviour of a model
- A number of distributed collision detection algorithms are developed. Analysis of the performance of these systems is described, and verified against implementations of each on two different architectures. Models for communication in Transputer clusters and the shared network medium, Ethernet, are created.

14.2 Analytical Simulation: Practice

14.2.1 Summary of the results

A natural decomposition of virtual reality systems is found to consist of three components: input, output and world modelling.

The effects of performing input using polled and interrupt-driven approaches are examined, and the performance implications of each approach are summarized. The use of input prediction is discussed, but this is not suited to quantitative analysis.

The output subsystems are based around the pipeline topology. Latency issues in pipelines are examined in detail, including the use of additional delays and feedback to reduce the effect of synchronization delays on the latency. The effect of buffering on the performance of the pipeline is investigated, and conditions are identified under which improvements in both latency and cycle time occur. Use of additional parallelism, both within the pipeline, and using replicated pipelines, is explored. The latter can substantially improve cycle time without detriment to latency, provided sufficient communication bandwidth is present. Case studies of output subsystems used in virtual reality systems provide details of the performance characteristics of these systems. Compared with a previous analysis of the PixelFlow system, the use of Analytical Simulation confirms the results and provides additional and more complete information about the cycle time and latency that can be expected for various operating conditions.

Analysis of different control and data distribution approaches shows that peer-to-peer communication performs best where well-connected communication networks are available. The performance of approaches that make use of central services scales better where communication is limited. Performance characteristics of a number of variations on these approaches are examined. The results show how these variations can improve the scalability of some approaches, and identify anomalies in the behaviour of others. The analysis also demonstrates methods of application of Analytical Simulation so as to produce the desired results rapidly.

Performance in a node of a complete virtual reality system is examined by creating models consisting of the components identified in Chapter 9, and modelled in Chapters 10, 11 and 12. These models demonstrate different interconnection strategies suggested by various virtual reality systems, and provide details of the tradeoffs in performance offered by each strategy.

14.2.2 Contributions of the work

The following contributions are made in this portion of the thesis (Chapters 9 to 13):

- The section provides a reference to the performance characteristics of parallel decomposition strategies in use in parallel and distributed virtual reality systems.
- It demonstrates that Analytical Simulation of message passing models is ideal for the analysis and comparison of parallel and distributed virtual reality systems.
- Techniques to make effective use of Analytical Simulation to extract the desired information from a model are described.

- Methods to reduce synchronization delays in pipeline constructs, such as feedback and delayed output are introduced and analysed.
- Analysis of the effects of buffering in pipelines proves that an improvement in both cycle time and latency can be obtained under the correct operating conditions.
- Performance trends for different decomposition strategies on a range of network topologies are compared.
- Models of four different network architectures: a well connected network, a star (switched hub) topology, a shared communication medium (Ethernet) and a wide-area network (such as the Internet) are developed and used in the analyses.
- Performance characteristics of specialized decomposition techniques on specific architectures are identified. Behaviour as number of processes increases is described, and optimal performance paths are identified.
- Tradeoffs between the two performance metrics, latency and cycle time, are described when considering different interconnection strategies that are possible using the components of virtual reality systems.
- The advantages of an automated analytical analysis technique are demonstrated with the greater range of metrics and improved detail that is achieved over previous analyses. Improved performance values are produced for a number of existing systems for which performance analysis has previously been attempted.

14.3 Assessment of this work

This section contains a few comments by the author on some of the aspects of the work which are not quantitative, and which cannot be isolated to any single preceding chapter. They reflect on the overall goal and attitude toward the analyses performed.

It is surprising to discover the complexity unveiled by the analysis of even small models of familiar systems. The behaviour of a few processes containing only a few communication constructs generates unexpected effects ranging from transients that can last indefinitely if the conditions are right, to tendencies to settle into stable states (often non-optimal) that are unintended in the design. It is disturbing to notice this behaviour occurring in a rigidly controlled simulation and to imagine its extrapolation to larger, more complex systems where only the overall effect can be observed. The possibility that these effects are occurring and being ascribed to other characteristics of a real system is a danger. Many of the cases examined use non-determinism, which often allows non-optimal performance. In some cases, such as that in section 12.3.7, optimal behaviour is an unstable transient which is only discovered after performance analysis using Analytical Simulation.

The analysis methodology is valuable for playing "what-if" games on the various models. It is easy to modify a model quickly to perform actions in a different order, or to impose additional restrictions to judge the change in behaviour. The analytical form of the output makes it simple to identify the variable determining the critical path, and to judge whether a particular path seems

reasonable for the expected behaviour of the model. Generating a trace of program behaviour provides a visual overview of the behaviour. This can reveal many things from its periodic nature alone. The processes that spend much of their time idle are identified, as are those which are continuously operating, and a probable bottleneck. Communication orderings that produce a shorter cycle can be identified at once, allowing the model to be modified to check if such an ordering represents a stable state.

The ability to try these alternatives rapidly makes it profitable to spend time at the design stage of a virtual reality system testing alternative decomposition strategies, so as not to experience unfortunate effects once a commitment is made to a particular approach.

Many of the models presented are simple and in no way approach the code complexity of a real virtual reality system (and nor should they). Much of the code of the systems examined during the course of this work is effectively irrelevant for performance analysis, consisting of conventional sequential code for performing specific computations. Replacing the most significant blocks by the *think* statement used in the models reduces these systems to much the same complexity as the models. The effective message passing calls are the same as those depicted in the models. More detail is only required when advanced flow control and buffering calls must be modelled. Successful analysis of such an instance is also feasible, as is illustrated in section 8.5.1.

The analysis technique developed in this document provides a valuable and practical tool for use during the design of parallel and distributed virtual reality systems. The analyses performed provide a comprehensive reference to the techniques commonly used in current systems. Design of future virtual reality systems will benefit from consideration of the issues uncovered by these analyses, and from analysis of additional variations and enhancements of these algorithms.

14.4 Future work

The long term goal of this work is to provide a solid foundation for the development of distributed virtual reality systems. Previous development of virtual reality systems has either been from scratch, or as in the case of some recent systems, based on a previous system developed at the same site. A limited amount of cross-pollination has occurred, influenced mainly by the profile of some of the projects. This can be seen by the increasing tendency of newer systems to overlap in their distribution techniques. While these techniques are sound, and can provide significant performance enhancement, they often overshadow approaches which may be more useful in other situations.

This work starts off by developing and testing a performance prediction approach that is capable of providing useful measures for virtual reality systems. The approach is simple enough to be quickly and easily used by a system developer, while sufficiently powerful to uncover significant behaviour patterns. It is able to provide predictions for an approach, based on the environment in which the system is expected to run. This document covers the analysis of parallel and distributed virtual reality systems, categorizing the different approaches and providing a performance analysis where appropriate. In this way, the attributes of the different approaches can be compared in a common and unbiased environment.

The next logical progression is the creation, testing and analysis of new approaches. This is

a major task in its own right, but one which requires a solid foundation. This foundation must provide both a mechanism for performing a comparative analysis, as well as an evaluation of existing approaches to prevent duplication of effort. It is this foundation that the work of this thesis is intended to provide.

This future task is part of the goal of the group developing the distributed virtual reality system RhoVeR [Ban96] (see also section 2.3.1). Part of the requirement when creating this system was to leave the distribution methods sufficiently modular so as to allow different approaches to be implemented, tested, compared and enhanced. The system provides a set of basic communication facilities and acts as a standard platform for benchmarking purposes.

The possible future directions for parallel and distributed virtual reality systems are numerous. The trends in system development are to expand the use of virtual reality across global networks. Already a number of multi-user virtual worlds have been created on the Internet, supporting thousands of users in vast and complex worlds. These are based around client-server systems which have evolved from the hypertext servers on the World Wide Web. A number of variations on the client-server paradigm are modelled in this thesis. These models can be refined as the nature of the communication in these systems becomes better understood. Increased use of peer-to-peer communication can be expected to reduce the load, and dependency, on a centralized server.

The changing nature of networking can also be expected to influence the nature of distributed virtual reality systems. The tradeoffs between higher speed networks and the requirements of realistic simulations can alter the performance issues of concern to designers of such systems. The algorithms underlying these systems can be expected to evolve to meet these changing requirements.

With the work described in this thesis as the foundation, it is now feasible to tackle this next set of exciting challenges.

Bibliography

[Aal94] Wil M.P. van der Aalst, "Using Interval Timed Coloured Petri Nets to Calculate Performance Bounds", *Proceedings of the 7th International Conference of Modelling Techniques and Tools for Computer Performance Evaluation*, G. Haring and G. Kotsis (eds), in *"Lecture Notes in Computer Science"*, Springer-Verlag, New York, Volume 794, 1994, 425-444.

