Autonomic Visualisation

David Chisnall

A thesis submitted to the University of Wales in
candidature for the degree of Philosophiae Doctor

Department of Computer Science
University of Wales, Swansea

January 10, 2008

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed	[illegible]	(candidate)
Date	January 10, 2008	

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed	[illegible]	(candidate)
Date	January 10, 2008	

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed	[illegible]	(candidate)
Date	January 10, 2008	

Summary

This thesis introduces the concept of autonomic visualisation, where principles of autonomic systems are brought to the field of visualisation infrastructure. Problems in visualisation have a specific set of requirements which are not always met by existing systems.

The first half of this thesis explores a specific problem for large scale visualisation; that of data management. Visualisation algorithms have somewhat different requirements to other external memory problems, due to the fact that they often require access to all, or a large subset, of the data in a way that is highly dependent on the view. This thesis proposes a knowledge-based approach to pre-fetching in this context, and presents evidence that such an approach yields good performance.

The knowledge based approach is incorporated into a five-layer model, which provides a systematic way of categorising and designing out-of-core, or external memory, systems. This model is demonstrated with two example implementations, on in the local and one in the remote context.

The second half explores autonomic visualisation in the more general case. A simulation tool, created for the purpose of designing autonomic visualisation infrastructure is presented. This tool, SimEAC, provides a way of facilitating the development of techniques for managing large-scale visualisation systems.

The abstract design of the simulation system, as well as details of the implementation are presented. The architecture of the simulator is explored, and then the system is evaluated in a number of case studies indicating some of the ways in which it can be used. The simulator provides a framework for experimentation and rapid prototyping of large scale autonomic systems.

Acknowledgements

First, I would like to thank Professor Min Chen for his patience and advice during his time as my supervisor while completing this PhD. I would also like to thank Professor Chuck Hansen for giving me the opportunity to vist the University of Utah in 2006.

Heartfelt thanks go to Beth Kupin, who helped keep me sane in the run up to submission deadlines for the papers that I wrote during the course of this PhD. Without her help, I would have found my PhD much more stressful.

I would also like to thank my parents for their support and advice over the years, especially for my mother for providing somewhere free of distractions where I could work while finishing this thesis.

Next, I need to thank Will Harwood, primarily for his coffee-making abilities, but also for his work on developing the official rules for the historic game of Labminton, without which the lab would have been a far less interesting place.

I would like to thank everyone in the Étoilé project for giving me some intersting problems to work on when I wanted to take a break from my thesis. Of particular note is Nicolas Roard, who persuaded me to stop complaining about the appaling state of desktop computing and do something about it and who helped turn nebulous ideas into ideas for working software.

I should probably thank Jo Gooch for providing various distractions when I was in need of a break, and ensuring that I rarely missed coffee time. Andy Gimblett and Basheera Khan have also been very helpful in this regard.

Finally, I need to thank all of the other inhabitants of the postgraduate lab, as well as many of the faculty of Computer Science, for making Swansea such an enjoyable place to study.

Contents

Part I

Introduction

Chapter 1

Aims and Objectives

Contents

Visualisation applications continue to require more and more computational power. Scientific datasets are now measured in gigabytes or terabytes, and show no sign of slowing their growth. The process of extracting meaningful information from datasets of this size through the process of visualisation can be computationally very expensive.

Current approaches to visualisation tend to follow a somewhat ad-hoc approach to the problem of very large scale deployments. Existing algorithms are applied to new problems with some slight modifications for their new environment. Chapter 2 outlines the problems involved in deploying a large-scale visualisation infrastructure, and some attempts to address these problems.

The term Autonomic Computing was coined by IBM [144] to describe an evolution in management of complex systems. This thesis will describe a vision for applying this and related concepts to the problem of large scale visualisation, and will demonstrate the progress made in this endeavour.

Delivering a full autonomic environment for visualisation would be far beyond the scope of a single PhD thesis, and so two parts of the problem will be addressed directly.

1.1 Large Dataset Management

The rapid growth of data from both complex simulations and high-resolution scanning instruments has outstripped the rate of growth of the physical memory, and in some cases even local disk storage. This problem is exacerbated by the rise in mobile computing. A modern laptop has a significant amount of processing power [287, 213] - often including a relatively fast GPU - and so is up to the task of performing a number of forms of visualisation. In the coming years it appears likely that this trend will extend to handheld devices.

The growth in data size as well as the shrinking of the devices on which the data is likely to be displayed presents a range of problems when it comes to data management. Most operating systems provide some support for paged virtual memory. These typically work on two key assumptions:

1. The computer has almost enough physical memory, and should only spill over in to swap space occasionally.
2. The working environment is multitasking, and idle applications can be completely swapped out.

The first assumption comes from the fact that most applications have a relatively small *working set*, which changes infrequently. When this change occurs, it is mildly irritating to the user for their computer to pause briefly, but does not have a great deal of impact on overall performance.

Neither of these assumptions necessarily holds for a visualisation task. The dataset might be several times the size of RAM, and the rendering algorithm may need several passes over the entire dataset. This causes most operating systems to experience *thrashing*, the situation in which most of their time is spent swapping data between disk and RAM rather than performing useful computations.

One application for which this is a significant problem is *discrete ray tracing* of point sets. This application suffers from the fact that the access patterns through the data are highly dependent on the data itself, with the positions of both the points themselves and the light sources within the scene affecting the paths of the rays.

The discrete ray tracer application will be used as a test case for examining prefetching techniques. Since this application has serious memory usage problems, both in terms of size and access patterns, it makes an ideal study for evaluating pre-fetching strategies. The situation in which no pre-fetching occurs, demand paging, produces highly sub-optimal results, and it is anticipated that the addition of a more advanced data management strategy can provide significant performance benefits.

It should be possible to design such a strategy by first understanding the way in which the ray tracing algorithm works. While this is useful in and of itself, doing so does not require the use of autonomic concepts, nor is it likely to produce a solution that will be useful in other situations.

Such an algorithm could, however, be used as a base-line for evaluating other techniques. Of more use in the general case would be an algorithm that did not rely on understanding the nature of the rendering algorithm. This could, potentially, be adapted to other rendering techniques, eliminating the need for such things to be created on a per-problem basis. This thesis aims to evaluate the possibility of designing such an algorithm, and to compare the performance of a system-specific approach with an autonomic one.

1.2 System Simulation

Designing a fully autonomic infrastructure for visualisation is a massive task, and building one presents an even greater challenge. One of the biggest problems of such an endeavour relates to the problems (and opportunities) associated with emergent properties.

Emergent properties arise when relatively simple systems are assembled to form complex ones. The complex systems exhibit behaviours that are not always apparent from the simple designs. In some cases, these can be highly advantageous, and can allow simple designs to be used in complex rôles. In others, they are problematic. If a systems is not designed with emergent properties in mind, then they can often cause pathological behaviour.

For this reason, it is necessary to test an autonomic infrastructure as early as possible in the design process and attempt to discover any emergent behaviours, then either take advantage of them or eliminate them. This, unfortunately, can not be done by techniques such as unit testing, since it requires the entire system to be tested simultaneously.

The proposed solution to this problem is simulation. A simulation environment matching the needs of autonomic computing would allow simplified models of individual components to have their interactions with other parts of the system tested, without the cost and effort of building the entire system.

This thesis aims to present a working system to address this need. Such a system must be easy to use and extend, and have the ability to work at varying levels of granularity. As a system is developed it will be possible to use prototypes and even final versions of components to provide more calibration data to the system. A good simulation environment must adapt to this growing amount of input data.

Scalability also extends in the other direction. It must also allow the user to perform highly detailed simulations of a subset of components, and more coarse simulations of the entire system distributed over a large grid.

Visualisation tasks are fairly similar to other tasks which could be performed by an autonomic grid environment, and so it is hoped that such a simulator would be usable for evaluating systems that were not directly tied to visualisation. To this end, it should not make any assumptions about the nature of visualisation tasks. This is important even within the context of visualisation, since such assumptions become outdated rapidly. Visualisation used to be solely the province of supercomputers, and later high-end workstations. Now a large number of visualisations can be performed on modest commodity hardware, and even hand-held computers often come with some form of graphics co-processor.

The gradual move from thin-, or dumb-client through desktop and laptop and to ubiquitous computing shows that assumptions about the location and distributions of workloads within a computational system rarely last. Autonomic computing is somewhat orthogonal to the underlying distribution paradigm, but is often identified closely with ubiquitous computing. A fully autonomic grid infrastructure would be most likely to exist within the overall context of ubiquitous computing, but deployment of some components would almost certainly begin before such a setting existed. For this reason, a simulator must be able to handle both existing and proposed systems, and to allow the gradual transition from one to the other to be effective.

The usability of such a system can only be evaluated experimentally, and so it is aimed that the simulation should be used for some real-world tasks and evaluated.

1.3 Summary of Aims

The aims of this thesis can be summarised as follows:

1. Create an out-of-core prefetching strategy for the rendering problem outlined above.
2. Determine whether it is possible to achieve equivalent performance from a general, knowledge-based, prefetching algorithm.
3. Devise a systematic model for building out-of-core systems.
4. Demonstrate the model is feasible by providing at least one implementation of it.
5. Design a simulation system that can be used for developing an autonomic computing infrastructure.
6. Implement and evaluate the system.

If this thesis meets the goals outlined here then it will provide a demonstration of the effectiveness of knowledge-based solutions and provide a systematic means for other researches to incorporate these ideas into their own specific problems. It is hoped that these will be able to augment, and even replace, existing application-specific strategies. The later goals relate more generally to the field of autonomic visualisation and, if met, will facilitate the development of this very young field to a significant degree by allowing rapid evaluation of algorithms aimed at this field.

1.4 Outline

The remainder of Part I describes the problem associated with visual supercomputing in general, and suggests how autonomic computing could be applied to give some possible solutions. The remainder of the thesis will then focus on some areas within this topic.

Part II describes the problem of managing large datasets, specifically those too big to fit in RAM and those too big to be stored locally. These are increasingly common in visualisation. Modern scientific instruments can generate data of a size limited only by the size of the disk arrays used to store them.

This part will describe a visualisation system for large data sets and a pre-fetching strategy tied to the renderer. It will then show how a self-optimising approach can be used to simplify the development of systems to handle them. An adaptive approach to data pre-fetching is proposed, as well as a systematic model for building out-of-core systems. Two example implementations of the model are discussed.

Part III describes a system for simulating the interactions between components running on a heterogeneous grid environment. This system was developed to aid the development of autonomic distributed visualisation systems, and some example uses are discussed.

Chapter 2

The Problem of Visual Supercomputing

Contents

2.1 Introduction

Today there is a variety of computational resources available to visualisation. While a huge number of users are contented with the visualisation capabilities provided through modern desktop computers and powerful 3D graphics accelerators, many are still relying on high performance computing facilities to visualize very large data sets or to achieve real-time performance in rendering a complex visualisation. In some areas, users have already demanded visualisation capabilities to be provided through mobile computing systems, such as PDAs (Personal Digital Assistants), most of which are yet to benefit from powerful 3D graphics accelerators. As the size of visualisation data (e.g., in visual data mining), the complexity of visualisation algorithms (e.g., with volumetric scene graphs), and demand for instant availability of visualisation (e.g., for virtual environments) continues to grow, it is unlikely that visualisation users can be served adequately, at least in the coming years, by an infrastructure largely based on desktop computers.

This leads to a series of questions:

- What would be an adequate infrastructure?
- In what way do the computational requirements of visualisation differ from other software technologies?

- Is it desirable or feasible to bring a range of technologies under one management (not necessarily under one roof)?
- If it were feasible to build such an infrastructure, what would be an appropriate virtual machine interface for the infrastructure?

In fact, the computer graphics and visualisation community has been seeking answers for these questions for the past few decades. The community has invested a huge amount of effort in developing specialized graphics hardware, has always been among the first to deploy the latest technologies for high performance computing, and has accumulated large volumes of research outputs in parallel, distributed, and web-based techniques for visualisation. Recently, the community has shown equally great enthusiasm to embrace the cluster, Grid and mobile technologies. However, in general, the community tended to address these questions mainly from the perspective of visualisation technology in isolation, rather than from the perspective of a coherent infrastructure design.

In this chapter, the historic route for deploying new technology in the context of visualisation is examined and a broad range of scientific and technological developments, including theories, algorithms, hardware, software and services, are described. The term *Visual Supercomputing* is used to encompass the concepts and technologies required to build a computing infrastructure for visualisation. An overview of the state of the art of technologies in hardware and software for visualisation are presented, and the impacts of the Internet, Grid and mobile technologies is discussed.

Those latest developments that are relevant, or potentially relevant, to visualisation infrastructure are highlighted. A set of technical challenges in realising a visual supercomputing infrastructure that manages visualisation tasks in complex networked computing environments, as well as manages users experience in accessing and interacting with visualisation resources are proposed. Later chapters in this thesis discuss the application of principles from the field of *autonomic computing* to the solution of some of the challenges presented.

There is now a growing body of experience in adapting applications to a Grid environment. What is emerging is a consensus that the original idea of a computational Grid that behaved like a utility Grid for computation is perhaps oversimplified. There may be several different structures for Grids depending on whether the resources aggregated in the Grid are to serve large-scale computation, large-scale data handling, complex data sources (e.g., bioinformatics databases) or perhaps to integrate business processes. In this, the visual supercomputing paradigm presents novel challenges to the Grid concept. A number of pioneering projects, described in this chapter, have been testing the implications of a Grid for various visualisation applications and have raised many technical issues including real-time processing, synchronisation of resource allocation and interaction between clients and Grid services.

This chapter is organized into the following sections. In Section 2.2, the term *Visual Supercomputing* is defined. In Section 2.3, major scientific and technological developments are reviewed by following the arrivals of different computing technologies, identify the state of the art technologies are identified that have are required for building an infrastructure for visual supercomputing. In Section 2.4, a collection of applications that would benefit enormously from such an infrastructure are examined, and their technical requirements discussed.

2.2 Visual Supercomputing

In this section, the term ‘Visual Supercomputing’ is defined. Its relevance to the three semantic contexts of visualisation is examined. Finally, the technical scope of visual supercomputing from the perspectives of *applications*, *users* and *systems* respectively.

2.2.1 Definition

Definition 2.1 (Visual supercomputing)**.** *Visual supercomputing is concerned with the infrastructural technology for supporting visual and interactive computing in general, and visualisation in particular, in complex networked computing environments.*

This thesis is focussed on the subject domain of visualisation, although most of the discussion in this chapter can be extrapolated to other subject domains involving visual and interactive computing, such as computer-aided design, computer animation, and computer vision.

As an *infrastructural technology*, visual supercomputing encompasses a large collection of hardware technologies and software systems for supporting the computation and management of visualisation tasks. It focuses on generic technologies for managing the specification, execution and delivery of visualisation tasks. It addresses issues such the scheduling of visualisation tasks, hardware and software configurations, parallel and distributed computation, data distribution, communications between different visualisation tasks, and communications between visualisation tasks and their couplings such as computation tasks or data collection tasks. In addition, it provides infrastructural support to users’ interaction with visualisation systems, and manages users’ experience in accessing and interacting with visualisation resources. Nevertheless, visual supercomputing does not concern a specific algorithm and technique for processing a specific type of data in order to generate visualisation results.

Explicit emphasis is given to complex networked computing environments, as this chapter is not intended as only a survey of the technologies that have been developed so far in the context of visual supercomputing, but as a report on technologies that are in place as well as that are desirable for a future infrastructure. No doubt such an infrastructure must take *web computing*, *Grid computing* and *mobile computing* into account. Hence it has to provide comprehensive support to visualisation tasks in complex heterogeneous networked computing environments. As current trends in hardware design are increasingly driven by power consumption as well as raw speed, there is an increasing reliance on dedicated hardware. This trend makes it appear likely that heterogeneous environments will be the norm, rather than the exception, in the near future.

The visual supercomputing environment is best considered as a global Grid infrastructure for visualisation. The above-mentioned technical features of visual supercomputing have clearly set it apart from the traditional subject domains such as *hardware architectures for visualisation*, *parallel and distributed computation for visualisation*, *web-based visualisation*, and *collaborative visualisation*. While the advances in these traditional subject domains will have significant influence in shaping the infrastructure of visual supercomputing, building such an environment will require not only integrating these technical advances together in an environment, but also bringing in, and developing new, technologies for significantly improving the quality of services (QoS) of such an infrastructure and users' experience.

2.2.2 Semantic Contexts

The term 'visualising' refers to a process that extracts meaningful information from data, and constructs a visual representation of the information. In the field of visualisation, this *process* is commonly considered in three different but interrelated semantic contexts as illustrated in Figure 2.1.

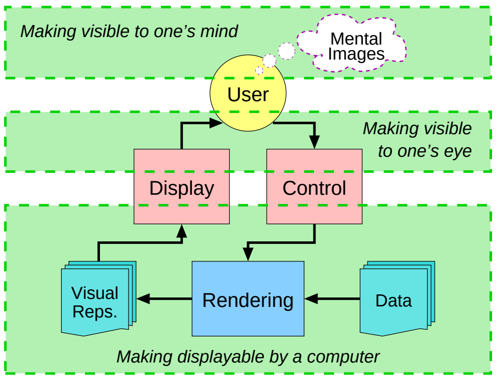

Making visible to one's mind
Mental Images
User
Making visible to one's eye
Display
Control
Visual Reps.
Rendering
Data
Making displayable by a computer

Figure 2.1: Three semantic contexts of visualisation.

Making displayable by a computer. This is concerned with the algorithmic and computational process of extracting information and rendering a visual representation of the information. In this semantic context, a visual supercomputing infrastructure should address issues such as allocating and scheduling computational resources for visualisation tasks, managing data distribution, and providing mechanisms for inter-process, and inter-task communications within an infrastructure.

Making visible to one's eyes. This is concerned with the process of specifying meaningful information, designing appropriate visual representations, and conveying visual representations to viewers. In this semantic context, a visual supercomputing infrastructure should address issues related to the interaction between users and their visualisation tasks, which can be conducted in a variety of forms, including interactive virtual environments, Internet-based collaborative environments, mobile visualisation environments, and so on.

Making visible to one's mind. This is concerned with users' thought process and cognitive experience of interpreting received information (not necessarily in a visual form) in one's mind and converting the information to knowledge in pictorial representations. In this semantic context, it is neither desirable nor perhaps feasible for a visual supercomputing infrastructure to manage the thought process of a user.

This section presents further detailed discussions on how visual computing relates to the above three semantic contexts, and provides rationalisation for focusing infrastructural support on the processes of *making displayable by a computer* and *making visible to one's eye*, with the state of the art technologies in visual supercomputing. It also argues for the necessity for introducing gradually new capabilities in a visual supercomputing environment, to support the process of *making visible to one's mind.*

2.2.3 Application Perspective

The demands for visualisation multiply in every direction, and there is an increasing number of new applications, resulting in new, and sometimes conflicting, requirements. For example:

- In some applications (e.g., bioinformatics), the size of datasets to be processed continues to grow, while in others (e.g., mobile visualisation), a careful control of data size is absolutely necessary. Even in cases where the data is sufficiently small to fit on a single machine, collaboration and security concerns may make it advisable to store it centrally. This makes the ability to quickly access the required subset of data that is available.

- In many applications (e.g., those involving 3D virtual environments), users still have plenty of appetite for photorealistic visualisation at an interactive speed, while in others (e.g., visual data mining), schematic visual representations and non-photorealistically rendered images are often able to convey more information. A visual supercomputing environment must be able to handle both latency- and throughput-dependent tasks.

- In many applications (e.g., virtual endoscopy), interactive visualisation can now be achieved with modern personal computers, hence small integrated systems provide a high degree of independence to users who operate in various practical situations. Meanwhile, other applications (e.g., those centralized around one or more data warehouses) require a substantial amount of computation for visualisation to be closely coupled with the source of data. Some applications, which have distributed data sources or dynamic data sources, demand a more complex computational model. A visual supercomputing environment must scale to all devices from hand-held machines up to supercomputers.

From the perspective of applications, an important requirement for a visual supercomputing infrastructure is choice. It has to provide support for a large collection of platforms, methods, mechanisms and tools to serve different categories of application, as well as offer each individual application a diverse selection of means to accomplish a visualisation task.

Section 2.4 considers several major applications, which collectively characterize the main requirements for a visual supercomputing infrastructure.

2.2.4 User Perspective

Visualisation users are no longer limited to scientists and engineers. At the same time, a visualisation process often requires a high degree of domain knowledge about the application concerned. While the diversity of applications demands a visual supercomputing environment to provide a large collection of platforms, methods, mechanisms and tools, users require the service to be tailored to individual needs, and to be delivered in a seamless manner. Many users, especially those less technically oriented, would very much hope for a secretary-like visualisation service, where they simply submit the data, give instructions and receive results. Although to get appropriate results may require a few feedback loops, many users certainly do not wish to get involved in choosing hardware, programming parallelism, organizing storage for input and output data, and so on. Further more, like a secretary, perhaps a visual supercomputing infrastructure should accumulate knowledge about various entities in the environments, profiling hardware capabilities, software usage, users' preference, etc. and gradually improving its quality of services to individual users.

Recent developments in business computing, such as electronic customer relationship management (e-CRM) [241, 183], has shown that it is possible to provide users with better quality of services with appropriate technologies that are capable of collecting and processing users' experience. The emergence of *autonomic computing* [165] is gathering further momentum in developing self-managed services in a complex infrastructure (see also Section 3.2). Therefore a visual supercomputing infrastructure should have the responsibility for managing:

- visualisation resources,
- visualisation processes,
- source data and resultant data,
- users' interaction and communication,
- users' experience in accomplishing a visualisation tasks.

2.2.5 System Perspective

From the system perspective, a visualisation task is a kind of computation task, which exhibits a specific class of characteristics. The infrastructure of visual supercomputing is built upon a range of underlying technologies, including computer hardware, operating systems, programming languages, data warehouses, communications, world-wide-web, Grid computing and knowledge-based systems. It is neither sensible nor feasible for the visualisation community to attempt to provide solutions in all these aspects. However, it is necessary for the construction of such an infrastructure to bring in the latest advances in other fields of computing and communications, and moreover, to influence the developments in these fields.

The following section, examines in detail the major advances and the state of the art in the relevant fields.

2.3 Trends in Visual Supercomputing

Visualisation is fundamentally an application of computing, and so trends in visual computing tend to follow overall trends in computing. The predominant trend in the industry since its inception has been towards miniaturisation. The one prediction that has always held in every aspect of computing has been 'you will be able to do this with a smaller machine next year.'

This section will provide a description of some of the more important trends and milestones in the computing industry, and their impact on visualisation. As visualisation is such a computationally-demanding application, it has always been near the forefront of uses for new advances.

For more information, please see the relevant surveys [332, 135, 27, 321, 104, 161, 40] and some major publications [94, 304, 127] [331, 281, 171].

2.3.1 The Dinosaurs

The pre-history of computing is dominated by large machines taking several rooms worth of space. The Cray-1 was possibly the first supercomputer deserving the name, although its computational power was small by modern standards. Visualisation was not the only task performed on this model, but Elwald and Mass's vector graphics library for this platform [103] represents some of the first efforts at directing supercomputing power towards visualisation.

Once the potential for computer visualisation began to be understood by the scientific community at large, the field began to get more attention. Supercomputers are difficult to define, as a breed. Some personal computers have been advertised as 'supercomputers' based solely on US export restrictions on processing power during the tail end of the cold war. This metric is not particularly meaningful, since the PDAs of one age are the supercomputers of an earlier one. Most supercomputers are based around the idea of massively parallel computational engines with very fast interconnects between them. This broad scope will be used for the remainder of this section, which will thus include some techniques that post-date the widespread use of supercomputers for visualisation, but inherit some concepts from this age.

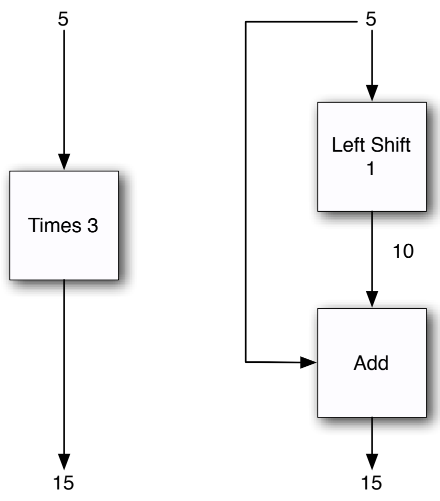


Figure 2.2: The decomposition of a multiply-by-three operation using functional parallelism.

2.3.1.1 The Growth of Parallel Computation

Computers are never quite fast enough. In 1965, Gordon Moore noted the trend that the number of elements that can be placed on an integrated circuit, for a fixed investment, doubles roughly every 18 months. This observation came to be known as Moore's Law [221].

Although Moore's Law says nothing about computational power, it has been a fairly accurate rule-of-thumb. If you double the number of transistors on a die, you can double the number of execution units, the amount of cache, etc. This is usually accomplished by a reduction in feature size, which comes with a corresponding increase in feasible clock speeds.

In spite of these gains, a single CPU is often still not fast enough. Or, more accurately, the cost of building a single CPU that is fast enough is prohibitive. A much cheaper alternative is to by two (or more) CPUs and join them together.

Most traditional programming models owe their heritage to very simple computers, and so have no concept of parallelism. New models were required, and two emerged. These were *functional parallelism* and *data parallelism*.

Functional parallelism, also called pipelining, describes a model by which an operation is split up into a number of stages, each of which can be executed sequentially. Figure 2.2 shows a trivial example, in which the function $f(n) = n \times 3$ is decomposed into functionally parallel stages. Assuming that each block in the diagram can be executed on a separate execution unit, the right hand pipeline could run twice as fast as the left. This is exploited by modern CPUs at the instruction level, but in visualisation it is commonly done at a much coarse granularity giving rise to the term *visualisation pipeline*.

Data parallelism is conceptually simpler, but often harder to implement in practice. Data parallelism splits the data, rather than the algorithm. A data parallel version of the multiply-by-three example would simply add a second multiplier. You could then achieve double the throughput, assuming that you could keep both supplied with data. The difficulty in implementing data parallelism comes from the fact that, for many applications, the data is not independent. Most modern hardware incorporates some form of data parallelism, from superscalar architectures in general purpose CPUs to multiple independent pipelines in GPUs.

In graphics, data parallelism is typically achieved using either *image space* or *object space* decomposition. Image space decomposition splits the workload up based on the pixels in the final image. Ray tracing is particularly suited to this approach, since each pixel in a ray-traced image is generated by a different ray, which is independent from the others. In rasterisation-based approaches, techniques such as *scan line interleaving* are common. In general, image space decomposition works better on *shared memory* architectures, since each thread needs access to all (or a large subset) of the source data for most techniques.

A final form of parallelism, known as *farm parallelism* is a hybrid of the two. In this, a problem is decomposed into computational tasks, encapsulating both a portion of data and some functional workload. These are typically independent, are are processed by whichever computational resources become free first. This form of parallelism is used by visualisation algorithms such as [242].

In terms of implementations, Flynn's taxonomy [111] still works well to describe the possible forms of parallel hardware in terms of the number of concurrent data and instruction streams. The simplest case, *single instruction, single data* (*SISD*) refers to the absence of any parallelism. At the other extreme, *multiple instruction, multiple data* (*MIMD*) refers to independent instructions working on multiple data streams, as is the case with independent processors. In the middle are *single instruction, multiple data*, (*SIMD*) and *multiple instruction, single data* (*MISD*) systems. The first, also known as vector processors, are common in supercomputer chips and increasingly as extra instructions on general purpose CPUs.

SIMD machines allow individual instructions to operate on vector rather than scalar quantities. While an add instruction in a SISD machine would be defined as $add(x, y) \rightarrow x+y$, the equivalent SIMD instruction might be defined as $add((x_1, x_2, x_3, x_4), (y_1, y_2, y_3, y_4)) \rightarrow (x_1 + y_1, x_2 + y_2, x_3 + y_3, x_4 + y_4)$. Vector processors typically have a very large peak throughput, however it is difficult to keep them saturated. In the four-way vector example, adding two numbers together would take as long as adding four pairs of numbers together. Visualisation algorithms tend to be well suited to this kind of architecture, since a vector can be used to store each colour component (for example), and acted on independently.

The final approach, MISD, is perhaps the hardest to pin down. It is difficult to define a MISD machine, because most machines could be categorised as MISD. A multiply instruction could be considered to be a series of add and shift instructions, for example (and a shift instruction, itself, could be considered a series of add instructions). Very few machines, if any, have instructions so simple that they can not be considered to be combinations of simpler operations.

2.3.1.2 Sharing Data

The simplest form of parallel computation is two completely independent computers, running two independent tasks. This could be regarded as the most widely-deployed form of parallel computation, since every computer in the world, past and present, is part of a grid implementing this form of parallelism. It is not usually regarded as a model of parallel computation, however. Some form of communication between computational resources is typically regarded as the basic entry requirement for being classed as a parallel computer.

There are two main ways of sharing data between a pair of processors; *shared memory* and *message passing*. The *Parallel Random Access Machine* (*PRAM*) model [112] represents the first method. PRAM describes a MIMD architecture with a number of independent processors with a shared clock and an unbounded memory space; the MIMD equivalent of an *unlimited register machine*. PRAM architectures can be categorised according to the capability of the memory; whether each location can be read from or written to concurrently. In modern machines, most RAM has no capability for concurrency, although specialised *Video RAM* (*VRAM*) has been designed to allow the DAC connected to the screen to read its contents while the CPU updated the contents, removing the need to lock the memory as the display was scanned. This has gone out of fashion in recent years as RAM prices have dropped enough to make double buffering the norm.

More recent models for parallel architectures have focussed on the layout of the memory, describing architectures according to whether access to the memory has a uniform cost [280, 50]. *Uniform Memory Architecture* (*UMA*) machines are those for which it does. UMA designs are typically found in systems which do not exhibit a large degree of parallelism[1]. The reason for this is that maintaining a uniform access cost for large numbers of processors to large amounts of memory requires either $n \times m$ interconnects, where n is the number of processors and m is the number of banks of memory, or a large constant cost. UMA systems where the processing units are heterogeneous are referred to as *symmetric multiprocessing* (*SMP*) systems, since a task can run as easily on any processing unit. Volume visualisation often relies on memory systems supplying conflict-free simultaneous access to multiple voxel values in a volume dataset [255].

Non-uniform memory architecture (*NUMA*) systems exhibit greater scalability [345]. In a NUMA architecture, there is some fast, local, memory for each processing resource. The speed of access to this memory is constant, irrespective of the number of nodes in the system. In contrast, adding one node to a UMA system either increases the cost of memory access to all nodes, or increases the cost of building the system (often by a large amount). The largest disadvantage of a NUMA system is that it requires locality of reference to be used effectively, delegating data distribution to the programmer. This makes NUMA machines more of a challenge to effectively utilise.

The other model for sharing data is *message passing*. A message passing system partitions the address space so that each processor has its own private memory and explicitly sends and receives data from others. While these two models seem different, it is important to

[1]What constitutes a ‘large degree’ depends largely on the year in which the system was designed.

note that it is relatively easy to implement one using the other. Shared memory can have message queues stored in it, and accesses to remote memory regions can be trapped and fetched using a message passing mechanism. It is important to distinguish between the underlying mechanism, and that exposed to the programmer.

A *distributed memory* system is one in which each processor has its own private memory, which is fast and not shared. A message passing mechanism is then used for non-local access. The Cray T3D is an archetypal example of this. A more modern implementation of this idea is found in the Cell microprocessor where each *synergistic processing unit* (*SPU*), of which there are eight on each die, has 256KB of very fast local memory and uses DMA transfers to access the system's main memory. A similar low-level design is used on most modern processors, although the fast local memory is not directly accessibly by the programmer; instead it is used as *cache* for the main memory, and the contents are determined at run-time by the cache controller. Distributed memory systems are often regarded as difficult to program, since they do not map to conventional programming languages very easily. CSP-based languages, such as Erlang [23] and Termite [123], are well suited to targeting these architectures, since their semantics more closely match the underlying hardware. In spite of this, distributed memory systems are popular for very high-end systems due to their ability to scale to many thousands of processors [299]. Distributed memory systems can have their nodes arranged in a variety of configurations.

Most UMA systems are *shared memory* systems at the low level. These typically make use of a distributed memory hierarchy, where each processor has a local cache that is not accessible to others. Maintaining coherence between these caches requires a *cache coherence protocol*. In the absence of cache coherence, a modification to a shared memory region might not be detected by one of the processors [299]. Some NUMA architectures also implement a cache coherency protocol. These are referred to as *cache coherent NUMA* (*ccNUMA*). The one of the most widespread ccNUMA architectures currently is AMD's Opteron, where each CPU has a memory controller and communicates with the others over a message-passing point-to-point HyperTransport interconnect. When a memory location is updated, the cache lines containing it in all of the connected processors are flushed. This is a good example of the importance of distinguishing between the implementation and the programmer-visible design, since most operating systems expose Opteron machines as UMA systems to the programmer.

2.3.1.3 Programming Models

Conceptually, the simplest model for programming parallel systems is shared memory, where each thread of execution has access to the same memory resources. While this is conceptually simple, it has several limitations. The most obvious is that it places a significant burden on the programmer, in terms of synchronisation. The second is that is is not ideal for NUMA systems. A shared memory system, where communication latencies are significantly under 1ms can present a shared memory view to a programmer using a structure such as a *dynamic interconnection network* [297] or a *crossbar switch* such as that found on the Cray Y-MP to keep latencies relatively constant. On NUMA systems, however, the cost of accessing memory can vary widely. This makes reasoning about performance very difficult [121].

Shared memory does not have to be exposed as an untyped melange of data. Linda [55] is a coordination system, which exposes a *shared tuple space* to developers. Linda supports C and Fortran. A similar model is available for Java in the form of JavaSpaces. The SR language [16] supports both shared-address-space paradigm and messaging passing paradigm.

Message passing as a programming model scales better than pure shared memory, and maps more cleanly to the underlying mechanisms of NUMA systems. Some languages, such as Erlang and Termite, have been designed around this model, but they require significant changes to the design of many existing algorithms. These languages are based on the *communicating sequential processes* formalism, where a problem is decomposed into independent parts, with no shared resources, that communicate by passing messages.

Message passing mechanisms have been attached to other languages at the library level. The most popular for high-performance visualisation, and computing in general, is *Message Passing Interface* (*MPI*) [51]. This specification has been implemented for a variety of languages, including C/C++ and Fortran.

An alternative is the *Parallel Virtual Machine* (*PVM*), from Oak Ridge National Laboratories. This uses an abstract virtual machine as the target, and an emulator for this machine. Although this incurs some performance penalty, it provides good fault tollerance and recovery, and so remains relatively popular [122]. A similar approach is taken by a derivative of the Plan 9 operating system, *Inferno*™ [99]. Inferno™ runs both as an operating system and in 'hosted mode' as a process in other operating systems, and is designed to run distributed applications written in the *Limbo* programming language and just-in-time compiled for the native platform.

In the late 1970s, Alan Kay proposed a method of software development where programs would be described as running on 'simple computers that communicate by message passing.' He termed this model *object oriented programming*. At the time, most computers had a single CPU and so the most efficient method for implementing message passing was synchronously, atop function calls. Languages such as Smalltalk built on this principle, however, present message passing as an abstraction and so allow different approaches to be used. Messages in Smalltalk-like languages can easily be replaced with trampolines which return a proxy object, which then blocks on access, implicitly extracting parallelism from synchronous algorithms[2].

Other languages, such as Java and C++ adopt some aspects of object orientation, and replace function calls with method invocations as the basic primitive for controlling program flow. Function calls and method invocations are very similar, both in implementation and use. A method invocation in these languages is basically a function call scoped to an instantiation of an abstract data type; a verb to the object's noun.

[2]An implementation of this idea in Objective-C by the author of this thesis is available from http://etoile-project.org

Function calls and method invocations are, at the local level, synchronous. Both have been extended to support parallel computation, however. Function calls have been extended as *remote procedure calls* (*RPC*). A number of implementations of these exist, with the most common being those of Sun and Microsoft. While the procedure calls themselves complete synchronously, it is common for them to be used to initiate a longer process which then completes asynchronously.

The concept of RPC was extended by NeXT Computers to give *Portable Distributed Objects* (*PDO*). The PDO system allowed both synchronous and asynchronous method calls to objects on remote machines. Since objects were passed as parameters, this gave more functionality than procedure calls. PDO was limited by the fact that it primarily worked on compiled languages, however which forced the classes to be distributed out-of-band before the system started.

The Java implementation, *remote method invocation* (*RMI*) overcame this limitation. Java classes are compiled as bytecode, which is machine-agnostic. This means that objects can be passed as parameters irrespective of whether their underling code had been distributed before hand. If they had not, then the bytecode could be distributed between nodes in the network and run.

Two PDO-like systems gained widespread adoption. The *Common Object Request Broker Architecture* (*CORBA*) [25] is a specification for method invocations across language and process boundaries. Microsoft's *Distributed Common Object Model* (*DCOM*) goes a step further, and specifies a binary interface, allowing DCOM objects to interact independently of the *object resource broker* on which they are running (although, in practice, the only DCOM implementation that has seen any widespread use is Microsoft's own). CORBA and COM, like PDO, use the object as the 'atom' — the unsplitable component — in the system. Globe [313] goes a step further, and permits a single object to be distributed amongst nodes.

In recent years, there has been a growth in distributed middleware, including coordination-based systems such as Jini [240], and document-based designs such as Globus [114]. Many of these use XML at the protocol level [43], especially those based on web, and later grid-service architectures.

The idea of *dataflow computation* [275, 276] is an alternate model for parallel programming, in which execution is driven by the dependencies within the data. The Polytypic Grid [100] provides a similar model, making use of the implicit parallelism that can be extracted from functional languages (moving from λ-calculus to π-calculus for the underlying formalism) in combination with lazy evaluation to perform on-demand execution of a parallel graphics pipeline.

Dataflow computation has seen a lot of use in the context of visualisation, as it maps well to the visualisation pipeline abstraction. Implementations including OpenDX [2], AVS [311], IRIS Explorer [117], SCIRun [243] and DDV [222] are all examples of dataflow environments aimed at visualisation. They treat the visualisation pipeline as a directed graph, where each node represents a computation. This provides support for dataflow parallelism [285], since each node can be executed in parallel. There are some limitations to most implementations of this approach, such as the inability to handle partial datasets [6], although this is a practical, rather than theoretical, limitation to the model.

One final model, that of *stream-based computation* provides an approach similar to data parallelism, but with some aspects of control parallelism. A stream-parallel system is composed of stream processing units, which take streams of commands and data and execute in parallel. This model is used by Chromium [149], which processes streams of OpenGL commands (which can be control or data, but are more often data). A set of extensions to VTK based on Chromium have also been proposed [223].

2.3.1.4 Measuring Performance

The main reason for moving to parallel computation is to achieve better performance, and so it is important to be able to measure the increase on performance derived from the addition of extra processing units.