[Abr94] Marc Abrams, Alan Batongbacal, Randy Ribler and Devendra Vazirani, "CHITRA94: A Tool to Dynamically Characterize Ensembles of Traces for Input Data Modeling and Output Analysis", Technical Report TR 94-21, Department of Computer Science, Virginia Polytechnic Institute and State University, June 1994.

[Adv94] Vikram S. Adve, Charles Koelbel and John M. Mellor-Crummey, "Performance Analysis of Data-Parallel Programs", Technical Report CRPC-TR94405, Center for Research on Parallel Computation, 1994.

[Ams95] Denis Amselem, "A Window on Shared Virtual Environments", *Presence: Teleoperators and Virtual Environments*, Volume 4, Number 2, Spring 1995, 130-145.

[And93a] Magnus Andersson, Christer Carlsson, Olof Hagsand and Olov Ståhl, "DIVE - The Distributed Interactive Virtual Environment, Tutorials and Installation Guide for DIVE version 2.2", Swedish Institute of Computer Science, Kista, Sweden, March 1993.

[And93b] Magnus Andersson, Christer Carlsson, Olof Hagsand and Olov Ståhl, "DIVE - The Distributed Interactive Virtual Environment, Technical Reference for DIVE version 2.2", Swedish Institute of Computer Science, Kista, Sweden, March 1993.

[And91] John B. Andrews and Constantine D. Polychronopoulos, "An Analytical Approach to Performance/Cost Modeling of Parallel Computers", Technical Report CSRD-1110, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1991.

[Azu95] Ronald Azuma and Gary Bishop, "A Frequency-Domain Analysis of Head-Motion Prediction", *Proceedings of SIGGRAPH '95 Conference*, Los Angeles, CA, August 1995.

[Bal91] G. Balbo, S.C. Bruell, P.Chen, G. Chiola, "An Example of Modelling and Evaluation of a Concurrent Program Using Coloured Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm", from "*High-Level Petri Nets*", editors K. Jensen and G. Rozenberg, Springer-Verlag, 1991.

[Ban93] Shaun Bangay, "Parallel Implementation of a Virtual Reality System on a Transputer Architecture", MSc Thesis, Department of Computer Science, Rhodes University, November 1993.

[Ban94a] Shaun Bangay, "Creating Virtual Reality Applications on a Parallel Architecture", available via URL: ftp://cs.ru.ac.za/www/vrsig/SDB03.ps.Z, March 1994.

[Ban94b] Shaun Bangay, Peter Clayton and David Sewry, "Application of Parallel Processing to Rendering in a Virtual Reality System", *Transputer Applications and Systems '94 - Proceedings of the 1994 World Transputer Congress*, Villa Erba, Cernobbio, Como, Italy, September 1994.

[Ban96] Shaun Bangay, James Gain, Greg Watkins, Kevan Watkins, "RhoVeR: Building the Second Generation of Parallel/Distributed Virtual Reality Systems", *Proceedings of the First Eurographics Workshop on Parallel Graphics and Visualization*, Bristol, UK, September 1996. Also to be published in *Parallel Computing*.

[Ben94] Steve Benford, John Bowers, Lennarte E. Fahlen, Chris Greenhalgh, "Managing Mutual Awareness in Collaborative Virtual Environments", *Proceedings of VRST '94*, Singapore, August 1994.

[Ber94] Joseph E. Berger, Loan T. Dinh, Michael F. Masiello and Jesse N. Schell, "NVR: A System for Networked Virtual Reality", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Ber91] Bernard Berthomieu and Michael Diaz, "Modeling and Verification of Time Dependent Systems using Time Petri Nets", *IEEE Transactions on Software Engineering*, Volume 17, Number 3, March 1991, 259 -273.

[Bir90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood, "The ISIS System Manual, Version 2.1", Cornell University, September 1990.

[Bri93] William Bricken and Geoffrey Coco, "The VEOS Project", HITL Technical Report TR-93-3, Human Interface Technology Laboratory, University of Washington, 1993.

[Bro95] Wolfgang Broll and David England, "Bringing Worlds Together: Adding Multi-User Support to VRML", *Proceedings of the Virtual Reality Modeling Language (VRML) '95 Symposium*, San Diego, December 1995.

[Car89] Nicholas Carriero and David Gerlernter, "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Computing Surveys*, Volume 21, Number 3, September 1989, 323-357.

[Coc92] Geoffrey Coco, "The VEOS project : VEOS 2.0 Tool Builders Manual", available via URL: ftp://milton.u.washington.edu/public/veos/veos.tar.Z, May 1992.

[Coc93] Geoffrey P. Coco and Dav Lion, "Experiences with Asynchronous Communication Models in VEOS, a Distributed Programming Facility for Uniprocessor LANs", Technical Report 93-2, Human Interface Technology Laboratory, University of Washington, 1993.

[Cod92] Christopher Codella, Reza Jalili, Lawrence Koved, J. Bryan Lewis, Daniel T. Ling, James S. Lipscomb, David A. Rabenhorst, Chu P. Wang, Alan Norton, Paula Sweeny, and Greg Turk, "Interactive simulation in a multi-person virtual world", *Proceedings of ACM Human Factors in Computing Systems (CHI'92) Conference*, May 1992, 329-334.

[Coh94] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, Madhav K. Ponamgi, "Interactive and Exact Collision Detection for Large-Scaled Environments", Technical Report TR94-005, Department of Computer Science, University of North Carolina, Chapel Hill, 1994.

[Cro93] Mark E. Crovella and Thomas J. LeBlanc, "The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation", Technical Report 479, Computer Science Department of University of Rochester, Rochester, New York, December 1993.

[Dem96] Kris Demunyck, Jan Broeckhove and Frans Arickx, "The impact of communication mechanisms on the performance in a distributed Virtual Reality system", in *"Lecture Notes in Computer Science"*, Volume 1067, Springer-Verlag, 1996.

[Div96] "Division On-Line Manuals", available via URL: http://www.division.co.uk/online_manual/division.html, May 1996.

[Fah93] Thomas Fahringer, "Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers", PhD Thesis, University of Vienna, Department of Software Technology and Parallel Systems, October 1993.

[Fen96] Clive Fencott, *"Formal Methods for Concurrency"*, International Thomson Computer Press, London, 1996.

[Gar94] Alejandro Garcia-Alonso, Nicolas Serrano and Jaun Flaquer, "Solving the Collision Detection Problem", *IEEE Computer Graphics and Applications*, Volume 14, Number 3, May 1994, 36-43.

[Ghe95] Steve Ghee, "Course Notes for the Programming Virtual Worlds course", presented at SIGGRAPH '95, Los Angeles, California, August 1995.

[Gib84] William Gibson, *"Neuromancer"*, Ace, New York, 1984.

[Gos93] Rich Gossweiler, Chris Long, Shuichi Koga and Randy Pausch, "DIVER: A Distributed Virtual Environment Research Platform", *Proceedings of the IEEE Symposium on Research Frontiers in Virtual Reality*, San Jose, CA, October 1993.

[Gos94] Rich Gossweiler, Robert J. Laferriere, Michael L. Keller, Randy Pausch, "An Introductory Tutorial for Developing Multi-User Virtual Environments", *Presence: Teleoperators and Virtual Environments*, Volume 3, Number 4, 1994, 255-264.

[Gra93] Robert Grant, Multiverse description and sources, available via URL: ftp://ftp.hitl.washington.edu/pub/scivw/multiverse/multiverse-1.0.2.tar.Z, April 1993.

[Gre92] Mark Green, "Minimal Reality Toolkit Version 1.2 : Programmer's Manual", Department of Computing Science, University of Alberta, Edmonton, Alberta, 1992.

[Gre95] Mark Green and Lloyd White, "Minimal Reality Toolkit Version 1.4 : Programmer's Manual", Department of Computing Science, University of Alberta, Edmonton, Alberta, 1995.

[Gre94] Chris Greenhalgh, "An Experimental Implementation of the Spatial Model", *Proceedings of the 6th ERCIM Workshop*, Swedish Institute of Computer Science, Stockholm, Sweden, June 1994.

[Hag95] Olof Hagsand, "SID2 Interface Specification", Swedish Institute of Computer Science, August 1995.

[Has93] George Hassapis, "High level Petri Net modelling and analysis of VME-based multiprocessors", *Microprocessing and Microprogramming*, Volume 36, Number 4, September 1993.

[Hon96] Yasuaki Honda, Kouichi Matsuda, Jun Rekimoto and Rodger Lea, "Virtual Society: Extending the WWW to support a Multi-user Interactive Shared 3D Environment", *Proceedings of the Virtual Reality Modeling Language (VRML) Symposium '96*, New York, NY, 1996, 109-116.