One of the most popular metrics is *speedup* [154]. This measures the increase in speed per added processing resource. Speedup is calculated as the ratio of the serial and parallel runtimes, multiplied by the number of processors. In the parallel case, the total time of all processors is taken, so a task which takes two seconds on each of two processors would have a parallel run time of four seconds. In the ideal case, the speedup for an algorithm will be the number of processors (P). Note that in many cases speedup is a function of P, rather than a constant. Some algorithms have a 'natural' degree of parallelism, and have a speedup of the order of P up to this value, after which little benefit is gained from adding more processors. In theory, the speedup can not exceed P, however in practice, concurrent applications may incur some overhead from context switching on a single processor that is removed when they are run on more. This is often apparent on highly-concurrent applications written in languages such as Erlang. The upper bound on speedup is given by Amdahl's Law [12]. Given a problem of size w that has a sequential fraction of size w_s and a parallel fraction of size $w_p = w - w_s$, the upper bound of speedup is w/w_s, regardless the number of processors.

It is important to track the efficiency of a parallel approach. The *efficiency* of parallelisation is defined as the speedup divided by the number of processors. In the ideal case, the efficiency will approach 1. In general (for most non-trivial algorithms), it tends towards 0 as P tends towards infinity. At some point, it ceases to be economical to keep adding processors to a problem. Another way of determining this is the *cost* metric. This is the total computation time needed (the run time multiplied by the number of processors). If the speedup is less than P (i.e. the efficiency is less than one) then the cost will grow as P grows.

The optimum value of P can be determined by the *scalability* [173] metric. This measures the rate of change of speedup, in terms of P. The *isoefficiency function* is an alternate metric based on the size of the problem required to maintain a constant efficiency, based on a known *overhead*. An isoefficiency function takes into account the structure of the parallel computational resources, including the communication speeds. A scalable parallel system is usually indicated by a small efficiency function [126].

For some applications, the *time-constrained scalability* [133] of the problem is a very important metric. This allows a fixed run-time to be defined, and reached by adjusting the size of the data. Similarly, *memory-constrained scalability* describes the way in which the memory usage of an algorithm can be adjusted in terms of its input data. This can be very important when running an algorithm on real hardware. In the case of visualisation, it is often possible to employ *level of detail* techniques to reduce the data size.

When describing parallel systems, it is important to be aware of the *granularity* of the parallelism. This describes the ratio of the number of processing units to the capacity of each. The smallest capacity is a single instruction, and most modern processors exhibit some degree of parallelism at this level, known as *instruction level parallelism* (*ILP*). This is the finest-grained form of parallelism that is likely to be encountered.

2.3.1.5 Parallel Visualisation

So far, parallel programming has been discussed in the general case, rather than the context of visualisation. This section will attempt to discuss how some of the concepts have been applied by the visualisation community

When creating a parallel system of any form, it is necessary to decompose the problem into smaller parts. In the scope of visualisation, this is usually done using either *image space* or *object space* [127] decomposition. These relate to how the scene is distributed across rendering nodes, either partitioning the input (object space) or the output (image space).

Image space approaches are often called *sort-first* algorithms, since the primitives are sorted during the rendering of the sub-images. *Sort-last* algorithms [220] are usually object-space. Different sets of objects are rendered to produce simpler objects which must then be composited (depth-sorted) together to produce the final image.

A parallel rendering task typically faces a problem when it comes to data distribution. Object space decomposition allows parts of the scene to be placed logically close to the processing resources rendering them, which helps to reduce the communication overhead. This is more true of visualisation tasks than many more general computational tasks, since visualisation typically involves a large reduction in the data size (for example, turning a large volume dataset into a 2D image). Image space decomposition typically requires all of the nodes to be able to access the entire dataset, which can allow more efficient distribution of tasks at the cost of a greater communication overhead.

For systems that do not have shared memory at the hardware level, distributing the data efficiently is important to minimise the communication overhead used. It is often possible exploit some characteristics of the data being visualised in order to produce an efficient distribution strategy. Some partitioning strategies take advantage of spacial locality [203], others image and frame coherence [124], and overlapping and exchange of boundary data [227]. These strategies are not easy to generalise. Typically, a distribution strategy is tied to a particular data type, and often to the rendering algorithm and even the underlying architecture of the system. Such strategies can be classified according to the type of rendering they support; image-space [203], object-space [337], or hybrid approaches [178]. Alternatively, a taxonomy based on the method of splitting the data can be used.

The simplest form of distribution is *complete data replication*, where a copy of the entire dataset is placed on each rendering node. This is common for image-space decomposition and rendering of read-only images. The cost, however, scales linearly in terms of the number of nodes, which can be prohibitively expensive, particularly on cheap rendering nodes which generally have fairly modest local storage abilities.

The next form is *block replications*, where the data is split based on coarse-grained slices across the object space. Typically, the blocks overlap, so that each node has slightly more than $1/p$ of the data. The overlapping area is typically much smaller than the size of the block, however, and so adding more nodes reduces the amount of data on each node, meaning the total storage space required scales much better than linearly. For workloads where the distribution of data is non-uniform, an *irregular block decomposition* method can be used to make blocks which have roughly the same number of features inside, and so have roughly equivalent rendering times.

The most complex distribution strategies are *structured* or *hierarchical partitioning* methods. These rely on an overlay structure used to organise the raw data. One of the simplest forms is an *occupancy map* [211, 159]. This makes use of an overlay structure which is a simple mapping from blocks to a boolean variable, indicating whether the block contains data that should be rendered. This can be used at a fairly coarse granularity to quickly eliminate empty regions.

The simplest forms of hierarchical partitioning are structures like *octrees* [97, 69]. These divide a region into a number (eight, for an octree) of equal-sized sub-regions. These are recursively re-divided until a threshold is reached, such as a minimum number of features (e.g. vertexes) in a leaf node. Kd-trees [31] provide a similar method of partitioning, although they split the data in such a way that the there is always an equal number of points in each node at a given depth, by placing the splitting plane on the median point. Hierarchical structures can be used to divide data on a variety of attributes, including spacial occupancy and rendering workload [316].

Structural partitioning is typically associated with object-space methods, although it is possible to make use of some image-space parallel rendering techniques with them, as they can be used to produce view-dependent information for efficient data access [194]. On generated data, the *scene graph* abstraction is often used to compose objects in 3D space. These can be used for managing sort-first, distributed memory parallel visualisation [32] and have been used in the context of real-time virtual reality [228].

After the data has been split, it is necessary to assign parts of the rendering task to different nodes for parallel execution. There two approaches to this problem. *Static task assignment* [337, 201] maps tasks to nodes at the start of execution. This works well for rendering algorithms that have a fixed, or easily computable, cost for a given set of input data. A preprocessing step is required to generate the mapping, but then the task can run with no further intervention. The alternative is *dynamic task assignment* [178], where tasks are assigned to to nodes as the job runs. This is typically done by splitting the rendering job into (many) more components than there are processing nodes, and issuing new ones from a queue to idle nodes.

The former is more commonly associated with object-space approaches, where a known amount of data is rendered by each node. The later is better suited to image-space algorithms, where rendering times are not constant. Consider the extreme case of a parallel ray tracer, where each ray is an individual task. Some rays will pass directly through the image, and be very cheap to render. Some will interact with complex structures and spawn secondary and tertiary rays, possibly in large numbers. Assigning a group of rays to each processor at the start of rendering would almost certainly cause some to terminate and sit idle before others.

The final step in most parallel rendering tasks is *image composition*. This is the process of re-assembling the results of the parallel rendering tasks into a single image, and often presents a bottleneck. The simple *direct send* method, where the results are sent directly to a compositing node, can cause some significant bottlenecks by saturating the compositing node's bandwidth. An alternative is to use a parallel composition algorithm [188, 337], where the image is gradually assembled in a hierarchical fashion. Another option is a scheduled linear image compositing algorithm, as a highly optimized direct send method [289], which offers better scaling to large numbers of processors. Finally, dedicated stream-processing hardware can be used for compositing, although this typically increases the cost.

2.3.2 Towards Local Visualisation

It is axiomatic that computers get smaller over time. Early visualisation was performed on large mainframe or supercomputing resources. This started to change in the late 1970s, with the arrival of the graphics workstation. These machines had the processing resources required for simple (by todays standards) visualisation tasks in the same device as a display. This reduced latency, and allowed more interactive visualisations. In the same timeframe, network-transparent graphics systems, such as NeWS and X11, were developed. These allowed similar interactivity to be gained over a fast network.

In the '80s, graphical computing began to take off in the mainstream, with the widespread deployment of GUI-based operating environments. Even cheap home computers began to have framebuffers, and some had more complex graphics acceleration, such as sprite-rendering hardware and simple vector drawing accelerators. At the higher-end, 3D visualisation became possible on workstation-class hardware. Volume rendering became practical using isosurface extraction techniques such as Marching Cubes [199] and direct volume rendering methods such as volume ray casting [191].

From the late 1980s to the early 1990s, a number of commercial products targetting visualisation began to appear, such as AVS [311], aPE [102] and Khoros [254], later joined by IRIS Explorer [117] and TGS Amira [302]. These use a visual representation of a visualisation pipeline as a user interface. They generally support a variety of modules, including some supplied by third parties, leading to their being described as *modular visualisation environments*. By presenting a visual model, rather than a programatic one, they lowered the barrier to entry for users wishing to exploit the capabilities of a visualisation environment.

These visualisation tools were regarded as the 'killer application' for high-end workstations for some time. Visualisation required large amounts of processing power, but the requirement for low latency made workstations much more appealing than thin-client solutions. Most modular visualisation environments were developed around this time to fuel demand for visual workstations, and were given away with the workstations, or made available at a very low cost. AVS was developed by Ardent to sell their workstations, and IRIS Explorer to sell Silicon Graphics workstations[3].

Just as graphics workstations took over from larger machines, their market began to be eroded by commodity hardware. This demand was largely driven by the gaming industry, which constantly demanded higher-quality visual effects than were possible with purely CPU-based graphics algorithms. Initially, 2D vector accelerators were introduced into commodity PCs. These accelerated simple line-drawing functions, and were often marketed as 'windows accelerators,' since they accelerated the drawing operations used by a GUI. Dedicated 3D hardware was introduced into the commodity market by manufacturers such as PowerVR (now owned by Intel) and 3dfx (now owned by nVidia), who provided support for texturing in hardware. Later, nVidia added support for offloading transform and lighting calculations to the *Graphics Processing Unit* (*GPU*). This was later extended to provide fragment, vertex and finally geometry shader support in hardware. Modern GPUs have a completely programmable rasterisation pipeline, and are sufficiently flexible that they are now being used for non-visualisation tasks.

This has been compounded in the last few years with the incorporation of graphics hardware into PDA form factor devices. These tend to be significantly slower than their desktop counterparts, however they are also typically limited by much lower bandwidth for transmitting images from a remote renderer, making the decision between local and remote rendering far less clear-cut.

2.3.3 Hardware for Visualisation

Visualisation tasks can often be offloaded effectively onto dedicated hardware. While dedicated silicon is generally more expensive than general purpose hardware, it can be significantly faster. When RAM was very expensive, dual-ported memory called *VRAM* [252] was popular. This had a read-write port for the CPU and a read-only port for the DAC. This allowed the CPU to write to the frame buffer while the display hardware was scanning it for output, without requiring any locking. This was obsoleted when RAM prices dropped sufficiently that two frame buffers could be used, and switched between.

[3]Silicon Graphics changed its name to SGI later, when the company began to focus more on the general HPC market than on visualisation

The CPU would write to one, while the display hardware read the other, and then they would be flipped. This also helped reduce visual artefacts, since the current frame will not be updated while it is being displayed (as long as the switching is synchronoised with the flyback period).

One of the earliest processors used as a dedicated graphics processor was the Intel i860 [129]. The i860 was intended as a general purpose CPU for workstation use, and was intended to replace the 8086-compatible line. It could achieve a theoretical peak of 66MFLOPS, compared to the 33MFLOPS of the i486 at the time. In general purpose use, particularly with compiler-generated code, the i486 performed better, however. This, combined with the very high context-switching overhead of the i860 caused it to be abandoned as a general purpose CPU. The instruction set was based on VLIW principles. This made it very good for running jobs that included large groups of unrelated instructions (since the compiler had to arrange them into bundles that could be executed in parallel). This made it particularly well suited to a number of graphics tasks, such as vector drawing. The chip was included in the NeXTdimension expansion board for the NeXT Cube. In this configuration, it ran a Mach kernel with a PostScript interpreter. Display elements would be sent to it as PostScript programs, which executed very quickly. Some systems, such as the IRIS Power series from SGI [8] moved in the opposite direction, and included large numbers of general purpose chips in a parallel configuration.

Texture mapping hardware began to appear in consumer hardware in the early '90s. This did not offload any of the geometry calculations from the main CPU, but did allow some more rendering techniques to be explored. Flow visualisation [259, 300, 192] took advantage of texture mapping to display flows on top of simple geometric primitives representing the structure of the flow. Splatting [296] used textures representing a 2D projection of a sphere (i.e. a shaded circle) mapped onto square primitives to quickly render point data sets, and other point-based techniques also made use of texturing hardware [249]. Recent consumer cards (and older workstation hardware) also includes support for 3D textures. These take up large amounts of memory; a 512 voxel cube in 32-bit colour will require 512MB of memory to store, without compression.

Modern GPUs are not exactly dedicated graphics hardware. They are highly parallel stream processors. They have some limitations over general-purpose hardware, such as the relatively high cost of branching. Early generations of GPU could not execute branches at all. Later ones did so by simply executing both parts and then throwing away the wrong result. The newer cards support branching natively, although their very long pipelines make them comparatively expensive. In spite of these limitations, a number of applications such as volume rendering and ray casting [212, 258, 264, 227, 89] have been ported to GPU architectures. Their shortcomings are largely made up for by the fact that they have very high throughput.

Modern GPUs have an interesting memory architecture. They generally have their own RAM and the ability to access the system RAM. Those connected via AGP are more able to quickly move data from main memory to their local storage, but moving in the other direction is much slower. PCI express addresses this by providing much more, symmetrical, bandwidth [333]. Roadmaps from both Intel and AMD[4] indicate that future x86 chips will incorporate GPU-like functionality, giving the GPU the same memory bandwidth as the CPU. In the interim, some pre-processing approaches have been proposed that help alleviate this bottleneck [82].

Hardware acceleration is not limited to surface rasterisation. Dedicated hardware such as the TeraRecon VolumePro allow real-time direct rendering of volume datasets [247]. This is based on an earlier research project in the same area [248], and delivers up to 30fps on data sets up to 512 voxels along each side.

As discussed earlier, 'more processors' tends to be cheaper than 'faster processors' and GPUs are no exception. A number of approaches have been proposed to take advantage of multiple GPUs in a cluster configuration [343, 227]. This has lead to dedicated designs such as the WireGL [148], which evolved into Chromium [149], allowing an OpenGL command stream to be transparently distributed amongst nodes in a cluster. Additional work has been done by others on using multiple CPUs in a cluster [219] and commercial solutions are available. Visualisation toolkits such as OpenRM and VTK have been integrated with Chromium [32, 223] and other commercial visualisation products, such as Mod-viz [218] have been developed for cluster deployment. On the hardware side, Sun Fire Visual Grid [293] and IBM DeepView [167] provide similar strategies from a hardware perspective.

Trends towards dedicated and general purpose hardware tend to be cyclic. At the time of writing, it appears that the trend is about to start its swing back towards general purpose hardware, with the next generation CPUs including design elements taken from GPUs and extensions to their instruction sets aimed towards graphics applications.

2.4 Applications of Visual Supercomputing

Visualisation is, first and foremost, a practical discipline. Technology is only interesting in the context of applications. There have been a number of trends in recent years that lead to the conclusion that a large-scale visualisation infrastructure is needed.

2.4.1 Alternative Reality

Total immersion virtual reality environments have been a dream of the visualisation community since the 1980s. This places a very large demand on computation [49, 281, 278]; either a very large display is required, or a smaller one with very frequent updates for movement (or some combination of the two).

Head mounted displays are fairly common for VR applications. They present a different screen to each eye, with a rendering from a slightly different viewpoint. The motion of the user's head must be tracked to maintain the illusion of immersion. For multiple users,

[4]Who now own ATi

projection-based approaches project the stereoscopic images onto a screen [283]. These can be projected at the same time, and filtered by coloured or polarised lenses, or protected alternately and split by shutter-glasses worn by the users. Since these displays do not track the motion of multiple users, there are only a small number of situations in which they are useful. Another alternative is to use a 'true' 3D display, where the individual pixels are drawn in a tank in 3D space, either using a holographic display [283], or some mechanism involving a rapidly rotating screen [36]. Since these draw an entire volume every frame, they require a huge amount of bandwidth, and are limited to the kind of 3D scene that can fit inside the tank. At the opposite extreme, it is possible to use very high resolution displays (for example, 63 million pixels [223]) to present large scenes in which there is little parallax. All of these approaches require a significant amount of computational power to maintain the illusion of presence.

The other half of the problem relates to input. Presenting a 3D world is only of use if it can be interacted with, and this helps maintain the illusion of physical immersion [290]. The simplest input devices are 3D mice, which track their position in three dimensions, or gloves. These work well in one direction — the computer can track the user — but do not work well in terms of feedback to the user [281]. Haptic systems, such as the phantom stylus [290], provide a better sense of interaction, since the computer and user can both affect the other.

The computational resources required to maintain a fully interactive visual and haptic environment are huge. The *Cave Automatic Virtual Environment* (*CAVE*) system provides a visually immersive environment by projecting a stereoscopic image onto each wall of a room, and tracking the head movements of the user. A typical CAVE is powered by a 12-CPU SGI Onyx and special purpose software such as DIVERSE [162] for management.

The issue of management is almost as important as that of processing cost. While processing power for a given investment doubles roughly every 18 months, the cost of a human managing the system does not. In fact, the converse is true; as system complexity grows the skills required to manage the system become rarer and thus more expensive. A *self-managing* system is highly prized.

Virtual environments are particularly attractive for collaboration, since they can present the illusion of physical proximity between remote collaborators. Examples of collaborative environments include DVIE [53], MASSIVE [128], VRML-extension [44], COVEN [236], DEVRL [282], and VirtuOsi [30]. These tend to adopt the *virtual world* model. This places great demands on the infrastructure to ensure that the individual users see a consistent view of the world, both in terms of rendering at the edges and ensuring distributing the data between them.

In many situations, a complete artificial world is not required, just some slight modifications to the existing one. The solution to this is *augmented reality*, where a virtual world is overlaid on the real one. AR typically uses a transparent display between the user and the real world [251] and a camera tracking objects in the real scene [182]. Examples of systems using this include th InfoTable [257] which detects objects and allows arbitrary metadata associated with them to be displayed. AR can also be used in a collaborative context, allowing users to see the same virtual object [271].

The Grid is becoming more and more important in the field of visualisation, particularly when computational resources required for real time interaction in a virtual environment are not locally available. Also the popular component-based programming paradigm, which has been adopted by many visualisation systems such as VTK, AVS and OpenDX, can make use Grid resources. This allows different computation steps of a visualisation pipeline to be distributed around the globe [274]. Other visualisation systems such as EnSight Gold [1] and VisIt [70, 318] also support parallel processing directly.

2.4.2 Distributed Visualisation

The biggest change to the computing landscape in the last decade has been the dramatic rise to prominence of the Internet, and more specifically the World Wide Web. The early web could display static images, and so presented a new mechanism for distributing the results of visualisation; a rendering could be generated on a large machine and then easily distributed to users without requiring them to install any specialised software.

The web has grown dramatically since the early days, both in size and capabilities. The ability of early browsers to launch 'helper applications' based on MIME types allowed custom applications for viewing visualisations to be triggered by visiting a web page [17]. The growth of plug-in architectures since then has allowed more complex visualisation tasks to be run from within the browser, using technologies such as VRML, X3D and Java.

The web gives rise to two conceptual structures for visualisation. The first, *server-side visualisation*, has been possible since the first web browsers; a user sends a request for a web page and receives a rendering as a result. This has been exploited by systems such as CGI interfaces to existing visualisation tools [342, 105]. The alternative is to delegate the rendering to the client side [215]. In this case, the web is used to distribute the data used for *client-side visualisation*. Both of these involve trade-offs. Server-side visualisations can be significantly more complicated, since the web server can be a front-end to a large rendering farm, while client-side renderings are limited both by the power of the client and the amount of data that can be sent over a network connection. The advantage of client-side visualisation is the low latency between the user and the renderer, giving much better interactivity.

These two approaches have, in combination, given rise to the web-services approach. This is similar to the server-side approach, but allows multiple servers. Data can be stored in one place and different stages in the visualisation pipeline can be run on different machines in a loosely-coupled environment.

Beyond the web, a number of other approaches for remote visualisation via the Internet have been developed. ORL's Virtual Network Computer (VNC) [322] uses a screen-scraping mechanism to send a copy of the framebuffer to a remote user, and remotely send mouse and keyboard input to workstation.

VNC provides basic collaboration abilities, however it does not provide any means of mediating access to resources. Different levels of collaboration are possible within distributed systems. The simplest is *data sharing*, which is supported by environments such as CUMULVS [229] and pV3 [253], where multiple viewers can inspect the same data, or COVISE [83] where geometry is shared, but collaborators retain their own viewpoint. This is in contrast with something like VNC, which provides *display sharing*, since each collaborator shares the same view of the data.

There is a growing need for collaborative visualisation, as the next section will explain, and some of the tools for building the required infrastructure are beginning to become available. The current state of visualisation, however, is not ready for deployment on the grid [274], and further work is required.

2.4.3 Visual Data Mining and Large-scale Data Visualisation

Never before in history have we had such capability for generating, collecting and storing digital data. Data repositories at terabyte level are becoming a common place in many applications, including bioinformatics, medicine, remote sensing and nano-technology. In some applications, such as network traffic visualisation [172] and video visualisation [87] (Figure 2.3), we are encountering the scenario that dynamic data streams are almost temporally unbounded. Many visualisation tasks are evolving into visual data mining processes [161].

These applications are placing a huge strain on the existing visualisation environments, and challenging the state of the art technologies in many ways. They demand a variety of infrastructural supports, such as,

- for providing sufficient run-time storage space to active visualisation tasks;
- managing complex data distribution mechanisms for parallel and distributed processing;
- choosing the most efficient algorithm according to the size of the problem;
- facilitating the search through a huge parameter space for the most effective visual representation.

Data management is the very first issue in handling large datasets. Many visualisation processes involve datasets that are much too large for the internal memory of a computer, and have to rely on external disk storage, usually under the virtual memory management of an operating system. The external disk access can become a serious bottleneck in terms of rendering speed. *Out-of-core algorithms* (also known as *external memory algorithms*) [321] are designed to solve a variety of batch and interactive computational problems by minimizing disk I/O overhead. Various out-of-core visualisation algorithms have been proposed to handle large structured and unstructured 3D datasets, for instance, in the context of (i) isosurface extraction [75, 67, 295], (ii) terrain rendering [194], (iii) streamline visualisation [308], (iv) mesh simplification [193], (v) rendering time-varying volume data [277], (vi) rendering unstructured volumetric grids [109], and (vii) ray tracing [250]. While some algorithms rely little on internal memory (e.g., [67, 109]), others utilize preprocessed data structures, such as octree [308] and indexing [277] to optimize disk I/O operations. Kurc et al. [175] recently reported their experience in visualizing large volume datasets using *Active Data Repository*, which is composed of a set of modular services and a unified interface for supporting the management of, and mapping between, in-core and out-core data.

There has been a similar amount of research effort, if not more, for developing techniques that synthesize a visualisation image using less than the full dataset. Two commonly used approaches for determining a subset of data to be visualized are *multi-resolution* and *view-dependent* data organisation.

Multi-resolution data organisation makes use of various hierarchical spatial structures to manage *levels-of-details* (LODs) of a graphical model or scene. Such structures facilitate real-time rendering by allowing an appropriate LOD to be selected according to the requirements of interactivity and the constraints of computational resources. In computer graphics and visualisation, there exists a large collection of works based on this approach. For example, octrees and min-max indexing were used for isosurface extraction [334, 155, 75]. Laur and Hanrahan [184] utilized an octree for progressive refinement in splatting. Wilhelms et al. [335] employed a k-D tree for direct rendering irregular and multiple volumetric grids.

View-dependent data organisation makes use of the concepts and algorithms of hidden surface removal, and prioritizes geometrical primitives according to their visibility to the viewer. For example, Livnat and Hansen [197] proposed a view-dependent isosurfacing algorithm. LaMar et al. [179] prioritized volume data based on its proximity to the viewer. Other view-dependent works include visible set estimation [168], visibility-based prefetching [82], and view-dependent progressive rendering [238].

While it is necessary to deal with problems arising from very large datasets, it is equally important to improve our capability for managing inter-related datasets in order to generate more meaningful visualisation. In computer graphics and computer aided design, *scene graphs*, built-upon the concept of constructive solid geometry, have played an indispensable role in combining simple objects into a complex object and bringing many objects together into a scene. It is common for graphics systems to support scene graphs, for instance, in RenderMan, OpenGL, OpenRM, VRML, Java3D, POV-ray and Open Scene Graph. However, support for combinational modelling in visualisation systems [32, 228] is largely based on surface-based scene graphs, relying on image-space composition. Early research efforts for modelling complex visualisations involving multiple datasets were focused on voxelisation [327]. In order to address the problems associated with voxelisation [156], such as excessive data size and data degeneration, Chen and Tucker [60] outlined the concept of *constructive volume geometry* for combining volumetric datasets and procedurally defined scalar fields. vlib [336], an open source volume graphics API, offers volumetric scene graphs as its fundamental data structure, and provides a discrete ray tracer for direct rendering *volumetric scene graphs*.

In large-scale data visualisation, high performance rendering techniques, such as massively parallel rendering (e.g., [202]), progressive rendering (e.g., [184]) and stream-based rendering (e.g., [149]), are essential to the the process of *making displayable by a computer* (Figure 2.1). However, facing very large datasets, *making meaningful in-*

formation visible to one's eyes is often more critical in visualisation. With very large datasets, 'meaningful information' is often featured in a visualisation at a sub-pixel level, in a large amount or in four or higher dimensions. This challenges us to develop visualisation techniques into tools for visual data mining [161].

A popular approach to the handling of a huge amount of visual information is the use of *focus and context* techniques, which highlights a 'focus' in detail and depict its 'context' with less details to provide an overview. Focus and context techniques such as *fisheye views* [268], *perspective wall* [204], *hyperbolic space* [226] and *rubber sheets* [269], have been deployed extensively in information visualisation. This approach has also been employed in scientific visualisation, deformation-based volume visualisation by Kurzion and Yagel [177], distortion viewing by Carpendale et al. [54], non-photorealistic rendering by Treavett and Chen [305], magnification lens by LaMar et al. [180], two-level rendering by Hadwiger et al. [134], digital dissection in cardiac visualisation by Chen et al. [61].

Data mining should be closely coupled with visualisation [339]. Interactive visualisation is an indispensable tool in many data mining activities [139, 142]. interactive visualisation of large datasets not only demands sufficient computational resources, but also requires effective interactive techniques for data exploration, view navigation, data segmentation, data filtering, data fusion and direct manipulation [161].

Perhaps one of the main challenges in the coming years is *computer-assisted design of visual representations*. Many techniques in information visualisation enable automated placement of information in a visualisation, for instance, *treemap* [279] and *Sunburst* [288] in hierarchy visualisation, *recursive pattern* [160] and *circle segments* [18] in time-series visualisation, and *spring models* [303] and *Kohonen networks* [170] for self-organisation and self-optimisation in the entire information space. In volume visualisation, initial attempts have been made to automate the specification of transfer functions. Marks et al. [206] proposed a *design galleries* approach to the problem, while Kindlmann and Durkin [166] developed a semi-automatic method for generating transfer functions.

The problems surrounding large scale data visualisation are collectively becoming an infrastructure issue, as it is unlikely an individual technique can provide a satisfactory solution alone. To process such large amounts of data at the speed required, it is necessary for a visual supercomputing infrastructure to provide dedicated computational resources and application software systems. It is no doubt useful for the infrastructure to select appropriate modelling, processing and rendering techniques according to the available resources and interaction requirements. It is also highly desirable for the infrastructure to offer a wide range of tools for visual data mining as such activities are often unplanned and the effectiveness of a particular tool cannot always be pre-determined.

2.4.4 Scientific Computation and Computational Steering

The scientific community is one of the largest consumers of visualisation. It is unusual to use visualisation in isolation. More typically, it is integrated with a simulation system or *problem solving environment*. Such an environment is usually designed in a domain-specific manner, targeting a particular class of problem [119]. Cactus is an example of an open source problem solving system used by physicists and engineers [10]. In scientific modelling, visualisation is one component in a *feedback loop* [311]. These feedback systems typically fit into one of the following categories, a taxonomy proposed by Marshall et al. [207].

Post-processing uses visualisation after the simulation has run, simply to display the results. The user has no control over the simulation, and can not abort it or modify the parameters as a result of the visualisation, without running the simulation again. Since these visualisations run offline, they may use expensive pre-processing steps to achieve interactive framerates during the visualisation.

Tracking refers to visualisations that run during the simulation, but have no direct feedback mechanism. A user may use a tracking visualisation to check that a simulation is progressing adequately, but not as a control system.

Steering is the most complex of the three. In a steering environment, the visualisation component is part of the user interface, and decisions that affect the simulation can be made by the user at run-time based on the output of the visualisation. This can also be used to provide an audit trail [38], where checkpoints from various steering phases are stored. A steering environment using visualisation can be quite different to a conventional user interface. The parameters of the simulation can be modified by the environment based on very high-level responses to the visualisations made by the user, as in [346].

A steering environment that is flexible enough for all uses has been attempted [314, 312, 225] by adding steering widgets to visualisation systems, allowing the same generic tools used for visualisation to be used for controlling simulations. SCIRun, from the Scientific Computation and Imaging Institute at the University of Utah is a visualisation environment designed from the ground-up for steering tasks [243]. Other tools for this purpose include the *Collaborative User Migration, User Library for Visualisation and Steering* (*CUMULVS*) [229], which was developed to provide tools for scientific programmers. This framework is designed to allow steering and visualisation components to be integrated with large-scale simulations.

The RealityGrid project, as shown in Figure 2.4, is another project aimed at assisting steering applications. This has been used to build a number of different grid applications, including large scale Lattice-Boltzmann simulations [45] running on a large number of distributed machines.

The gViz project [340] has extended the IRIS Explorer application, described earlier, to allow components of the visualisation pipeline to run on different machines. In a grid environment, it is expected that communications between two machines are not necessarily private, and so a system like this is likely to require encryption, and a mechanism for authenticating trusted machines.

It is clear from the interaction between computation and visualisation that high-level interprocess communication mechanisms are required for the grid environment. These typically involve moving very large amounts of data between parts of the pipeline, and so the current grid services architecture, built using XML and HTTP, might not be appropriate. Systems such as Kepler [11] integrate scientific workflows with visualisation.

Simulations have been the focus of much of this section, but they are not the only things that require visualisation, nor are they the sole recipients of steering activities. The same feedback mechanisms can be applied to control real systems, such as experimental apparatus, medical systems or industrial equipment. A nuclear power plant, for example, is too dangerous to directly observe, and so can only be controlled based on output from sensors and is a good target for visualisation driven steering.

While simulations can typically be paused, interrupted, or restarted, real systems can (usually) not. They require real-time processing of datasets that are often very large, and low-latency feedback. This category of *mission critical visualisation* does not solely encompass systems requiring immediate feedback, but such systems do usually have a fixed time window between availability of data and the need for a user to make a decision. An example of this would be a flight simulator used to train pilots [284]. The simulator throws situations at a pilot, which must be visualised in time for the pilot to control the simulated aircraft in a way that reacts to the simulated situation.

The medical community is likely to be another area in which this form of visualisation will be used extensively. Visualisations from patient scans are commonly used during the planing phase of surgical procedures [246], and it is not a great imaginative leap to think that they could be used for training and even remote operations conducted by specialists on a different continent in the near future.

Figure 2.5 shows a system for delivering interactive volume interrogation of patient data in the operating theatre [209]. In this example, visualisation tasks are carried out on a large, remote visualisation server and delivered over the network.

This form of visualisation is typically conducted using custom (expensive) hardware at the moment, which makes it an ideal target for a visualisation infrastructure. The computing power available to a large scale infrastructure is likely to be sufficient to perform many of these tasks, however the management of such a system, the allocation of resources, and the interaction of components are still very much open problems. Quality of service concerns are significant for mission critical situations. In a lot of existing applications a few dropped frames are irritating. In a remote surgery application, a sudden drop in computing resource assignment or bandwidth allocation causing dropped frames or increased latency could potentially be fatal. Synchronisation and data distribution are also important problems [118], which must be addressed when designing an infrastructure for visualisation.

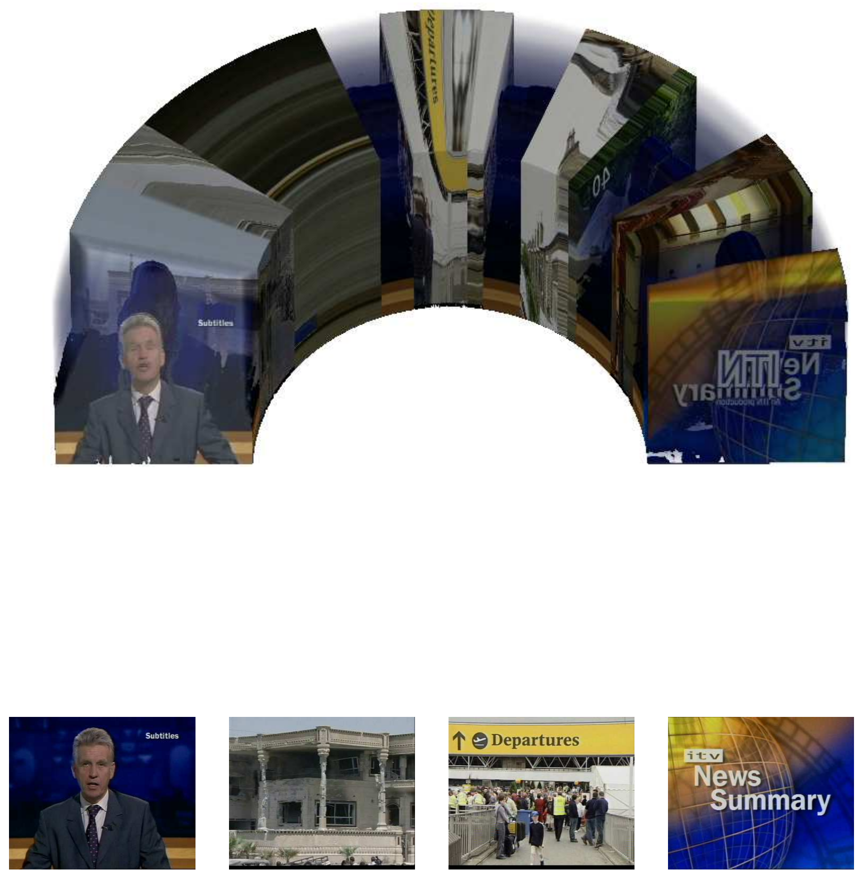
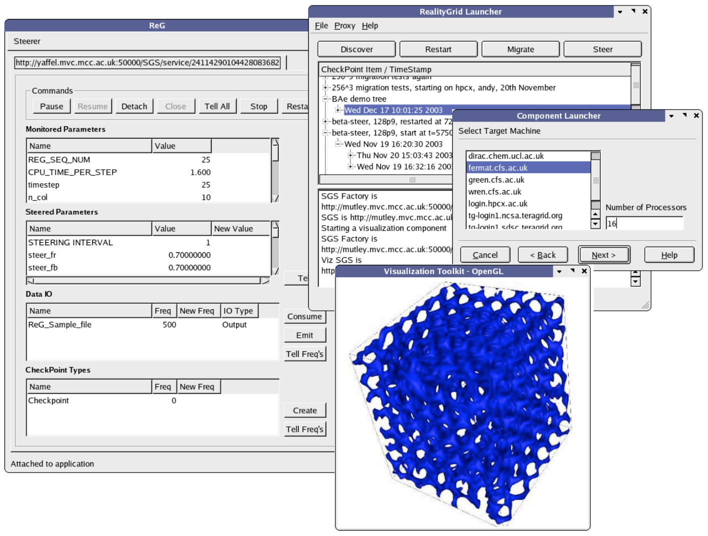


Figure 2.3: Video visualisation needs to deal with data streams of an arbitrarily large size. Stream-based rendering can be effectively deployed to visualize video streams [87].



http://yaffel.mvc.mcc.ac.uk:50000/SGS/service/24114290104428083682

Monitored Parameters

Name	Value
REG_SEQ_NUM	25
CPU_TIME_PER_STEP	1.600
timestep	25
n_col	10

Steered Parameters

Name	Value	New Value
STEERING INTERVAL	1	
steer_fr	0.70000000	
steer_fb	0.70000000	

Data IO

Name	Freq	New Freq	IO Type
ReG_Sample_file	500		Output

CheckPoint Types

Name	Freq	New Freq
Checkpoint	0	

Attached to application

CheckPoint Item / TimeStamp
- 256^3 migration tests, starting on hpcx, andy, 20th November
- BAe demo tree
  - Wed Dec 17 10:01:25 2003
- beta-steer, 128p9, restarted at 72
- beta-steer, 128p9, start at t=5750
  - Wed Nov 19 16:20:30 2003
    - Thu Nov 20 15:03:43 2003
    - Wed Nov 19 16:32:16 200

SGS Factory is
http://mutley.mvc.mcc.ac.uk:50000/
SGS is http://mutley.mvc.mcc.ac.uk
Starting a visualization component
SGS Factory is
http://mutley.mvc.mcc.ac.uk:50000/
Viz SGS is
http

Select Target Machine
- dirac.chem.ucl.ac.uk
- fermat.cfs.ac.uk
- green.cfs.ac.uk
- wren.cfs.ac.uk
- login.hpcx.ac.uk
- tg-login1.ncsa.teragrid.org
- tg-login1.sdsc.teragrid.org

Number of Processors: 16

Figure 2.4: A computational steering environment developed in the RealityGrid project [45].

Figure 2.5: Visualisation-guided surgery is a typical application of mission-critical visualisation [209].

Chapter 3

Autonomic Computing and Visual Supercomputing

3.1 The Question of Scale

The Grid concept refers to a distributed computing infrastructure for 'co-ordinated resource sharing' [116]. This extends the concept of the World Wide Web, which is a distributed computing infrastructure for sharing data. While a grid infrastructure allows sharing of data, it also allows sharing of storage space and computational resources.

The Grid is a large heterogeneous system in which different components with different resources and policies interact to further their own goals. Unlike cluster computing, the Grid idea does not include a single administrator responsible for the whole system, nor a single set of policies for use. A Grid infrastructure must provide a set of services [243], but the way in which these services are provided is not well defined. There are a number of standards, but many of them are incompatible. These are typically embodied by a Grid middleware, which should implement the following services:

- networking quality of service (QoS),
- resource co-scheduling,
- load balancing,
- message passing,
- file transfer mechanisms,
- data security, integrity and coherence.
- authentication.

One widely-deployed middleware is that provided by the Globus Project [115], the *Globus Meta-computing Toolkit* [114]. This contains a range of tools, which can be used for building grid services. Globus makes a distinction between local and global services. The former are simple to deploy, and are not publicly visible. They are then exported via global services, which typically wrap a small group of local services. Local services can be tied to particular hardware resource, and more instances of the same service added without changing the public appearance of the global service. This also allows fault-tolerance; a global service could run a single job on two local services exported by different machines, and if one fails return the result from the one that didn't, without the external user being aware of the failure.

Globus is not the only option for Grid middleware. Others include *UNICORE* [106], which allows legacy closed-source applications to be re-packaged as grid services, which was then extended to provide EUROGRID [107] and GRIP [130]. The *Imperial College e-Science Networked Infrastructure* is designed to allow software components to be deployed in a federated resource pool, as is Legion [58] which assembles the pool into a virtual supercomputer. Condor [79] could be classed as a Grid middleware application, although it predates the idea of the Grid, and is mainly focussed at numerical computation tasks, while the Grid is typically regarded as a more general idea. Codine, which was designed for scheduling tasks across a distributed infrastructure, was developed into Sun Grid Engine [294].

Several Grid infrastructure projects have resulted in deployment to date. The UK e-Science Grid, NASA's Information Power Grid (IPG) and the European Data Grid are all examples which are usable for research purposes, although not necessarily for general use. The AccessGrid [3] is a very specialised Grid environment, based around providing conferencing services, which has seen some use in visualisation since it is designed to share visual data for video conferencing, and can be extended further.

The plethora of Grid 'standards' is expected to converge at some point. The Open Grid Services Architecture [113] was formed by the Globus project and IBM. This aims to build standard interfaces for Grid services on top of existing Web Services. The first draft specification was released in 2003.

Current Grid systems are not ideally suited to running interactive tasks, they are better suited to batch-processing. Since most visualisation tasks require some form of interactivity, this is less than ideal. Some attempts have been made to modify existing Grid middleware to fulfil these needs, such as the *Interactive Access* plug-in to the UNICORE client [106], which allows end-users to interact, via the UNICORE middleware, with simulation processes running at multiple locations.

The Grid is the embodiment of large scale computing. Many of the participants on the Grid (i.e. those offering Grid services) are likely to be clusters or supercomputing centers. Mobile devices occupy the other end of the scale, with very limited computing power and connectivity. The addition of mobile devices to the Grid brings us closer to the vision of *ubiquitous computing*, where computational resources are widely deployed, and consumer electronic devices participate in a Grid-like environment, along with mobile and even wearable computers [328]. This is particularly interesting from a visualisation perspective, since it provides a number of new and exciting opportunities which are largely unexplored.

Some work has been done in this area already. The FUSE system has been proposed as a tool for developing collaborative systems that are to be deployed on multiple platforms. PDAs have been used for a variety of tasks, including as a remote display for visualisations rendered on a graphics workstation [181], or a smart pointer [338]. Tweek [137] presents another alternative, a middleware system which displays a 2D GUI to a virtual environment on a variety of devices, including PDAs.

An alternative use is to use the on-board processing power of the PDA to create visualisations. D'Amora and Bernardini [86] developed a PDA 3D viewer that can access a remote database of CAD models. Some current PDAs have quite powerful 3D accelerators, when taken in the context of their screen size.

Izadi et al. [151] proposed the FUSE system as a development tool for collaborative systems across multiple platforms. Lamberti et al. [181] demonstrated a mobile graphical interactive rendering task running on a PDA which is provided by a remote graphics workstation. Wolf et al. [338] proposed the Smart Pointer as a role for PDA devices, where it either presents a subset of the visualisation when part of a larger visualisation environment (such as a CAVE) or it aims to provide the same overall image as other (desktop) clients, both approaches using a remote visualisation server. Hartling et al. [137] presented a middleware system, Tweek, which displays a 2D GUI to a virtual environment using a PDA. The user may interact with the virtual environment via the PDA. Apart from the technical aspect, human factor issues in using PDAs for visualisation need to be addressed [244].

We have categorized the demands upon both the mobile device and the visualisation service into the following classes ordered according to their communication requirements:

Remote scheduling — A device, such as a PDA, can be used to monitor the account status of the user on a visualisation server. The user should be able to consult their account, see the current state of any jobs, and perform basic management tasks, such as *start*, *stop*, *hold* and *remove*. This requires a low bandwidth duplex channel for textual communications.

Remote monitoring — Higher level monitoring functions can take advantage of the colour displays on the device. Users may query their account to retrieve still images which are visualisations of their data. They may (pre)select parameters for rendering (such as rendering method and transfer function), and be presented with the image. Such parameters may be used to assist with scheduling decisions. This class requires a duplex channel with a higher bandwidth downstreaming traffic.

Remote steering — A remote user can be notified on job (or intermediate result) completion, and may view a visualisation of the result. Some limited interaction with the visual representation is possible as the user's feedback can be used to generate modifications to the current job. This is most useful for checking intermediate results during batch mode without having to be tied down to one location. Some steering of the simulation is possible as jobs can be stopped and restarted from a recent state with new parameters. The bandwidth requirement is higher as the wait time for several images may be undesirable. The computational demands on the PDA are higher due to the need to zoom, pan, and interact with the data. At this stage, transmission and interaction with small 3D models may be desirable and possible.

Remote visualisation — The user interacts freely with the simulation, using the visualisation to explore all aspects of their data. This places a high demand on the PDA as well as the server. The visualisation could be in the form of a sequence of images generated by the server and transmitted compressed to the PDA, or the server could send a stream which could be processed by the limited graphics hardware available on the PDA. User interface widgets could be overlayed over the data, and the user will send interaction data back to the server in order to steer the simulation. Some frame loss, and some pauses in results are inconvenient but not critical.

Figure 3.1: Mobile technology has offered an exciting scope for developing new visualisation applications (image courtesy of Dr. Mark Jones).

Mobile visualisation (Figure 3.1) introduces an interesting design problem for a visual supercomputing environment. It reminds us of the desktop technology two decades ago when low resolution displays and limited computation resources were supported by main frame computers. However, it also exhibits a completely new scenario where the requests for visualisation, or managing visualisation tasks, can come from anywhere with often unreliable communication channels in terms of both bandwidth and security. The infrastructural support to mobile visualisation may significantly broaden the application scope of visualisation, and transform this largely laboratory-based technology to a pervasive technology.

3.2 Autonomic Computing

A Grid infrastructure, or more generally, a pervasive infrastructure, will be considerably complex, and the difficulties in managing such an infrastructure raise a serious question as to whether it is adequate for it to be managed by human administrators, and whether it requires a much more system-level automation than what is currently implemented. Researchers and developers in many fields, such as distributed systems, data communications, Internet technology, Grid computing, agent technology, database systems, expert systems and business management systems, are embracing the concept of *autonomic computing* in managing large and complex infrastructures and services.

Autonomic computing [165] refers to computing systems which possess the capability of self-knowing and self-management. Such a system may feature one or more of the following attributes:

Self-configuring — The system can integrate new and existing components without low-level intervention from an administrator.

Self-optimizing — The system can continually try alternate configurations to determine if the current one is optimal.

Self-healing — The system can detect, and recover from, failure of components, hardware or software.

Self-protecting — The system can detect attempts to compromise it, perhaps from hackers or viruses, and react accordingly.

A noticeable amount of research effort in autonomic computing has been placed on the self-management of system infrastructure and business services. Examples of this include self-configuration in patching management [101] and in Grid service composition [4], self-optimisation in power management [158], business objectives management [7], and resource management in dealing with network traffic spikes [237], and self-healing in online service management [62] and distributed software systems [217].

Efforts have also been made to broaden the scope of autonomic computing, addressing a wide range of related research issues, such as economic models [108], physiological models [187], interaction law [216], preference specification [326], ontology [195, 307], human-computer interaction [14], and so forth.

Though the development of generic software environments for autonomic applications is still in its infancy, several attempts were made, which include projects such as QADPZ [80], AUTONOMIA [98] and Almaden OptimalGrid [88].

QADPZ (*Quite Advanced Distributed Parallel Zystem*) [80] provides an open source framework for managing heterogeneous distributed computation in a network of desktop computers using autonomic principles. In QADPZ, the system complexity is hidden in the middleware layer, facilitating self-knowledge, self-configuration, self-optimisation and self-healing.

AUTONOMIA [98] is a prototype software development environment that provides application developers with tools for specifying and implementing autonomic requirements in network applications and services. It features an *application management editor* for requirements specification, a *mobile agent system* as a uniform execution interface to underlying hardware and operating systems, an *autonomic middleware service* for managing autonomic services, an *application delegated manager* as a broker between components and resources in the context of Jini lookup service [240], and a *fault handler* for self-healing.

OptimalGrid is a self-configuring, self-healing and self-optimizing grid middleware, using a set of distributed whiteboards for communication between the different nodes. A computational problem is expressed using Original Problem Cells (OPCs), which describe the connectivity of the cells with their neighbours and the calculations to be performed using the neighbours information. OPCs are aggregated in collections which are themselves part of Variable Problem Partitions (VPP), assigned to grid nodes. The OptimalGrid system is then able to self-configure, using a list of available compute nodes with their characteristics, and can optimize the repartition of OPCs after each computation cycle. As the communication history between nodes is saved in the whiteboards, if a node is lost the system is able to recover and catch up with the computation, rather than restarting the entire problem. The use of these different autonomic features permits to deliver a grid system more robust and easier to use. Future plans include integrating support for the Open Grid Services Architecture (OGSA) [113].

By mimicking the behaviour of the human autonomic system especially in dealing with homoeostasis, autonomic computing is believed to be a solution to the increasing administrative complexity of computing infrastructures. Hence, no visual supercomputing infrastructure can afford to ignore this emerging technology.

The above discussions have clearly indicated the need for encompassing a large collection of infrastructural issues related to the management of visualisation tasks in a common framework, for which we have introduced the subject domain of *visual supercomputing*. The requirements from applications, such as visual data mining, computational steering, mission-critical visualisation and mobile visualisation, have indicated a high research priority to the infrastructure of visual supercomputing. While such an infrastructure can benefit from the state of the art technologies in visualisation, we are still facing many new challenges in order to realize a well-designed, serviceable and cost-effective infrastructure for visual supercomputing.

Autonomic computing has provided some ideas which can be used for managing parts of this infrastructure, such as the Laundromat Model [136], however the field is still young and has far more ideas than concrete implementations at present.

Hoare outlined a set of criteria for a grand challenge in computer science [143]. According to these criteria, building a visual supercomputing infrastructure can be considered as a grand challenge in the field of visualisation. It raises a series of scientific questions such as:

Architectural Design — Would it be desirable or feasible to build an infrastructure for visual supercomputing based on that of the Grid? How would it accommodate the different needs for centralized, distributed or independent services from various applications? How would such an infrastructure provide generic support to the management of visualisation data, distributed visual data mining, very large scale data visualisation, mission-critical visualisation and mobile visualisation?

Technology Deployment — Should special purpose graphics hardware form the central core of a visual supercomputing environment? If so, what would be the relationship between such central hardware and graphics hardware available on personal computers? What would different hardware attributes impact upon visualisation algorithms, and how would visualisation tasks are managed to take such attributes into account?

Quality of Service — How would a visual supercomputing infrastructure provide seamless services to many users and for many applications, instead of just another 'remote login' service? What would be the role of the infrastructure in managing interaction, data and knowledge about users' experience? In what way could users benefit from a knowledge-based infrastructure?

One emerging strategy for developing complex computing infrastructure is *autonomic computing* [165] (see also Section 3.2), which seeks inspiration in self-adaptive biological systems and self-governing social and economic systems.

Adapting the deployment model, proposed by IBM [150], for the gradual evolution of complex system-wide self-managing environments, one can envisage a similar five-level deployment model for visual supercomputing, which can be developed evolutionarily.

Level 1: Basic — At this level, a visual supercomputing infrastructure is an integrated *system* platform that provides visualisation applications with necessary computation and communication resources. Typically, users are fully involved in identifying appropriate tools, locating computation resources, and managing data distributions. It is often necessary for users to navigate themselves through complicated technical obstacles, such as networking, security, parallel computing, data replication, and so forth.

Level 2: Managed — At this level, a visual supercomputing infrastructure will have a managed *service* layer between the user interface and the system platform. The service layer is aware of the availability and ontology of data and resources, and can provide services to various visualisation applications according to dynamic requirements of users and applications as well as dynamic states of the system platform. To a large extent, the development of the Grid technology is aiming at the delivery of a general-purpose infrastructure. To managing visualisation applications effectively, it is necessary to incorporate more advanced service features into the Grid technology for supporting a variety of visualisation needs such as interactive, distributed, mobile, and mission-critical applications in a more transparent manner.

Level 3: Predictive — At this level, a visual supercomputing infrastructure will have an *information* layer between the user interface and the service layer, which collects, monitors and correlates various user interaction data and system performance data. It provides users with analytical data, which may indicate the quality of visualisation results, effectiveness of visualisation tools, and so on. In addition, this layer can enable faster and better task specification by reporting potential problems and recommending suitable tools and visual representations. It is at this level, the infrastructure starts to manage users' experience in carrying out visualisation tasks.

Level 4: Adaptive — At this level, a visual supercomputing infrastructure will have an *adaptation* layer between the information layer and the service layer. Based on the information collected, the adaptation layer has the functionality for self-configuring and self-optimizing the computational requirements of a visualisation task, as well as the functionality for self-managing the system platform and various visualisation services dynamically. It is at this level, visualisation users can be largely freed

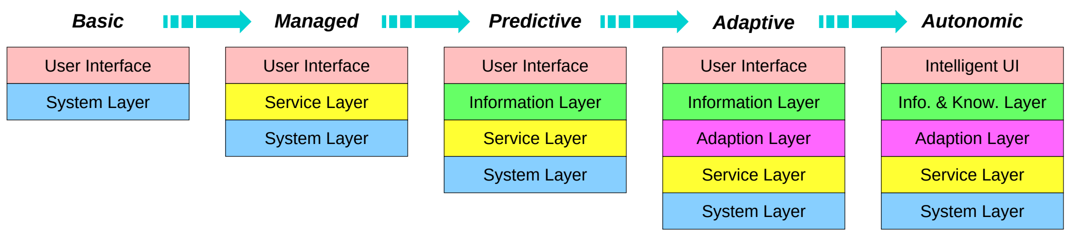


Figure 3.2: developing new visualisation applications.

from software management, and are able to focus on their core business, that is, visualisation.

Level 5: Autonomic — At this level, the traditional user interface in a visual supercomputing infrastructure will be replaced by an intelligent user interface, for instance 'a virtual secretary', which is capable of transform information to knowledge and provides users with a wide range assistance. Such assistance may include specifying visualisation tasks, scheduling inter-dependent jobs, organizing raw data and visualisation results, managing security, checking the quality of the service and results, and arranging the sharing of the data with other users.

Figure 3.2 illustrates evolutionary advance of the infrastructure through the five levels. In this deployment model, each layer is merely a conceptual placeholder for a collection of functional components (e.g., services, tools, agents, databases, knowledge-bases, and so on). It is not necessary for the development and deployment of each level to follow a temporal order. Nor is it desirable to make each layer a centralized bottleneck in the process of visualisation. It is most likely that the infrastructure will be realised with a large number of autonomous, interacting, self-governing functional components.

3.3 e-Viz

Building a visual supercomputing infrastructure is no doubt an ambitious grand challenge. However, a solid foothold already exists at Level 1, and developments are rapidly approaching Level 2. A noticeable amount of research effort is being made to develop system-level autonomic computing techniques in many fields, including distributed systems, data communications, Internet technology, Grid computing, agent technology, database systems and business management systems. Some of such effort can be viewed as 'horizontal' deployment of autonomic computing at the *system layer* and *service layer* of a visual supercomputing infrastructure (Figure 3.2), while others can provide new concepts, methods and tools for the development of the *intelligent user interface*, *information and knowledge layer* and *adaptation layer*.

The e-Viz project [39, 260, 42], begun in 2005 intends to explore the needs of such an infrastructure and begin developing solutions. This project is a collaborative endeavour between four sites in the UK; Bangor, Leeds, Manchester and Swansea. This PhD was conducted as part of the e-Viz project and this thesis presents work performed towards

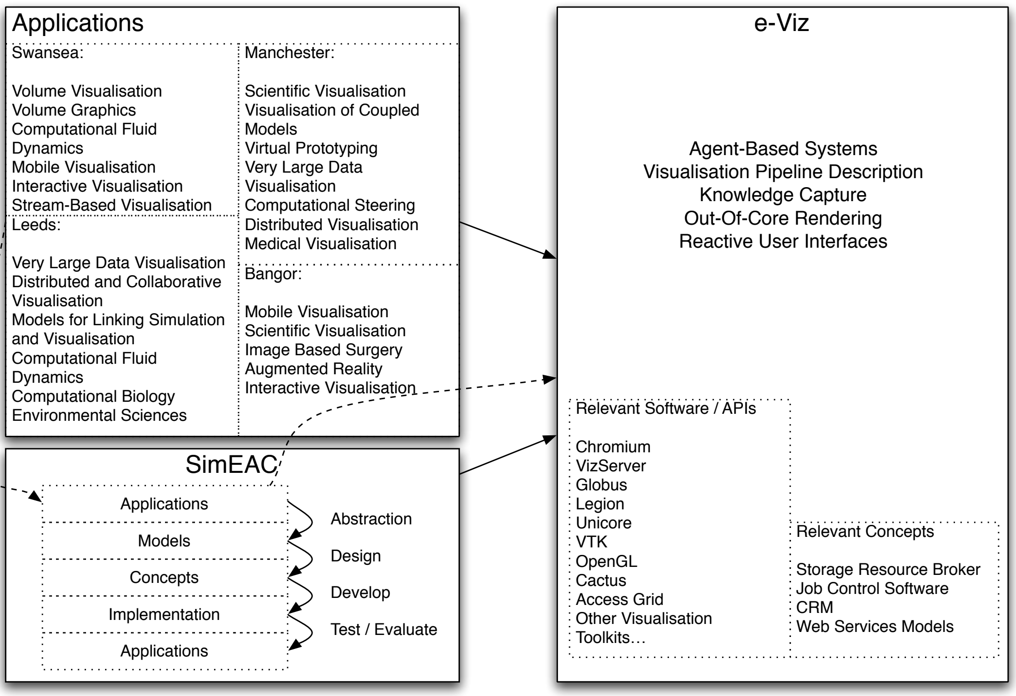


Figure 3.3: The e-Viz project overview.

this overall goal.

It is understood that a complete solution to the problem of designing an infrastructure for visual supercomputing is well beyond the scope of a single PhD. The work presented here intends to provide some of the framework for such a system. Figure 3.3 shows an overview of the e-Viz project. The project had three main components. The e-Viz system, a prototype for an autonomic visualisation system, a set of applications to be evaluated for deployment on this infrastructure, and a simulator for modelling the overall system.

Work on reactive user interfaces for this was system was conducted at Leeds and Manchester [42] and applications were tested at all sites. For example, work was conducted at Bangor on using an adaptive grid infrastructure for augmented reality tasks [146, 147].

One of the goals of the e-Viz project was to explore the application of autonomic concepts to visualisation problems. Part II of this thesis describes one such application. Rendering large datasets will be a problem that any visualisation infrastructure will have to address and out-of-core rendering is a necessity for this. This part of the thesis will explore an autonomic, knowledge-based, approach to data management in an external memory context. Approaches such as this would need to be built in at the lowest levels of a truly adaptive infrastructure for visualisation, such as that proposed by the e-Viz project and be part of the system pipeline described in the diagram.

Other work conducted at Swansea as part of the e-Viz project dealt with an agent-based system for defining this pipeline [262]. During the design phase, this work was simulated using the SimEAC simulator described in Part III.

SimEAC was originally entered in the e-Viz proposal as SimuVis, a system for simulating a final e-Viz system. While working on the requirements for such a system, it became clear that any system which met the needs of e-Viz would be more generally useful. The final system, renamed to make this more apparent, is presented in detail in this thesis.

Once the pipeline is created, it is often important to interactively control the visualisation. Work on this was conducted at Leeds [341] and the pollution modeller application used was also simulated running on several different configurations by SimEAC.

The e-Viz project demonstrated the advantages of an autonomic infrastructure for visualisation. Ongoing work should make this a reality. Other contributions to the research community from this project include:

- The five-level deployment model for autonomic visualisation from Swansea [39].
- Interactive remote server-based visualization from Manchester [260].
- Adaptive user interfaces for visualisation and computational steering from Leeds and Manchester [42].
- Agent-based management of visualisation pipelines from Swansea [262].
- Applications in augmented reality from Bangor [147, 146].
- An application in volume visualisation from Swansea [262].

Part II

Managing Large Datasets

Chapter 4

Out-of-Core Techniques

Contents

4.1 Motivation

Moore's law specifies that the number of transistors it is possible to fit on an integrated circuit for a fixed financial investment doubles roughly ever 18 months. Since the number of transistors gives a rough indication of performance, we can claim that the speed of commodity CPUs follows a similar trend.

Hard drive capacities have followed a similar pattern, however seek times have not. Hard drive seek times are limited by the angular movement of the platter and the linear movement of the head. The head speed has remained relatively constant for the last twenty years (although the platter size has shrunk somewhat, meaning that the longest possible movement has shrunk). Rotational speed has increased somewhat. The first graph in Figure 4.1 shows the number of millions of instructions per second of commodity x86 chips over time along with the rotational speed of hard drives of equivalent eras.

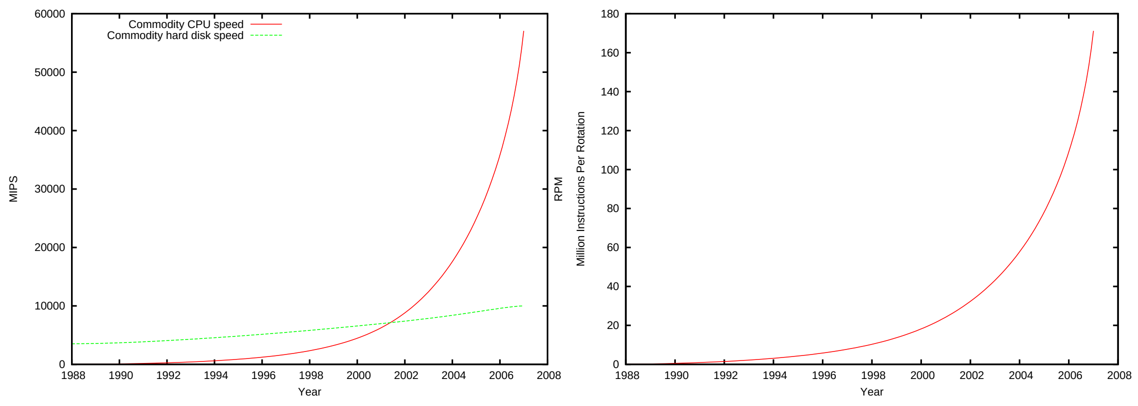


Figure 4.1: Disk and CPU speed over the last twenty years.

The maximum amount of time for a seek that doesn't involve moving the head is given by the reciprocal of the rotational speed (i.e. the time for one complete rotation). The second graph in Figure 4.1 shows the number of instructions that a CPU can execute in one revolution of a hard disk of the same vintage. Note that this follows a roughly exponential curve. If an algorithm depends on data on the disk, then this gives an idea of how much time the CPU will spend idle instead of doing useful work.

The data used to produce this graph is shown in Table 4.1. The CPU MIPS numbers are taken from the speeds of commodity x86 chips. Within the same instruction set architecture, MIPS is a relatively accurate reflection of speed. The disk RPMs for the corresponding years are taken from a variety of sources, including manufacturer press releases and adverts dating from the relevant year. These figures are not to be considered completely reliable, however they give a good indication of the speed of consumer technology for a given year.

The seek time for any hard disk may be approximated by the following formula:

$$average\ seek\ time = max(\frac{1}{2 \times rotational\ speed}, pivot\ time + settling\ time)$$

The settling time and pivot time are closely related; the faster (and further) the pivot moves, the larger the settling time due to induced vibrations caused by sudden motions. The pivot time is also influenced. The graph in Figure 4.1 assumes some locality of data, and so that the pivot time will be negligible compared to the rotational latency. While this is not always valid, it gives a sufficiently good approximation for demonstration purposes.

In many cases, seek time is already a significant bottleneck, and this is likely to increase. Since the CPU is spending so much time waiting for data, a significant performance increase is possible by using some of this spare CPU time to load the data before it is needed. Note that sustained transfer rates of hard drives are proportional to the rotational speed and the data density. Since the density has increased at a faster rate than CPU

Year	CPU	MIPS	Hard Drive RPM
1988	Intel 386SX (25MHz)	8.5	3524
1992	Intel 486DX (66MHz)	54	3524
1996	Intel Pentium Pro (200MHz)	541	5400
1999	Intel Pentium III (500MHz)	1354	5400
2000	AMD Athlon (1.2GHz)	3561	7200
2002	AMD Athlon XP (2GHz)	5935	7200
2003	Pentium 4EE (3.2GHz)	9726	7200
2006	AMD Athlon FX (2.6GHz)	18938	10000
2007	Intel Core 2 (3.33GHz)	57063	10000

Table 4.1: Disk rotational speed and CPU speeds.

speeds, sustained transfers are much less of a bottleneck[1].

This bottleneck is even more apparent when the data is stored on a remote server; in this case the latency can be 200ms or more, giving two orders of magnitude more wasted cycles.

The next three chapters describe some approaches to alleviating the problem of latency. An algorithm-specific approach is proposed for a specific rendering method, and then an attempt to apply autonomic concepts to the problem is discussed.

4.2 The Anatomy of an Out-of-Core Algorithm

An out-of-core algorithm is divided into three components. These each deal with a different stage in the life cycle of cached data.

- A data layout strategy
- A prefetching strategy,
- An eviction strategy

The *data layout strategy* relates to how data is stored out-of-core. On a hard disk, sequential reads are significantly faster than random seeks. If data is organised in a way that preserves locality then the entire working set for an algorithmic step can be loaded into memory with a single contiguous read. Unfortunately, the inherent unstructured nature of some data, such as the point sets described later, make it very difficult to gain any performance benefit from layout.

Another problem with data layout strategies is that they rely on the user having access to something approaching the physical structure of the underlying hardware. A modern hard disk uses linear block addressing (LBA) giving the appearance of a sequential structure, hiding the details of the cylinder and track layout from the operating system. The OS

[1]Whether sustained transfer speeds are actually a bottleneck depends on the processing algorithm under discussion.

will then split files for more efficient storage in different parts of the disk. Something that appears to a userspace application to be a linear write has to pass through two layers of abstraction and can end up being nothing like a sequential operation. This is ignoring the fact that disk accesses by competing processes may end up being interleaved, moving the disk head a significant distance in the middle of a large write. Any data layout strategy, to work properly, has to be aware that it might be fighting against the operating system. Research conducted at the University of Utah[2] has attempted to build an adaptive data management strategy into the physical device, automatically re-ordering data based on access patterns, using the same principle as wear-leveling in Flash storage devices.

The next component of an out-of-core algorithm is the *prefetching strategy*. This determines which data should be loaded speculatively. It is important for performance that this provides a good prediction accuracy, as discussed earlier. The average seek time of a modern disk is of the order of 9ms. This means that every time the required data is not pre-fetched the CPU has to stall for several thousand cycles waiting. As the size of this gap grows, more complicated prediction algorithms can be implemented without introducing a new bottleneck.

The final component is the *eviction strategy*. This defines which parts of the data should be evicted to make room for the newly loaded components. A perfect eviction strategy would evict the data that are not going to be used for the longest time. Usually, however, this information is not available, and so it is necessary to make some educated guesses. Common strategies for eviction include Not Recently Used (NRU), Least Recently Used (LRU), and Least Frequently Used (LFU). A hinted LRU strategy is used by the algorithms described herein. Each time our prediction algorithm is run, it provides a set of (node, confidence) pairs. These are then used by the eviction code so that it will not evict nodes that have been predicted with a high confidence value, and are not currently being used.

While not part of the data management strategy for an out-of-core algorithm, the processing algorithm used can also have an effect on the effectiveness of the approach. Some algorithms are designed to minimise the size of the working set at any given time interval. These tend to work better in an out-of-core context.

4.3 A Taxonomy of Approaches

A taxonomy of out-of-core approaches is proposed. This divides out-of-core algorithms into three categories:

- Algorithm Based
- Data structure Based
- Knowledge Based

[2]Not yet published, at the time of writing.

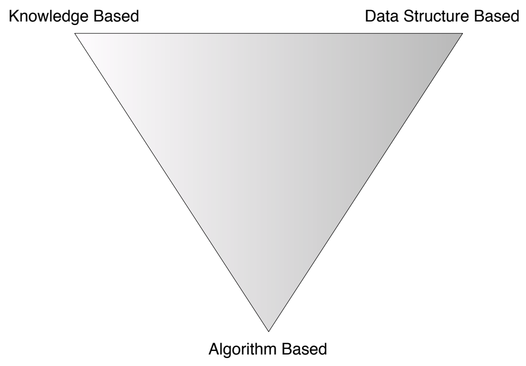


Figure 4.2: The continuum of out-of-core approaches.

Definition 4.1 (Knowledge Based)**.** *Using no a priori information; learning everything required to make predictions from previous interactions.*

These categories refer to where the algorithm acquires the information required to make predictions, and are specific to the prefetching and eviction strategies of the algorithms.

An *algorithm based* strategy is one that is tied to a particular processing algorithm. These rely on the algorithm to be able to make predictions about their future access patterns. If the algorithm is able to make good predictions, then these will perform well. The disadvantage is that they are not portable between problem domains; each instance of a problem has to be extensively modified to work in an out-of-core setting.

A *data structure based* strategy is slightly more general. These are specific to the structure of the data, but not to the algorithm used for processing. Usually these work by understanding some dataset-specific concept of locality, and prefetch data 'near' the currently processed blocks. An example of this might work on regular structured volumes and load every voxel adjacent to the ones being currently processed. Data structure based approaches are typically closely tied to a specific data layout strategy; data is stored on disk in a way that preserves the concept of locality specified by the prefetching strategy.

Knowledge based algorithms are the most general of the three. They do not require any knowledge of the data being processed. These learn access patterns through the data from previous interactions and use them for prediction. Most operating system virtual memory implementations fall loosely into this category, although they are very simple implementations. They have no knowledge of the structure of the data, or what is being done to it and apply simple heuristics, such as 'Not Recently Used' to handle eviction and linear read-ahead for pre-fetching.

In practice, most algorithms fit somewhere in the middle. This can be represented this using a continuum as shown in Figure 4.2. This depiction will be used in later sections to classify out-of-core strategies.

4.4 Related Work

There is a close relationship among volume visualization, implicit modeling and point-based techniques. *Points*, as modeling primitives, are extensively featured in all three classes of techniques.

The advances in *volume visualization* have produced a collection of methods for rendering *volume datasets*, including isosurfacing [199], ray casting [191] and forward projection [329]. Though in most cases, point primitives, i.e., voxels, are organized into a grid or a mesh, in some cases, volume modeling with scattered data is necessary, (e.g., [230]), though existing approaches mostly involve the construction of a mesh structure connecting these points together.

Implicit modeling [34, 231, 344] facilitates the composition of complicated objects from elemental field functions, each of which is often defined on a point primitive. The octree method has been used for polygonizing implicit surfaces [35] and computing ray-surface intersections [157]. Due to the computational costs of polygonization and ray tracing, the emphasis has always been placed on the use of a small set of point primitives or elemental field functions.

One of the major advances in recent years is point-based modeling and rendering. The most significant examples of this development include Surfels [249] and QSplat [266]. Other important developments include [132, 9, 286, 347]. In addition to the splatting approach commonly adopted in point-based rendering, ray tracing point clouds through intersection has been examined [270].

Recently a number of researchers addressed the convergence of these techniques, for example, approximating volume datasets with implicit models [145], building implicit surfaces upon point clouds, using the point-based approach for isosurfacing volume datasets [347, 323, 198], and combining point clouds and volume datasets in volume scene graphs [59].

Many visualization processes involve datasets that are much too large to fit into the internal memory of a computer, and have to rely on external disk storage, usually under the virtual memory management of an operating system. The external disk access can become a serious bottleneck in terms of rendering speed. *Out-of-core algorithms* (also known as *external memory algorithms*) [321] are designed to solve a variety of batch and interactive computational problems by minimizing disk I/O overhead.

Various out-of-core visualization algorithms have been proposed to handle large structured and unstructured 3D data-sets, for instance, in the context of (i) isosurface extraction [75, 67, 68, 64, 295], (ii) terrain rendering [194], (iii) streamline visualization [308], (iv) mesh simplification [193], (v) rendering time-varying volume data [277], (vi) rendering unstructured volumetric grids [190, 109, 64], (vii) ray tracing [250], and (viii) radiosity [301]. While some algorithms rely little on internal memory (e.g., [67, 109]), others utilize preprocessed data structures, such as octree [308] and indexing [277] to optimize disk I/O operations. Use of *Active Data Repository* for visualizing large volume datasets was also reported [176].

While point datasets are usually excessively large, there has been little existing work on out-of-core methods for handling point datasets [131]. This motivates us to investigate the feasibility of multiple large point sets in the context of discrete ray tracing.

4.4.1 Out-of-core

The concept of External Memory is not unique to visualisation applications. Operating systems have supported some form of *virtual memory* [92] for decades. The term virtual memory is used to refer to the combination of two concepts; protected memory and out-of-core storage. The term 'out-of-core' derives from the old systems which used ferrite core memory. In these systems, there were two places data could be stored; in the ferrite core, or out of it. Ferrite core memory was fast (for the time), but expensive. In a modern system, the 'in-core' storage is no longer ferrite core, but the term remains.

Since the late 1960s, the most common way of implementing the external memory portion of this has been through *paging* [81], a system whereby memory is divided into equal-sized regions which are swapped between in-core and out-of-core existence. These pages are loaded back into memory when they are accessed, using a process known as *demand paging* [174].

Processes which exhibit a rapid change in the accessed memory can cause *thrashing* [91] on systems which employ demand paging. Thrashing involves the computer spending most of its time swapping data between internal and external memory, rather than doing useful work. This is not a problem for a lot of uses. Modern operating systems are typically designed on under the assumption that exhausting in-core storage is unusual, an argument made by Prof. A. Tanenbaum for omitting support for swapping from Minix 3 [298]. While this is a valid assumption for general purpose computing, it does not hold for a number of visualisation tasks.

Proposed solutions to this problem include allowing the application to control the operation of the demand paging system [84, 138]. This allows more intelligent decisions about which data to swap to be made.

Beyond minor modifications to paging systems, a number of problem-specific approaches have been proposed, including data structures designed for on-disk access [320, 110] and algorithms that promote efficient access paths through the data [319, 65, 19, 140]. Much of the literature in this area focusses on complete traversals of tree data structures. This is not particularly applicable to many rendering algorithms, which rely on a partial traversal of a tree in an order determined by the current viewport.

4.4.2 External Memory

Towards the end of the 1960s, it became clear that the larger storage capacity of out-of-core devices, such as tapes, could allow computers to work on much larger problems. Work from this era [90, 169] focussed largely on sorting problems. Research on sorting gives a number of streaming algorithms (such as [234]), based on a distribution sort, conceptually similar to a generalised radix sort. In these, each pass over the data places it into a number of bins on disk. Each of these is then read in, and further sub-divided, finally giving a completely sorted set of files. Much research in this area hinges on finding good partitioning mechanisms [95]. A number of merge-sort derivatives [120, 169, 235] have also been developed, mainly aimed at systems with multiple disks.

Many of these approaches generally make use of the *Parallel Disk Model* [320], where multiple, independent, external memory stores are available, although some assume a single disk.

Beyond sorting, a number of areas such as matrix and grid computations [153, 233, 189], computational geometry [125, 20, 22], and graph traversal [310, 66, 21]. While most of this work deals with algorithms with low I/O complexity, some, particularly in the field of graph manipulation [5, 232] focusses on efficient on-disk storage structures for a particular form of access.

The work in this area measures algorithm performance using the traditional metrics of spacial and temporal complexity, but also in terms of the number of I/O operations required. Much of the focus is on developing entirely new algorithms to solve existing problems, rather than attempting to allow existing algorithms to achieve good performance using external memory. This is not necessarily a good approach for visualisation, where a considerable amount of work has been done in designing in-core algorithms with good visual output.

4.5 The 5-Layer Model

A five tiered approach to out-of-core strategies is proposed, as shown in Figure 4.3. Each of the layers is conceptually independent and needs only to communicate with those directly above and below it in the stack. For efficiency reasons, it is possible that some layers may be omitted or combined. An out-of-core stack running on a single machine might not need the network layer, while many existing out-of-core systems lack the knowledge layer.

In the implementation described earlier, the lines between the knowledge and data structure layers are often blurred. The block and record layers are discrete, and multiple implementations of each were evaluated. In ascending order, the layers are:

1. Block Layer
2. Record Layer
3. Knowledge Layer
4. Network Layer
5. Data Structure Layer

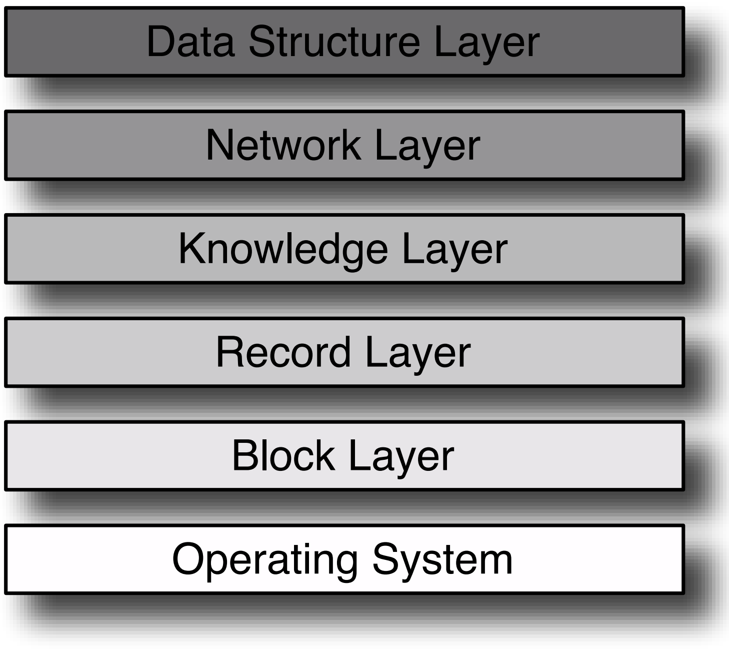


Figure 4.3: The five layers in the model, in relation to the OS and application.

The remainder of this chapter will discuss the layers individually and the interactions between them, as well as presenting an example implementation.

4.5.1 The Block Layer

The very bottom of the stack is the block layer. This is responsible for moving data between in-core and out-of-core storage. This layer responds to messages from the next layer up instructing it to load byte ranges from the external memory device, and to evict (modified) data from the in-core cache.