[Hub93] Philip M. Hubbard, "Interactive Collision Detection", *Proceedings of the IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[Hub95a] Philip M. Hubbard, "Collision Detection for Interactive Graphics", PhD Thesis, Department of Computer Science, Brown University, March 1995.

[Hub93b] Roger Hubbold, Alan Murta, Adrian West and Toby Howard, "Design Issues for Virtual Reality Systems", *Proceedings of the First Eurographics Workshop on Virtual Environments*, Barcelona, September 1993.

[Hug95] Charles E. Hughes, J. Michael Moshell, "Shared Virtual Worlds for Education: The ExploreNet Experiment", available via URL: http://www.cs.ucf.edu/~ExploreNet/papers/VA.Experiment1195.html, November 1995.

[Ibe93] Oliver C. Ibe, Hoon Choi and Kishor S. Trivedi, "Performance Evaluation of Client-Server Systems", *IEEE Transactions on Parallel and Distributed Systems*, Volume 4, Number 11, November 1993.

[Kal93] Roy S. Kalawsky, *"The Science of Virtual Reality and Virtual Environments"*, Addison-Wesley, Wokingham, 1993.

[Kan92] Krishna Kant, *"Introduction to Computer System Performance Evaluation"*, McGraw-Hill, New York, 1992.

[Kav86] Krishna M. Kavi, Bill P. Buckles and U. Narayan Bhat, "A Formal Definition of Data Flow Graph Models", *IEEE Transactions on Computers*, Volume C-35, Number 11, November 1986.

[Kaz93] Rick Kazman, "Making Waves: On the Design of Architectures for Low-end Distributed Virtual Environments", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1993 (VRAIS '93)*, Seattle, Washington, September 1993.

[Koh75] Walter H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems", *IEEE Transactions on Computers*, Volume C-25, December 1975, 1235-1238.

[Kol80] Bernard Kolman and Robert E. Beck, *"Elementary Linear Programming with Applications"*, Academic Press, New York, 1980.

[Lee94] Insup Lee and Patrice Bremond-Gregoire, "A Process Algebra Approach to the Specification and Analysis of Resource-Bound Real-Time Systems", *Proceedings of the IEEE, Special Issue on Real-Time Systems*, January 1994.

[Lei96] Jason Leigh, Andrew E. Johnson, Christina A. Vasilakis, Thomas A. DeFanti, "Multi-perspective Collaborative Design in Persistent Networked Virtual Environments", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1996 (VRAIS '96)*, Santa Clara, California, March 1996.

[Li95] Zhiyong Li, Peter H. Mills and John H. Reif, "Models and Resource Metrics for Parallel and Distributed Computation", *Proceedings of the 28th Annual Hawaii Conference on System Sciences (HICSS-28 Parallel Algorithms Software Technology Track)*, Wailea, Maui, Hawaii, January 1995.

[Loc93] John Locke, "An Introduction to the Internet Networking Environment and SIMNET/DIS", Computer Science Department, Naval Postgraduate School, available via URL: ftp://sunee.uwaterloo.ca/pub/vr/documents/DISIntro.ps, February 1993.

[Mac95a] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, Paul T. Barham, "Exploiting Reality with Multicast groups: A Network Architecture for Large-Scale Virtual Environments", *IEEE Computer Graphics and Applications*, Volume 15, Number 5, September 1995, 38-45

[Mac95b] Michael R. Macedonia, Michael J. Zyda, "A Taxonomy for Networked Virtual Environments", *Proceedings of the 1995 Workshop on Networked Realities*, Boston, MA, October 1995.

[Mac95c] Michael R. Macedonia, Donald P. Brutzman, Michael J. Zyda, David R. Pratt, Paul T. Barham, John Falby, John Locke, "NPSNET: A Multi-Player 3D Virtual Environment over the Internet", *Proceedings of the ACM 1995 Symposium on Interactive 3D Graphics*, Monterey, California, April 1995.

[Mak90] Victor M. Mak and Stephen F. Lundstrom, "Predicting Performance of Parallel Computations", *IEEE Transactions on Parallel and Distributed Systems*, Volume 1, Number 3, July 1990.

[Mal94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, "Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers", *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, Cancun, Mexico, April 1994.

[Man95] Jon Mandeville, Thomas Furness III, Masahiro Kawahata, Dace Campbell, Paul Danset, Austin Dahl, Jens Dauner, Jim Davidson, Jon Howell, Kigen Kandie and Paul Schwartz, "GreenSpace: Creating a Distributed Virtual Environment for Global Applications", *Proceedings of the 1995 Workshop on Networked Realities*, Boston, MA, October 1995.

[Mas95] Thomas W. Mastaglio and Robert Callahan, "A Large-Scale Complex Virtual Environment for Team Training", *Computer*, Volume 28, Number 7, July 1995.

[Men93] Celso L. Mendes, "Performance Prediction by Trace Transformation", *Proceedings of the Fifth Brazilian Symposium on Computer Architecture*, Florianopolis, Brazil, September 1993.

[Men94] Celso L. Mendes and Daniel A. Reed, "Performance Stability and Prediction", *Proceedings of the IEEE Workshop on High Performance Computing (WHPC'94)*, March 1994.

[Men95] Celso L. Mendes, Jhy-Chun Wang and Daniel A. Reed, "Automatic Performance Prediction and Scalability Analysis for Data Parallel Programs", *Proceedings of the CRPC/Rice Workshop on Automatic Data Layout and Performance Prediction*, Houston, April 1995.

[Mil89] R. Milner, *"Communication and Concurrency"*, Prentice-Hall, Hemel Hempstead, 1989.

[Mol89] Faron Moller and Chris Tofts, "A Temporal Calculus of Communicating Systems", Technical Report ECS-LFCS-89-104, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, December 1989.

[Mol92] Steven Molnar, John Eyles and John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH '92*, Chicago, Illinois, July 1992, 231-240.

[Mor95] John Morrison, "The VR-Link Networked Virtual Environment Software Infrastructure", *Presence: Teleoperators and Virtual Environments*, Spring 1995.

[Mur94] Praveen Murthy, "On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules", Memo. No. UCB/ERL M94/46, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, June 1994.

[Nyg94] Erik Nygren, "VETTNet Design and Use", available via URL: http://mimsy.mit.edu/VETTnet/vettnet.html, May 1994.

[OCo95] Karl O'Connell, Vinny Cahill, Andrew Condon, Stephen McGerty, Gradimir Starovic and Brendan Tangney, "The VOID Shell: A Toolkit for the Development of Distributed Video Games and Virtual Worlds", *Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*, July 1995.

[Pan95] Igor Sunday Pandzic, Tolga K. Capin, Nadia Magnenat Thalman and Daniel Thalman, "VLNET: A Networked Multimedia 3D Environment with Virtual Humans", *Proceedings of Multi-Media Modeling (MMM '95)*, Singapore, 1995.

[Par92] Mark Parris, Carl Mueller, Jan Prins, Adam Duggan, Quan Zhou, Erik Erikson, "A Distributed Implementation of an N-body Virtual World Simulation", Technical Report TR92-020, Department of Computer Science, University of North Carolina, Chapel Hill, 1992.

[Pau95] Randy Pausch, Tommy Burnette, A.C. Capehart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga and Jeff White, "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality", *IEEE Computer Graphics and Applications*, Volume 15, Number 3, May 1995, 8-11.

[Pes94] Mark D. Pesce, Peter Kennard and Anthony S. Parisi, "Cyberspace", available via URL: http://vrml.wired.com/concepts/pesce-www.html, June 1994.

[Pig95] Laurent Piguet, Terry Fong, Butler Hine, Phil Hontalas, and Erik Nygren, "VEVI: A Virtual Reality Tool For Robotic Planetary Exploration", *Proceedings of Virtual Reality World '95*, Munich, Germany, 1995, 263-274.

[Pig96] Laurent Piguet, Butler Hine, Philipp Hontalas, Terrence Fong and Erik Nygren, "The Virtual Environment Vehicle Interface: A Dynamic, Distributed And Flexible Virtual Environment", available via URL ftp://artemis.arc.nasa.gov/papers/imagina.paper.pdf, 1996.

[Pin96] Alexander del Pino, "MPSC - A Model of Distributed Virtual Environments", *Proceedings of the 3rd Eurographics Workshop on Virtual Environments*, Monte Carlo, Monaco, February 1996.

[Pol93] "3SPACE InsideTrak User's Manual", Polhemus Incorporated, Colchester, Vermont, USA, December 1993.

[Pou91] Dick Pountain, "ProVision: The Packaging of Virtual Reality", *Byte*, Volume 16, Number 10, October 1991.

[Pra93] David R. Pratt, "A Software Architecture for the Construction and Management of Real-Time Virtual Worlds", PhD Thesis, Naval Postgraduate School, Monterey, California, June 1993.

[Reg92] Matthew Regan and Ronald Pose, "A Low Latency Virtual Reality Display System", Technical Report 92/166, Department of Computer Science, Monash University, Monash, 1992.

[Reg93] Matthew Regan and Ronald Pose, "An interactive graphics display architecture", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1993 (VRAIS '93)*, Seattle, 1993.

[Roe95] Bernie Roehl, "Distributed Virtual Reality - An Overview", available via URL: http://sunee.uwaterloo.ca/~broehl/distrib.html, June 1995.

[Sar94] Sekhar R. Sarukkai, Jerry Yan and Jacob K. Gotwals, "Normalized Performance Indices for Message Passing Parallel Programs", *Proceedings of the International Supercomputing Conference*, Manchester, England, June 1994.

[Sch96] Dieter Schmalstieg, Michael Gervautz, Peter Stieglecker, "Optimizing Communication in Distributed Virtual Environments by Specialized Protocols", *Proceedings of the 3rd Eurographics Workshop on Virtual Environments* (ed M. Goebel), Monte Carlo, Monaco, February 1996.

[Seg94] Mark Segal, Kurt Akeley and Chris Frazier, "The OpenGL Graphics System: A Specification (Version 1.0)", Technical Report, Silicon Graphics Inc., Mountain View, CA, July 1994.

[Sha93] Chris Shaw and Mark Green, "The MR Toolkit Peers Package and Experiment", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1993 (VRAIS '93)*, Seattle, 1993.

[Sha95] Chris Shaw, "The MR Toolkit Peer Package", Department of Computing Science, University of Alberta, 1995.

[Sin95] Gurminder Singh, Luis Serra, Willie Png, Audrey Worg and Hern Ng, "BrickNet: Sharing Object Behaviours on the Net", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1995 (VRAIS '95)*, Research Triangle Park, North Carolina, March 1995.

[Sin91] Mukesh Singhal and Thomas L. Casavant, "Distributed Computing Systems", *Computer*, Volume 24, Number 8, August 1991, 12-15.

[Sin96] Sandeep K. Singhal, "Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments", PhD Thesis, Department of Computer Science, Stanford University, August 1996.

[Sno92] Michael Snoswell, "Overview of Cyberterm, a Cyberspace Protocol Implementation", available via URL ftp://sunsite.unc.edu/pub/academic/computer-science/virtual-reality/papers/Snoswell.Cyberterm, July 1992.

[Sno93] David N. Snowdon, Adrian J. West and Toby L.J. Howard, "Towards the next generation of Human-Computer Interface", *Proceedings of Informatique '93: Interface to Real and Virtual Worlds*, March 1993, 399-408.

[Sno94a] Michael Snoswell, "Documents on Cyberterm", available via URL: ftp://ftp.adelaide.edu.au/pub/cybertem/ctdocs.zip, April 1994.

[Sno94b] David N. Snowdon and Adrian West, "The AVIARY Distributed Virtual Environment", *Proceedings of the 2nd UK VR-SIG Conference*, Theale, Reading, UK, December 1994, 39-54.

[Som93] Sukhamoy Som, Roland R. Mielke and John W. Stoughton, "Prediction of Performance and Processor Requirements in Real-Time Data Flow Architectures", *IEEE Transactions on Parallel and Distributed Systems*, Volume 4, Number 11, November 1993.

[Sri95] Sudhir Srinivasan and Bronis R. de Supinski, "Multicasting in DIS: A Unified Solution", Technical Report CS-95-17, Computer Science Department, University of Virginia, Charlottesville, March 1995.

[Sta92] Dave Stampe and Bernie Roehl, "REND386 - A 3-D Polygon Rendering Package for the 386 and 486 : LIBRARY Documentation Version 4.01 - September 1992", available via URL: ftp://sunee.uwaterloo.ca/pub/rend386/devel4.zip, September 1992.

[Sta94] Dave Stampe, "vr_api.h", available via URL: ftp://psych.toronto.edu/pub/vr-386/vr_api.h, 1994.

[Str76] Gilbert Strang, *"Linear algebra and its applications"*, Academic Press Inc., New York, 1976.

[Sty96] Martin R. Stytz, "Distributed Virtual Environments", *IEEE Computer Graphics and Applications*, Volume 16 Number 3, May 1996.

[Tre82] Philip C. Treleaven, David R. Brownbridge and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", *ACM Computing Surveys*, Volume 14, Number 1, March 1982.

[Van93] George Vanecek Jr. and James Cremer, "Project Isaac: Building Simulations for Virtual Environments", Technical Report CSD-TR-93-068, Computer Science Department, Purdue University, November 1993.

[VRM96] "The Virtual Reality Modeling Language Specification - Version 2.0", ISO/IEC CD 14772, available via URL: http://webspace.sgi.com/moving-worlds/spec/, August 1996

[Wal87] David Walker, "Introduction to a Calculus of Communicating Systems", Technical Report ECS-LFCS-87-22, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, March 1987.

[Wan94] Ko-Yang Wang, "Precise Compile-Time Performance Prediction for Superscalar-Based Computers", *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994, 73-84.

[Wan95] Qunjie Wang, Mark Green and Chris Shaw, "EM - an Environment Manager for Building Networked Virtual Environments", *Proceedings of the IEEE Virtual Reality Annual International Symposium 1995 (VRAIS '95)*, North Carolina, March 1995.

[War95] Robert Warriner, "Re: Collision Detection", Message to VRML discussion list, available via URL: http://www.eit.com/www.lists/www-vrml.1995q4/0253.html, October 1995.

[Wat94] Kevan Watkins, "A Virtual Modelling Environment", Honours Thesis, Computer Science Department, Rhodes University, November 1994.

[Wes92] A.J. West, T.L.J. Howard, R.J. Hubbold, A.D. Murta, D.N. Snowdon, D.A. Butler, "AVIARY - A Generic Virtual Reality Interface for Real Applications", An invited paper for "*Virtual Reality Systems*" sponsored by the British Computer Society, May 1992.

[Wlo95] Matthias M. Wloka, "Lag in Multiprocessor Virtual Reality", *Presence: Teleoperators and Virtual Environments*, Volume 4, Number 1, Winter 1995, 50-63.

[Wu95] Jiann-Rong Wu and Ming Ouhyoung, "A 3D Tracking Experiment on Latency and Its Compensation Methods in Virtual Environment", *Proceedings of the ACM Symposium on User Interface Software and Technology*, Pittsburgh, Pennsylvania, November 1995, 41-50.

[Yas92] Masahiro Yasugi and Akinori Yonezawa, "An Object-Oriented Parallel Algorithm for the Newtonian N-Body Problem", Technical Report 92-6, Department of Information Science, University of Tokyo, 1992.

[You95] Neil Youngman, "DIS Frequently Asked Questions", available via URL: http://www.crg.cs.nott.ac.uk/~dxl/DIS/dis.faq, October 1995.

[Zyd93] Michael J. Zyda, William D. Osborne, James G. Monahan and David R. Pratt, "NPSNET: Real-Time Vehicle Collisions, Explosions and Terrain Modifications", *Journal of Visualization and Computer Animation*, Volume 4, Number 1, 1993, 13-24.

Appendix A

Petri Nets and Data Flow Graphs

This chapter contains a brief introduction to Petri Nets and Data Flow Graphs. These constructs are used in performance analysis techniques considered in Chapter 3.

A.1 Petri Nets

This description is drawn from a description of Petri Net based performance modelling given in [Kan92].

A Petri Net consists of a 5-tuple (P, T, I, O, M), where:

P	is a set of places
T	is a set of transitions
I	is a set of input functions, mapping places to transitions
O	is a set of output functions, mapping transitions to places
M	is a set of markings, associating a non-negative integer with each place

Petri Nets are usually represented graphically using circles for places and bars for transitions. Input functions are given by arcs from circles to bars, output functions by arcs from bars to circles. Markings are represented by placing small filled circles (tokens) in the places.

The behaviour of Petri Nets is described in terms of firing. When a transition fires, it removes a token from each place connected to it via an input arc and adds a token to each place connected to it via an output arc. The transition can only fire if each input place contains at least one token.

Petri Net performance is often given relative the time spent in particular states. The state of a Petri Net is given by its marking (the number of tokens in each place).

Times can be associated with the transitions to produce a Timed Petri Net. If the firing times are random variables, then a Stochastic Petri Net (SPN) is produced. In this case, some simplifying assumptions are often made regarding the distribution of the firing times. Often a memoryless distribution is chosen, allowing the marking alone to provide an adequate description of the state of the system.