It is advisable that the block layer operate asynchronously, even if the remainder of the stack is synchronous. If I/O operations are performed synchronously then the only effect is to move the delay from the usage phase to the prefetching phase; often increasing the total delay, since the lower bound for the amount of data prefetched is the amount of data used. The reference implementation described in 4.8 includes several block layers, including one built on POSIX asynchronous I/O facilities which generally provides the best performance. A synchronous version of the block layer is also provided for reference. This serves two purposes; it is easy to understand, and so can be used as a base for building better ones, and it provides more deterministic results, making it useful for algorithm evaluation. In real-world use, one of the other back-ends is likely to give better performance.

The POSIX specifications provide a number of operating system facilities that can be used to implement this layer including, but not limited to, asynchronous I/O and memory-mapped files. Which provides the best performance can vary between platforms. For example, on Mac OS X, memory mapped file access is an order of magnitude slower than asynchronous I/O, while the difference between the two is negligible on FreeBSD. The rôle of the block layer is to provide an abstract interface to the underlying operating system functionality, allowing the most efficient implementation for a particular scenario to be used without modifying the higher-level code.

4.5.2 The Record Layer

The record layer is responsible for translating between abstract record number and byte-ranges. In the case of fixed record sized data this is a simple multiplication. For other data structures it may be more complicated. The record layer, in simple terms, can be described by the following function:

$$r(n \in \mathbb{N}) \rightarrow \{O, L\} \tag{4.1}$$

This maps a record, n, to an offset/length pair ($\{O, L\}$) within the out-of-core storage. Since this mapping can vary between data sets, the initialisation of this layer may involve the passing of a higher-order function. The distributed implementation discussed later uses this mechanism.

The record layer is also responsible for tracking which records are currently being used by maintaining a reference count. When notified that a record is no longer required, the record layer may retain a cached copy or evict it immediately.

4.5.3 Knowledge Layer

The knowledge layer is responsible for making predictions based on past actions. It receives prediction information from the higher layers, but is free to disregard speculative prediction information; only non-speculative requests must be handled. This layer can be viewed as being represented by the following function:

$$k(\{X_0 ... X_n\}, P, S) \rightarrow X_{n+1} \tag{4.2}$$

In this, X_0 represents the first accessed record and X_n is the current record. S is a value representing the initial conditions of the accessing process. In a visualisation context, this may be the viewpoint position and direction, for example. The set of predictions, P, should be a set of pairs of the form $\{PX_{n+1}, C\}$, where PX_{n+1} is the predicted value of X_{n+1} from the higher layers, and C is a confidence value. The only stipulation on the nature of the implementation of this layer is that the following must hold:

$$k(\{X_0 ... X_n\}, \{X_p, C_{max}\}, S) \rightarrow X_p \tag{4.3}$$

That is to say, any prediction passed into this layer with the maximum confidence value (C_{max}) must be returned as the prediction by this layer. This provides a mechanism for a 'pass through' request; the client is totally sure of the record it requires (which always happens as the record is used, and may happen before) and informs the knowledge layer of this. In this case, the knowledge layer does not provide a prediction — it simply passes on the existing prediction — but it can update its internal state based on the incoming prediction.

In the existing implementation, $C \in \{0, 1, ..., 255\}$, and hence $C_{max} = 255$. This was an implementation decision to allow small, byte-indexed, hash tables, and should not necessarily constrain future implementations.

The knowledge layer is situated below the network layer to enable it to take advantage of the knowledge acquired by multiple different users. In these cases, inferences can be made from similar values of C. If there is no network layer in a given implementation (i.e. the out-of-core store is local), the knowledge and data structure layers may be merged.

4.5.4 Network Layer

The network layer is an optional layer that is only required in distributed implementations. In cases where the data storage and processing are performed by different machines, the network layer mediates between the two.

The network layer is responsible for maintaining a local cache of nodes that have been received, the size of which is dependent on the amount of local storage space available. A cluster node, for example, might not have a local hard drive. In this case, the local storage would be limited by the amount of memory available. The same might hold for a PDA. A workstation, conversely, might have enough local storage to cache a significant fraction of the working set.

In addition to caching, the network layer is responsible for transferring data between the data server and the processing machine. Timely delivery of data is important for the network layer, since faults will cause a significant slow-down. Since more predicted records are typically sent than are needed, reliable delivery is not as important. For this reason, TCP is not a good match for the network protocol. On IP networks, UDP or SCTP provide a better match. Of the two, SCTP is likely to better solution in the long term. Unfortunately, support for SCTP is not at an ideal level in major operating systems at the time of writing; while support generally does exist, at least in add-on form, for most platforms, performance is not yet at a sufficiently high level for it to be usable.

4.5.5 Data Structure Layer

The data structure layer provides an abstract view of the data to the processing algorithm. In the example given in the previous chapter, this view was of an octree data structure and an indexed point set.

This layer represents a concrete instance of an abstract compound data type. Each data structure should have its own implementation. This layer is able to make predictions based on an understanding of the structure of the data and the algorithm used to process it.

It is possible to further subdivide this layer, and provide predictions based solely on the processing algorithm or solely on the data structure. In the case of the ray tracing example, a pure data-structure prediction may come from the locality of nodes in a spacial partitioning system, while a pure algorithmic prediction might come from the point along a ray path, with no reference to the node containing this. In practice, it is sufficiently difficult to separate out these components that they are placed in the same layer.

4.6 Interaction Between Layers

Each layer in the hierarchy communicates with those directly above and below. Above the top layer is the application, which delivers requests to the data structure layer. These requests are often in an abstract form, such as 'the voxels neighbouring coordinates x,y,z' or 'the octree node which is the parent of this one.'

The interface between the application and the data structure layer depends on the data structure. A simple example data structure might be an array with get and set element methods to set and get elements at a specified index. Listing 4.1 shows a C interface for such a data structure. This is taken directly from the reference implementation, and the line numbers reflect those in the source file (`Array.h`).

Listing 4.1: Interface to an out-of-core array.

```
30 //Create a new out-of-core array
31 ooc_array_t ooc_array(char * fileName,
32                       BOOL readOnly,
33                       BOOL grows,
34                       unsigned int nodeSize,
35                       unsigned int size,
36                       unsigned int inCore);
37 //Add a new node to the array
38 void * ooc_array_new_node(ooc_array_t array, unsigned int * position);
39 //Precache a specific element
40 void ooc_array_precache(ooc_array_t array, unsigned int position);
41 //Get the specified element from an array
42 void * ooc_array_get_element(ooc_array_t array,
43                              unsigned int position,
44                              BOOL predict);
45 //Move the specified element out of core
46 void ooc_array_free_element(ooc_array_t array,
47                               unsigned int position);
48 //Free an array
49 void ooc_array_free(ooc_array_t array);
```

4.6.1 Retaining an Element

In this example, the array appears to be a simple wrapper around the fixed-size record layer. In addition to this function, it also attempts to predict linear accesses.

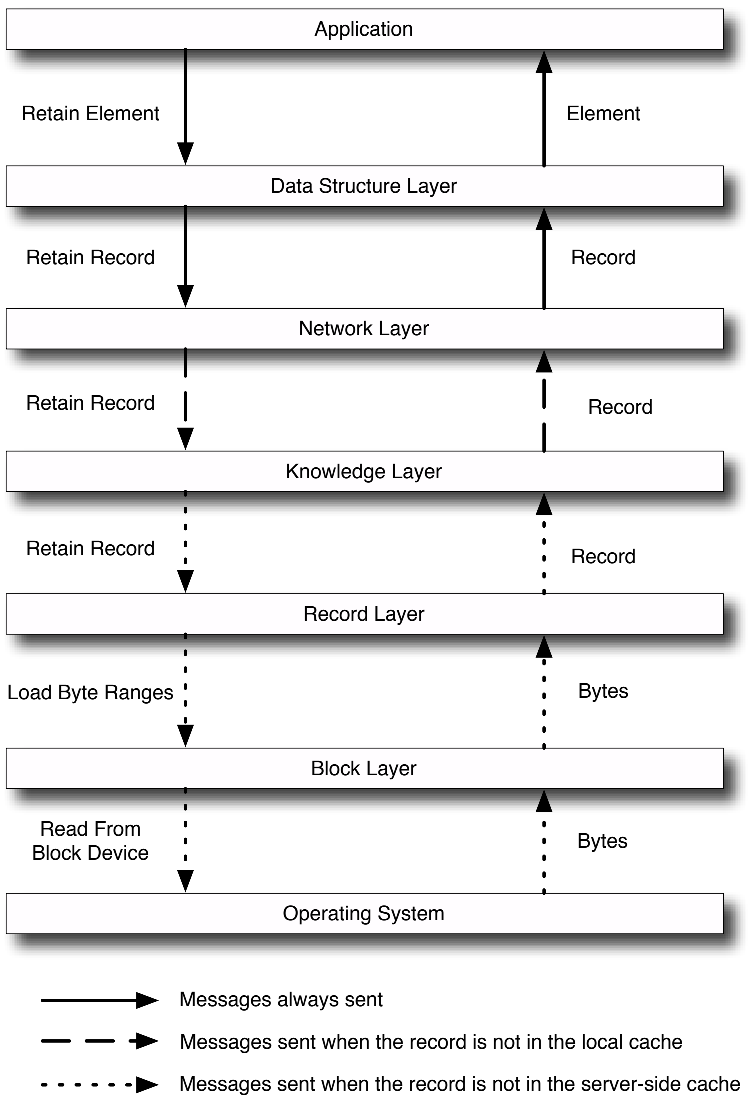


Figure 4.4: Message flow retaining an element in an out-of-core data structure.

Figure 4.4 shows the flow of messages through the system when retaining an element in a data structure managed by an out-of-core system. The application sends a message[3] to the data structure layer instructing it to retain a specified element. The data structure layer then translates this into a record request and passes it on to the network layer.

At the network layer, two outcomes are possible. If there is already a copy of the requested record in the local cache then some of the next steps can be skipped. If not, then it must be requested over the network.

[3]The term message is used here in the object oriented sense, and may be implemented by a function call or a method invocation.

The server's record layer receives the request for a record. It first checks to see if it is in the server's cache. If it is not, then this layer maps the record number to a set of byte ranges and instructs the block layer to fetch them.

The block layer provides an abstract interface to the operating system's disk I/O facilities. When it receives a request for a set of block ranges, it provides the data to the record layer. This then percolates up to the top of the stack and is read by the application.

It should be clear from this example that best performance is achieved when the network layer already has a cached copy of the requested record. As such, accurate predictors in the data structure and knowledge layers are important.

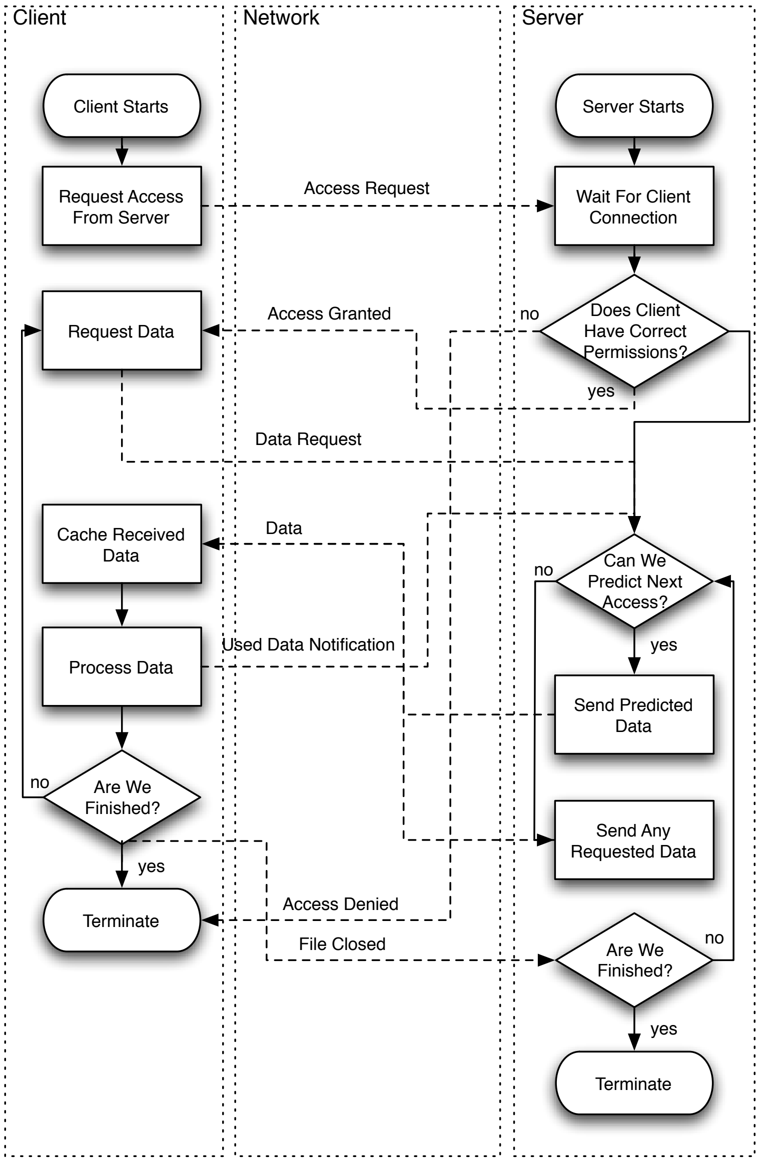


Figure 4.5: Message flow between an out-of-core client and server.

Figure 4.5 shows an example message flow between an out-of-core client and server. This is a slightly simplified flow, omitting the data structure prediction mechanism and the caching.

The server in this example has access control features in its half of the network layer, and so the first step is authentication of the client. At each iteration through this flow, the client requests a group of records from the server. The server is then sends a set of records in response to this request.

4.6.2 Predicting Usage

Predictions occur at two levels: the data structure and knowledge layers. Figure 4.6 shows the flow of prediction-related messages between the layers in response to an application retaining an element in an out-of-core managed data structure.

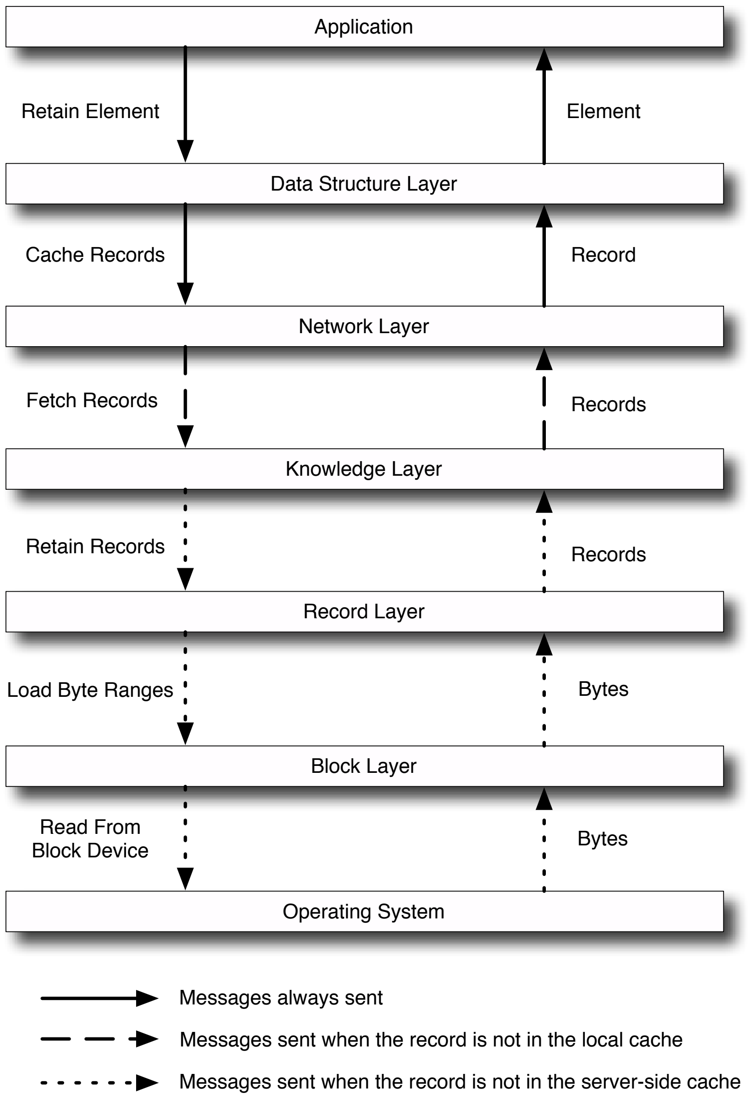


Figure 4.6: Message flow related to predictions when retaining an element in an out-of-core data structure.

The first predictions are made at the data-structure layer. These are based on an understanding of the structure of the data, and are then passed to the network layer. The network layer determines if the predicted records are cached locally, and if not then it requests them from the server.

When the server receives the caching request, it should then return some records. Note, however, that the server is not required to return the records that were requested. The knowledge layer takes these requests as input and uses them to predict which records the client is likely to need next.

The records predicted by the knowledge layer are then loaded from the disk (if they are not cached) and sent to the client.

4.7 Interface Specification for Layers

The algebras listed in this section depend on the String and Nat algebras which are omitted for the sake of conciseness. These should be taken to represent strings and natural numbers with their standard definitions. On some implementations, the range of natural numbers may be restricted to machine words (or multiples thereof) for efficiency.

4.7.1 Block Layer

The algebraic specifications in this section depend on the existence of List, Nat and Bool algebras (and associated sorts) corresponding to the standard definitions of lists, natural numbers and Boolean values respectively. The List algebra is expected to implement head and tail operations with their conventional meanings.

The block layer must conform to the specification shown in Algebra 4.2. The operations described here allow the reading and writing of ranges of bytes from files on permanent storage.

For efficiency reasons, a block layer implementation may provide the operations for batching loads and stores (loadRanges and storeRanges) in a way that is not built from the primitive operations, as long as they are semantically equivalent.

4.7.2 Record Layer

The record layer has two main functions; caching and mapping between records and byte ranges.

Note that a record must be cached immediately after caching, but at no other times. The semantics for evicting records from the cache are not defined, and the cache may be implemented in any way that the developer chooses.

Algebra 4.2: Interface specification for the block layer.

```
Algebra BlockLayer is

    including String Nat TypeList

    sorts String Nat TypeList  BlockFile ByteRange Block Data

    op openFile     : String → BlockFile
    op loadBytes    : BlockFile ByteRange → Data
    op storeBytes   : BlockFile ByteRange Data → BlockFile
    op loadRanges   : BlockFile List → Block
    op storeRanges  : BlockFile List Block → BlockFile
    op makeBlock    : List → Block
    op blockData    : Block → List

    var F : BlockFile
    var D : Data
    var B : Block
    var R : ByteRange
    var L : List

    eq loadBytes(storeBytes(F,R,D), R) = D
    eq storeBytes(B, R, loadBytes(B, R)) = B
    eq loadRanges(storeRanges(F, L, B), L) = B
    eq storeRanges(F, L loadRanges(F, L)) = F
    eq loadRanges(storeRanges(F, L, B), L) = B

    eq storeRanges(F, emptyList(ByteRange), B) = F
    eq storeRanges(F, L, B) = storeRanges(storeBytes(F, head(L), head(
        blockData(B))), tail(L), tail(blockData(B)))

    eq makeBlock(blockData(B)) = B

end Algebra
```

4.7.3 Knowledge Layer

Algebra 4.4 shows the required semantics for the Knowledge Layer. Again, this specification only describes the interface, leaving the mechanism for making knowledge-based predictions up to the implementation. Some examples of algorithms that can be used at this layer are evaluated in Chapter 6. Due to the large number of potential ways in which this layer can be implemented, some of which were described in the last chapter, this is specification is very generalised.

In this specification, note that there are two functions which alter the state of the knowledge layer. The first is called whenever a record is accessed by the application, while the second is used as input when the data structure layer makes predictions. Note that all functions in this layer take a client as an argument. The type of the client is not specified, all that is required is that it be a unique value for each user or application connected to the system.

Algebra 4.3: Interface specification for the record layer.

```
Algebra RecordLayer is

    including BlockLayer Bool

    sorts Nat BlockFile RecordFile List Cache Record Bool

    op makeRecordFile : BlockFile → RecordFile
    op blockFile : RecordFile → BlockFile

    op cacheRecord : RecordFile Nat → RecordFile
    op isCached : RecordFile Nat → Bool
    op recordData : RecordFile Nat → Record
    op retainRecord : RecordFile Nat → RecordFile
    op releaseRecord : RecordFile Nat → RecordFile

    op recordRanges : RecordFile Nat → List
    op makeRecord : RecordFile Block → Record

    var R : RecordFile
    var B : BlockFile
    var I : Nat

    eq blockFile(makeRecordFile(B)) = B
    eq isCached(cacheRecord(R, N), N) = tt
    eq releaseRecord(retainRecord(R, N), N) = R
    eq retainRecord(R, N) = retainRecord(cacheRecord(R, N), N) if ¬
       isCached(R, N)

    eq recordData(R, N) = makeRecord(R, loadRanges(blockFile(R),
       recordRanges(R, N)))

end Algebra
```

4.7.4 Network Layer

The interface to the network layer is very similar to that of the record layer. It is responsible for handling caching records on the local machine, as well as passing predictions between the data structure and knowledge layers. The interface specification is described in Algebra 4.5.

4.7.5 Data Structure Layer

The interface of the data structure layer depends on the data structure represented. It is recommended that the public (i.e. application-side) interface to the data structure layer be as abstract as possible to give the layer as much information as possible when performing deterministic predictions.

Algebra 4.4: Interface specification for the knowledge layer.

```
Algebra KnowledgeLayer is

    including RecordLayer KnowledgeLayer

    sorts Nat RecordFile KnowledgeLayer Client

    op newKnowledgeLayer : RecordLayer → KnowledgeLayer
    op recordLayer : KnowledgeLayer → RecordLayer

    op predict : KnowledgeLayer Client Nat → KnowledgeLayer Nat
    op accessed :  KnowledgeLayer Client Nat → KnowledgeLayer
    op predicted :  KnowledgeLayer Client Nat → KnowledgeLayer

    op loadRecord : KnowledgeLayer Nat → Record
    op storeData : KnowledgeLayer Nat Record → KnowledgeLayer

    var K : KnowledgeLayer
    var R : RecordLayer
    var N : Nat

    eq loadRecord(K, N) → recordData(recordLayer(K), N)
end Algebra
```

A data structure layer representing a spacial partitioning system, for example, could be written so that the application requests individual nodes. A better solution would be for the application to request the node at a specific depth corresponding to a given location. This would allow the data structure layer to make predictions based on an understanding of the relationships between the nodes.

In general, more complex data structures will encode more semantic information and thus give better predictions. It would be possible to construct a data structure layer representing a resizable array, for example, and use this for a wide variety of applications. In practice, however, this would not be useful since accesses to an array either follow a simple pattern or are very difficult to predict (with the majority of cases being the latter). It is, however, possible for an implementation to stack multiple data structure layer components.

In the case studies in the previous chapter, different data structure layer implementations were used for the point set and the spacial partitioning scheme. The point set was represented by a simple array, however predictions for point accesses were generated from the octree rather than directly from the array.

4.8 Reference Implementation

Two implementations of the model have been constructed to date. The first omits the network layer, and is intended for local use. The second implements a client-server ar-

Algebra 4.5: Interface specification for the network record layer.

```
Algebra NetworkLayer is

    including KnowledgeLayer KnowledgeLayer

    sorts Nat Record KnowledgeLayer NetworkLayer

    op cacheRecord : Nat → LocalCache
    op

    var N : NetworkLayer
    var C : LocalCache
    var I : Nat

    op newNetworkLayer : KnowledgeLayer → NetworkLayer
    op knowledgeLayer   : NetworkLayer → KnowledgeLayer

    op netPredict : NetworkLayer Client Nat → NetworkLayer Nat
    op netAccessed :  NetworkLayer Client Nat → NetworkLayer
    op netPredicted :  NetworkLayer Client Nat → NetworkLayer

    op netLoadRecord : NetworkLayer Nat → Record
    op netStoreData : NetworkLayer Nat Record → NetworkLayer

    var W    : NetworkLayer
    var K    : KnowledgeLayer
    var C    : Client
    car R    : Record
    var N, M : Nat

    eq knowledgeLayer(newNetworkLayer(K) = K
    eq knowledgeLayer(netPredicted(W, C, N)) = predicted(
       knowledgeLayer(W), C, N)
    eq knowledgeLayer(netAccessed(W, C, N)) = accessed(knowledgeLayer(
       W), C, N)
    eq knowledgeLayer(netPredict(W, C, N)) times M = predict(
       knowledgeLayer(W, C, N)) time M

    eq netLoadRecord(W, N) = loadRecord(knowledgeLayer(W), N)
    eq knowledgeLayer(netStoreData(W, N, R)) = storeData(
       knowledgeLayer(W), N, R)

end Algebra
```

chitecture, with the server written in Erlang and the client code in C. The remainder of this chapter discusses the two implementations, and shows some results from the second implementations.

4.8.1 Local Implementation

The implementation described here was used for the evaluation of prefetching algorithms in the last two chapters of this part.

The API defines one scalar data type, BOOL, which can take values of YES or NO. This type follows the C conventions for Boolean values, and so can be used in if statements and similar. If compiling as C99, Objective-C, or C++ this will use the built-in boolean type (bool, BOOL or bool respectively).

4.8.1.1 Block Layer

The block layer is the lowest layer above the kernel, responsible for moving bytes to and from the disk. As described earlier, the rôle of this layer it to load and store ranges of bytes in an on-disk file. There are a number of possible ways of doing this, and so several implementations of the block layer were created. These all implement the same interface, and so can easily be swapped.

block_aio.c is a POSIX asynchronous I/O implementation. All file reads are performed asynchronously, and grouped together into a single system-call when possible. This is accomplished using the lio_listio system call, which permits multiple AIO operations to be dispatched simultaneously. This implementation tends to provide the best performance.

block_fread.c uses the POSIX synchronous I/O calls. This may be faster than virtual memory, since it will probably not require as many small reads, however all pre-caching operations will block the calling process so it is not very efficient. This is included as a relatively simple implementation on which others can be based, rather than for production use.

block_null.c is based on the synchronous implementation, however it ignores all pre-caching requests. Using this is effectively the same as relying on virtual memory, and so can be used to test the efficiency of out-of-core algorithms.

block_mmap.c uses memory mapped I/O on files. This implementation provides hints to the virtual memory subsystem in the kernel, rather than explicitly loading the data itself. On 32-bit systems, this implementation is limited by the amount of address space available to the process, which will be under 4GB. On 64-bit systems, this should give high performance as long as the operating system respects the hints given by the madvise system call.

Two opaque data types were defined for this layer. The ooc_block_file_t type represents a file mapped at the block layer, while the ooc_cached_fragment_t type is used to represent a fragment which has been cached by the block layer. This is stored in the block layer, and contains some meta-data relating to the cached fragment (exactly what varies between implementations).

Listing 4.6: Two structures defined by the block layer.

```
typedef struct
{
    off_t address;
    int length;
    int count;
} ooc_block;

typedef struct
{
    ooc_cached_fragment_t* list;
    unsigned int fragments;
}* ooc_fragment_list_t;
```

Listing 4.6 shows the two structures defined by the block layer. The ooc_block structure contains a request for a set of contiguous, constant-sized, blocks to be read; these are specified by the address, length of a single block, and the number of blocks. This is used to reduce the overhead of reading contiguous blocks from the disk. The ooc_fragment_list_t structure is used to return a list of fragments that have been cached.

The following functions provide the interface to the block layer:

```
BOOL ooc_block_init(void)
```

This function must be called before any other block layer functions, and performs any set-up required of the block layer. In general, this should only ever be called by implementations of the record layer, and should only be called once.

This function returns YES on success and NO on failure.

```
ooc_block_file_t ooc_map_block_file(char * fileName, BOOL
    readOnly);
```

This function maps the specified file to memory and returns a ooc_block_file_t . The returned value must be used with any subsequent calls to block-layer functions involving this file.

A return value of OOC_INVALID indicates that the call failed.

Parameters:

fileName The path to the file to be loaded.

readOnly Whether the file should be modifiable. If it is marked as read only, then changes will not be flushed back to disk which can give higher performance.

```
void ooc_unmap_file(ooc_block_file_t file)
```

This function is the inverse of ooc_map_block_file(). It takes a file identifier and frees the resources associated with it. Further calls to other functions using the same file are invalid.

```
ooc_fragment_list_t ooc_cache_bytes(ooc_block_file_t file,
    ooc_block bytes[], unsigned int blocks)
```

Instruct the block layer to cache the specified blocks. The return value is a list of fragments, which must be freed using ooc_block_free_fragment_list(). The arguments taken by this function are:

file The relevant ooc_block_file_t created by a call to ooc_map_block_file().

bytes An array of ooc_block structures containing the fragments to be cached.

blocks The number of elements in the array.

Note that the fragments returned in the fragment list by this function are likely to be futures, rather than real instances of the fragments. This function should begin an asynchronous operation to cache the data, and return immediately[4]. The ooc_get_bytes function will block if the read operation has not completed by the time the data is actually required.

```
void ooc_block_free_fragment_list(ooc_fragment_list_t list);
```

Free an ooc_fragment_list_t returned by ooc_cache_bytes(). This function must be called to free these structures, they may not be freed manually. This function exists purely for efficiency reasons, allowing the block layer to re-use ooc_fragment_list_ts. Since one of these is created every time a caching request is issued, re-using them eliminates a large number of malloc() and free() calls.

```
void ooc_grow_block_file(ooc_block_file_t file, size_t bytes);
```

Increases the size of a mapped file. This will silently fail if the file was marked as read only when it was mapped, as write to a read-only fail are invalid operations.

Arguments:

file The relevant ooc_block_file_t created by a call to ooc_map_block_file().

bytes The number of bytes by which the file size should be increased.

```
void * ooc_get_bytes(ooc_block_file_t file,
    ooc_cached_fragment_t fragment);
```

Retrieves the data associated with a fragment. This function will block if the underlying caching operation on the fragment (e.g., aio_read()) has not yet completed. In the case of the null implementation, the caching operation is a no-op, and this function will perform a synchronous load.

Arguments:

file The relevant ooc_block_file_t created by a call to ooc_map_block_file().

fragment The fragment, created with a call to ooc_cache_bytes().

[4]This is not the case in the synchronous and null implementations of this layer.

```
        void ooc_uncache_bytes(ooc_block_file_t file,
            ooc_cached_fragment_t fragments[], unsigned int
            fragmentCount);
```

Uncaches an array of fragments, freeing the memory associated with them and flushing the changes back to disk if required. The exact operation of this function depends on whether the file was opened read-only. If so, the in-core copy will be discarded. If not, the data will be flushed back to disk. Note that some block-layer implementations use memory protection mechanisms to prevent modification of data loaded from read-only files, so the ability to modify data should not be assumed.

Arguments:

file The relevant `ooc_block_file_t` created by a call to `ooc_map_block_file()`.

fragments The array of fragments to be flushed.

fragmentCount The number of fragments in the array.

4.8.1.2 Record Layer

Currently, a single implementation of the record layer is included. This is designed to be efficient, and makes extensive use of object pools to eliminate overhead of allocating and de-allocating memory (around twenty percent of run time in profiling).

One opaque data type is defined by the record layer. `ooc_record_file_t` represents an instance of the record layer, and is used to identify the file to the system. A small set of functions provided to interface with this layer:

```
        BOOL ooc_init(void)
```

This function must be called before any other record layer functions and performs any initialisation required. A return value of `YES`. This is responsible for calling the matching function in the block layer.

```
        ooc_record_file_t ooc_map_file(char * fileName, unsigned int
            recordSize, BOOL readOnly, unsigned int memorySize)
```

Designates a file used as out-of-core storage by the record layer. This function will create a block layer instance corresponding to the specified file, and associate it with a new instance of the block layer.

fileName The name of the file to open.

recordSize The size of a single record. Note that records at this layer are fixed size in the current implementation.

readOnly A flag indicating whether this file should be read only. Read only out-of-core files have significantly higher performance than read-write files, and so should be used whenever possible.

memorySize The amount of memory to allocate in-core as a cache for this file. Note that this does not include the size of control structures.

```
void ooc_grow_record_file(ooc_record_file_t file, unsigned int
    growBy)
```

Increase the size of a previously mapped file. Note that in most implementations this is a relatively expensive operation and so should be called as infrequently as possible.

The new part of the file is initialised with zeros.

file The file to increase in size.

growBy The number of bytes by which to increase the size of this file.

```
BOOL ooc_precache_record(ooc_record_file_t file, unsigned int
    recordNumber, unsigned char priority)
```

Caches a single record with a specified priority. This is a convenience method for occasions when a single record is required. Since caching a single record is not a very efficient thing to do (assuming the out-of-core storage device is mechanical), it should be avoided for performance reasons.

file The file containing the record.

recordNumber The index of the record to be cached.

priority The priority with which the record should be cached.

This is a convenience method, and is implemented as an inline function using ooc_precache_records. Future implementations may use this to provide an optimised method for caching a single record.

```
void ooc_precache_records(ooc_record_file_t file,unsigned int
    records, unsigned int recordNumbers[],unsigned char
    priority)
```

Caches a group of records. For highest performance, the records should be sorted, although this is not required. Adjacent records will be fetched with a single disk read, allowing for more efficient use of disk bandwidth.

file The file containing the records.

records The number of records to be cached.

recordNumbers The indexes of the records to be cached.

priority The priority of the cached records.

```
BOOL ooc_in_core(ooc_record_file_t file, unsigned int record)
```

Determines whether a particular record is in-core. Note that this only actually indicates whether the block layer has been instructed to load a record, not whether the loading actually succeeded in the current implementation.

file The file containing the record.

record The index of the record to be inspected.

```
        void * ooc_retain_record(ooc_record_file_t file, unsigned int
            recordNumber)
```

Increments the reference count of the indicated record and returns a pointer to the record's data. A record whose reference count its non-zero will not be evicted from main memory, so it is important that no more records are retained than fit in the memory allocated as a cache for any given file. Records with a reference count of zero will gradually have their priorities decayed until they are evicted.

The return value is a pointer to the data represented by this record.

Arguments:

file The file containing the record.

recordNumber The index of the record to be retained.

```
        void ooc_release_record(ooc_record_file_t file, unsigned int
            recordNumber)
```

Decrement the reference count of a record. Once the reference count reaches zero, a record may be evicted from main memory (although this typically does not happen immediately).

Arguments:

file The file containing the record.

recordNumber The record to be released.

```
        void ooc_free_record_file(ooc_record_file_t file)
```

De-allocates a record-layer file and flushes all cached data back to disk. This must be called after operations on a file are completed to ensure that resources are freed correctly and data is written back to the disk. This function also frees the associated block layer instance.

file The record-layer file to free.

4.8.1.3 Data Structure Layer

This layer presents higher level abstractions of data structures to the user, and must be easily extensible by third party developers. Currently, an out-of-corearray implementation is provided, which either generates predictions based on linear accesses, or takes them from an external source. An octree implementation is also provided for point clouds. The points are stored in an out-of-corearray, and hinting is provided based on which nodes are loaded.

Four versions of the octree are provided, one for each of the prediction algorithms described in the previous chapter.

4.8.2 Client-Server Implementation

The client-server implementation consists of two components, a server written in Erlang and a client interface written in C. The server component provides a mechanism for implementing hierarchical structures of agents which generate predictions.

Location	Memory	Hard Disk	Network
Latency	5ns	10ms	1-3000ms
Throughput	2GB/s	50MB/s	8KB/s-128MB/s

Table 4.2: Approximate timings for different storage locations

Considering the relative speeds of disk and network access (as shown in Table 4.2), a significant speed benefit can potentially be obtained by using the hard disk as an intermediate cache of data, between the network and main memory. Disk accesses, however, are much faster when sequential, so it is important to lay the data out on the disk sensibly, and to minimise the amount of memory used to store translations between in-core and disk addresses.

The protocol defined for this implementation uses UDP. The reason for this is that UDP does not guarantee delivery or delivery order. This means that latency, the primary cause of slow-down in out-of-core algorithms, will be kept to a minimum. Using TCP, or another protocol which guarantees delivery order, would potentially increase latency dramatically since a single dropped packet would result in every subsequent packet being delayed until the original packet had been re-sent. There are two possible cases for a dropped packet. It is either speculatively transmitted based on predictions from either the client or the server, or it is required immediately. If it is immediately required then the dropped packet will cause the client to stall until it can be retransmitted. TCP will perform this retransmission automatically, while UDP requires the retransmission to be performed higher up the protocol stack. If the packet contains data that is not immediately required, then TCP will buffer all subsequent packets until the retransmission has occurred. In this case, the packet containing the data which is required immediately could be left waiting in the client's receive buffer until some speculatively requested data is received.

All strings in the following sections are encoded as UTF-8, and all integers encoded in network byte order.

4.8.2.1 Message Types and Flow

Four types of message can be sent by a client:

1. A request for access to a file.
2. A request for a group of records.
3. A notification of use of records.
4. A notification that the file is no longer required

Similarly, a server may send two messages in response to these requests:

1. An acknowledgement of a request for access to a file, either granting or denying access.
2. A group of records from a file.

No data may be sent to a client which has not yet registered for access to a file, however data may be sent as soon as a client has been authorised. For an out-of-core octree, for example, it is certain that the first node any client would wish to access would be the root node, and it is probably that the next node would be one of the children. If the knowledge layer in the server has determined these trends then it is at liberty to pre-emptively send un-requested data to a client.

4.8.2.2 Message Detail

This section includes detailed description of the internal layout of each of the message types described in the previous section.

The first byte of each message shall be an identifier indicating the type of the message. Since we are only defining 6 message types, this gives a significant amount of room for expansion of the system.

Several message types apparently permit an arbitrary number of values. In these cases that the total message size should always be less than 536 bytes. This, added to the 40-bytes required for an IP header gives 576 bytes, the minimum packet size defined by RFC 894 which all Internet nodes must be able to transmit without fragmentation. On other networks, this size can be increased, for example a system deployed entirely over ethernet should be aware that the MTU for ethernet is 1500 bytes and adjust the message size accordingly.

Client Messages Upon initialisation, a client should send a message of the type defined in Table 4.3 requesting access to a particular file. The user is identified by a combination of their IP address and a 128-bit integer which is negotiated out of band. Each user ID is only valid for a single connection, eliminating the possibility of replay attacks.

bytes	value	meaning
0	1	Message type
1	1 $\rightarrow$read, 2 $\rightarrow$write	Values anded together to produce requested access permissions.
2-17	User ID	User ID negotiated out-of-band.
2-3	Length	Length of the file identifier.
4-	File ID	UTF-8 encoded file identifier.

Table 4.3: A request for access to a file.

A client requests records from a file by sending messages of the type defined in Table 4.4. The server must transmit all records requested with a priority of 255. It may, at its own discretion, ignore all others or use them as input for its own prediction mechanisms.

bytes	value	meaning
0	2	Message type
1-4	File ID	A unique identifier returned when the file is opened.
5	1-255	Priority of records.
6	1-255	Number of records requested.
7-	Array of 32-bit values	Indexes of the records requested.

Table 4.4: A request for a group of records.