The analysis given in Chapter 3 assumes the use of Generalized Stochastic Petri Nets (GSPN). These allow both timed and untimed transitions. The firing times are exponentially distributed. Firing occurs by first delaying for the firing time and, if still enabled, removing tokens from the input places and adding them to the output places in a single instantaneous operation. Firing rate is allowed to depend on the marking.

States of a GSPN can be tangible (only timed transitions enabled), or vanishing. Vanishing states occur only momentarily since untimed transitions always fire before timed ones.

Markov processes (and Markov chains) are used in the solution of GSPNs. The Markovian property states that the probability of being in a particular state at any time is independent of the states occupied previously. A random process is a random variable that is a function of time. Thus a Markov process is a continuous time random process for which the Markovian property holds. The discrete time equivalent is called a Markov chain. The state residence time of a Markov process has an exponential distribution, that for a Markov chain has a geometric distribution.

Solution of a GSPN gives the state probabilities for the tangible states of the Markov process for the model. The solution process involves the removal of the vanishing states (to reduce complexity), followed by the solution of the Markov chain embedded at the transition points of the Markov process.

A.2 Data Flow Graphs

This description of Data Flow Graphs is based on that given in [Kav86] and [Tre82].

A Data Flow Graph is a bipartite directed graph whose two types of nodes are called links and actors.

$$G = (A \bigcup L, E)$$

where

A	is a set of actors
L	is a set of links
E	is a set of edges ($E \subseteq (A \times L) \bigcup (L \times A)$)

Semantically, the actors can be regarded as processors of information while the links act as storage points for data items. The arcs joining them can be regarded as channels for the transfer of data. The Data Flow Graphs in this document do not explicitly show the links, only the tokens that are stored in them.

The actual interpretation of the data tokens is not relevant for the Data Flow Graphs considered in this document. Movement of the tokens occurs when actors fire. Firing occurs when the actor has tokens on all its input links. Firing occurs by removing a token from each input link, delaying for the firing time of the actor, and then placing a token in each output link.

Appendix B

The use of the Analytical Simulation tool

This section is intended to provide a description of the use of the Analytical Simulation tool that was implemented to perform the analyses shown throughout this document. The emphasis is on describing the capabilities of the prototype system: to show the degree to which the analysis can be automated, as well as to provide a description of the syntax used in the models presented in this thesis.

The tool implements both the original Analytical Simulation algorithm, referred to as finite-cycle analysis, as well as the state space extensions to this approach.

B.1 Creation of a Model

The various statements that may be used in the description of a system are described in the sections following. Each statement is found on a separate line. A pre-processing step occurs during which some statements are expanded.

Variables can be used as arguments where applicable.

B.1.1 Comments

Comments in the file occur on lines starting with //. Blank lines may also be present.

B.1.2 Variables

Variable names in a model definition are strings containing upper case characters. Surrounding a variable name with square brackets means that the value of the variable is substituted. This is used to distinguish cases where both a variable name, or its value would be valid arguments. Substitution and evaluation can take place during both pre-processing and model simulation, depending on the availability of the variable values.

B.1.3 Expressions

Expressions are built of constants, variables and binary operators. Precedence is defined through the liberal use of parentheses. The operators available are:

==	equal (returns 1 if the arguments are identical, 0 otherwise)
!=	not equal (returns 0 if the arguments are identical, 1 otherwise)
>	greater than (returns 1 if the left argument is greater than the right, 0 otherwise)
<	less than (returns 1 if the left argument is less than the right, 0 otherwise)
+,-,*,/,%	addition, subtraction, multiplication, division and modulus (standard mathematical operations)

Constants and variables values must be limited to integers for evaluation by most of these operators to succeed. Only the first two (== and !=) may be successfully applied to strings.

B.1.4 Analysis control statements

This category of statement usually occurs at the beginning of the model description file and is used to describe the environment in which the model is analysed, as well as the format of the output produced after analysis.

B.1.4.1 Generate statement

This statement specifies the type of result required. The statement has the form:

```
generate arg1 [- [arg2 [arg3]]
```

The arguments are labels defined in REPORT statements elsewhere in the model. With only one argument it produces a list of times at which that report is executed. This is only meaningful when doing finite-cycle analysis. With the minus symbol included, differences between two sequential executions of the report are generated. This is useful for producing cycle times (the difference between the time that a process reaches a certain point, and the time at which it next reaches that point).

With three parameters, the difference between corresponding occurrences of two different reports is given. The meaning of corresponding occurrences is discussed in the main text in section 7.2. This is used for latency calculations.

The variation used for calculating latency also allows an additional argument. This corresponds to a cut marker, a report marker that prevents any further exploration of the execution paths on which it occurs. This can be used to limit the exploration of the reachability graph in state space, simplifying the analysis process, and eliminating paths that do not represent actual flow of data.

Any number of generate statements may be used.

B.1.4.2 Assume statement

The ASSUME statement allows certain assumptions to be preset for the model. This allows the state space search to be limited, predefined restrictions to be specified, or different forms of the assumptions to be tested. The format is:

```
assume exp1 > exp2
```

where each of the expressions consists of a sequence in the following format:
$C_1V_1 + C_2V_2 \ldots C_nV_n$ where the C_i are numeric constants and the V_i are variable names.

B.1.4.3 Declare statement

This statement is used to identify the variables in a particular process that are to be included in the state definition. As a rule of thumb, all variables should be declared in this way for the state space analysis to yield valid results. The format is given below:

```
declare var1, var2, var3, ...
```

B.1.4.4 Define statement

The DEFINE statement sets variable values during the pre-processing stage. This is the only way to define global variables. These values are only available during the pre-processing stage. The format is:

```
define variablename value
```

B.1.5 Executable statements

B.1.5.1 Process statement

The PROCESS statement is used to start the definition of a process in the model. It also defines the process name, required for message passing purposes. The code following a PROCESS statement until the next PROCESS statement or the end of the file, and excluding everything contained within an INIT-ENDINIT pair, is modelled as running within an infinite loop. The format is:

```
process processname
```

B.1.5.2 Send statement

The send statement is used to transfer data from one process to another. The format is:

```
send destinationprocess argument [time]
```

If time is not specified then it defaults to zero. The sending process blocks until the receiving process executes a RECEIVE statement whose source process field matches the name of the sending process, and whose argument matches the argument of the SEND statement. Both processes then synchronize for the amount of modelled time given by the time field.

A variation on the SEND statement, named SENDNOTRACE, has the same syntax and semantics but indicates that the message sent is a control message and not a data path to be considered when calculating latencies.

B.1.5.3 Receive statement

This statement takes the form:

```
receive sourceprocess argument
```

The argument can be a variable name which is instantiated with the value in the argument field of the SEND statement when synchronization occurs. If it is not a variable, then the argument of the sending process must match that of the receiving process. The source process can also be given as a variable. In this case the sender is chosen as the first process (in modelled time) to attempt communication (provided arguments can match). If more than one source process matches, then one is chosen non-deterministically. If the source process is given as a variable then it is instantiated with the name of the sending process when communication occurs. If no process is currently sending (in modelled time) the receiving process blocks.

B.1.5.4 SendIfReady statement

This is a variation on the SEND statement which either fails or succeeds non-deterministically. It uses the same format as the send statement, and the same semantics when it succeeds. When it fails, no action occurs and execution continues at the statement following the SENDIFREADY.

B.1.5.5 Think statement

This statement delays the process for a period in modelled time. The format is:

```
think time
```

B.1.5.6 Report statement

This marks points in the program which are starting or ending points for the calculation of cycle times and latencies. The format is:

```
report reportname
```

B.1.5.7 Assign statement

This is used to assign values to variables. All variables are initially given the value *undef*. The format is:

```
assign variable expression
```

B.1.5.8 If statement

This is a simple form of the IF statement found in most conventional programming languages and is used for simple flow control. The format is:

```
if expression
    body
endif
```

The body consists of a sequence of executable statements, excluding the PROCESS statement. The body is executed if the expression evaluates to a non-zero value.

B.1.5.9 Init statement

The INIT-ENDINIT pair are used to surround portions of code to be executed only once in the life time of a process. The code is run at the beginning of the first cycle. The format is:

```
init
    body
endinit
```

The body consists of a sequence of executable statements, excluding the PROCESS statement.

B.1.5.10 Replicate statement

This statement is only available during the pre-processing stage. It duplicates all code between it and the matching ENDREPLICATE statement. The number of times this duplication occurs is set by the value of its argument. For each iteration a hash variable is set to the iteration count, which ranges from 1 to *value*. The name of the hash variable is a string of # with a length equal to the nesting level. The format is:

```
replicate value
    body
endreplicate
```

The REPLICATE statement acts as a pre-processor version of a FOR loop. It is worth noting that the body can contain processes.

B.2 Analysis of a model

This section provides a detailed description of the process of analysis of a model. Complete details are given, as opposed to the examples given in the main body of this text which have had irrelevant details removed. The various issues involved in constructing a model for efficient analysis are also mentioned.

B.2.1 Construction of the model

A simple token ring system is chosen as the example of the system to be modelled. This system consists of a number of processes passing a message from one to another in a circular fashion. As an extra constraint, the communication medium allows each process an equal chance of placing messages on the medium, allocated on a round robin basis. The communication slot is allocated for at least a period K, independent of whether it is used or not.

The description of the model of this system is shown below:

```
// simulate a token being passed sequentially from one token process
// to the next via a token controller which allocates slots to each
// process in turn
// let N be the number of token processes

generate havetoken3 -
generate havetoken3 - tokenmark

define N 6

replicate [N]

    process token#
        declare HAVETOKEN
        declare SOMEWHERE
        declare SOMETHING
        init
            if # == 1
              assign HAVETOKEN got
            endif
        endinit
        receive SOMEWHERE SOMETHING
        if SOMEWHERE == tokencontrol
            if HAVETOKEN == got
              report tokenmark#
              send token[(#%N)+1)] token C
              assign HAVETOKEN undef
            endif
            send tokencontrol finished
        endif
        if SOMETHING == token
            assign HAVETOKEN got
            report havetoken#
        endif

endreplicate
```

```
process tokencontrol
    replicate [N]
        send token# yourturn
        think K
        receive token# finished
    endreplicate
```

The number of processes in the system is set by the variable N, defined in the DEFINE statement. In this case it is set to the value 6, but can be easily changed without having to alter any other parts of the model.

The *token* processes are inside a REPLICATE statement whose argument is the variable N. The # suffix is instantiated when the REPLICATE is expanded, generating processes: *token1, token2, ..., tokenN.* Each of the *token* processes declare three variables, two to hold information about a communicating process, and the third to determine which process holds the token at any point. The INIT statement gives the token initially to process *token1.* Based on whether the incoming message is the *tokencontroller* allocating a communication slot, or just the token arriving from another *token* process, the process either sends the token onward if it holds it, or accepts the token. When the token is passed on it takes time C before the completion message is sent back to the *tokencontroller*, which allows it to allocate the next communication slot.

The *tokencontroller* process sends a message to each *token* process in turn, offering it a communication slot. It then waits a period K before accepting a completion message and allocating the next slot. These communications are instantaneous (in modelled time) since they are constructs used to enforce the nature of the model, and not necessarily present in the actual implementation of such a system.

The technique of assigning the value *undef* to variables simplifies the state space analysis since duplicates of an earlier state occur much sooner. Using a third value to represent the absence of a token would increase the number of possible states that the system could assume. The variables SOMEWHERE and SOMETHING could also be given the value *undef* at the end of the *token* process to reduce the reachability graph. This optimization does not affect the nature of the model but can simplify the state space search by a substantial factor.

The placement of REPORT markers needs to be performed with a view to the required results. The first GENERATE statement is used to calculate cycle times for one of the *token* processes. Since the system is (almost) symmetrical, any *token* process can be chosen. The value required is the inter-token arrival time, hence the placing of the *havetoken* markers. The actual body of the process repeats more often, each time a communication slot is allocated, or a token arrives.

The second GENERATE statement calculates the latency for a message from process *token1* to *token3.* This gives the time taken for a token to pass from the one process to the other.

B.2.2 Analysis of the model

The model after it has gone through the pre-processing stage is shown below, for $N = 3$:

```
Process   token1
[3]   *** Initialization section
[4]     If      1       ==      1
[5]       Assign HAVETOKEN      got
[6]     Endif
[7]   *** End of Initialization
[8]   Receive SOMEWHERE SOMETHING
[9]   If      SOMEWHERE ==      tokencontrol
[10]    If      HAVETOKEN       ==      got
[11]      Report  tokenmark1
[12]      Send    token2        token (1 C)
[13]      Assign HAVETOKEN      undef
[14]    Endif
[15]    Send    tokencontrol    finished
[16]  Endif
[17]  If      SOMETHING         ==      token
[18]    Assign HAVETOKEN        got
[19]    Report  havetoken1
[20]  Endif

Process   token2
[3]   *** Initialization section
[4]     If      2       ==      1
[5]       Assign HAVETOKEN      got
[6]     Endif
[7]   *** End of Initialization
[8]   Receive SOMEWHERE SOMETHING
[9]   If      SOMEWHERE ==      tokencontrol
[10]    If      HAVETOKEN       ==      got
[11]      Report  tokenmark2
[12]      Send    token3        token (1 C)
[13]      Assign HAVETOKEN      undef
[14]    Endif
[15]    Send    tokencontrol    finished
[16]  Endif
[17]  If      SOMETHING         ==      token
[18]    Assign HAVETOKEN        got
[19]    Report  havetoken2
[20]  Endif
```

```
Process   token3
[3]   *** Initialization section
[4]     If      3       ==      1
[5]       Assign HAVETOKEN      got
[6]     Endif
[7]   *** End of Initialization
[8]   Receive SOMEWHERE SOMETHING
[9]   If      SOMEWHERE ==      tokencontrol
[10]    If      HAVETOKEN       ==      got
[11]      Report  tokenmark3
[12]      Send    token1        token (1 C)
[13]      Assign HAVETOKEN      undef
[14]    Endif
[15]    Send    tokencontrol    finished
[16]  Endif
[17]  If      SOMETHING         ==      token
[18]    Assign HAVETOKEN        got
[19]    Report  havetoken3
[20]  Endif

Process   tokencontrol
[0]   Send    token1    yourturn
[1]   Think   K
[2]   Receive token1    finished
[3]   Send    token2    yourturn
[4]   Think   K
[5]   Receive token2    finished
[6]   Send    token3    yourturn
[7]   Think   K
[8]   Receive token3    finished
Dimension of state space:  13
```

The last line shows the dimension of state space: three *token* processes, three variables in three *token* processes and the *tokencontroller* process. For finite cycle analysis, limited to at most ten cycles of any process, the output produced is shown below:

```
Trace 1:  Program with assumptions:
Example:  C = 1000 K = 500
1 C       >       1 K
Change in havetoken3    3 C (4)
havetoken3 - tokenmark1         2 C (5)
```

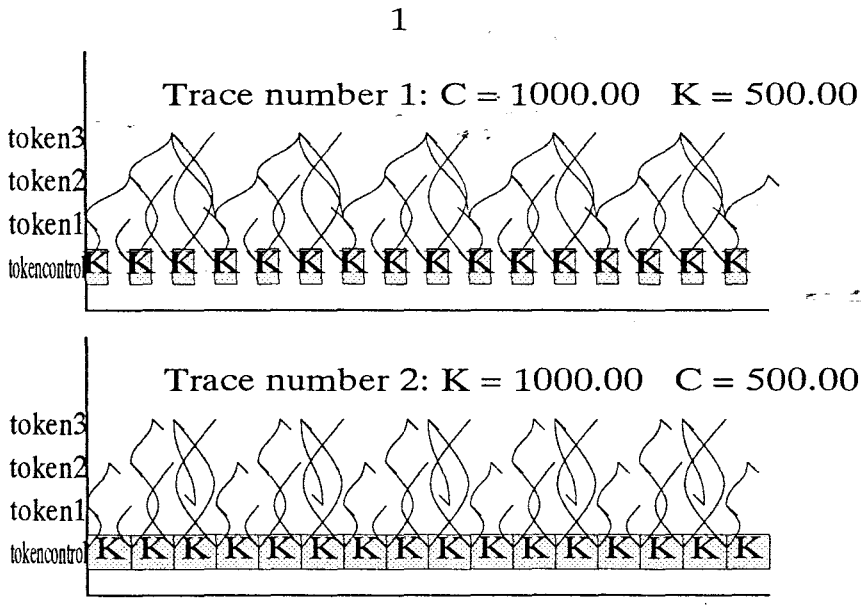

Figure B.1: Trace of process activity for the three process token passing system

```
Trace 2:  Program with assumptions:
Example:  K = 1000 C = 500
1 K       >       1 C
Change in havetoken3     3 K (4)
havetoken3 - tokenmark1          1 C + 1 K
havetoken3 - tokenmark1          1 K + 1 C (4)
```

The two cases resulting from the single assumption are shown, with an example of the values of K and C which would satisfy each constraint region. The output specified by the generate statements is also given, the values in parentheses after each result is the number of times that the particular result occurs. Thus four token cycles occurred, and five tokens could be traced from *token1* to *token3*.

The program also generates a graphical trace of the system for this interval, see Figure B.1, for verifying the model. This view is also useful for understanding unexpected performance characteristics.