Once the client actually makes use of a record, as opposed to speculatively caching it, then it is required to send a notification to this effect of the form described in Table 4.5. For efficiency reasons, it is recommended that the client batch these notifications. The current implementation maintains a buffer of used records, and sends a notification of all of them as soon as it has enough to efficiently fill a packet. The size of a packet is defined by the MTUSIZE constant, which is tunable at compile time. Typical values of this would be 576 bytes for the Internet, or 1500bytes for an ethernet network.

bytes	value	meaning
0	3	Message type
1-4	File ID	A unique identifier returned when the file is opened.
5	1-255	Number of records used.
6-	Array of 32-bit values	Indexes of the records used.

Table 4.5: A notification of use of records.

Once a client has finished with a particular file, it should send a notification to this effect, in the form of a message of the type defined in Table 4.6. The server is then able to free all resources associated with this client.

bytes	value	meaning
0	4	Message type
1-4	File ID	A unique identifier returned when the file is opened.

Table 4.6: A notification that the file is no longer required

Server Messages The server should return a client-unique, non-zero identifier for each request where access is granted as shown in Table 4.7. A response with the identifier set to 0 indicates that access was denied.

bytes	value	meaning
0	5	Message type.
1-4	File ID	A client-unique identifier.
5-8	File Size	The number of records in this file
9-12	Record Size	The size of a single record
13-16	Fragment Size	The size of each transmitted fragment. Equal to record size if
17-20	Record Fragments	The number of fragments a record is divided into for sending.
21-24	Last Fragment Size	The size of the last fragment in a record. This allows for record sizes that are not divisible by the fragment size.

Table 4.7: An acknowledgement of a request for access to a file, either granting or denying access.

Records are sent to the client should be in the form described in Table 4.8. If the record size is greater than the MTU size, then the Split Format value from the acknowledgement packet will have been set. If this is the case, then the record count indicates the ordered fragment of the message within the record, rather than the number of records (which will always be less than one in these cases).

bytes	value	meaning
0	6	Message type
1-4	File ID	A unique identifier returned when the file is opened.
5-8	Record Count	The number of records included in this message
9-12	Record ID	Index of first record.
13-	Record	The first record in raw format.
n-	Records	Subsequent records in the same format as the first.

Table 4.8: A group of records from a file.

Security Message-level security is beyond the scope of this document. If it is required then the connection should go via IPSec [163], which provides packet-level security.

4.8.2.3 Cache File Layout

In order to facilitate the efficient and fast use the local hard disk as an out-of-core cache, it is important that the data is laid out in a way that closely mirrors the layout of the remote file. It is also important that the local file be smaller than the remote file, for obvious reasons.

For this reason, the chosen approach is adapted from CPU caching strategies. A file is created on disk of a fixed size that is $r \times 2^a$ bytes, where r is the number of records and a is an arbitrary value defined by the amount of on-disk space allocated. Record indexes are split into two sections, the most significant n and the least significant m bits. n is used as an index in a hash table to look up lines of the file, while m is the index within a line.

A count of all retained records within a line should be maintained at all times, and a line only replaced with a different start index when this count is zero.

4.8.2.4 Client Interface

The client interface consists of the following functions:

Listing 4.7: Client interface to remote out-of-coreserver

```
void ooc_send_packet(struct ooc_file * file, struct ooc_packet *
    packet);
struct ooc_record_request_packet * ooc_request_get_request(struct
    ooc_file * file, unsigned char priority, unsigned int elements);
void * ooc_retain_record(struct ooc_file * file, uint32_t index);
void ooc_release_record(struct ooc_file * file, uint32_t index);
struct ooc_file * ooc_connect(char * server, uint16_t port, char *
    filename);
void ooc_close(struct ooc_file * file);
```

These are similar to the record layer in the local implementation, and correspond to the top half of the network layer in the five layer model. There is no standard function for requesting records defined, for efficiency reasons. Instead, the functions for creating and sending request packets are exposed directly to data structure layer developers. The SET_RECORD_INDEX macro can be used to easily create these packets. Listing 4.8 shows an example of a series of 100 records being requested from record i to record i+99.

Listing 4.8: Example showing 100 records being requested

```
        struct ooc_record_request_packet * request =
            ooc_request_get_request(file, 200, precacheSize);
        for(unsigned int j=0; j<100; j++)
        {
            SET_RECORD_INDEX(request, j, j+i);
        }
        ooc_send_packet(file, (struct ooc_packet*)request);
```

The retain and release functions work exactly as their local counterparts (in terms of interface). Note that notifications of use do not have to be explicitly generated by developers using this interface. Instead, they are automatically created and set when the retain function is invoked.

4.8.2.5 Agent Interface

Agents are defined in the server by a set of four higher-order functions. The first take no arguments and returns an value representing the initial state of the agent. For complex agents, this will be the pid of a running (Erlang) process.

The next two functions change the state of the agent. These are called to update the state when a message indicating that records have been used or requested is received. Both take the state and a tuple containing the client, file and list of records as arguments, and return a new state.

The final function is used to generate predictions. It takes the state and a tuple containing the client, file and last set of records to be requested as arguments, and returns a tuple containing a list of records and a confidence. This allows the agents to be combined into trees using maximum or threshold operators; the combining agent will either return the best prediction, or all predictions with a confidence over a certain threshold.

Agents are loaded from a configuration file when the server is started. When a new connection occurs, agents in three categories are started:

Global Agents contain simple logic that can apply to any file (e.g., linear predictions). These are shared between all files shared by the server.

File Agents predict file-related behaviour. Some of these might contain a priori information about the structure of the file, others might be general knowledge-based agents that learn access patterns based on previous uses.

File-User Agents are similar to File Agents, but are unique to the file and user pair. This allows for user-specific knowledge to be used for prediction.

Note that agents in different rôles are able to share a knowledge base. A knowledge base implementation is provided that permits concurrent access to a persistent tuple store.

4.8.2.6 Preliminary Results

The distributed implementation is the subject for future work and is not yet ready for detailed evaluation, however preliminary results indicate that this approach has potential.

Linear access to a data set is fairly common in a number of applications. It is also trivial to write a predictor which accurately predicts linear access, providing some calibration. The results from this preliminary testing show the benefit of prefetching when applied to a remote data server.

The linear access test program read each record in a remote data set in order. For each record, the client process slept for one μs to simulate some computation before going on to read the next. It performed two passes through the data set.

Three different tests were performed using sequential, linear accesses:

1. No prediction; records are requested as they are used.

RTT (ms)	Predictor	Time (s)	Cache Hit Rate
100	None	6136.59	0.005%
100	Knowledge-based	4864.29	21.17%
100	Perfect	4864.29	21.17%

Table 4.9: Results from linear read tests.

2. Knowledge-based prediction using an algorithm which attempts to predict linear accesses.
3. Perfect prediction; records are requested by the client in blocks of 200 before they are used.

The difference between the two prediction strategies is which end of the system performs the prediction. In the 5-layer model, both the data structure layer and the knowledge layer are allowed to perform predictions. In our second test (the 'perfect' predictor) the data structure layer, on the client side, performs the prediction. In this case, it is guaranteed to be accurate since our test program is purely deterministic.

In the third test, the client makes no predictions. The knowledge layer on the server detects the presence of a sequential access pattern and begins sending data speculatively. This has two advantages over performing predictions in the data structure layer. The first is that, in the distributed setting, the knowledge layer can build a detailed knowledge base from multiple users, enabling predictions to be refined more accurately. The second is that moving the predictions to the server frees up the client to devote all of its processing power to actually processing the data. The last point is particularly important when systems such as PDAs are considered on the client side.

It can be seen from the results in Table 4.9 that the knowledge-based approach works well on predictable data accesses and that accurate prediction provides a significant performance increase on remote accesses. The results from both timings were identical in this simple test since the access pattern was predicted with equal accuracy by the predictors at both ends. In a more complex example, the knowledge-based implementation would be expected to perform less well than a perfect predictor, however writing a perfect predictor for more complex access patterns is less likely to be feasible. Future work on this system will involve adding prediction agents which work on less obviously predictable access patterns.

These results were collected over a 100Mbit network with an average round trip time (RTT) of 100ms.

4.9 Conclusions

This chapter has presented a 5-layered model for out-of-core data management systems. Two implementations of this model were described, indicating that it is a feasible model for real-world use.

The model incorporates network transparency as well as both algorithmic and knowledge-based prediction. The self-optimising component required for an autonomic system is provided in the knowledge layer.

Chapter 5

Out-of-Core Sample Problem

Contents

This chapter describes a sample problem where external memory is important, and discusses a prefetching strategy. The prefectching strategy described here fits predominantly into the algorithm category, but has some data structure-specific elements. Work described here was originally presented in The Visual Computer [74].

5.1 Introduction

Point-based modeling and rendering is a collection of techniques that enable direct processing of complex geometric objects represented by large discretely sampled point clouds [249, 266]. Point clouds are usually rendered directly, using forward projection and image-space composition of point splats [329, 85, 224]. In terms of computational costs, this approach is highly attractive, facilitating the use of graphics hardware and stream-based data processing. However, by minimizing the interaction between points in the object space, it does not easily permit the generation of some visual effects, such as shadows, reflection and refraction, especially when considering a complex and arbitrary scene composed of multiple point-based objects.

An alternative direct rendering approach is to organize point-based objects in a volume screen graph and synthesize images using discrete ray tracing [59]. Not only does this approach address some of the shortcomings of splatting, but it also facilitates combinational and comparative visualization of volume and point datasets by allowing both to co-exist in the same volume scene graph. For example, in Figure 5.1, a translucent bunny, built from a digitized point set, is combined with a heart captured as a volume dataset. In Figure 5.2, four dragons, also built from a digitized point set, are immersed in artificial clouds that are modeled as a volume dataset. Despite these benefits, the approach is yet to deliver a real-time solution. Nevertheless, it can no doubt bring benefits to some visualization and graphics applications; and with the rapid advances in both hardware technology and distributed computing, its usability will be further enhanced in the coming years.

The fundamental bottleneck of ray tracing a volume scene graph with multiple point clouds is that it demands an overwhelming amount of memory to accommodate all point datasets and their associate control structures such as octrees. Let $\{P_1, P_2, \ldots, P_n\}$ be a set of point clouds contained in a volume scene graph, and $\{C_1, C_2, \ldots, C_n\}$ be their corresponding control structures. To address the *scalability* of this approach, one needs to consider the following issues:

- *The size of each individual point cloud*, $|P_i|$ — With modern digitization technology, P_i may easily contain millions, or even billions of points.
- *The size of the control structure for each point cloud*, $|C_i|$ — A discrete ray tracer benefits particularly from a spatial partitioning scheme, such as an octree. However, to achieve optimal speed efficiency, it is not uncommon that the control structure for partitioning a point cloud may consume more space than the corresponding raw dataset, i.e., $|C_i| > |P_i|$.
- *The number of point clouds in a volume scene graph*, n — The growth of n has a profound impact on the total space requirement for both point datasets and their control structures.
- *The complexity of ray path predication* — For a simple ray-casting algorithm, it is possible to preprocess a volume scene graph and predetermine a static data prefetching strategy. The more visual effects the ray tracer incorporates, the more difficult the ray path predication will be. With some complex visual effects, it is likely that static preprocessing would not yield much benefit.

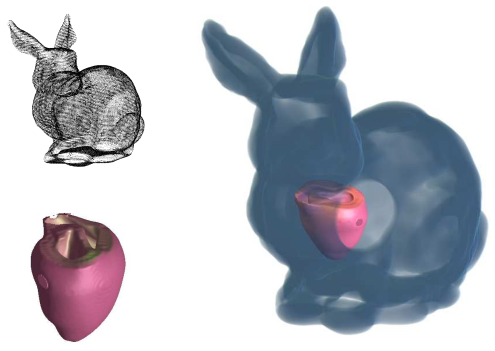


Figure 5.1: Combining the Stanford Bunny point set with a conventional volume dataset (San Diego rabbit heart) in a volume scene graph. (Image courtesy of Prof. Min Chen)

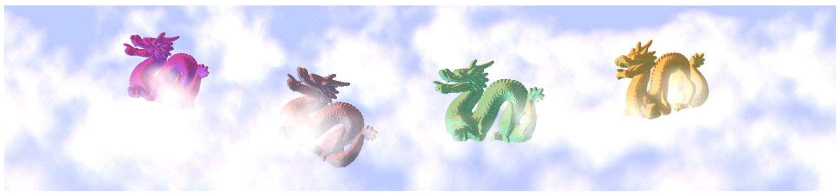


Figure 5.2: A volume scene graph consisting of four dragons modeled using a point-set (Stanford Dragon) and artificial clouds represented by a volume dataset (Erlangen Clouds). (Image courtesy of Prof. Min Chen)

This section describes an out-of-core approach for ray tracing volume scene graphs in a scalable manner. A technique based on a dynamic, in-core working sets, is introduced, which addresses the combined difficulties in pre-determining data mixing patterns and ray paths, and hence data access patterns. It is assumed that the number of points in each individual point cloud is much larger than the number of point datasets in a scene, i.e., $|P_i| >> n$. Hence octrees are utilised to partition individual point datasets, and use the bounding boxes of scene graph nodes to partition the scene. During ray tracing, the point datasets and their octrees are stored out-of-core, and the required octree nodes and point data are pre-fetched automatically according to access patterns predicted based on captured knowledge of ray-data intersection. Testing results have shown that this technique is scalable, and enables volume scene graphs composed multiple point clouds to be rendered directly on desktop computers.

The remainder of this chapter is organized as follows. A brief review of point-based modeling and rendering techniques, and out-of-core techniques, is given in Section 4.4. In Section 5.2, the basic in-core method for modeling and rendering volume scene graphs with multiple point clouds is outlined, and highlight the correlation between modeling quality and memory consumption. In Section 5.3, a ray-driven technique for predicting the working set automatically is presented, and the dynamic algorithms for predicting octree access and for pre-caching octree nodes and the corresponding point data are outlined. In Section 5.4, experiments on the scalability of the technique are demonstrated using working sets and datasets of different sizes, and qualitative and quantitative analyses are presented. This is followed by concluding remarks in Section 5.5.

5.2 Modeling and Rendering Multiple Point Sets

5.2.1 Point-based Volume Object (PBVO)

Volume objects can be defined procedurally as well as built from discretely sampled datasets such as CT and MRI scans. In particular, they can be defined on point clouds using appropriate *radial basis functions* [59], and can therefore be integrated into a volume scene graph as elemental volume objects at terminal nodes. For example, consider a discretely sampled point cloud $P = \{p_1, p_2, \dots, p_m\}$, where each p_i is associated with a confidence value and an intensity value. We can map the confidence value to a radius r_i, and the intensity value to an opacity value, o_i. The former defines the *radius of influence* of p_i, and the latter contributes to the visibility of points within its radius of influence.

Consider a radial basis function, $\omega(q, p_i, r_i)$, such that,

$$\omega(q, p_i, r_i) = 0, \forall q, \parallel q - p_i \parallel > r_i$$

where $\parallel q - p_i \parallel$ denotes the Euclidean distance between q and p_i. For a collection of opacity values, $o_1, o_2, \ldots, o_m$, associated with $p_1, p_2, \ldots, p_m$ respectively, a scalar field O is therefore defined using a *blending function* as:

$$O(q) = \sum_{1 \le i \le m} \omega(q, p_i, r_i) v_i. \tag{5.1}$$

Several blending functions were considered in [59]. Images in this chapter were rendered using either the function proposed by Wyvill *et al.* [344]:

$$\omega^W(q, p_i, r_i) = \begin{cases} 1 - \frac{4u_i^6 - 17u_i^4 + 22u_i^2}{9} & \text{if } \parallel q - p_i \parallel < r_i \\ 0 & \text{if } \parallel q - p_i \parallel \ge r_i \end{cases},$$

or that proposed by Chen [59]:

$$\omega^C(q, p_i, r_i) = \begin{cases} \left(1 - \frac{2u_i^{2\alpha_1}}{1 + u_i^2}\right)^{\alpha_2} & \text{if } \parallel q - p_i \parallel < r_i \\ 0 & \text{if } \parallel q - p_i \parallel \ge r_i \end{cases}.$$

In both functions, $u_i = \parallel q - p_i \parallel / r_i$.

$O(q)$ in Eq. (5.1) in effect defines the opacity of every point in a 3D volumetric domain which is the union of the spherical bounding volumes of all points in P. Hence, $O(q)$ defines the essential component of a volume object and can be rendered using discrete ray tracing. Such a volume object is called a *point-based volume object* (PBVO). We can specify luminance properties of a PBVO using transfer functions, or by building the relevant scalar fields in a similar manner to $O(q)$. Figure 5.3 shows a PBVO defined on a very large point cloud of over 14 million points. Its opacity field was constructed using a radial basis function with $r_i = 2, o_i = 1, i = 1, 2, \ldots, m$.

5.2.2 Volume Scene Graphs

In the theoretic framework of *Constructive Volume Geometry* (CVG) [60], a *volume scene graph* is an algebraic expression, called a *CVG term*, which involves a class of spatial objects and a family of constructive operations. In practice, a CVG term can be represented by a tree, where constructive operations are defined at non-terminal nodes, and elemental volume objects are defined at terminal nodes of the tree. Each subtree defines a composite volume object, while the root represents the final composite volume object, or the *scene*. To facilitate the sharing of low level object data, we allow a CVG term to be realized

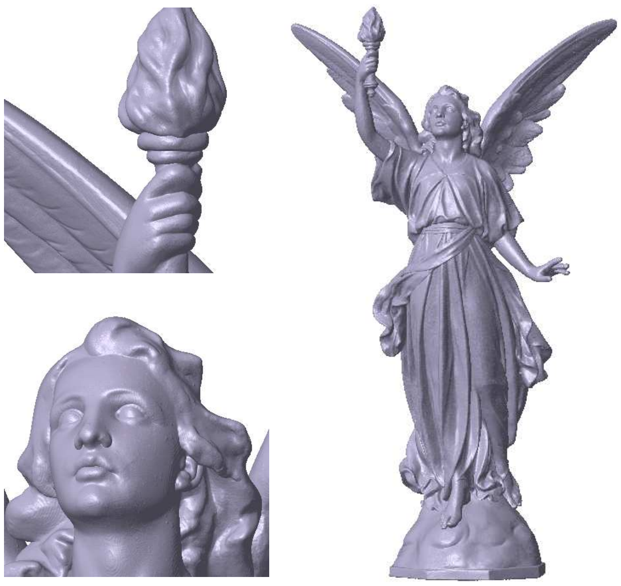


Figure 5.3: A point-based volume object defined on the the Stanford Lucy dataset of 14,027,872 points.

using a directed acyclic graph with a single root, hence resulting in a volume scene graph. Geometrical transformations and transfer functions can be applied at each graph node.

Figure 5.4 shows the results of applying CVG operations to a PBVO, **r**, built from the Stanford bunny point set, and a procedurally defined cylindrical object, **c**. The example shown in Figure 5.1 involves the use of a union operation and a difference operation. The latter is used, in conjunction with a cylindrical object, to remove part of bunny for exposing the heart. The example in Figure 5.2 demonstrates that multiple PBVOs in a volume scene graph can share the same point set.

5.2.3 Discrete Ray Tracing

So far, discrete ray tracing is still the most appropriate means for directly rendering a volume scene graph which features multiple volume objects, solid or translucent. The basic ray tracing mechanism is to sample at regular intervals along each ray cast from the view position. At each sampling position s, we recursively determine if s is inside the bounding box of the current CVG subtree, until we reach a terminal node. If s is inside the bounding box of the terminal node which contains an elemental volume object, we evaluate its opacity field $O(s)$ and possible other luminance attributes.

When an opacity field is defined on a point cloud $P = \{p_1, p_2, \ldots, p_m\}$, it is necessary to identify a subset of points, $P' \subseteq P$, such that

$$P' = \{p'_i \,|\, p'_i \in P \text{ and } \parallel s - p'_i \parallel \leq r_i\}.$$

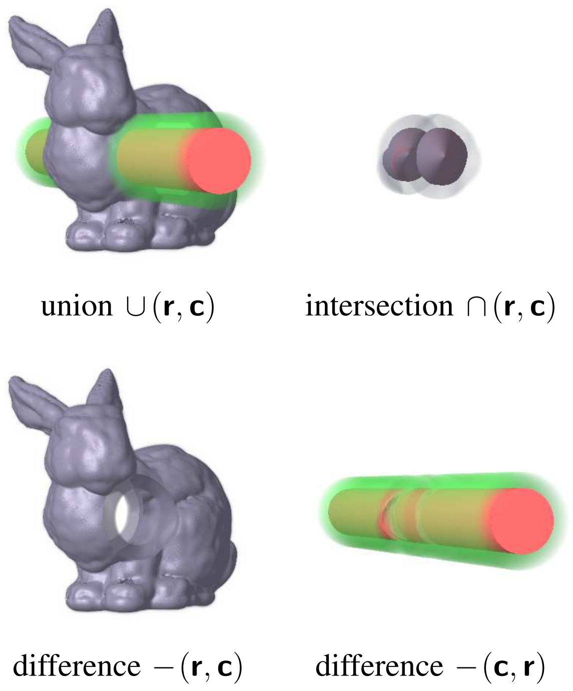

union $\cup(\mathbf{r},\mathbf{c})$
intersection $\cap(\mathbf{r},\mathbf{c})$
difference $-(\mathbf{r},\mathbf{c})$
difference $-(\mathbf{c},\mathbf{r})$

Figure 5.4: Applying three basic CVG operations to a PBVO $\mathbf{r}$ and a procedurally defined cylindrical object $\mathbf{c}$.

Given such a subset, we can sample the radial basis function of each $p'_i \in P'$, and obtain a scalar value by using the above-mentioned blending function.

5.2.4 The Benefits and Costs of Using Octrees

For a large point cloud, the most expensive cost in rendering a point cloud P is the identification of the subset P', as it involves a distance calculation against every point $p_i \in P$, thereby limiting the scalability when $|P|$ increases.

For each large point cloud P in the volume scene graph, we therefore utilize an *octree* structure for partitioning the points in the local data coordinate system of P. In each level of the hierarchy, a subtree contains only those points, which have some influence in the bounding box of the subtree. It is important to note that due to the non-zero radius of influence of each point, and the likely overlaps among the 'volumes of influence' of different points, a point element can belong to more than one leaf nodes. We therefore store only indices, rather than the records of points, in the leaf nodes of an octree.

In comparison with a brute force ray tracer, an octree-based ray tracer can have almost linear speedup in relation to the sizes of point clouds, if there is sufficient space for an octree that provides a sufficiently fine partition of a point cloud. Table 5.2.4 shows the speedup pattern in rendering three point sets different sizes, where points are placed randomly on a spherical surface.

However, Table 5.2.4 also shows that, for large point clouds, the amount of space con-

height ▼	# points ▶	1000	10000	100000
	speedup	23.75	36.62	26.92
H = 3	space	20 MB	155 MB	1499 MB
	η_{max}	45	411	3895
	speedup	49.91	318.84	446.81
H = 5	space	196 MB	741 MB	5708 MB
	η_{max}	15	89	739
	speedup	46.61	429.01	1122.27
H = 7	space	901 MB	12189 MB	63634 MB
	η_{max}	9	48	348

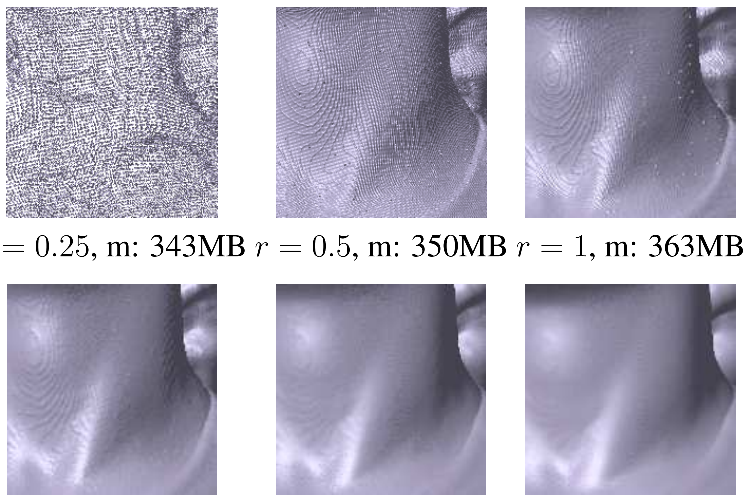
Table 5.1: Testing results for ray casting three randomly generated point clouds. The octree height is limited to 3, 5 and 7 respectively. η_{max} is the highest number of points that occupy a leaf node in the octree.

sumed by an octree can be quite noticeable, especially when points are densely placed, the radius of influence is set to a relatively large value, or the limit for octree height is generously set.

Note that the size of the octree can grow significantly faster than the point set. This is due to the fact that each point must be present in every octree leaf node in which there exists a point where its radial basis function evaluates to a number greater than zero. Depending on the radial basis function, the point value, and threshold value used, a single point may have a radius of effect spanning a large number of octree leaf nodes.

Figure 5.5 illustrates the effect of varying the radius of the radial basis function. It shows a zoomed-in section from the neck of the statue shown in Figure 5.3. As the radius is increased, the points blend together more smoothly to form the appearance of an iso-surface, and at the same time, the octree structure consumes more space for dealing with the increasing ‘volume of influence’ of each point. In fact, when we ran our in-core ray tracer on a desktop computer with 1GB memory, any setting with $r > 6$ encountered some difficulties, resulting in excessive virtual memory swapping and an unreasonable amount of system time overhead. Additionally, the octrees used in this set of examples are limited to only six levels, no where near the optimal depth for such a large dataset. Often some leaf nodes of the octree contained over 100,000 indices to points.

It has often been suggested that a kD-tree [186] could be deployed instead of an octree in this application as it has been successfully used in conjunction with many rendering algorithms such as ray tracing (e.g., [324, 325]). We found that this application is not able to take advantage of the kD-tree technique as easily as surface-based approaches, for several reasons. Firstly a kD-tree is most effective when points are considered to be unrelated or have the same small radius. Neither condition holds in this application, as it is assumed that points may have different radii (see Section 5.2.1 and [59]). Secondly, a kD-tree, like a BSP tree, focuses the precise order of primitives in relation to a given viewing (or ray) direction. It relies on search to identify neighboring primitives, for instance,



$r = 0.25$, m: 343MB $r = 0.5$, m: 350MB $r = 1$, m: 363MB

$r = 2$, m: 512MB $r = 4$, m: 676MB $r = 6$, m: 889MB

Figure 5.5: Rendering of Lucy's neck with varying radii for the radial basis function, from 0.25 to 6. One can observe the improvement of image fidelity proportionally to the increase of the radius (r), which also leads to the additional consumption of memory (m).

in rendering point-based implicit surfaces [325]. Hence, the fast detection of an opaque surface closest to the viewing position minimizes the need for the search, the cost of which depends on the radius of points. In volume rendering, translucent objects are a common feature, which do not benefit from the precise ordering as much as opaque surfaces. On the other hand, minimal search requirement at each sampling point is better suited for rendering point-based volume objects.

If a kD-tree, or similar structure, were to be designed to accommodate overlapping radial basis functions, it would encounter the same space issue as an octree. Hence, this study of out-of-core methods is also applicable to other spacial partitioning strategies.

5.3 Out-of-core Rendering

The problem of insufficient memory has been around for almost as long as stored-program computers. Most operating systems provide some form of virtual memory [92] to help alleviate this problem. Unfortunately, visualisation tasks where the working set [91] changes rapidly do not mix well with the demand paging strategy used by most operating systems.

Cox and Elleworth [84] proposed a method by which the application could control the demand paging strategy, allowing data to be evicted in an intelligent way. The authors discovered in their analysis of demand paging systems in existing operating systems that a significant performance increase could be gained by using smaller page sizes than are common, increasing the granularity of their strategy.

An effective out-of-core, or external memory [321] strategy requires an efficient prefetching algorithm such as that proposed by Varadhan and Manocha [315] in order to prevent disk latency being the limiting factor in rendering. Traditionally such algorithms are designed based heavily on a priori knowledge of access patterns. It is theorised that this is not required for an efficient prefetching strategy.

All of the caching algorithms tested interact with the renderer by the same simple interface. They sit between the rendering code and an abstract out-of-core file controller. The renderer requests an octree node containing a specified point in 3D space. This request is processed by the caching algorithm, which requests individual nodes from the controller. The caching algorithm may also provide pre-caching hints to the controller, with an associated priority. If there is enough cache space to fulfill these requests, then the requested nodes will be loaded, with lower priority nodes being evicted to make space for higher priority ones. The priority of loaded nodes is gradually decayed over time. This gives slightly less-than-optimal performance, since there is a clear separation (preventing some compiler optimisations such as aggressive inlining) between layers, but it allows for fair evaluation of the different algorithms. Figure 5.6 shows how the various layers of the test system interact.

The asynchronous I/O controller in this system implements the block layer, as described in the previous chapter. The cache implements the record layer. The data structure layer is implemented in the prediction algorithm component. This system does not implement the knowledge or network layers from the five-layer model.

The LRU algorithm is the simplest of all. Every octree node that is requested is loaded if not cached, and stored at the head of a queue. If the cache is full, then the oldest node is evicted to make room.

The intelligent algorithm works on the assumption that access is going to be along the path of a ray. Each pair of points accessed sequentially are used to extrapolate the line of which the ray is a segment. The algorithm then determines a point on this line which falls just outside of the current octree node. It then attempts to navigate the octree until it reaches the required node. If, at any point on this navigation, it would need to enter a node which is not already in the cache, it instructs the lower layer to cache it. The next time a point is requested, it tries again to reach the next node.

This provides good cache hit rates, but often at the cost of a significant overhead in terms of disk usage.

5.3.1 Algorithm Overview

The concept of *working set* was first introduced in the context of memory management in operating systems [91]. The concept used in this work as an approach to the out-of-core management in some way resembles many typical methods found in operating systems, such as anticipatory paging.

Consider the running of an algorithm as a series of algorithmic steps $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_i, \ldots\}$, and each step is merely a functional group of an arbitrary number of instructions. A *work-*

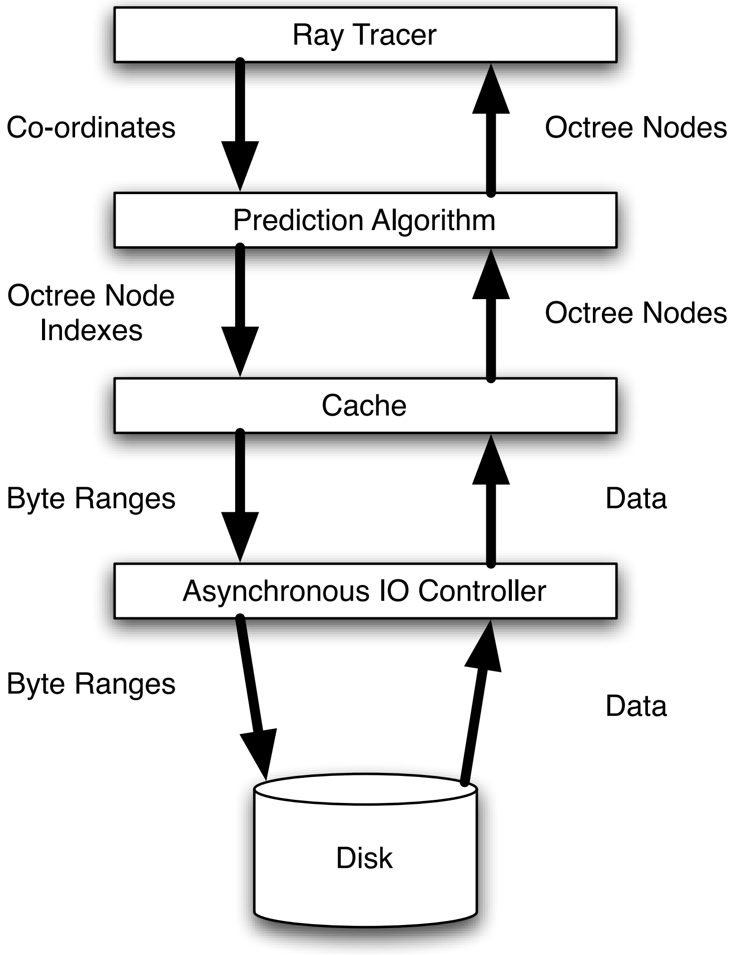


Figure 5.6: The architecture of the test system.

ing set of an algorithm in execution is the subset of the associated data structures being accessed during an algorithmic step Λ_i. Note that unlike the definition commonly used in the context of operating systems, here the duration of a working set is not a constant time window. In general, different algorithmic steps may require different execution time. If an algorithmic step is taken to be the processing of a single octree leaf node, then the running time will be $\mathcal{O}(n)$ in terms of the number of points contained within the node.

In the case of the discrete ray tracer, the primary algorithmic steps are sampling individual volume objects in a volume scene graph. For an octree defined on a point-based volume object (PBVO), the working set of a sampling operation is basically the leaf node of the octree that contains the sampling point, and the points referenced by the leaf node. Assuming that it is not possible to have all parts of the octree and the entire point set in-core at all times, the aim of the data management strategy is therefore to ensure that the working set for an algorithmic step Λ_i is located in-core, before and during Λ_i. Without such a data management strategy, the renderer will quickly encounter a situation that the working set for an incoming step Λ_j is out-of-core, stalling the renderer until it can be swapped in.

Figure 5.7 gives an overview of the data environment of a volume scene graph to be rendered by the out-of-core ray tracer. For each individual point cloud, there is a complete out-of-core copy of the entire point dataset and the corresponding octree. Each out-of-core point cloud has an in-core memory cache. The amount of in-core memory available for each point cloud is set when an out-of-core copy is created, and can be modified by the user. This allows the algorithm to be easily scaled down in highly constrained memory situations.

Since it is assumed that the number of points in each individual point cloud is much larger than the number of point datasets in a scene, comparatively the actual memory requirement for storing a volume scene graph (without the actual data for its elemental objects) is negligible. We thereby maintain the data structure for the volume scene graph in-core. Note that it is possible for different PBVOs to share the same point clouds.

The implementation consists of three main functional components, namely the *discrete ray tracer*, an *out-of-core octree controller*, and an *out-of-core point set controller*, connected as shown in Figure 5.8. The ray tracer sends requests for discrete sampling points to the octree controller. This then determines whether the node containing the requested point is currently cached in-core. If it is, then it simply returns it, and attempts to predict the next one to be accessed. The prediction strategy will be detailed in 5.3.3.

Once the octree controller has determined the next node or subtree most likely to be accessed, it attempts to pre-emptively fetch it from disk. At the same time, it informs the point set controller of the point list from the accessed leaf node. On receipt of the point list, the point set controller checks that these are cached in-core, and if not attempts to load them asynchronously.

The ray tracer then attempts to retrieve points identified by the octree node from the point set. By this time, they should already be cached in-core. Hence, it is the octree controller and point set controller that try to make sure the working set related to each sampling

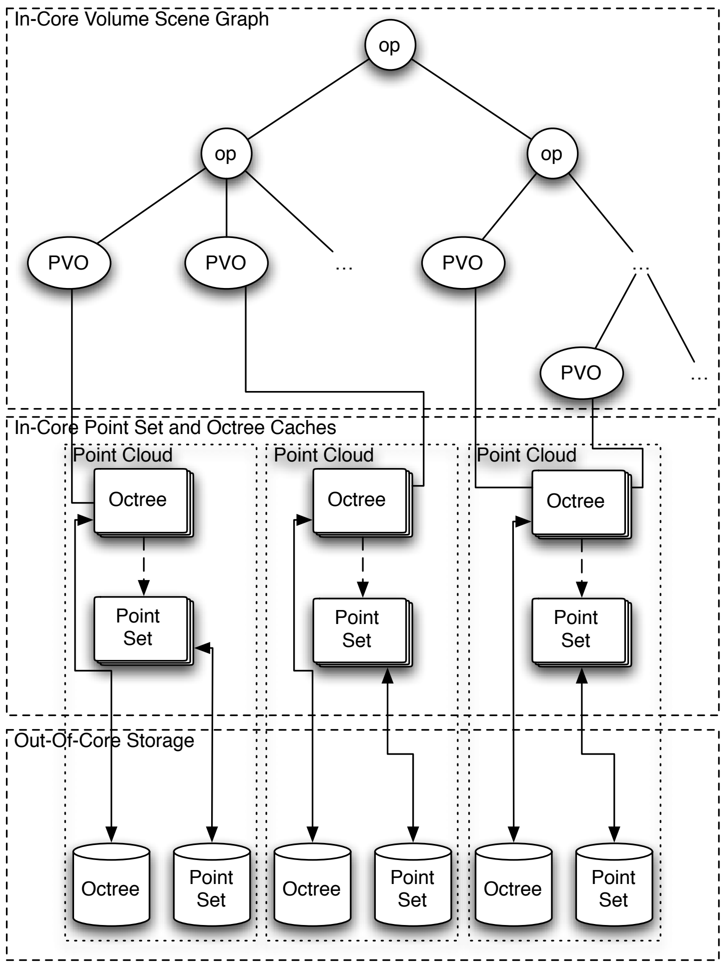


Figure 5.7: The data environment of a volume scene graph.

point is cached in-core before the ray tracer progresses to the sampling point.

5.3.2 I/O Management during Rendering

Most modern hard disks have a sufficiently high sustained transfer rate to keep the ray tracer fed with data, if it could be read in a linear fashion. Consider a single point cloud. In the described implementation, each octree node is composed of 244 bytes on disk[1], and references a set of points which are 24 bytes each. The maximum transfer rates for hard disks usually rely on reads of consecutive blocks of 512 or more bytes, making the required reads fairly inefficient.

The layout of the octree on disk is such that the child nodes of a single node are contiguous, allowing them all to be read with a single disk read. Unfortunately, this does

[1]This space is required to store the bounds and extents of the node, as well as the addresses of the parent, children and/or the points in each node, as well as some pre-computed values to speed up rendering calculations. It also includes the indexes of the points referenced by the octree node.

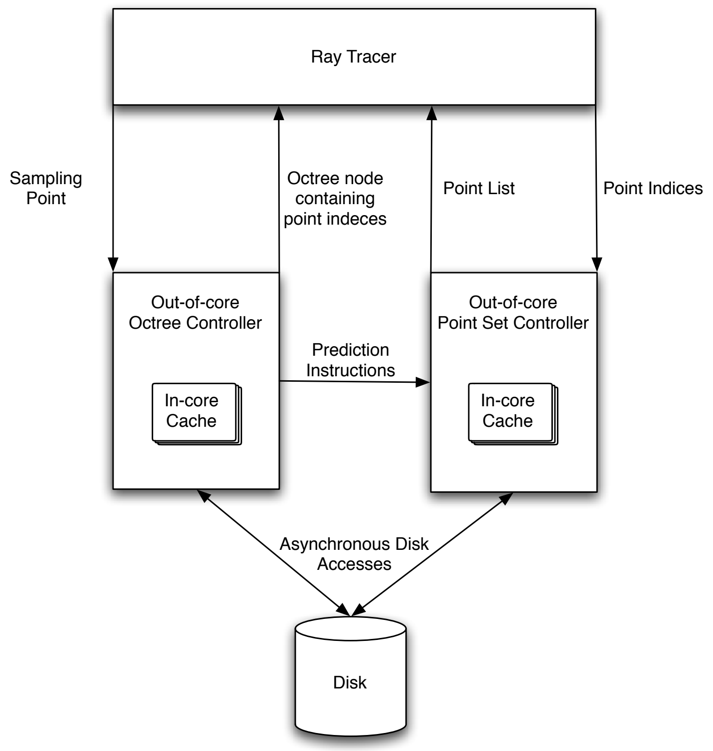

Ray Tracer
Sampling Point
Octree node containing point indeces
Point List
Point Indices
Out-of-core Octree Controller
In-core Cache
Prediction Instructions
Out-of-core Point Set Controller
In-core Cache
Asynchronous Disk Accesses
Disk

Figure 5.8: The main components of the out-of-core renderer.

not give a significant benefit since navigation of an octree is performed by ascending and descending the tree, rather than large lateral movements. Ordering the nodes on the disk to allow linear reads to load paths to the leaf nodes is not feasible, since there are eight possible paths down from each node.