A state space analysis is performed to guarantee that all performance characteristics of a particular model are shown. The same input data is used for this analysis. A number of output files are produced from this analysis, depicting the state space at various points of the analysis. The most useful is indubitably the final result which in this case consists of the file shown below:

```
0:  --1 K > 1 C
   ----Cycle havetoken3 [3 K(1) -> 3 K(1)] |
       Delay tokenmark1:havetoken3 [1 K + 1 C(1) -> 4 K + 1 C(1)]
```

```
1:  --1 C > 1 K
    ----Cycle havetoken3 [3 C(1) -> 3 C(1)] |
        Delay tokenmark1:havetoken3 [2 C(1) -> 5 C(1)]
```

The results given are the minimum and maximum values. The format is the cost for a sequence of values followed by the sequence length in parentheses. The value per cycle can be obtained by taking the total sequence cost and dividing by the sequence length. The large upper limit on latency is due to the presence of a second message path in the system: A message can go from *token1* to *token3* via the token passing SEND, or through the messages passed between *token* processes and the *tokencontrol* process. The path of interest is the former, corresponding to the lower bound on the results.

To determine the dependency on N, the results for different values of N are combined:

N	Cycle havetoken1	Delay tokenmark1:havetoken1
2	$2K$ if $K \geq C$ $2C$ if $C \geq K$	$K+C$ if $K \geq C$ $2C$ if $C \geq K$
3	$3K$ if $K \geq C$ $3C$ if $C \geq K$	$2K+C$ if $K \geq C$ $3C$ if $C \geq K$
4	$4K$ if $K \geq C$ $4C$ if $C \geq K$	$3K+C$ if $K \geq C$ $4C$ if $C \geq K$
5	$5K$ if $K \geq C$ $5C$ if $C \geq K$	$4K+C$ if $K \geq C$ $5C$ if $C \geq K$
6	$6K$ if $K \geq C$ $6C$ if $C \geq K$	$5K+C$ if $K \geq C$ $6C$ if $C \geq K$
7	$7K$ if $K \geq C$ $7C$ if $C \geq K$	$6K+C$ if $K \geq C$ $7C$ if $C \geq K$

The latency measured above is that for a complete traversal of the ring, since the process *token3* used previously is not present for all values of N. Thus the results for an arbitrary value of N are:

$K \geq C$:

Cycle Time : NK

Latency : $(N-1)K+C$

$C \geq K$:

Cycle Time : NC

Latency : NC

Other variations can be explored easily. For example the token might not be propagating in the same direction as the communication slots. Changing *tokencontrol* to reverse the order as

follows:

```
process tokencontrol
    replicate [N]
        send token[(N+1)-#] yourturn
        think K
        receive token[(N+1)-#] finished
    endreplicate
```

gives the following result for the $N = 3$ case:

```
0:  --1 K > 1 C
   ----Cycle havetoken1 [6 K(1) -> 6 K(1)] |
       Delay tokenmark1:havetoken1 [4 K + 1 C(1) -> 10 K + 1 C(1)]

1:  --1 C > 1 K
   ----Cycle havetoken1 [3 K + 3 C(1) -> 3 K + 3 C(1)] |
       Delay tokenmark1:havetoken1 [3 C + 2 K(1) -> 6 C + 5 K(1)]
```

This shows the change in performance for a token ring trying to pass information the "wrong" way.

Appendix C

Translation of Modelling Language into CCS

This appendix describes the translation of models specified for use with the Analytical Simulation tool into CCS, Milner's Calculus of Communicating Systems [Mil89]. This transformation has a number of benefits. It allows the models used for performance analysis to be translated easily to a system which allows additional properties to be proven. CCS also has a well defined semantics, and the translation process defines the semantics of the constructs used in the Analytical Simulation models (Given with rather less precision in Appendix B).

This section starts by giving a brief description of CCS and the extensions involved in dealing with time and variables. The translation process is then described, followed by a simple example in which a semaphore construct used in many previous models is transformed into a CCS specification used to provide exclusive access to critical regions.

C.1 CCS

The following simple introduction to CCS is based on that given in [Wal87] and [Fen96].

Processes contain ports for communicating, these ports occur in complementary pairs consisting of an input port and an output port. The same label is used for both the input and output ports of a pair, the latter being distinguished by being marked with an overbar. The agent expressions, E, in the language are defined by the following BNF:

$$
\begin{aligned}
\epsilon \quad ::= \quad & 0 \\
| \quad & A \\
| \quad & \epsilon \backslash K \\
| \quad & (\epsilon) \\
| \quad & \alpha.\epsilon
\end{aligned}
$$

$$
\begin{array}{l}
\mid \quad \epsilon|\epsilon \\
\mid \quad \epsilon+\epsilon \\
\mid \quad \text{if } b \text{ then } \epsilon \text{ else } \epsilon
\end{array}
$$

The *Nil* agent, **0**, describes a process that performs no actions. The other agent constants, represented by A above, require definitions of the form: $A \stackrel{def}{=} E$. Prefixing an expression with a port label indicates that communication must occur on that port before any actions in the rest of the expression can occur. Restricting an agent E to a set of labels K removes the ability of the agent to communicate on a port contained in K. Summation of two expressions, $E + F$, allows the combined agent to undertake actions from either of the two expressions. Composition of two agents, $E \mid F$ allows the two to run in parallel.

A rigorous definition of the operational semantics of CCS is given below. The format of the rules given specifies the hypotheses above the bar, and the conclusion below. Additional conditions are given in brackets to the right.

$$Act \frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E}$$

$$Sum_1 \frac{E \stackrel{\alpha}{\rightarrow} E'}{E+F \stackrel{\alpha}{\rightarrow} E'} \qquad Sum_2 \frac{F \stackrel{\alpha}{\rightarrow} F'}{E+F \stackrel{\alpha}{\rightarrow} F'}$$

$$Com_1 \frac{E \stackrel{\alpha}{\rightarrow} E'}{E|F \stackrel{\alpha}{\rightarrow} E'|F} \qquad Com_2 \frac{F \stackrel{\alpha}{\rightarrow} F'}{E|F \stackrel{\alpha}{\rightarrow} E|F'} \qquad Com_3 \frac{E \stackrel{l}{\rightarrow} E' \; F \stackrel{\bar{l}}{\rightarrow} F'}{E|F \stackrel{\tau}{\rightarrow} E'|F'}$$

$$Res \frac{E \stackrel{\alpha}{\rightarrow} E'}{E\backslash K \stackrel{\alpha}{\rightarrow} E'\backslash K} (\alpha, \overline{\alpha} \notin K)$$

$$Def_1 \frac{E \stackrel{\alpha}{\rightarrow} E'}{A \stackrel{\alpha}{\rightarrow} E'} (A \stackrel{def}{=} E) \qquad Def_2 \frac{E \stackrel{\alpha}{\rightarrow} E'}{E \stackrel{\alpha}{\rightarrow} A} (A \stackrel{def}{=} E')$$

$$Cond_1 \frac{E \stackrel{\alpha}{\rightarrow} E'}{\text{if } b \text{ then } E \text{ else } F \stackrel{\alpha}{\rightarrow} E} (b \text{ true}) \qquad Cond_2 \frac{F \stackrel{\alpha}{\rightarrow} F'}{\text{if } b \text{ then } E \text{ else } F \stackrel{\alpha}{\rightarrow} F'} (b \text{ false})$$

$$Brac_1 \frac{E \stackrel{\alpha}{\rightarrow} E'}{(E) \stackrel{\alpha}{\rightarrow} E'} (E \equiv E_1 + E_2) \qquad Brac_2 \frac{E \stackrel{\alpha}{\rightarrow} E'}{(E) \stackrel{\alpha}{\rightarrow} (E')} (E \equiv E_1|E_2)$$

The syntax $E \stackrel{\alpha}{\rightarrow} E'$ may be read as E performs action α and becomes E'. A state transition occurring on a communication between two agents is written as a τ action. The rules above define the semantics of the prefix, summation, composition, restriction, constant definition, conditional and brackets respectively.

To effectively translate the timing information, an extension to CCS is required. The introduction of a delay construct produces a calculus known as the Temporal Calculus of Communicating Systems (TCCS) [Mol89]. This includes the delay operation and an extra choice (summation) construct. The construction $(t).E$ represents a process that behaves as E after t units of time. The summation as used above, $E + F$, can behave as E or F, the choice being made at the time of the first action and is referred to as strong choice. The weak choice operator, as used in $E \oplus F$, can behave as E or F, the choice being made at the time of the first action or at a point where

only one process may continue to delay. The semantics for the two summation operators are as for the summation operator in CCS.