In order to permit the fast traversal of the octree, the access of any non-leaf node causes all of that node's children to be pre-emptively swapped in. Additionally, the path between a currently accessed node and the root node is locked in-core, allowing movement up the tree to occur without requiring any disk accesses. The disadvantage of this approach is that it locks more nodes into in-core memory than are strictly required, however for discrete ray tracing applications, where adjacent octree nodes are frequently required, the benefit is worthwhile.

Individual nodes in the octree are accessed using a reference counted retain-release mechanism. Retained nodes are locked in-core. Once a node is released, and its reference count becomes zero, it is not immediately swapped out. Instead, its priority is decayed. No node is evicted until the allocated in-core space is exhausted. Since the point set and octree data are not modified during rendering, and hence does not need to be written back to disk, it costs little to free in-core nodes individually.

Unfortunately, it is not possible to group the points together on disk for linear reads without incurring a significant space penalty. The reason for this is that each individual point can be referenced by several octree nodes, the number varying with the radius of the radial basis function associated with each point.

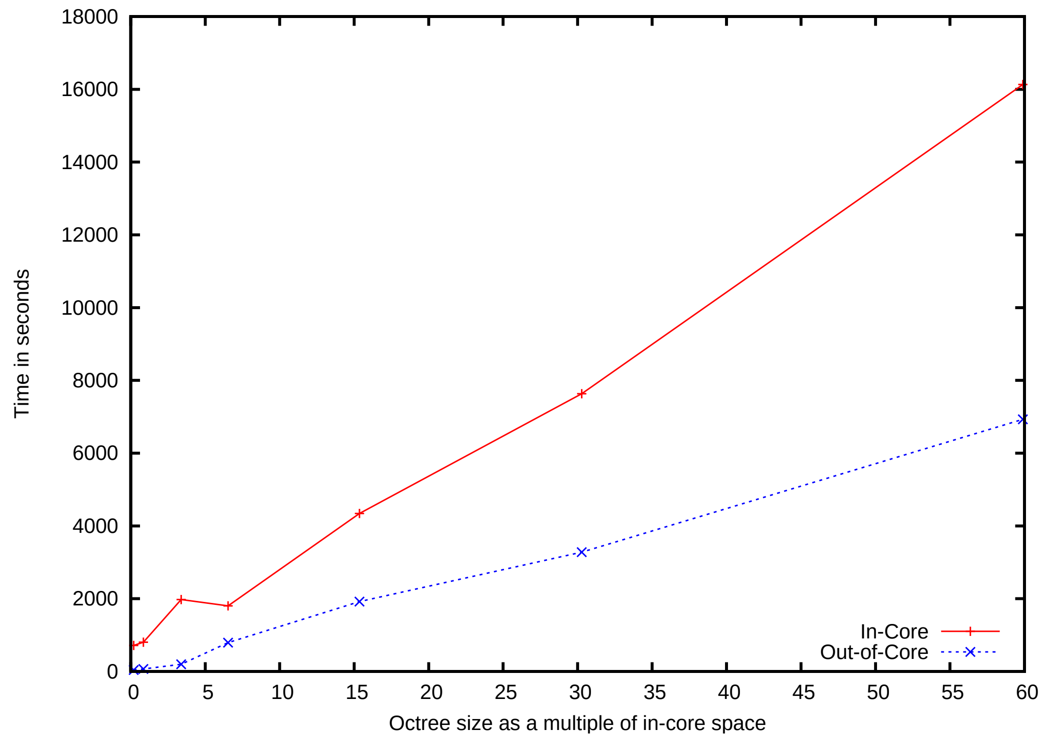


Figure 5.9: A performance comparison between the in-core only rendering and the out-of-core approach with prefeching.

Various parts of the discrete ray tracer, such as opacity sampling and shadow sampling, trigger the pre-caching of out-of-core data. Each of these assigns a priority value between 0 and 255 to the record, representing the confidence of its prediction algorithm. When the allocated in-core space has been exhausted, records are swapped out based on priority, with the least recently used record of the lowest priority being accessed first.

The method provides two major benefits over the built-in paging strategy in a conventional operating system:

- *Fine grained access* — The operating system relies on the granularity of the paged memory system, and will only swap entire pages in and out of main memory. This means that, in order to load a single byte, an entire page (typically around 4KB) must be swapped out to make room for it. We are able to swap out only the relevant working sets actually needed.
- *Lower latency* — The operating system will typically wait until a page is accessed which is currently out-of-core before acting. It will then issue a page fault, evict the least recently used page, and then load the required page. The problems with this are that the evicted page may be the one needed next, and that the process accessing the page is stalled while it waits for the page to be loaded.

Figure 5.9 shows a comparison of the time taken to render a randomly generated point set using both in-core and out-of-core approaches. The octree size is expressed as a multiple of the amount of RAM space available to the rendering process. For example, from Table

5.2.4, we can observe that with 100,000 points, a 7-level octree tree would require 60 times more space than typical RAM space available on a PC.

In the pure in-core case, once this value of multiples exceeded one, the demand-paging subsytem in the operating system is automatically invoked to handle swapping. In fact, most everyday systems are configured with the size of the swapping space set to between 50% and 100% of the RAM space. This means, when the value of multiples reaches about 1.5 $\sim$ 2, the allocatable physical address space of the operating system will be exceeded and rendering will be aborted. Thus relying on the demand-paging subsytem is not scalable.

In order to observe the effectiveness of the prefetching strategy described in this paper in comparison with the demand-paging subsytem, we reconfigured an operating system by allowing significantly more swapping space than a typical configuration. It can be clearly seen that the out-of-core approach performs better in terms of raw speed. The in-core approach, using the operating system for paging takes more than twice as long to complete in all cases.

5.3.3 Prediction Scheme

The *prediction algorithm* developed for the ray tracer requires two out-of-core data structures, one representing the octree and the other representing an array of points. The algorithm detects octree access patterns based solely on its captured knowledge of previous accesses. The array of points has an internal predictor which predicts regular accesses, and additionally accepts hints from an external source. The first predictor is used when performing pre-processing — the point set is streamed into in-core memory and each point is processed in order. The second predictor is used during rendering, with the hints being generated from the octree.

Algorithm 1 Predict Octree Access

```
**Require:** $s$: the new sampling point
**Require:** $s_1$ and $s_2$: the last two sampling points accessed
**Require:** $c$: a Boolean value indicating if the next cell for $s\prime$ is cached
  $N \leftarrow$ the leaf node containing $s$
  **if** $s_1$ and $s_2$ are set **then**
    **if** $s$, $s_1$ and $s_2$ are not on the same line **then**
      $s\prime \leftarrow$ the next sampling point on the ray ($s$, $s_2$) but not in $N$
      $M \leftarrow$ the node containing $s\prime$ {navigating from N to M}
      $c \leftarrow$ conditional test if $M$ is now in-core?
    **end if**
  **else**
    $s_1 \leftarrow s_2$
    $s_2 \leftarrow s$
    $c \leftarrow$ NO
  **end if**
```

Algorithm 1 shows the basic outline of the prediction algorithm. Accesses to nodes in the octree from a discrete ray tracing algorithm are along the paths of rays. The algorithm makes use of this fact, along with the fact that only two points are required to uniquely identify a line.

Every time a point is accessed, the algorithm first determines whether it is on the same line as a previous access. If this is the case, then it checks whether the last attempt to pre-cache the node was successful. If not, then it tries again. The attempted pre-caching is performed by Algorithm 2.

If the new point is not on the same line as previous ones, then it attempts to predict a new line based on the last accesses. This works for a secondary ray from a point, but fails for a new initial ray after the previous one terminates.

It is not difficult to extend this algorithm to record the start positions of rays and generate new predictions based on rays originating at these points. However doing so with current hardware results in performance degradation. The computation is relatively expensive, and the special case covered is infrequent. If processing speeds continue to increase at a rate faster than disk access times, then this may become a more attractive proposition within a few years.

Algorithm 2 handles the pre-caching. It attempts to navigate to the required node, without accessing any nodes stored out-of-core. First, it navigates up the tree until it finds a node which contains the point. This is guaranteed not to require accessing any out-of-core nodes, since the path between the root node and a currently retained leaf node is always locked in-core.

Algorithm 2 Pre-cache an Octree Node

```
Require: $s$: the new sample point
Require: $N$: the starting node
  while $s$ is not in $N$ do
    $N \leftarrow$ the parent of $N$
  end while
  while $N$ is not a leaf node do
    $M \leftarrow$ the child of $N$ containing $s$
    if $M$ is in-core then
      $N \leftarrow M$
    else
      pre-cache $M$
      return NO
    end if
  end while
  return YES
```

The second phase navigates down the tree towards the node as far as it can without accessing a node that is not stored in-core. If it reaches an out-of-core node, then it stops, and pre-caches the node.

Figure 5.10 illustrates this pre-caching mechanism. The node marked as A is the original node, containing s, and the node marked as B is the one containing the point $s\prime$. In other words, the last search within the octree returned A, and the next search along the same line which does not return A, will return B. This diagram indicates how the prefetching algorithm will attempt to navigate from node A to node B. Nodes which are not accessed in this navigation are omitted, or summarised with an ellipsis.

If no pre-caching were accomplished, this access would require three disk reads. To make matters worse, there is no way of determining where each node is on disk until its parent node has been read, and so none of these reads could be initiated until the previous one has been completed. Given common hard drive seek times of 8ms, this would stall the algorithm for 24ms — 24,000,000 clock cycles[2] on a 1GHz CPU — when leaving an octree node in this way. Even adjacent octree nodes would incur an 8,000,000 cycle penalty.

The first time a point inside A is accessed, the prefetching algorithm will navigate along the branches indicated by the number ‘1’. Once it reaches the first node that is not cached in-core, it will issue a request that this node be pre-cached, and then return. The rendering algorithm can then perform processing on the points in A while the node is loaded. The second time a point in A is accessed, this node should be in-core, and so it will get two levels down the tree, to the end of the arrow indicated by a ‘2’, before encountering an out-of-core node. Again, this node will be marked for fetching and the predictor will return. The third time, indicated by a ‘3’, it will get to node B and cache this.

5.3.4 Prefetching Scheme

Prefetching data efficiently is very important to the overall performance of the rendering. Moving from a pure in-core implementation to an out-of-core approach caused a significant, yet unavoidable, performance hit. Previously, moving to the next node in an octree was as simple as de-referencing a pointer, something which can be accomplished in a very small amount of time. In order to perform the same operation in an out-of-core setting, the following steps are required:

1. Translate the record index to a disk location.
2. Calculate the hash of the location.
3. Determine whether the node is in-core.
4. Fetch it, if not.

The hashing technique is adapted from hardware cache implementations — the lowest n bits of the record index (not the disk address) are taken and used as the hash. This has the advantage that it is very computationally cheap to perform, and allows sequential accesses to fill the in-core hash table without collisions.

[2]Note that modern CPUs complete more than one instruction per clock cycle

With a fast and efficient hashing algorithm, the time taken to determine whether a record is in-core is not high, however the access process still potentially takes several function calls — each of which has a cost associated with it — making it far slower than a simple pointer access.

If the prediction algorithm fails, then the cost is very high, since the data must be synchronously accessed from disk. This may stall the process for several hundred clock cycles while it waits for the disk access to return. In the case of the pure in-core implementation, this is the situation encountered whenever the required memory exceeds the available physical memory — each access to out-of-core virtual memory creates a page fault which stalls the process until a page is read from disk.

The prefetching process makes use of the operating system's asynchronous I/O capabilities. When a record is identified as requiring loading in-core, an asynchronous read request is dispatched, and the rendering process continues. When the record is accessed, or when the asynchronous I/O buffers have all been used, the request is completed. In most cases, the prefetch requests are dispatched sufficiently far in advance that the asynchronous read has completed by the time it is required.

In order to alleviate some of the system call overhead incurred from large number of small reads, the POSIX `lio_list` system call is used, which allows up to 16[3] asynchronous reads to be initiated with a single system call. In many cases, this allows all of the points contained within a single octree leaf node to be loaded with a single system call. An additional benefit of this is that it allows the kernel to re-order the disk reads to reduce seek time on the disk.

Both the point set and octree make use of the same underlying code for shifting data in and out of core. This code receives the pre-caching requests and priority information from the high level code, and evicts low-priority records when their space is needed.

5.4 Experimental Results

5.4.1 Scene Graph Complexity

We have run a number of tests on different desktop computers, including a legacy Alhlon 1.4 PC (1.4GHz, 70MB), a Pentium 4 PC (3GHz, 1GB), a Pentium M 770 laptop (2.13GHz, 0.5GB), an PowerMac G5 (2×2.5GHz, 2GB) and an Apple G4 laptop (1.5GHz, 0.5GB). The render used was a that presented in [59], modified to support out-of-core storage of data.

Figure 5.11 demonstrates that this technique can be used to synthesize images from complex volume scene graphs on a desktop computer. The volume scene graph is composed of six point-based volume objects, built from two point sets (Stanford Lucy of 14,027,872 points, and Stanford Bunny of 35,947 points). They are partially immersed in artificial

[3]This number is implementation dependent, however 16 is common

clouds represented by a volume dataset (Erlangen clouds of $512 \times 512 \times 32$ voxels). The scene is lit by three point light sources, casting shadows in different directions. From Figure 5.11, we can observe hard shadows cast by the bunnies and the Lucy statue, soft shadows by the clouds, and self-shadows by Lucy's arm. As the radial basis function used for the Lucy point set has a much smaller radius, it requires much finer sampling intervals.

5.4.2 Memory Usage

The memory usage of the algorithm is entirely configurable, with the only constraint being that there must be enough in-core space allocated to store the maximum number of nodes which need to be processed at once. As discussed in 5.3.2, the path between a current access node and the root node of each octree is locked, this maximum number is thus related to height of the octree h and the number of individual point sets n. In terms of space complexity, this is of $O_{space}(n \cdot h) = O_{space}(n \cdot \log m)$, where m is the average number of leaf nodes in each octree, which is related the average size of each point set.

For example, in a single PBVO test with the Bunny point set, the total memory used by the out-of-core ray tracer — including the in-core data cache and all other memory allocated by the process — was under 20MB. In contrast, the in-core implementation used around 300MB. For larger datasets, it is possible to keep the memory usage at a similar level, however this comes at the expense of speed. In the following section, the trade off between memory usage and performance is examined in more detail.

5.4.3 Performance

The performance of the algorithm is dependent on the amount of memory allocated to it. This can be controlled in two ways — the amount allocated to each point set and the amount allocated to each octree can be varied independently. Table 5.4.3 shows how the performance varies as each of these is changed.

number of octree	number of points in core			
nodes in core	32768	16384	8192	4096
131072	94.59	98.67	100.31	107.15
65536	90.67	96.25	96.62	101.63
32768	88.28	95.20	94.89	100.91
16384	88.78	96.58	95.84	103.38
8192	92.39	98.56	100.31	108.25

Table 5.2: Performance (in seconds) as memory is constrained while rendering the Stanford Bunny.

The timing data in Table 5.4.3 gives the time, in seconds, taken to render a scene containing a single PBVO (Stanford Bunny of 35,947 points). It includes both preprocessing

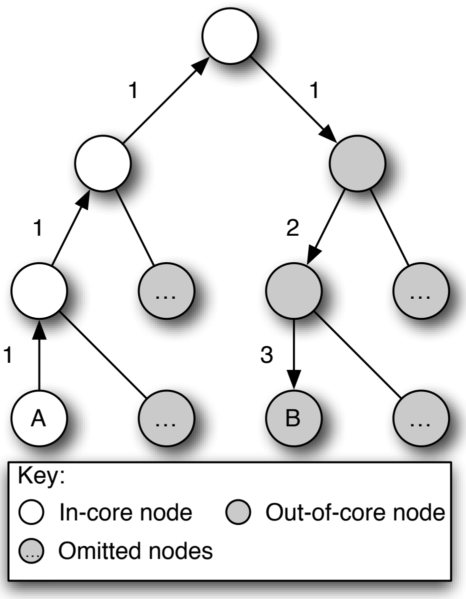


Figure 5.10: Pre-caching nodes in an octree . Unused nodes omitted for brevity. Each step in the algorithm is shown numerically.

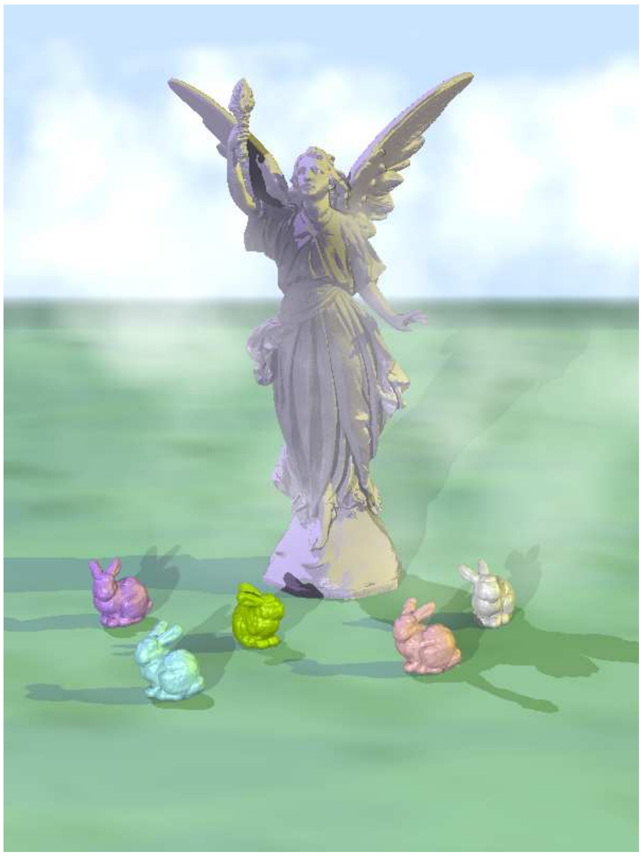


Figure 5.11: A volume scene graph composed of six PBVOs (built from two point sets, Stanford Bunny and Lucy), a volume dataset (Artificial Clouds from Erlangen) and a procedurally defined floor.

Table 5.3: Performance for multiple point sets.

number of point sets	1	2	4	8	10
Preprocessing	57.0	114.8	230.9	470.7	589.0
Rendering	13.8	13.9	14.5	16.8	16.8

time and rendering. The preprocessing times vary between 55-65 seconds. The results were taken from a PowerMac G5 with 2×2.5GHz CPUs. The current implementation currently only makes use of a single CPU directly, however the second CPU is used to process asynchronous I/O requests.

The results in Table 5.4.3 indicate that increasing the size of the point cache has a more noticeable impact on performance than increasing the size of the octree cache. This is due to the fact that the points required change dramatically over short octree traversals, meaning that a small increase in the point cache size can dramatically cut down the total amount of disk I/O required.

Increasing the amount of memory allocated to the octree cache does not always give a performance benefit. The pre-fetching algorithm requires very little space to ensure that all of the required nodes are in-core before they are accessed. Once the cache reaches this size, increasing it delivers no benefit — the extra space is not required. Increasing this amount further increases the length of time required to find a single cache record, incurring a performance penalty. For larger cache sizes, this penalty is sufficient that increasing the allocated memory results in a performance penalty.

An in-core ray tracer would have a great difficulty in dealing with multiple point sets, such as the volume scene graph in Figure 5.11. The out-of-core ray tracer can easily handle such a volume scene graph in terms of memory allocation, without experiencing inefficient disk I/O managed by the operating system. The rendering process in this case involves the full data environment as shown in Figure 5.7.

To evaluate the performance under the condition of multiple point sets, the volume scene graph in Figure 5.12 was rendered by associating the twenty bunnies to different numbers of point sets. Though the same point set is being used, for the purpose of scalability test, repeated uses are treated as independent point sets. For example, in the first test, one point set is used for all bunnies. In the second, half use one point set, and half use another.

The evaluation was performed in this manner to reduce the number of variables affecting the results. For each test, the final scene was the same. The differences arise solely from the way in which the scene graph was created. The only difference between each rendering of the scene is how the points are accessed. In the simplest case, the scene graph contained twenty references to the same point set (and octree). Each subsequent test modified the scene graph to use new instances of the data set, increasing the memory usage without modifying the rendered scene.

Table 5.3 gives both preprocessing (i.e., octree building) and rendering times in seconds. The point set used to generate these results contains 35,947 points. The scene with 10 instances of this dataset thus contains 359,470 points. The preprocessing time scales lin-

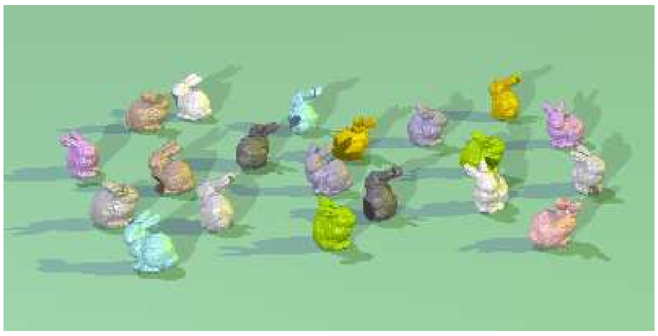


Figure 5.12: A volume scene graph composed twenty point-based volume objects (Stanford bunny), ray traced with three light sources.

early as new point sets are added, but the actual rendering time changes little and remains almost constant in relation to the increasing number of point sets from 1 to 10. Note that the point sets are independent, and have separate octrees, and thus the preprocessing time is lower than it would be with a single, larger point set.

The results are also shown graphically in Figure 5.13. It can be seen here that the preprocessing time scales linearly as more points are added. The rendering time, however, remains almost constant. Recall that adding point sets to the scene increases the memory requirements without modifying anything else in the rendering path. Increasing the number of points by a factor of ten increased the rendering time by approximately 20% without requiring an increase in in-core memory usage.

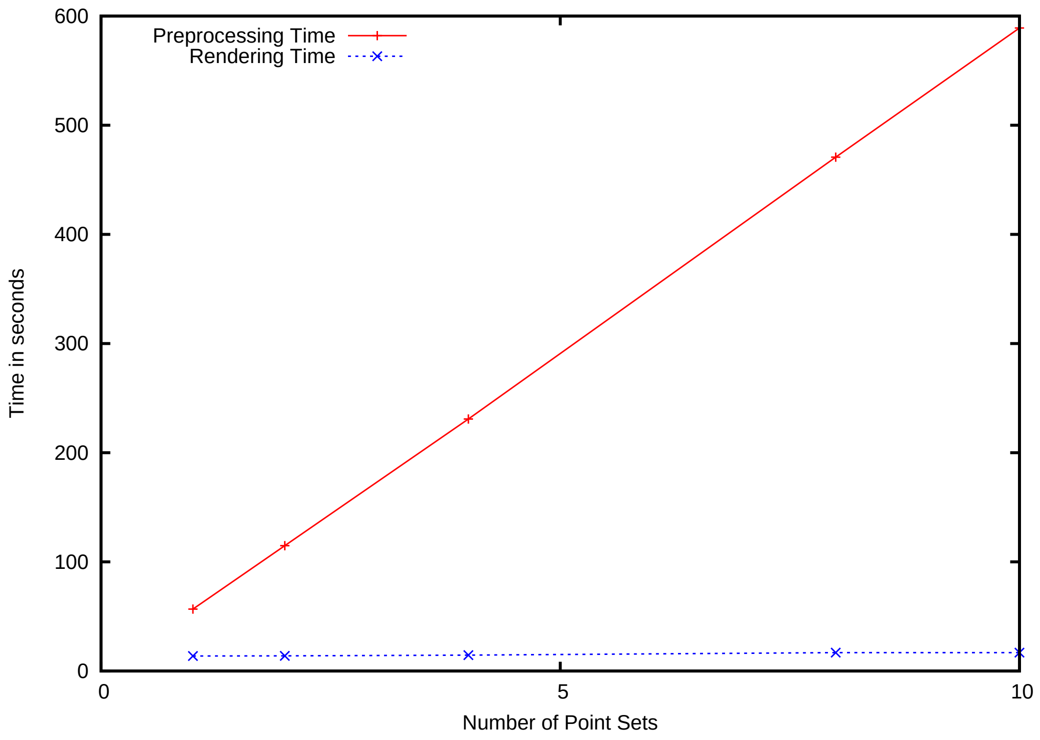


Figure 5.13: Graph showing performance for multiple point sets.

5.5 Conclusions

This chapter has presented an out-of-core solution to a difficult problem for rendering multiple point-based volume objects using discrete ray tracing. The I/O management involves a dynamic, in-core working set, and uses a ray-driven algorithm for predicting the working set automatically. The results have shown that the algorithm scales well to very low memory conditions. Performance increases can be gained by increasing the size of the point cache up to the size of the point dataset, and by increasing the size of the octree cache up to a limit dependant on the structure of the data and the number of in-core nodes required at any given time. We have demonstrated that this approach allows the rendering of multiple large PBVOs in a volume scene graph on common commodity desktop computers.

The following chapter will use the algorithm described here as a bench-mark to determine whether it is possible to produce comparable results from a prefetching strategy that doesn't incorporate knowledge of the rendering algorithm. The algorithm described in this chapter is evaluated in more detail and compared to other approaches.

Chapter 6

A Comparison of Algorithm and Knowledge Based Approaches

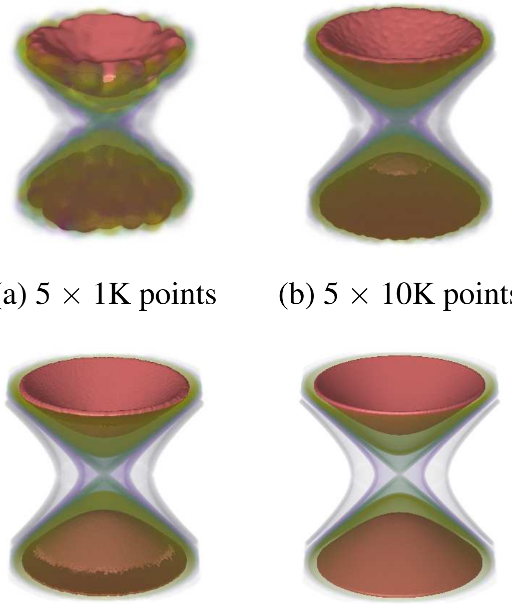
Contents

It is believed that it is possible (and practical) to design a generic knowledge-based prefetching algorithm that achieves performance commensurate with algorithms specific to the rendering method. This chapter attempts to demonstrate that a knowledge-based algorithm can perform as well as one which uses problem-specific information. Work described in this chapter was originally presented at EuroVis 2006 [73].

6.1 Introduction

In this chapter, four caching algorithms are compared. The first is a naive approach employing no prefetching, the second uses domain-specific intelligence as described in the previous chapter, and the remaining two are knowledge based.

The first algorithm uses a simple demand paging strategy coupled with least recently used (LRU) eviction, and is used as a base-line for evaluation of all algorithms. The second, the ray driven predictor (RDP) was described in more detail in the previous chapter. It makes use of domain-specific intelligence to make predictions. The remaining two algorithms use a knowledge-based approach - both make predictions based on previous access patterns, and are described in the next section.



(a) 5 × 1K points (b) 5 × 10K points

(c) 5 × 100K points (d) 5 × 1M points

Figure 6.1: Five iso-surfaces of a hyperbolic field are visualized using point sets that are randomly placed on the surfaces with different resolutions. Each iso-surface is approximated by (a) 1,000, (b) 10,000, (c) 10,0000 and (d) 1,000,000 points respectively.

This chapter aims to demonstrate that it is possible to achieve good performance in a specific problem domain using an algorithm which does not exploit problem-domain specific logic.

6.2 A Knowledge-based Approach

A knowledge-based system is one which functions using inferred knowledge. As the system runs, it learns how to function efficiently, fulfilling the *self optimising* component of the definition of an autonomic system. In this case, the inferred knowledge relates to access patterns within the data. A good knowledge-based prefetching algorithm will learn the order in which data is accessed and automatically fetch data accessed next.

The two knowledge-based algorithms in this chapter are aware of the structure of the data to a limited degree. They (along with the other two predictors) make use of the fact that the octree can be decomposed into individual nodes. The algorithms are not of the process being used to render the data. They must infer knowledge of the access patterns within the data based on previous accesses.

Two knowledge-based approaches are evaluated, and their performance compared with two other algorithms. In this implementation, both knowledge-based algorithms work by storing some extra data in the unused child node pointers in leaf nodes in the tree. The advantage of this approach is that it does not change the data layout in RAM of the dataset

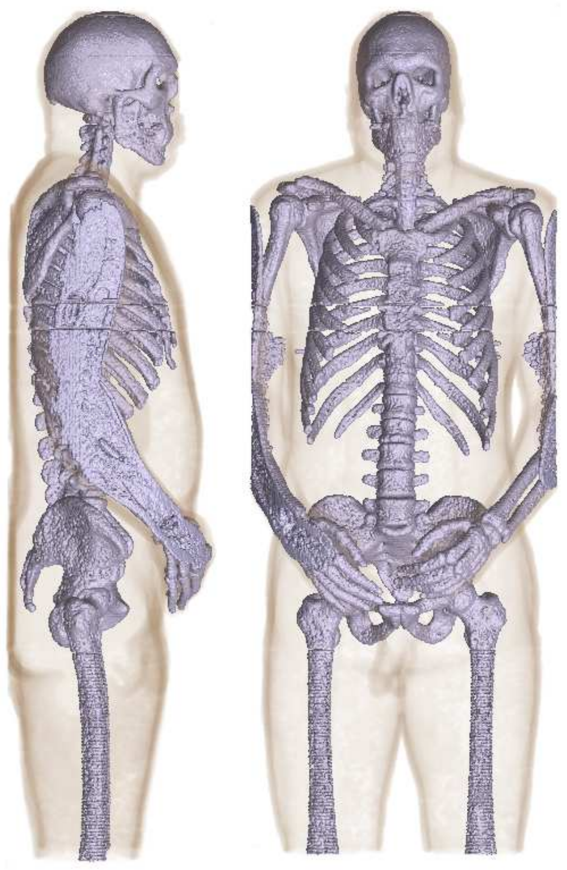


Figure 6.2: Two Visible Human point sets, representing bones (1,218,973 points) and skin (267,303 points) respectively, are combined together using a volume scene graph. The point sets were part of the polygonal model provided by William E. Lorensen [200] and made available by Georgia Institute of Technology.

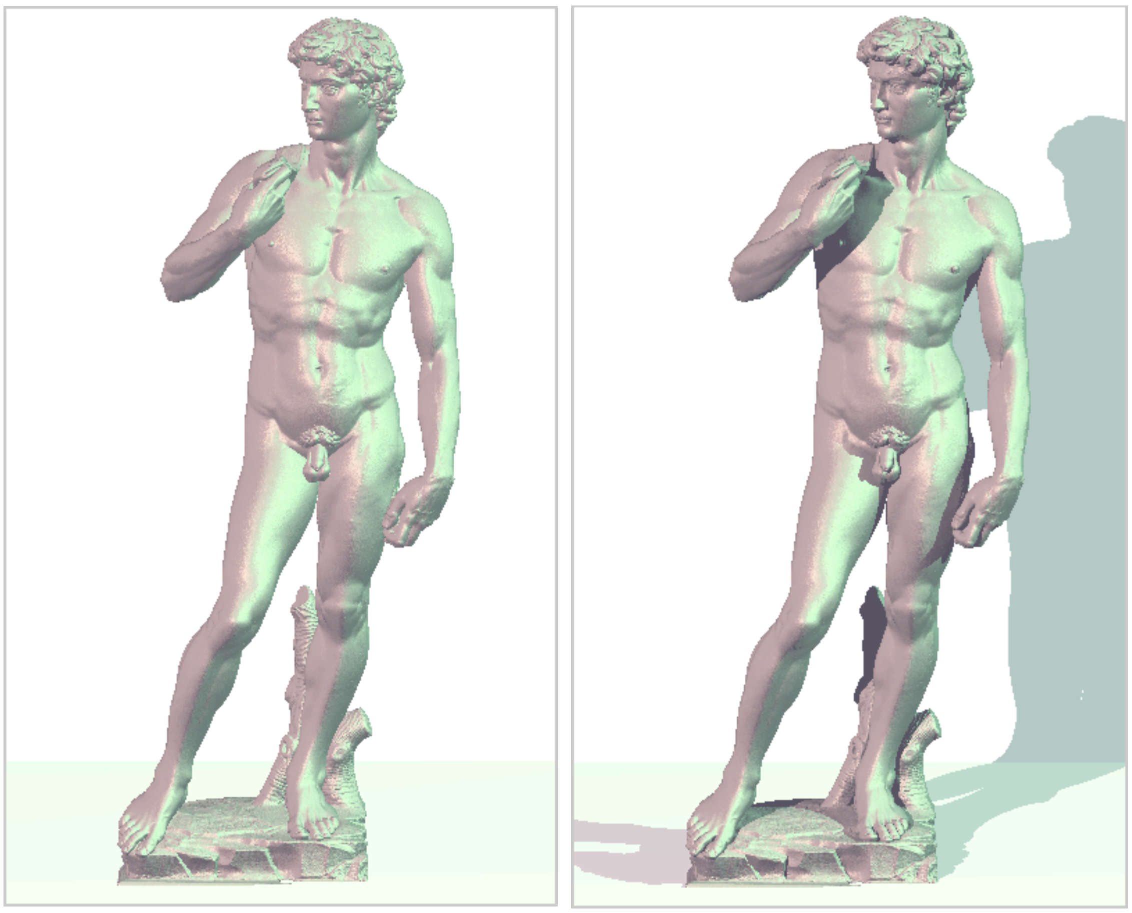

(a) without shadows (b) with shadows

Figure 6.3: The David dataset (Stanford) contains 28,184,526 points, for which an octree with 10 levels takes about 64 GB space. The visualization with shadows gives extra visual cues about the spatial relationship between the object and its surrounding, and between different parts of the object.

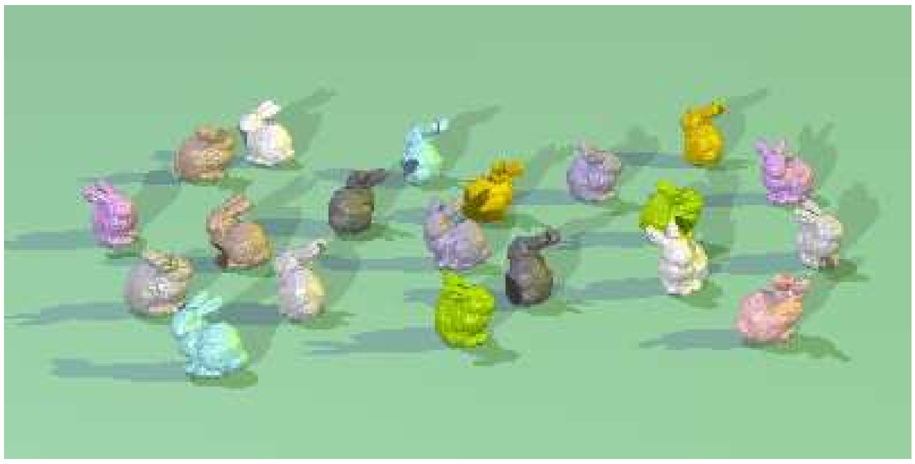


Figure 6.4: A volume scene graph composed of twenty bunny objects, which is rendered with shadow effects. The bunny dataset (Stanford) contains 35,947 points.

between the different algorithms any more than is absolutely necessary, which helps to prevent differences in CPU cache accesses skewing the evaluation.

This design decision limits the available space to 8 addresses. If the algorithms were to be implemented in a purely general case, the extra data would be stored in a separate data structure associated with the page table. It is believed that eight addresses per node is an acceptable number for this situation, since the algorithm is required to search through them every access to update the prediction, and is not allowed to take more time to run than it saves by predicting accesses.

Algorithm 3 Knowledge-based access path predictor

```
Require: New sample point $p$, last two octree nodes $Node$, $LastNode$
  if $p$ is in $Node$ then {If the new point is in the old node, we don't need to do anything.}
    $NewNode \leftarrow Node$
  else if $LastNode$ previously visited before $NewNode$ then {Try to predict along an existing vector}
    $NextNode \leftarrow$ node visited after $Node$ previously.
    if $p$ in $NextNode$ then
      $NewNode \leftarrow NextNode$
    end if
  end if
  if $NewNode$ unset then
    if $p$ in any of the four nodes visited after $Node$ then {Use an historical next nodes as a next guess}
      $NewNode \leftarrow$ next node from $Node$ containing $p$
    else {Fall back to octree navigation}
      $NewNode \leftarrow$ node reached navigating the octree from $Node$ to $p$.
    end if
  end if
  Replace oldest vector in $Node$ with $LastNode$ and $NextNode$ {Update knowledge}
  if $Node$ previously visited before $NewNode$ then
    $NextNode \leftarrow$ node visited after $NewNode$ previously.
    Cache $NextNode$ with a high priority
  else
    Cache all four nodes visited after $NewNode$ with a low priority.
  end if
  return $NewNode$
```

When a node is requested, its associated knowledge is updated. In the case of the access path predictor , described in Algorithm 3, the knowledge stored is comprised of two parts describing a vector through the dataset. Note that this is a vector through the dataset itself, not a vector through the three dimensional space represented by the dataset.

The access path predictor divides the eight available child addresses into four pairs. When three nodes, a, b, and c, are accessed in sequence, node b is updated to contain the addresses of a and c. The next time b is accessed, if a were the last node then it pre-caches c. If there is space in the cache, then the other nodes accessed afterward are also cached.

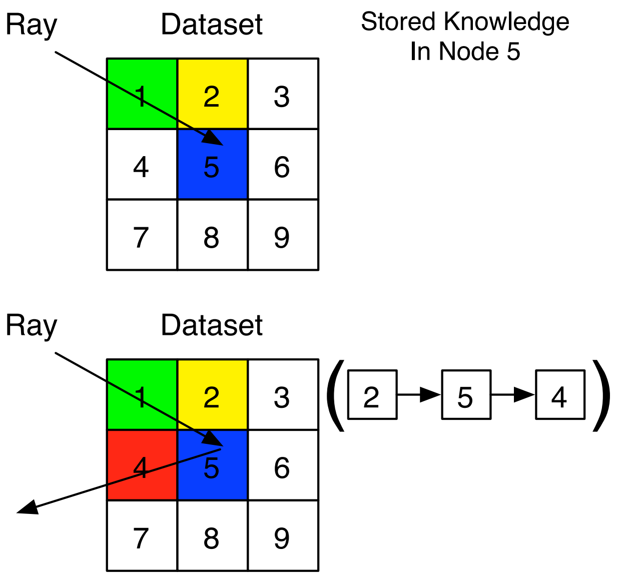


Figure 6.5: First ray entering a node with access path predictor .