The time delay semantics, given below, are dependent on the function $|\cdot|_\tau$ which gives the maximum time delay before a computation must occur and is defined as follows:

$$\begin{aligned}
|0|_\tau &= 0 \\
|A|_\tau &= 0 \\
|a.E|_\tau &= 0 \\
|(t).E|_\tau &= t + |E|_\tau \\
|E \oplus F|_\tau &= max(|E|_\tau, |F|_\tau) \\
|E + F|_\tau &= min(|E|_\tau, |F|_\tau) \\
|E|F|_\tau &= min(|E|_\tau, |F|_\tau) \\
|E\backslash a|_\tau &= |E|_\tau
\end{aligned}$$

The time delay semantics are then:

$$Del_1 \frac{}{(s+t).E \xrightarrow{s} (t).E} \qquad Del_2 \frac{}{(t).E \xrightarrow{t} E} \qquad Del_3 \frac{E \xrightarrow{s} E'}{(t).E \xrightarrow{s+t} E'}$$

$$StrongSum \frac{E \xrightarrow{t} E' \; F \xrightarrow{t} F'}{E+F \xrightarrow{t} E'+F'}$$

$$WeakSum_1 \frac{E \xrightarrow{t} E'}{E \oplus F \xrightarrow{t} E'} (|F|_\tau < t) \qquad WeakSum_2 \frac{F \xrightarrow{t} F'}{E \oplus F \xrightarrow{t} F'} (|E|_\tau < t) \qquad WeakSum_3 \frac{E \xrightarrow{t} E' \; F \xrightarrow{t} F'}{E \oplus F \xrightarrow{t} E' \oplus F'}$$

$$Com \frac{E \xrightarrow{t} E' \; F \xrightarrow{t} F'}{E|F \xrightarrow{t} E'|F'}$$

$$Res \frac{E \xrightarrow{t} E'}{E\backslash K \xrightarrow{t} E'\backslash K}$$

Extensions to CCS to incorporate value passing using variables are described in [Fen96]. The value-passing calculus involves only the relabelling of processes to convert it back to CCS, and so the semantics above still hold.

C.2 Translation of Analytical Simulation Modelling Language into CCS

The transformations given apply only to the constructs present after the preprocessing stage of the Analytical Simulation process, thus limiting the range of statements that need to be translated. The problem of defining suitable restriction sets is not addressed in detail; it is assumed that all labels are unique within the system, and restricted so as not to be visible outside the translated model.

C.2.1 Process

Each model consists of a number of processes running concurrently, each containing a sequence of statements that is repeated in an infinite loop. Thus in CCS:

$$Model = (Process_1 \mid Process_2 \mid \ldots \mid Process_n)$$

and

$$Process_i = \epsilon_i.Process_i$$

where ϵ_i is a sequence constructed by translating the sequence of statements in the *ith* process as specified below.

Each variable in a process is defined by the following definition:

$$\begin{aligned} Variable &= variable(undef) \\ variable(x) &= variable \bullet assign(y).var(y) + variable \bullet read(x).var(x) \end{aligned}$$

where it is assumed that *Variable* and *variable* are replaced by suitably unique descriptors.

C.2.2 Assign

The ASSIGN statement of the form

```
assign variable value
```

translates into the following sequence:

$$\overline{variable \bullet assign}(value)$$

C.2.3 Think

The think statement translates directly into a delay. Thus

```
think t
```

translates to:

$$(t)$$

C.2.4 Send and Receive

The RECEIVE statement provides different functionality depending on its syntax. It can either receive a message from a specific process, or select one non-deterministically from a number of processes. Variables can be involved in the operation, by both the sending and receiving process. These can be found in the field specifying the second process, or as the datum to be transferred during the communication.

The SEND statement can be translated for the different cases as shown below. The statements:

```
send processname constant commtime
send processname variable commtime
send procvariable constant commtime
send procvariable variable commtime
```

translate to

$\overline{processname}(sourceprocess, constant).(commtime).sourceprocesss$
$\overline{variable \bullet read}(x).\overline{processname}(sourceprocess, x).(commtime).sourceprocess$
$\overline{procvariable \bullet read}(v).\overline{v}(sourceprocess, constant).(commtime).sourceprocess$
$\overline{procvariable \bullet read}(v).\overline{variable \bullet read}(x).\overline{v}(sourceprocess, x).(commtime).sourceprocess$

respectively. The variables x and v must be replaced by suitably unique local variable identifiers. The name of the process containing the SEND replaces the identifier *sourceprocess.*

The receive operation translates similarly.

```
receive processname constant
receive processname variable
receive [procvariable] constant
receive [procvariable] variable
receive procvariable constant
receive procvariable variable
```

translate to

$destinprocess(processname, constant).\overline{processname}$
$destinprocess(processname, x).variable \bullet assign(x).\overline{processname}$
$procvariable \bullet read(v).destinprocess(v, constant).\overline{v}$
$procvariable \bullet read(v).destinprocess(v, x).variable \bullet assign(x).\overline{v}$
$destinprocess(v, constant).procvariable \bullet assign(v).\overline{v}$
$destinprocess(v, x).procvariable \bullet assign(v).variable \bullet assign(x).\overline{v}$

respectively. The last two cases correspond to the situation where the source process is a variable and where the RECEIVE is performed non-deterministically.

The use of two messages ensures that both processes must block for the required length of time. A simpler form leaving out the final step can be used should the communication time be zero.

C.2.5 SendIfReady

This construct either performs a send or it does not. If σ is the translation of the corresponding form of the send statement, and ϵ is the translation of the remainder of the statements in the process, then the required TCCS for this construct is:

$$(\sigma.\epsilon) \oplus \epsilon$$

C.2.6 If

This statement translates using the equivalent construct under CCS. Given the statement:

```
if b
  body
endif
```

let β be the translation of the body, and ϵ be the translation of the remainder of the statements in the process. The translation of the IF statement is:

$$\text{if } b \text{ then } \beta.\epsilon \text{ else } \epsilon$$

If evaluation of the condition, b, requires values of variables then $\overline{read}$ actions may need to be issued before the translation of the IF statement to retrieve the values.

C.3 Restrictions on concurrent access

This section shows the effect of the translation procedure applied to a simple sample system, demonstrating mutual exclusion. The model for Analytical Simulation is given below:

```
process p1
    think A
    send mutex start
    think B
    send mutex stop

process p2
    think C
    send mutex start
    think D
    send mutex stop

process mutex
    receive ANY start
    receive [ANY] stop
```

The translation using the rules above is:

$$
\begin{aligned}
Program &\stackrel{def}{=} (P_1 \mid P_2 \mid Mutex) \\
P_1 &\stackrel{def}{=} (a).\overline{mutex}(p_1, start).(0).p_1.(b).\overline{mutex}(p_1, stop).(0).p_1.P_1 \\
P_2 &\stackrel{def}{=} (c).\overline{mutex}(p_2, start).(0).p_2.(d).\overline{mutex}(p_2, stop).(0).p_2.P_2 \\
Mutex &\stackrel{def}{=} mutex(x, start).\overline{any \bullet assign}(x).\overline{x}.\overline{any \bullet read}(y).mutex(y, stop).\overline{y}.Mutex \\
ANY &\stackrel{def}{=} Any(undef) \\
Any(z) &\stackrel{def}{=} any \bullet assign(w).Any(w) + any \bullet read(z).Any(z)
\end{aligned}
$$

Simplifying, by removing the extra synchronizations which are unnecessary due to the zero communication time gives:

$$
\begin{aligned}
Program &\stackrel{def}{=} (P_1 \mid P_2 \mid Mutex) \\
P_1 &\stackrel{def}{=} (a).\overline{mutex}(p_1, start).(b).\overline{mutex}(p_1, stop).P_1 \\
P_2 &\stackrel{def}{=} (c).\overline{mutex}(p_2, start).(d).\overline{mutex}(p_2, stop).P_2 \\
Mutex &\stackrel{def}{=} mutex(x, start).\overline{any \bullet assign}(x).\overline{any \bullet read}(y).mutex(y, stop).Mutex \\
ANY &\stackrel{def}{=} Any(undef) \\
Any(z) &\stackrel{def}{=} any \bullet assign(w).Any(w) + any \bullet read(z).Any(z)
\end{aligned}
$$

By examining the specification above, one can see that the variable is assigned a value which is then immediately read from it. The variable is not used anywhere else, and can be removed, leading to a specification closely resembling that given as an example of a mutual exclusion system in [Fen96] and which is reproduced below:

$$
\begin{aligned}
Mx &\stackrel{def}{=} (P_1 \mid P_2 \mid Sem)\backslash\{get, put\} \\
P_1 &\stackrel{def}{=} \overline{get}.c_1.\overline{put}.P_1 \\
P_2 &\stackrel{def}{=} \overline{get}.c_2.\overline{put}.P_2 \\
Sem &\stackrel{def}{=} get.put.Sem
\end{aligned}
$$