The eight addresses allow for the storage of four $(predecessor, successor)$ pairs (actually, $(predecessor, current, successor)$ tuples, but the $current$ node is implicit) for each node in the octree. The access path predictor orders these according to their most recent access, and discards the oldest one whenever a new one is found. If the new pair is already present, then it is simply promoted to the front of the queue.

Figure 6.5 shows a simplified version of the predictor in operation. Here, a two dimensional grid is used to represent a segment of the dataset. Nine leaf nodes are shown, along with a ray and the collected data at each point.

The ray is shown having past through nodes one and two and into node five. It then continues into node five (presumably having hit an object and being reflected, or being a secondary ray fired at a light source). Node one does not feature in this interaction, but due to having been recently accessed it is presumed to still be in the cache. Node two is the immediate predecessor and node four the immediate successor, so the tuple $(2, 5, 4)$ is stored for this node.

Figure 6.6 shows a second ray entering the same node. This one comes from node four, and so there is no existing relation that can be used for prediction. At this point, the algorithm would typically fall back to simply pre-fetching all of the successor nodes, however the only successor node it has stored here is node four.

This ray is not deflected, and passes straight through the node. As it passes, the tuple $(4, 5, 6)$ is recorded into the node's knowledge base for later predictions.

When the next ray enters, as shown in Figure 6.7, there is an existing tuple that can be used to make a prediction. This uses the $(4, 5, 6)$ tuple to predict node six will be accessed next, and begins prefetching it.

The same mechanism would be used in every single node in the dataset, so once the ray

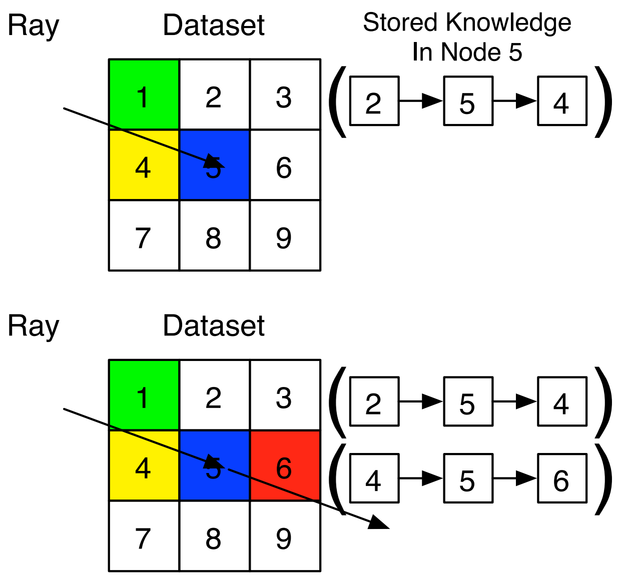


Figure 6.6: Second ray entering a node with access path predictor .

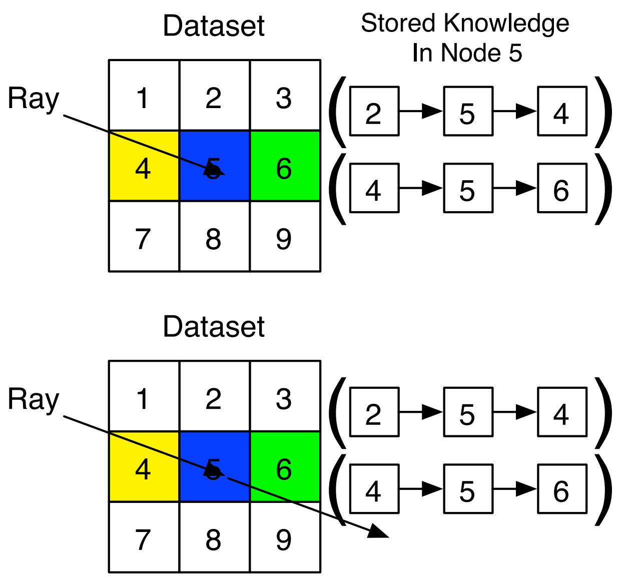


Figure 6.7: Third ray entering a node with access path predictor .

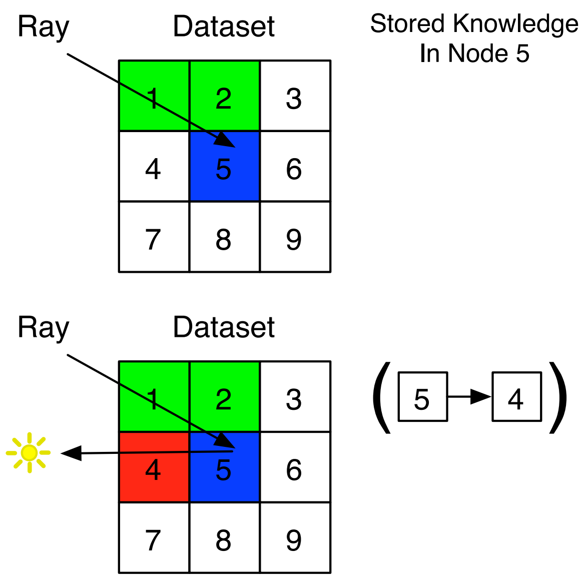


Figure 6.8: First ray entering a node with history access predictor .

has entered node six, it would again have its next node predicted based on prior accesses. Since the last node in the example exited via node nine, this would probably be pre-cached, giving an two correct predictions in a row.

One important potential benefit of the access path predictor over the ray driven predictor is that it is able to predict access patterns caused by secondary rays trivially. Predictors tied to the ray tracing algorithm have some difficulty doing this, as the existence of secondary rays is dependent on the contents of a particular node in the spacial partitioning scheme (in this case, the octree). The access path predictor will miss the first secondary ray in each direction, but then has a good chance of predicting future ones.

The history access predictor is simpler. It was designed due to concerns that the computational cost of the access path predictor might be too high. Unlike the access path predictor , the history access predictor simply stores $(current, successor)$ pairs. Again, the $curent$ node is implicit, and so only the address of the eight successors are stored. Every time a ray enters a node, all potential successors are loaded.

An example of the history access predictor in operation is shown in Figures 6.8 and 6.9.

In the first diagram, a ray enters a node in a similar two dimensional arrangement to the last example. The ray enters from node two, having passed through node one. A secondary ray is then fired towards a light source. The ray exits into node four which, as before, has not been predicted and so causes a page (or, rather, node) fault and is loaded from the disk, unless it was already in the cache for some other reason.

The second ray enters from the same direction, and has a similar experience. Again, the secondary ray exists through node four, but this time it is pre-cached. A description of the use of the history access predictor algorithm in this context is given in Algorithm 4. Note that the mechanism used in history access predictor is also used as a fall-back in access path predictor for cases where no the predecessor does not match any in the node's knowledge base.

Algorithm 4 Knowledge-based history access predictor

```
Require: New sample point $p$, last octree node $Node$,
  if $p$ is in $Node$ then {If the new point is in the old node, we don't need to do anything.}
    $NewNode \leftarrow Node$
  end if
  if $p$ in any of the eight nodes visited after $Node$ then
    $NewNode \leftarrow$ next node from $Node$ containing $p$
  else {Fall back to octree navigation}
    $NewNode \leftarrow$ node reached navigating the octree from $Node$ to $p$.
  end if
  Replace oldest successor in $Node$ with $NextNode$ {Update knowledge}
  Cache all eight nodes visited after $NewNode$ with a medium priority.
  return $NewNode$
```

When looking at these examples, it is important to remember that they both represent small segments of a much larger data set. In between two rays entering node five, a large number of other nodes might be accessed. In the last example, two consecutive rays to enter node five exited through node four, so it might be assumed that node four would already be in the cache, and thus no prefetching would be needed. Unfortunately, these two rays are likely to have traversed other parts of the data set before reaching node five, and even more after leaving it. The two accesses to node five might be a long way apart, and thus the second may not benefit from caching at all.

It is also important to remember that, while these examples were described in terms of rays, this was to illustrate how the algorithms are used, not how they are designed. Neither algorithm has any awareness of the rays, just the accesses. From the perspective of the predictors, the rays are simply a sequence of node requests.

Both of these algorithms work solely by storing knowledge about leaf nodes. This means that in both cases it is frequently possible to hop from one leaf node to another. In the absence of this knowledge, it would frequently be necessary to navigate several nodes up and then back down the tree, and each of these nodes could potentially require a disk access.

The secondary advantage of the ability to move directly between leaf nodes in an octree is that it reduces the size of the working set. There is a lower probability that non-leaf nodes will be required, and so more cache space can be used to store the leaf nodes and points.

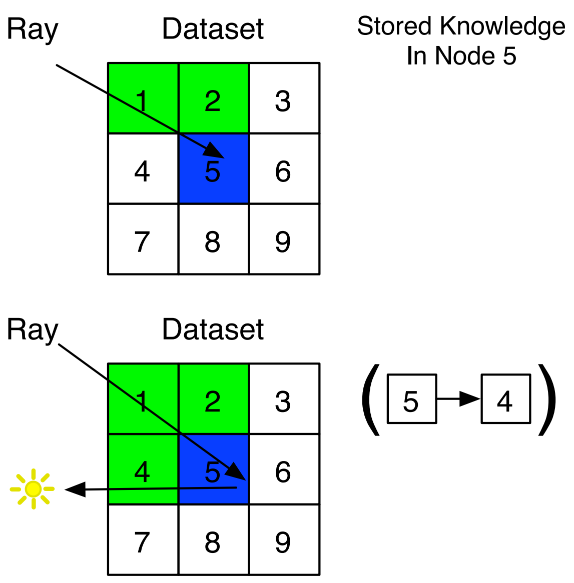


Figure 6.9: Second ray entering a node with history access predictor .

6.3 Algorithm Evaluation

The four algorithms evaluated are:

- *The least recently used (LRU) strategy* — This is a simple memory management algorithm discussed extensively in many textbooks on operating systems. This is the most generic among the four considered and contains a very limited amount of application-specific logic. It is used in this work as a base-line for evaluating all algorithms.
- *The ray driven predictor (RDP)* — This is the least generic among the four considered. It relies on a significant amount of hard-coded application-specific logic, including both the data structures and the rendering algorithm, to make predictions.
- *The access path preictor (APP)* — This algorithm assumes that the most likely access pattern is a predecessor-current-successor pattern. Unlike RDP, it does not attempt to hard-code such logic mathematically, and instead makes predictions based on previous access patterns. Hence, it is a knowledge-based algorithm.
- *The history access predictor (HAP)* — This algorithm also adopts a knowledge-based approach, but is more generic than APP without the assumption about the likelihood of any access pattern. It maintains a relatively fuller record of access history.

All algorithms were evaluated with random point sets and with a small number of real datasets. The random datasets contained 100, 1,000, 10,000, 100,000 and 1,000,000 points respectively. For each number of points, two different formations were generated; one in which all of the points were on the surface of a sphere and the other in which all points were in a volume.

	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
Best	1	0.4	0.3	0.8
Average	1	1.8	0.8	1.9
Worst	1	5.1(38)	1.4	4.9
Rank	2	3	1	4

Table 6.1: Summary of normalised disk access results over algorithm tests.

	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
Best	99.9%	100%	99.9%	99.9%
Average	98.9%	99.8%	99.8%	99.7%
Worst	91.1%	99.3%	99.2%	99.0%
Rank	4	1	2	3

Table 6.2: Summary of cache hit rates over algorithm tests.

The surface point sets results indicate the performance when rendering an object created by sampling a solid object, such as a statue. The volume point sets resemble the kind of data found in a CT scan, or similar. Between the two different types of data, they represent the majority of extant point data sets.

Tests were performed both with and without shadows for the point sets on the surface of a sphere. The shadow rendering involves firing secondary rays through the data set. This means that each octree node will have rays passing in multiple directions passing through it.

In assessing these algorithms we use two metrics; cache hit rate and total disk reads. Every time there is a cache miss, the renderer stalls while the data is fetched from disk. Thus, every cache miss slows the system down. The hit ratio gives an idea of the number of false negatives generated by each algorithm; times when the algorithm failed to accurately predict the data that was required.

Disk reads give an indication of the number of false positives of each algorithm. Without an in-core cache or prefetching, the number of disk reads will equal the number of octree node accesses. Caching will reduce the number, since some nodes will already be in-core when they are needed. Prefetching is likely to increase the number of disk reads since data will be speculatively moved from the disk to the cache. If an algorithm is speculatively caching data that is not used, it will increase the number of disk reads.

A least recently used algorithm was used as a base-line. This is close to the strategy employed by modern operating systems, but finer grained. The least recently used strategy employed here has a granularity of a single octree node, while OS-level implementations can be as course-grained as evicting entire process address spaces and then demand-paging in the part that is actually used - in the best case they will work on 4K pages, each of which can store around 16 octree nodes.

	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
Best	1	0.88	0.87	0.96
Average	1	0.91	0.90	1.01
Worst	1	1.05	1.03	1.05
Rank	3	2	1	4

Table 6.3: Summary of timing results over algorithm test.

Detailed results can be found in the final section in this chapter. Tables 6.1 and 6.2 contain a summary of the information presented here.

Table 6.2 shows the cache hit ratios of the four algorithms on the random data sets (detailed data can be found in Tables 6.4, 6.6, 6.8, and 6.10. The least recently used algorithm falls short of the other three, while the ray driven predictor has the best results. Both of the knowledge-based algorithms perform better than the naive approach, and come close to the performance of the algorithm-specific approach in terms of hit ratios. Since hit ratios indicate false negatives, this shows that all three prediction algorithms have roughly the same number of accurate predictions. The average cases for the three are very similar, however the history access predictor has the lowest worst-case performance.

Note that the values in the summary here are all over 98%. It may seem that there is little variation between the algorithms. Keep in mind that each cache miss represents an algorithm stall. In the last chapter it was shown that a modern CPU can execute around 350 million instructions in the time taken to process one in-core cache miss, and this number is climbing rapidly. Within a few years, increases in CPU speed will have a negligible impact on the performance of this kind of algorithm, since processing all in-core nodes will take less long than fetching one from out-of-core. At this point, a cache hit rate of 99% will (all other things being equal) equate to an overall performance of twice that of a hit rate of 98%.

Looking at the detailed results it can be seen that all of the algorithms often get better cache hit rates when a coarser sampling interval is used. This is quite apparent in the stanford bunny and hyperboloid datasets (Table 6.4) where the cache hit rates at the coarsest sampling interval are significantly worse than at other times. This is due to the fact that as the sampling interval tends towards zero the probability of each step in the rendering process using the same data as the previous one tends towards one. In a coarse sampling, the distance between sampling points is greater and so the probability of new data required for each sample is larger.

Table 6.3 shows some summarised timing results from these test runs. These are not considered part of the evaluation since the test system was designed for fair evaluation rather than performance. In spite of this, an average 10% performance increase can be seen with the ray driven and access path predictors. An implementation designed for higher performance would perform the prefetching asynchronously on a separate CPU (or even a dedicated I/O controller) and would avoid a number of function calls in the data loading by removing some of the abstraction between the layers.

The most impressive improvements in the knowledge-based approach are highlighted when looking at disk reads. These are summarised for the random datasets in Table 6.1 (detailed results can be found in Tables 6.5, 6.7, 6.9, and 6.11). The results are shown in a normalised form. The number of disk reads is heavily dependent on the number of points in the scene and the sampling interval. This means that the number of disk reads in a 100 point scene can not be directly compared against the number in a 1,000,000 point scene in any meaningful way. To allow comparison, the number of disk reads required when prefetching is not performed is used as a baseline. The number of disk reads in each table gives the number required by each algorithm divided by the number required without prefetching. An algorithm which predicts nodes which are not used before being evicted from the cache (false negatives) will require more disk accesses than a run without prefetching. An algorithm which predicts accesses to nodes which are still in the cache, preventing them from being evicted just before they are used, can give much better performance.

The disk access results contain one pathological result for the ray driven predictor, requiring 38 times as many accesses as the uncached version in the smallest point set with the coarsest sampling. A similar pathological case is apparent on all of the 100-point sets with this algorithm. The result of 5.1 is the worst case discounting this pathological case. When this pathological case is discounted, the ray driven predictor performs slightly better than the history access predictor in terms of disk reads. The history access predictor has the best performance by this metric.

6.4 Conclusions and Future Work

These results show that a knowledge-based algorithm can provide cache hit rates commensurate with, and often exceeding an algorithm-specific one. In many cases, this improved hit rate does not come at the expense of increased disk access. On average, the access path predictor requires 20% fewer disk accesses than the LRU strategy while the algorithm-specific predictor required between 80-90% more.

This chapter has shown that applying autonomic principles to a particular facet of visualisation can have significant benefits. The knowledge-based algorithms described in this chapter were simpler — both conceptually and in terms of implementation — than that described in the last chapter. In spite of this, they achieved similar performance and had a greater potential for re-use.

While not part of the evaluation, Figure 6.10 gives some quite interesting results. This was generated by modifying the predictor slightly to log the outcome of each request. Some were incorrectly predicted, some were predicted as the result of a last to current to next relation, and some as a result of the fall-back mechanism that simply loaded the next nodes. The graph shows the number that were predicted by either mechanism, those predicted with the full mechanism, and those not predicted. It is interesting to note that the performance of both components of the predictor quickly improves (showing that this is a self-optimising algorithm), but also rapidly reaches a plateau. The plateau gives an

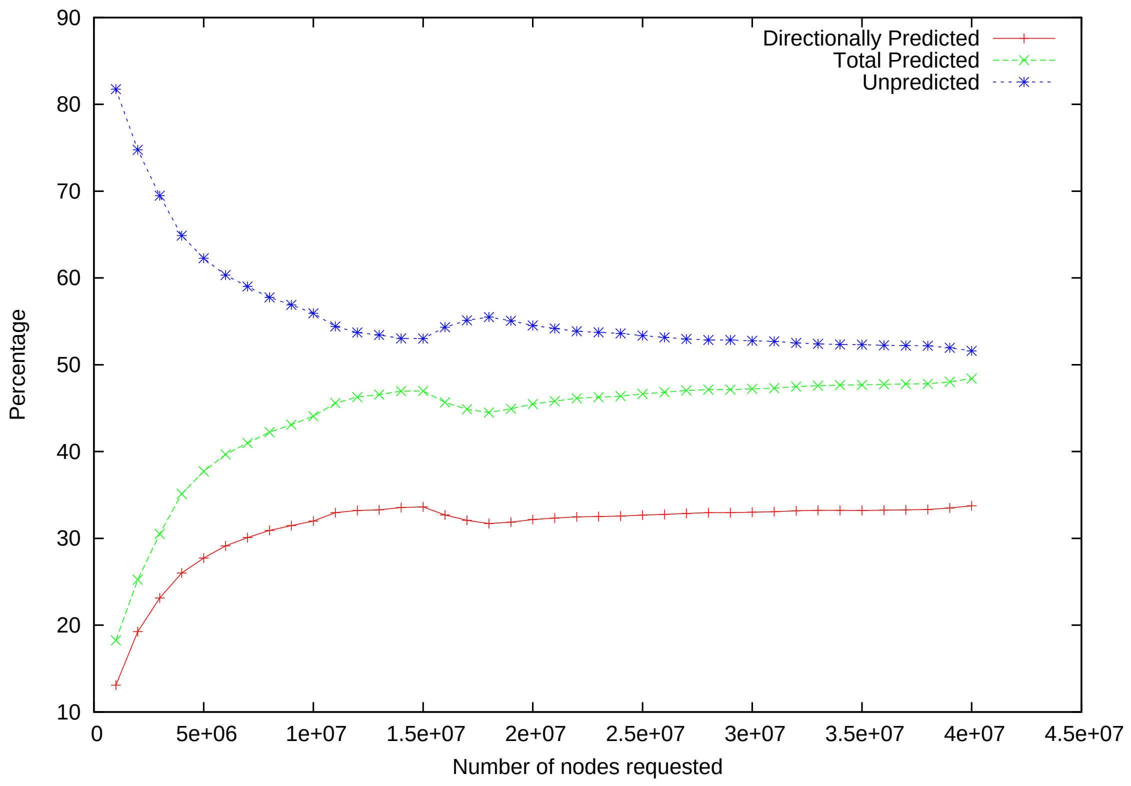


Figure 6.10: Prediction percentages over a rendering run.

indication of the directions future work should take:

1. Attempting to reach the plateau faster, and
2. Attempting to raise the height of the plateau.

The rate at which the plateau is reached is a function of the rate at which knowledge is acquired. When these algorithms begin running, they have no knowledge, and must acquire it from the renderer as they run. Pre-seeding the knowledge base could provide some better results, and this is the aim of ongoing work related to the multi-user variation of this problem, since knowledge acquired from one user can be used by another.

The height of the plateau relates to the quality of the knowledge acquired. In a typical rendering run, it is likely that more than four paths through a particular node exist. Increasing the size of the prediction cache is not (currently) feasible in the local setting, since the extra cost of using it has an adverse effect on performance. In the remote context, where latencies are typically one to two orders of magnitude larger, this is likely to give some beneficial results. Note that the hit rates achieved in testing were higher than the prediction peak on this graph. This is due to the fact that the graph only shows whether the requested octree node was predicted as the next one, not whether it was in the cache from a previous prediction.

These algorithms will also be used in the distributed system described in Chapter 4. In this application, the predictors run on a data server and speculatively transmit information to the clients. In this setting, the predictors will be able to store knowledge accumulated from multiple users. They will also be able to spend more processing time performing predictions, since the overhead of a cache miss in the distributed setting is one or two orders of magnitude more than in the local setting.

6.5 Detailed Results

This section contains the detailed results from the testing runs with the three predictors. All disk read figures are normalised against the least recently used (no prefetching) numbers indicating relative performance for the specified data set.

Dataset	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
Stanford Bunny	0.1	91.272%	98.218%	95.473%	95.901%
Stanford Bunny	0.01	97.579%	99.688%	97.795%	98.812%
Stanford Bunny	0.001	99.723%	99.968%	99.733%	99.857%
Twenty Bunnies	0.1	91.272%	98.218%	94.222%	95.896%
Twenty Bunnies	0.01	97.579%	99.688%	97.714%	98.813%
Twenty Bunnies	0.001	99.723%	99.968%	99.733%	99.857%
Hyperboloid	0.1	95.443%	98.865%	99.185%	99.302%
Hyperboloid	0.01	98.756%	99.880%	99.828%	99.855%
Hyperboloid	0.001	99.290%	99.910%	99.864%	99.890%
VH Bone	0.1	99.131%	99.967%	99.896%	99.904%
VH Bone	0.01	99.714%	99.997%	99.970%	99.974%
VH Bone	0.001	99.777%	99.995%	99.980%	99.983%
VH Skin	0.1	99.581%	99.986%	99.929%	99.941%
VH Skin	0.01	99.900%	99.999%	99.984%	99.988%
VH Skin	0.001	99.877%	99.997%	99.982%	99.987%
Mean		97.908%	99.623%	98.886%	99.197%
σ		2.857	0.618	1.755	1.350

Table 6.4: Cache hit rates from rendering real data sets

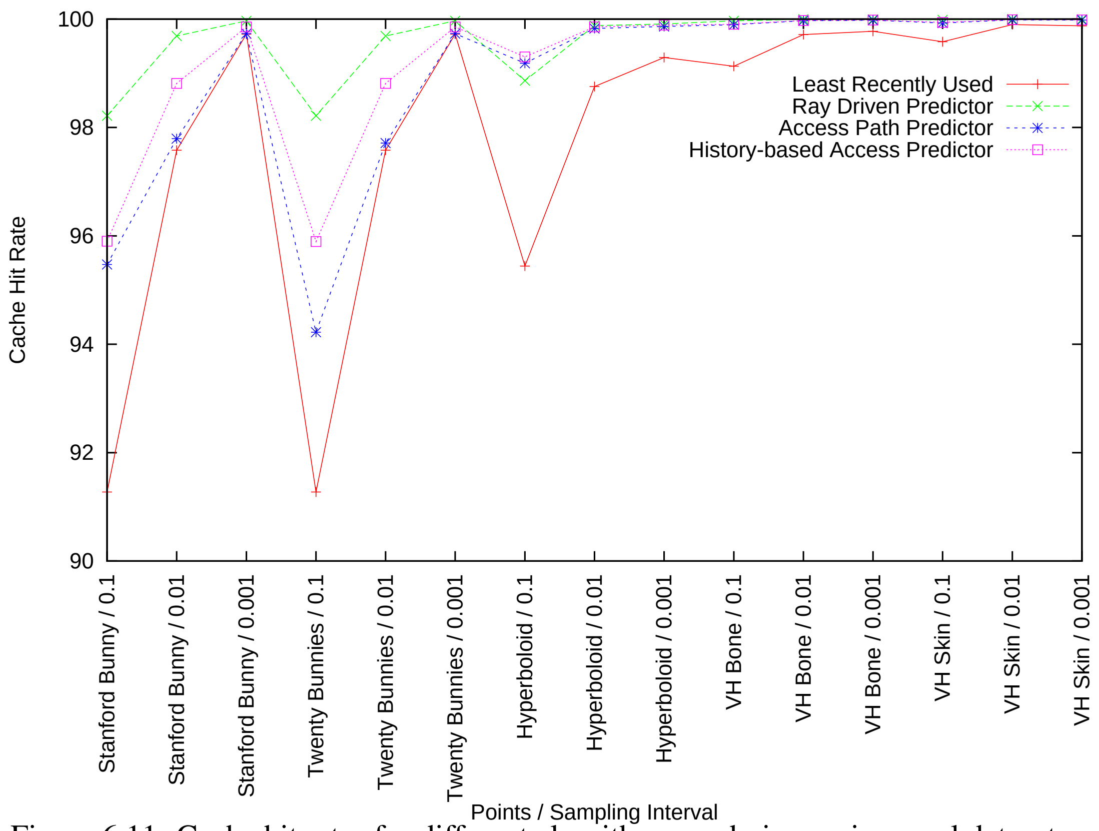


Figure 6.11: Cache hit rates for different algorithms rendering various real datasets

Dataset	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
Stanford Bunny	0.1	1	1.30	1.30	1.24
Stanford Bunny	0.01	1	1.46	1.20	1.64
Stanford Bunny	0.001	1	1.86	1.14	1.84
Twenty Bunnies	0.1	1	1.30	1.15	1.41
Twenty Bunnies	0.01	1	1.46	1.23	1.64
Twenty Bunnies	0.001	1	1.86	1.14	1.84
Hyperboloid	0.1	1	1.25	0.95	1.11
Hyperboloid	0.01	1	1.21	0.93	0.99
Hyperboloid	0.001	1	1.26	0.94	1.10
VH Bone	0.1	1	1.00	0.98	1.02
VH Bone	0.01	1	1.06	0.99	1.13
VH Bone	0.001	1	1.11	0.32	1.01
VH Skin	0.1	1	1.19	0.97	1.07
VH Skin	0.01	1	1.11	0.99	1.20
VH Skin	0.001	1	1.13	0.87	0.94
Mean:		1	1.30375	1.00635	1.28015
σ		0	0.251	0.229	0.301

Table 6.5: Normalised disk read values from rendering real data sets

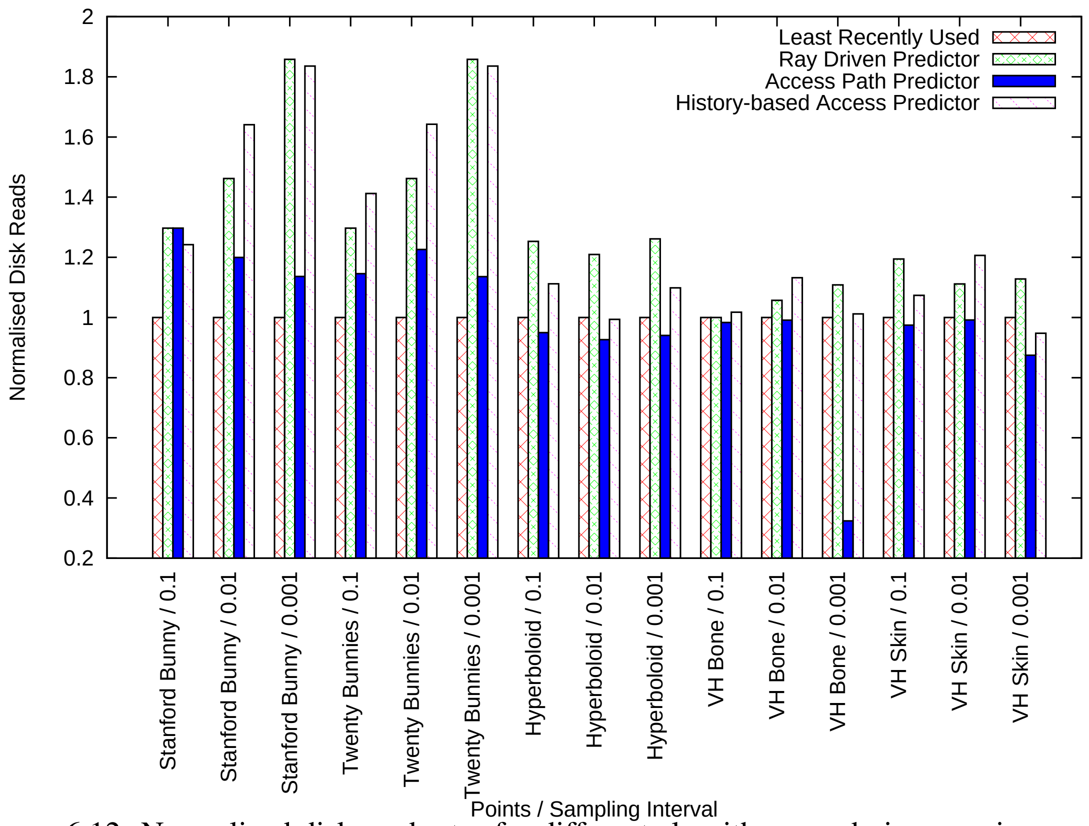


Figure 6.12: Normalized disk read rates for different algorithms rendering a various real datasets

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	99.998%	99.996%	99.996%	99.997%
100	0.01	99.992%	99.999%	99.987%	99.992%
100	0.001	99.999%	100.000%	99.997%	99.997%
1000	0.1	99.528%	99.887%	99.967%	99.979%
1000	0.01	99.981%	100.000%	99.996%	99.997%
1000	0.001	99.961%	99.995%	99.971%	99.981%
10000	0.1	98.173%	99.362%	99.204%	99.374%
10000	0.01	99.663%	99.877%	99.589%	99.743%
10000	0.001	99.932%	99.934%	99.921%	99.950%
100000	0.1	98.173%	99.362%	99.204%	99.374%
100000	0.01	99.663%	99.877%	99.589%	99.743%
100000	0.001	99.932%	99.934%	99.921%	99.950%
1000000	0.1	91.056%	99.511%	99.166%	99.274%
1000000	0.01	98.215%	99.882%	99.859%	99.875%
1000000	0.001	99.644%	99.962%	99.969%	99.971%
Mean		98.927%	99.838%	99.756%	99.813%
σ		2.209	0.220	0.310	0.002

Table 6.6: Cache hit rates for different algorithms rendering a dataset containing points on the surface of a sphere with shadows disabled

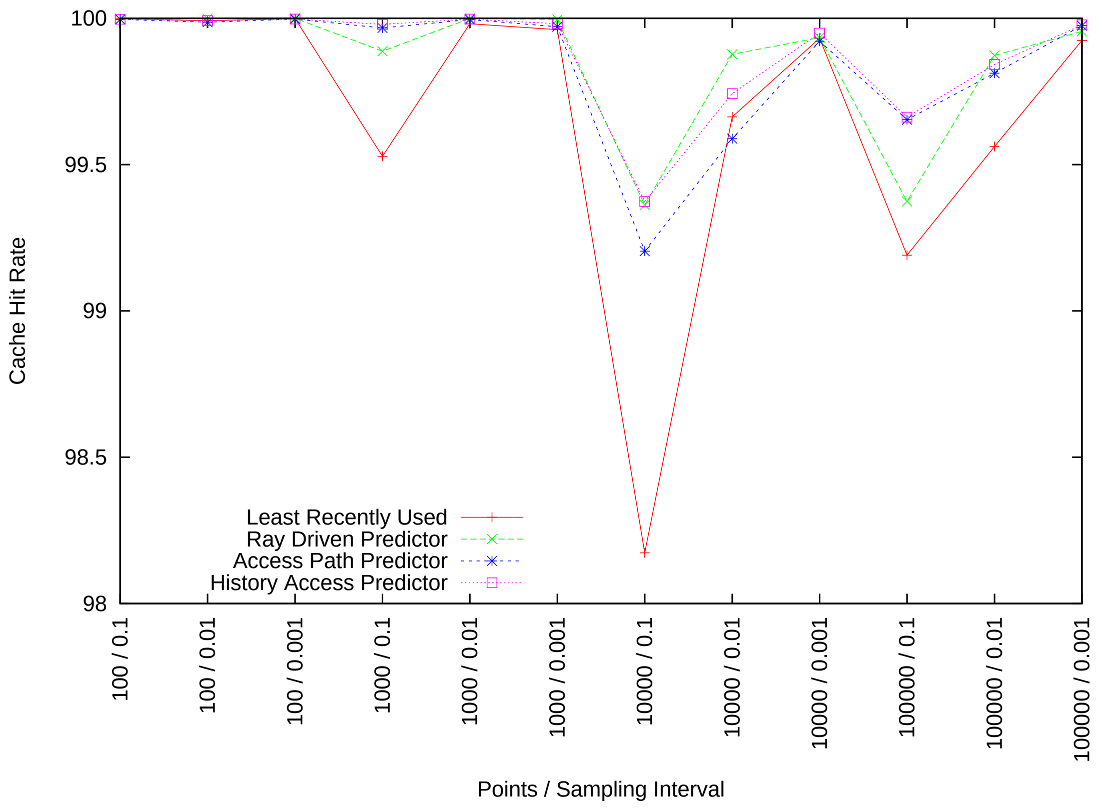


Figure 6.13: Cache hit rates for different algorithms rendering a dataset containing points in a volume with shadows enabled

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	1	38[1]	1.01	1.00
100	0.01	1	1.29	0.64	2.36
100	0.001	1	4.44	1.44	4.80
1000	0.1	1	1.61	0.20	0.47
1000	0.01	1	0.38	0.40	0.84
1000	0.001	1	0.73	1.40	4.89
10000	0.1	1	1.62	0.34	0.80
10000	0.01	1	1.68	1.04	2.82
10000	0.001	1	4.29	0.72	2.13
100000	0.1	1	1.62	0.34	0.80
100000	.01	1	1.68	1.04	2.82
100000	0.001	1	4.29	0.72	2.13
1000000	0.1	1	1.07	0.98	1.05
1000000	0.01	1	1.13	0.95	0.98
1000000	0.001	1	1.27	0.89	0.96
Mean		1	1.80630	.80900	1.92449
σ		0	1.392	0.375	1.394

Table 6.7: Normalized disk read counts for different algorithms rendering a dataset containing points on the surface of a sphere with shadows disabled

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	99.999%	99.994%	99.997%	99.997%
100	0.01	99.992%	99.999%	99.987%	99.992%
100	0.001	99.999%	100.000%	99.997%	99.997%
1000	0.1	99.570%	99.895%	99.967%	99.979%
1000	0.01	99.981%	100.000%	99.996%	99.997%
1000	0.001	99.962%	99.996%	99.972%	99.981%
10000	0.1	98.250%	99.385%	99.095%	99.268%
10000	0.01	99.687%	99.884%	99.656%	99.784%
10000	0.001	99.928%	99.934%	99.927%	99.952%
100000	0.1	98.250%	99.385%	99.095%	99.268%
100000	0.01	99.687%	99.884%	99.656%	99.268%
100000	0.001	99.928%	99.934%	99.927%	99.952%
1000000	0.1	91.124%	99.510%	99.181%	99.284%
1000000	0.01	98.232%	99.884%	99.863%	99.878%
Mean		98.899%	99.834%	99.737%	99.757%
σ		2.229	0.219	0.389	0.312

Table 6.8: Cache hit for different algorithms rendering a dataset containing points on the surface of a sphere with shadows enabled

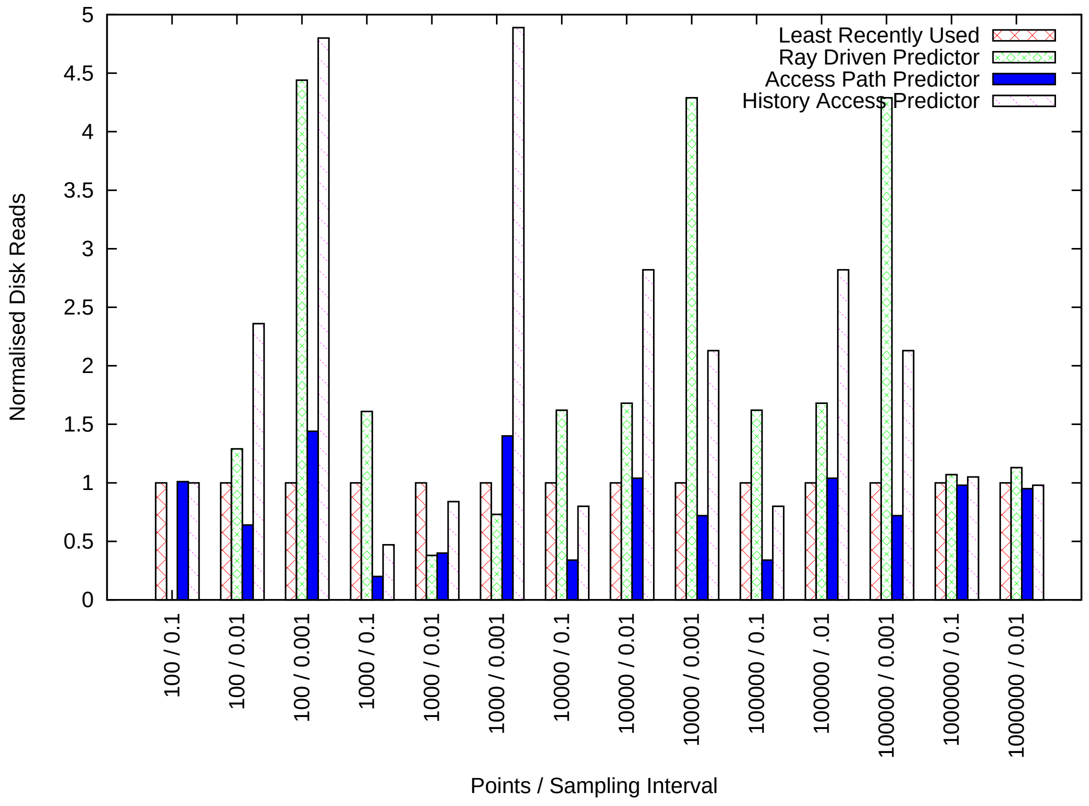


Figure 6.14: Normalized disk read rates for different algorithms rendering a dataset containing points on the surface of a sphere with shadows disabled

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	1	38.0	1.00	1.00
100	0.01	1	1.29	0.70	2.40
100	0.001	1	5.11	1.46	4.83
1000	0.1	1	1.59	0.19	0.45
1000	0.01	1	0.38	0.40	0.82
1000	0.001	1	0.73	1.37	4.74
10000	0.1	1	1.58	0.43	1.28
10000	0.01	1	1.75	0.99	2.57
10000	0.001	1	4.13	0.66	1.85
100000	0.1	1	1.58	0.43	1.28
100000	.01	1	1.75	0.99	2.57
100000	0.001	1	4.13	0.66	1.85
1000000	0.1	1	1.07	0.98	1.05
1000000	0.01	1	1.13	0.95	0.98
Mean		1	1.87308	.80173	1.97597
σ		0	1.452	0.357	1.313

Table 6.9: Normalized disk read counts for different algorithms rendering a dataset containing points on the surface of a sphere with shadows enabled

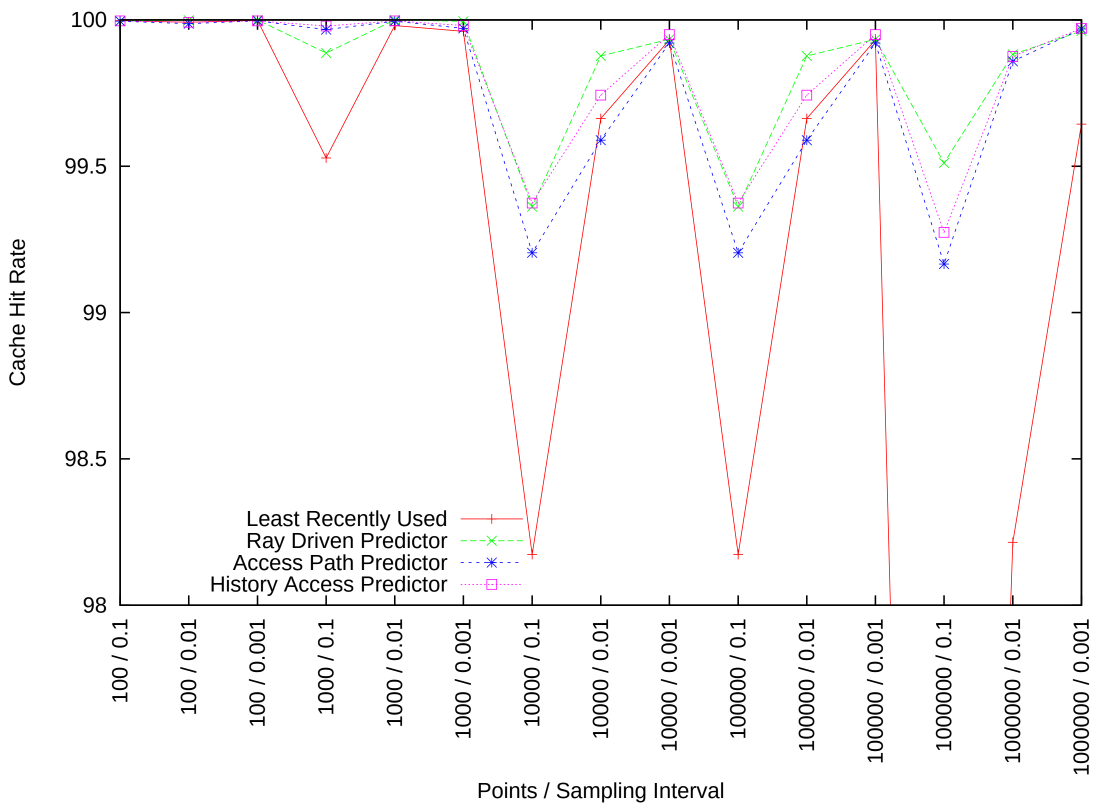


Figure 6.15: Cache hit rates for different algorithms rendering a dataset containing points on the surface of a sphere with shadows disabled

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	99.998%	99.996%	99.997%	99.997%
100	0.01	99.992%	99.999%	99.987%	99.992%
100	0.001	99.999%	100.00%	99.997%	99.997%
1000	0.1	99.528%	99.887%	99.967%	99.979%
1000	0.01	99.981%	100.00%	99.996%	99.997%
1000	0.001	99.961%	99.995%	99.971%	99.981%
10000	0.1	98.173%	99.362%	99.204%	99.374%
10000	0.01	99.663%	99.877%	99.589%	99.743%
10000	0.001	99.932%	99.934%	99.921%	99.950%
100000	0.1	99.190%	99.374%	99.653%	99.662%
100000	0.01	99.561%	99.873%	99.812%	99.842%
100000	0.001	99.924%	99.955%	99.975%	99.977%
Mean		99.6587%	99.8544%	99.8391%	99.8743%
σ		0.511123	0.222618	0.233923	0.185261

Table 6.10: Cache hit for different algorithms rendering a dataset containing points in a spherical volume with shadows enabled

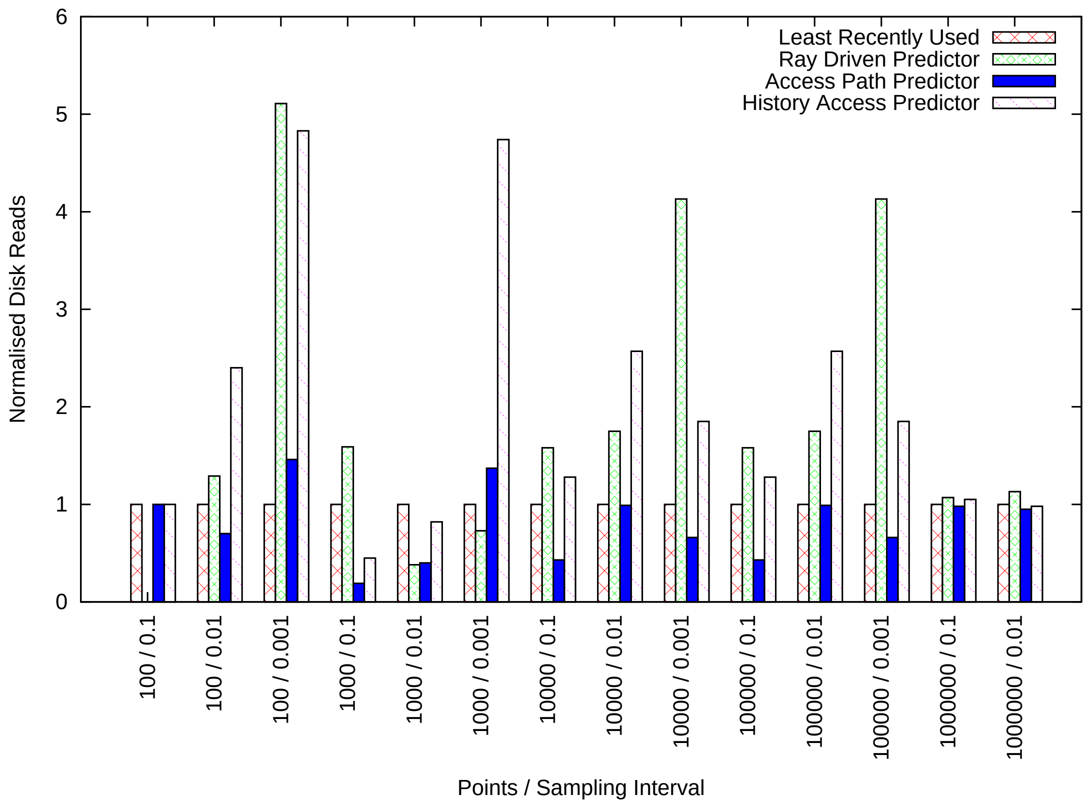


Figure 6.16: Normalized disk read rates for different algorithms rendering a dataset containing points on the surface of a sphere with shadows enabled

Points	Sampling Interval	Least Recently Used	Ray Driven Predictor	Access Path Predictor	History Access Predictor
100	0.1	1	33.2	1.00	1.00
100	0.01	1	1.28	2.35	2.35
100	0.001	1	4.44	4.79	4.79
1000	0.1	1	1.60	0.47	0.47
1000	0.01	1	0.38	0.84	0.84
1000	0.001	1	0.72	4.88	4.88
10000	0.1	1	1.62	0.80	0.80
10000	0.01	1	1.67	2.82	2.82
10000	0.001	1	4.29	2.13	2.13
100000	0.1	1	4.23	1.04	1.04
100000	0.01	1	1.60	0.98	0.98
100000	0.001	1	2.52	0.79	0.79
Mean		1	4.79583	1.9075	1.9075
σ		0	8.66716	1.48156	1.48156

Table 6.11: Normalized disk read counts for different algorithms rendering a dataset containing points in a spherical volume with shadows enabled

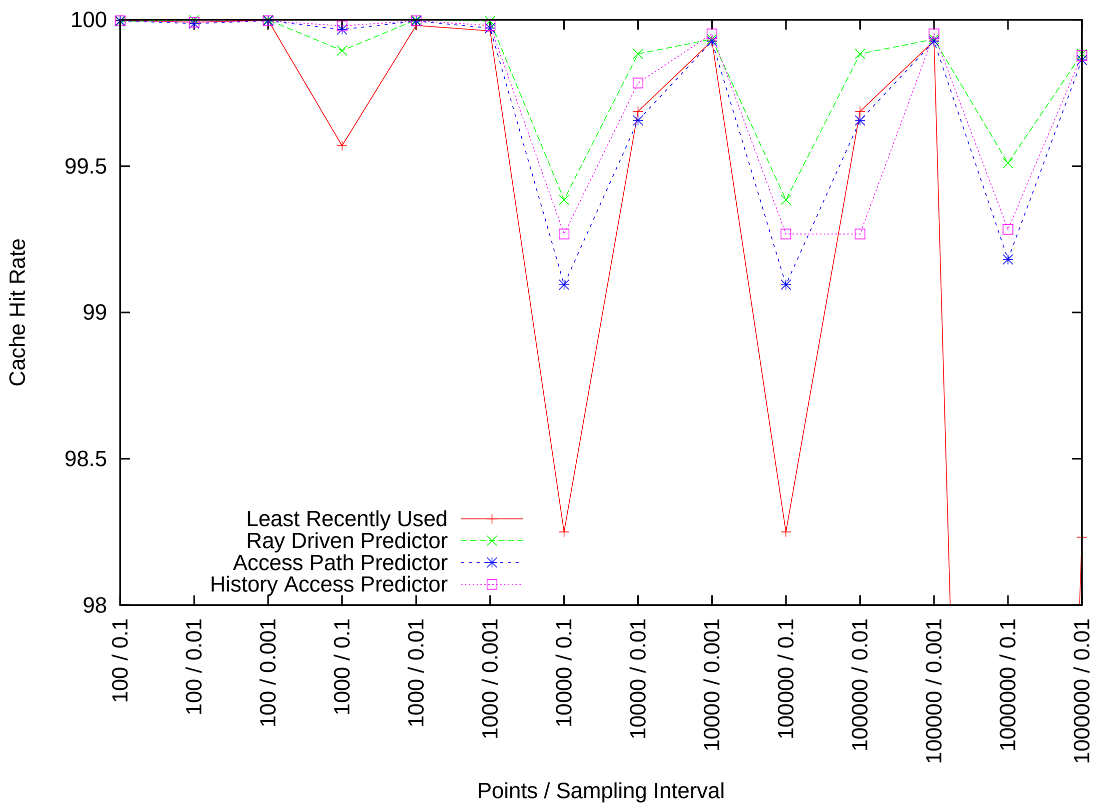


Figure 6.17: Cache hit rates for different algorithms rendering a dataset containing points on the surface of a sphere with shadows enabled

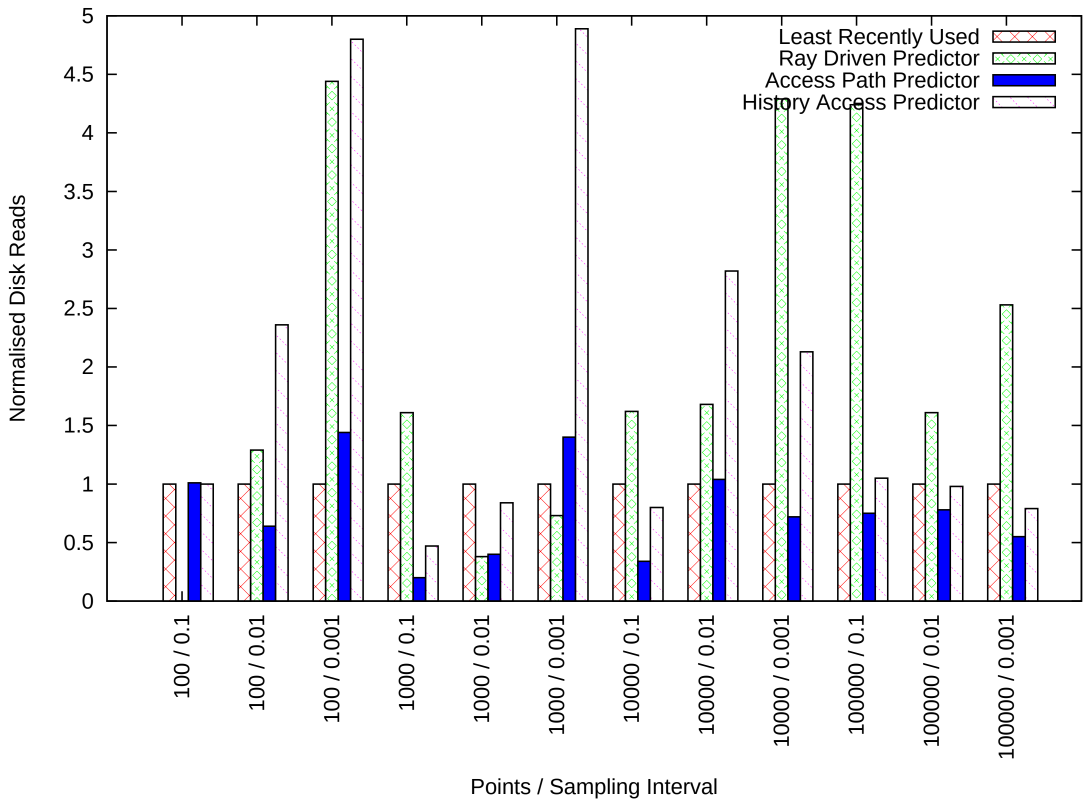


Figure 6.18: Normalized disk read rates for different algorithms rendering a dataset containing points in a volume with shadows enabled

Part III

Autonomic System Simulation

Chapter 7

An Overview of SimEAC

Contents

The next three chapters discuss the motivation behind the development of the SimEAC system, the design and implementation of the simulator and some case studies of the simulator in action. Work described in these chapters was presented at the International Conference on Autonomic Computing 2006 [72].

SimEAC was originally conceived as part of the e-Viz project as a means of simulating parts of the system that could not feasibly be developed within the project and for refining the components that were built. In some e-Viz publications, SimEAC was described as SimuVis (a simulated e-Viz). The name was changed when it became clear that a simulator that met the requirements of the e-Viz project would also be applicable in a more general context.

7.1 Introduction

The development of an autonomic system will inevitably involve the design, test, verification, and optimization of a collection of autonomic elements. As pointed out by [164], there is a significant engineering challenge in the development of such elements for large-scale systems. It is hard to anticipate the combined effects of many autonomic elements when they interact with the system, users, and one another. It is difficult to create a variety of possible scenarios in which autonomic elements are expected to function, and to recreate a particular condition for the purpose of testing and optimization. It is risky to install any autonomic element that has not been through an adequate engineering process. It is costly to provide a replica of the live system in order to engineer autonomic elements.

In computer science and applications of scientific computation, the common wisdom is to use simulation to aid the understanding and prediction of the behaviors of phenomena that exhibit non-deterministic characteristics, and are too complex to model algorithmically. This leads to the motivation for providing a simulation tool to support the engineering of autonomic elements. The core of an autonomic element usually consists of one or a few algorithms that encode strategies, methods and operations for self-management in response to the dynamic environment where it functions. The simulation of an autonomic system and its operational environment can thus feature different granularities, with relatively fine-grained simulation for the core autonomic algorithms and strategies, and coarse-grained simulation of the operational environment, including the various system components, user tasks, and dynamic events that influence the behaviors of the autonomic algorithms and strategies.

In this work, we focused on the provision of such software development support through a simulation system, SimEAC (Simulated Environment for Autonomic Computing). SimEAC enables a system designer to create a system infrastructure to be simulated by specifying a collection of hardware components and performance attributes, and it provides a degree of control over the operating systems and tasks, hence allowing new autonomic algorithms to be prototyped, tested and optimized on a virtual system infrastructure. SimEAC supports modeling and simulation of a variety of system architectures, including large scale networked systems, in a relatively abstract manner. It provides a user interface for configuring a virtual system infrastructure to be simulated, and for specifying the statistical and stochastic behavior of the system and running tasks, including hardware and software failures, viruses and dynamic job loads. SimEAC allows an algorithm designer to conduct experimentation on different virtual architectures without the risks of bringing down a production system, and enables more scientific evaluation of autonomic algorithms in analyzing typical attributes of an autonomic system such as scalability and problem localization.

SimEAC is specially designed for simulating autonomic systems. Differing from conventional system simulators, it places its emphasis on the fine-grained algorithmic specification of the autonomic algorithms through program code, while providing support to modeling of the operational environment with different levels of granularities, including abstract models and XML-based specification, build-in procedural models, interactive parameterization, and programmable procedural models. It addresses the needs for testing various self-* features of an autonomic system by allowing the specification of dynamic behaviors, for instance, simulating system failures for testing self-healing, simulating viruses for testing self-protection, and simulating stochastic system reconfiguration for testing self-optimization. In addition, it allows the core autonomic algorithms and strategies to change the configuration of the operational environment for testing self-reconfiguration. Nevertheless, SimEAC itself is not an autonomic system, because it is essential to maintain a high degree of deterministic and predicable behaviors of the simulator, ensuring that the simulation results can be reproduced with the same parameter settings.

The chapter is organized as follows. Following a brief review of the related work in Section 2, we discuss the motivation and requirements of a simulation system for autonomic computing in Section 3. We present the abstract design model of SimEAC in Section 4. In Section 5, we outline the main features of an extensible markup language (XML) based Application Programming Interface (API) and the user interface for configuring a virtual system. This is followed by a description of our implementation of the simulation engine in Section 6. To demonstrate this feature, we present two case studies in Sections 7 and 8 respectively. One case study involves a simulation of a self-organization algorithm for managing distributed computation in an ad hoc network. Another is a simulation of an agent-driven resource manager in a virtual cluster-based visualization environment. We give our concluding remarks and suggestions for future work in Section 9.

7.2 Related Work

Autonomic computing is a relatively new research area, and many of the tools required for designing and building a fully autonomic system do not yet exist. The pioneering attempts typically fall into the following categories:

Software development environments for agent-based systems Software agents are commonly used in implementing autonomic components. Toolkits such as, [239], [77], [152], [24], [33] have provided developers with tools for quickly developing agent systems by allowing agents to be created and used in a generic fashion. For example, [78] describes an electronic marketplace system built atop ZEUS.

Generic software environments for autonomic applications Though the development of such development tools is still in its infancy, several attempts were made, which include projects such as QADPZ [80], AUTONOMIA [98] and Almaden OptimalGrid [88]. QADPZ [80], provides an open source framework for managing heterogeneous distributed computation in a network of desktop computers using autonomic principles. AUTONOMIA [98] is a prototype software development environment that provides application developers with tools for specifying and implementing autonomic requirements in network applications and services. OptimalGrid is a self-configuring, self-healing and self-optimizing grid middleware, using a set of distributed whiteboards for communication between the different nodes.

New programming languages for semantic rules There are new developments of ontology languages for encoding knowledge and services on the Web, including RDF [28], RDFS [37], OWL [210], and OWL-S [208]. Although such languages do not themselves support directly the programming or evaluation of autonomic features of a system, they provided a means for encoding ontological representations of the rules that determining certain types of autonomic behaviors. More recently, [47] proposed the WSML rule languages, which uses logical formalisms to describe the semantics of a semantic web service.

Tools for protocol development Communication between autonomic components is difficult to manage and its correctness is usually assured through correct communication protocols. Examples of these include wireless testbeds [317, 306].

Other specialized development tools have also been designed for features such as security and trust [63], context awareness [291], performance analysis [196] and business support [7].

Dobson et al. [96] discussed the design and analysis of autonomic algorithms for communication, while providing a comprehensive survey of a number of design techniques. Their survey also highlighted the gap between the needs for testing and evaluating autonomic algorithms, in terms of their stability and reliability.

To address the needs for testing and evaluation of autonomic algorithms, which exhibit typically non-deterministic behaviors, simulation can play a vital part in the development process, as it enables us to determine how well they would work without devoting significant time and effort to their creation.

Simulation is a vital part of the development process for these tools, as it enables us to determine how well they would work without devoting significant time and effort to their creation.

The majority of existing tools focus on emulating a particular kind of hardware. Alpha [93], PowerPC [15] and Itanium [309] simulators are available allowing code for these platforms to be examined in great detail. Some systems, such as Bochs [185], QEMU [29], Simics [205] and Xen [26, 71], provide a complete simulated (or virtualized) system.

Outside the full-system simulation area are simulators such as the Emulab Network Emulation Testbed [330]. This provides a highly accurate way of simulating a variety of network conditions, and has been used in a number of activities related to *Autonomic Computing*, including [13] and [52]. While a useful tool, Emulab has a limited range of applicability; it focuses on simulating one aspect of a system — the network — very accurately.

Tools such as Emulab, OPNET [57] and ns2 [245] are very important to development of a large category of autonomic systems. They provide very accurate testbeds for analyzing the interactions between components in a network. This is useful for a large subset of autonomic computing research, for example routing strategies in mesh networks, but is of limited use elsewhere. These simulators are very useful in the prototyping stage, but less so in the design stage where only a high-level overview of the system is desired.

Such fine-grained systems are usually intended for the last stage in the development of system-level software. This is not particularly relevant to *Autonomic Computing* in general, since a large number of autonomic systems involve the global behaviors of complex systems and are often affected by — or rely on — emergent properties of these systems.

At a slightly higher level of abstraction are systems like CSIM [272] and GridSim [292]. CSIM is a relatively flexible simulation language based on C. It is designed to work at a very fine granularity, giving the ability to evolve the simulated design into a finished product. While a useful tool in a number of areas, it is not designed for testing the applicability of high-level autonomic concepts.

GridSim works well in the Grid arena, but makes certain assumptions on the design and structure of the underlying Grid infrastructure that may make it unsuitable for application to Autonomic Computing problems. GridSim runs each simulated object in a separate thread. The developers of MICSim [48] discovered that this introduces a degree of non-determinism into the simulation due to scheduler interaction, which can make it difficult to reproduce results.

At the most abstract end of the spectrum are theoretical models such as *Communicating Sequential Processes* (CSP) [46], [256] and systems such as CSP-CASL [265] and Maude [214], [76], which allow the simulation of mathematical specifications for complex systems. These are useful tools for evaluating algorithms in an abstract context, but they are less good at indicating real-world performance.

7.3 Motivation

In theory, it is possible to build a composite simulation system for autonomic algorithms and their operational environment by integrating a mixture of simulators into a single environment, providing a testbed for a variety of configurations. In practice, the interfaces between different simulators will be costly to implement, and the granularity of different simulators will determine the granularity of the autonomic system to be simulated, giving little flexibility in varying the levels of granularity for different components of the autonomic system. The level of details provided by these simulators is often neither required nor beneficial in prototyping and testing autonomic systems, and the effort for specifying various components in such details would render such a composite simulation system totally cost-ineffective.

While [164] outlined the overall challenges in engineering autonomic systems, [273] expressed unequivocally the need for a simulation system in the context of Grid research where autonomic computing has a great role to play. Such a simulation system, to be useful in an Autonomic Computing context needs to meet a number of requirements.

The system must allow for fine-grained specification of core autonomic algorithms and strategies to be simulated. It is the most natural way to provide a placeholder where algorithms in the form of computer programs can be run by the simulation system. Since the simulation system is designed to be used by designers and developers of autonomic components of a system, a programming interface is not a burden but a necessity for fine-grained simulation and tuning of autonomic algorithms and strategies.

The system must enable the specification of the operational environment of an autonomic component at different levels of granularity. It is most unlikely that a designer and developer of autonomic components is also a hardware architect, an operating system designer, and a system, network and security manager. It is usually difficult and often unnecessary to model the exact behaviors of the underlying system infrastructure, network traffic, job loads, user interactions, and other dynamic features of a real world environment where an autonomic component will operate. Hence, allowing some coarse-grained specification of the operational environment is highly desirable, while providing a means for medium to fine-grained specification whenever it is necessary. The simulation must support the analysis of a variety of performance attributes of an autonomic system, specially, scalability, stability, reliability, problem localization and efficiency. Hence, it should enable the modeling of large and complex configuration of a variety of computing and networking devices, without the burden of specifying every single component with the same level of granularity.

The system should ideally provide tools for specifying building blocks of the operational environment at an abstract level, allowing simulation of systems, which are highly theoretical or futuristic. Autonomic systems are frequently intended to be deployed on top of systems and middleware that are still experimental. Any assumptions made based on current system designs are likely to quickly become obsolete.

The system must provide support to testing and evaluation of various self-* features of autonomic algorithms. While it is for the users of the simulation system to provide autonomic solutions to various problems, it is for the simulation system to provide a platform where such problems can be easily modeled and simulated, and to provide an interface between the simulated problems and the solutions to be tested. Based the main self-* features outlined in [164] and the numerous instances given in [96], one may consider the following specific requirements for a simulation system.

To support *self-configuration*, the simulation system must allow the operational environment to be reconfigured dynamically under the influence of the autonomic components being tested.

To support *self-optimization*, the simulation system must be easily re-configurable, through simple specification interfaces. It should also allow the operational environment to be reconfigured dynamically in response to various stochastic events beyond the influence of the autonomic components being tested. Thus, autonomic algorithms can be tested in a variety of contexts against their capability of self-optimization. An autonomic system is usually most useful in environments where there is little prior knowledge about the structure of the operational environment available in advance (e.g., in ad hoc networks). Being able to quickly test the autonomic algorithm in a large number of situations is important for robustness and reliability.

To support *self-healing* and self-protection, the simulation system much be able to create a variety of failures (including hardware, network, system and application software), and a variety of security threats and their impacts on system resources and running users tasks.

To support *self-awareness*, the simulation must provide an interface allowing the autonomic component to gather global and local information about the operational environment to be simulated. If it can, highlight this feature in the text.) It must provide dynamic event (and message) capturing, and the event-driven (and message-driven) programming paradigm.

SimEAC has thus been designed to fulfill these requirements.

7.3.1 Emergent Properties

One of the key features of Autonomic Computing is the design of systems based on the concept of emergent behaviours. Autonomic systems are typically constructed from simple components with a simple set of interactions. Complex behaviours evolve from these simple rules, as with [267].

The problem comes from the fact that, while we have a plethora of theoretical models for reasoning about software on a small scale, we do not have theoretical tools available for designing systems to create a given emergent property. Ideally, we would like to be able to first state the desired emergent behaviour, and then work backwards and develop the simple rules that will allow create the desired outcome. The current state of the art, however, relies significantly on the intuition of system designers in order to be able to build such systems.

One of the goals of SimEAC is to create a system whereby designers can quickly implement models of their ideas for simple behaviours, and then see how they interact on a large scale. While the system does not provide a theoretical model for designing the rules from the behaviours, it does allow the current trial-and-error approach to be performed much more quickly and easily.

7.4 An Abstract Model

SimEAC has been designed to meet the goals outlined above. The remainder of this chapter discusses the design choices from a high-level perspective. The next chapter covers some of the implementation detail, and how the system is used.

SimEAC is constructed in three layers as shown in Figure 7.1. The first layer describes the capabilities of the hardware, including connections between individual components.

The second layer contains *resource managers*, which allocate access to hardware resources from the first layer to tasks in the third layer. *Resource managers* can be arranged hierarchically, and represent operating systems and autonomic middleware. The precise arrangement of resource managers is defined in a simulation specification, and hence modifiable by a user of SimEAC.

The third layer represents userspace components of the system. Unlike resource managers, these are not tied to a particular piece of hardware. This allows tasks to be mi-

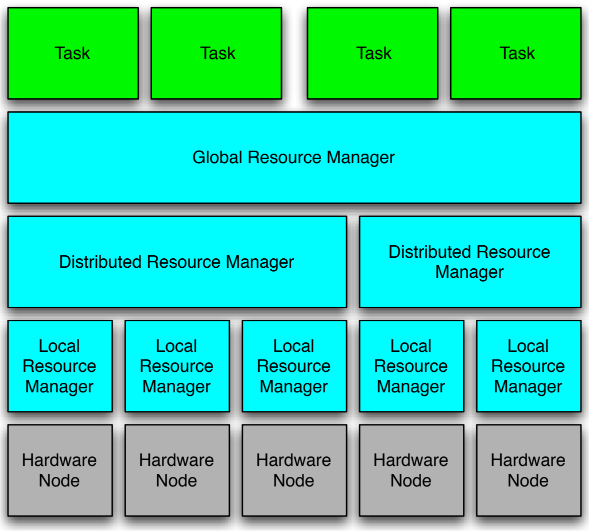


Figure 7.1: SimEAC architectural overview.

grated between hardware components, assuming the resource managers responsible for them support this.

7.4.1 Hardware Simulation

SimEAC, at the lowest level, provides a mechanism for describing hardware grids. The basic building blocks of these are:

- Storage Units represent anything that holds data.
- Processing Units represent anything that is capable of executing code, including general purpose CPUs, GPUs and DSPs.
- Interconnects are used to join components together.
- Containers provide a mechanism for grouping objects.
- Routers allow multiple interconnects to be connected.

These are combined using an XML format described in Section 8.3.1. Each of the building blocks is described by an algebraic specification (similar to the one shown in Algebra 7.1) and a semantic definition.

The arrangement of these building blocks is hierarchical. All items, including other containers, can be contained within containers giving a simple inductive definition of the entire system. Most of these building blocks can be configured in various ways. A processing unit has a specified instruction set and speed, etc. An an overall interconnect has a latency and a throughput as well as a maximum latency and throughput for each device. This allows asymmetric links to be added easily.

Algebra 7.1: Interface specification of a processing unit

```
Algebra InstructionSet is

    sort InstructionSet

    ops GPU PPC x86 SPARC IA64 MIPS DSP : → InstructionSet

end Algebra

Algebra ProcessingUnit is

    including Nat
    including InstructionSet

    sorts Nat ProcessingUnit InstructionSet

    op newProcessingUnit : InstructionSet Nat Nat Nat → ProcessingUnit
    op instructionSet : ProcessingUnit → InstructionSet
    op floatSize : ProcessingUnit → Nat
    op intSize : ProcessingUnit → Nat
    op vectorSize : ProcessingUnit → Nat

end Algebra
```

The container node is an abstract grouping of hardware components. It allows a collection of hardware components to grouped at the granularity desired by the user of the system. An individual computer would typically be represented by a computer. Components in this computer might also be grouped in containers, for example a GPU and some VRAM might be aggregated into a container representing a graphics card. Any container may have a resource manager and an arbitrary number of tasks associated with it. Containers may also be used for higher-level groupings, such as clusters or entire grids. Containers used in this way provide a semantic grouping for the system's user and allow resource managers representing grid middleware or distributed operating systems to be simulated.

7.4.2 Resource Managers

Resource managers sit between the hardware and executing tasks. Resource managers are generally divided into two categories, local and distributed. A *distributed resource manager* may have children, while a local one may not. A *local resource manager* is responsible for scheduling tasks on the processing units under its control, while a distributed resource manager may be responsible for process migration between nodes.

The lines between the two categories of resource manager can be blurred in some cases. A resource manager for a computer containing some special purpose execution units might directly handle the general purpose hardware (as a local resource manager would) but delegate the scheduling of tasks on the special-purpose hardware to another resource manager (as a distributed resource manager would).

While an operating system is usually implemented as a local resource manager, this is not always the case. A system such as DragonFly BSD, which runs a kernel image on each processor would be best implemented by a local resource manager on each processing unit and a distributed resource manager mediating between them. As this example shows, the division or responsibilities among resource managers does not have to directly reflect the structure of the operating system in question.

The primary responsibility of a resource manager is to manage process scheduling; it assigns time on the processing units under its control to running tasks. Part of this responsibility includes maintaining processor-affinity sets. When a scheduler determines that it is overloaded, it can notify its parent in the resource manager hierarchy that it has a set of under-scheduled processes which it wishes to migrate. Conversely, a distributed resource manager can request that a child resource manager accept responsibility for a migrated task.

Resource managers are responsible for mediating access to all hardware. This includes reserving space on storage units. Space is reserved by name, allowing different processes to request access to the same storage resource. Virtual memory is built on top of this mechanism. Tasks request space on a volatile storage device. The resource manager must then determine which device to allocate this space on, and optionally create some backing space on a slower device. The task can then detect its in-core size and schedule page faults. This mechanism also permits the creation of cache hierarchies.

All interaction between tasks, and between tasks and hardware, happens through a specified interface, and many of the functions of this are delegated to the resource manager responsible for the task. As well as those already described, this functionality also include routing; a resource manager can replace the system's default algorithm for finding the path for messages, if required.

Any container may have a resource manager associated with it. As with other parts of the system, resource managers are required to conform to a formal algebraic specification, a reduced version of which is found in Algebra 7.2.

7.4.3 Tasks

Tasks can be created either when the system starts, or by other tasks as the system runs. They may:

- Consume processing unit run time.
- Increase or decrease their virtual memory size.
- Issue page faults.
- Create and resize files on long-term storage devices.
- Read and write data from files.
- Create new tasks or acquire handles to named tasks.

Algebra 7.2: Abridged specification for a resource manager

```
Algebra ResourceManager is

    including Nat
    including ProcessingUnit
    including Task
    ...

    sorts Nat Bool Container ProcessingUnit ... ResourceManager

    op newResourceManager : Container ResourceManager →
       ResourceManager
    op canRun : ResourceManager Task → Bool
    op addTask : ResourceManager Task → ResourceManager
    op removeTask : ResourceManager Task → ResourceManager
    op parent : ResourceManager → ResourceManager
    op setVMSizeForTask : ResourceManager Task Nat → ResourceManager
    op run : ResourceManager Nat → ResourceManager
    ...

    var R : ResourceManager
    var C : Container
    var T : Task

    eq parent(newResourceManager(C,R)) = R
    eq removeTask(addTask(R, T), T) = R
    ...
end Algebra
```

- Send and receive messages.

Algebra 7.3: Abridged specification for a task

```
Algebra Task is

    including Nat
    including Task
    including String
    including Message
    including TaskDelegate
    ...

    sorts Nat Bool Container ProcessingUnit ... ResourceManager

    op null → Task
    op newTask : String → Task
    op setDelegate : Task Delegate → Task
    op delegate : Task → Delegate
    op run : Task Nat ProcessingUnitList Nat → Task
    op threads : Task → Nat
    op retain : Task → Task
    op release : Task → Task
    ...

    var S : String
    var T : Task
    var R : Task
    var M : Message
    var D : Delegate

    eq release(newTask(S)) = null
    eq release(retain(T)) = T
    eq delegate(setDelegate(T,D)) = D
    ...
end Algebra
```

This small set of operations allows the user to model the interaction of any computational task with a wide variety of systems and gain feedback as to where bottlenecks might occur. An abridged specification of a task is given in Algebra 7.3. Tasks in SimEAC closely correspond to *actors* in the *Actor Model* [141] of concurrent computation. From a certain perspective, SimEAC could be viewed as an implementation of this model with the hardware and resource manager components providing some extra syntactic sugar for mapping the results back on to real computational systems, but not affecting the expressive power of the system.

7.5 Simulating Failure

One of the key components of an autonomic system is that it should be self-healing. In order for a system to heal, part of it must break. In general, any failure in a system can be categorised as either a hardware or a software failure. Simulating software failure is trivial; the simulated task simply terminates itself at a random (or specific, but undesired) point in time. Alternatively, a task can enter a failure mode, consuming all CPU time available to it, for example, or leaking memory, without doing any useful work.

Listing 7.4: Simulating failure in a task.

```
- (sim_time_t) runFor:(sim_time_t)nanoseconds
         onProcessors:(NSArray*)processors
               atTime:(sim_time_t)now
{
    switch(failureMode)
    {
    case terminate:
        [delegate terminate:self];
        return 0;
    case infiniteLoop:
        return nanoseconds;
    default:
        return [super runFor:nanoseconds
         onProcessors:processors
               atTime:now];
    }
}
```

Listing 7.4 shows how a task could perform both of these forms of failure. This code assumes that the task's class has an instance variable failureMode, where terminate and infiniteLoop are valid values. This can be set by any message handlers, or at a given time, to either of the described failure modes. The first will result in the task terminating immediately (and uncleanly). The second will result in the task consuming all CPU time allocated to it and never performing any useful work. If no failure mode is set, the task will use the superclass's implementation of the run method. If this task inherits from SimpleTask then this will use the dynamic dispatch mechanism to pass messages on to different handlers.

In order to simulate hardware failures, SimEAC provides a mechanism for disabling and re-enabling hardware components (software components can trivially disable and enable themselves by simply stopping responding). While SimEAC provides the mechanism, implementation of the policy is left up to users of the system. A task can cause individual components to fail either at a specified time or in response to stimuli from within the system. This allows failures to be configured as required.

Listing 7.5: Simulating hardware failure.

```
HardwareNode * node = [[delegate container] componentNamed:component];
[node setEnabled:isEnabled];
isEnabled = !isEnabled;
```

Listing 7.5 contains a snippet of code from the included DeterministicFailure task class. This task is started with the name of a component and a list of times as arguments. As each of the times is passed, it toggles the state of the named component. The first two lines of this get a reference to the named component in the container, and toggle its state. The final line updates an instance variable to the state of the piece of hardware after the next time interval is reached.

Since this is a task, it can only run whenever it has CPU time. If you are simulating failure of a processing unit, then the easiest solution is likely to be run the task outside. For complicated situations, you may wish to simply add a failure controller node to your grid. Unlike every other interaction in the system, enabling and disabling hardware resources can be done without a path to the target. A failure controller can be set up as a single container with a single processing resource without the need for any interconnects between it and the rest of the grid. As such, it will have no impact on the simulation other than to enable and disable components.

Note that the setEnabled: method can be called from any piece of code that can get a reference to the named hardware, including a resource manager.

7.6 Simulation Engine

The concepts of discrete time and message passing are central to the simulation engine. The simulation system is a blend of two approaches, *discrete time simulation* and *discrete event simulation*. At the lowest level, the system is able to act as a nanosecond-granularity discrete time simulator. Time within the simulation elapses one nanosecond at a time. This is not quite smaller than the smallest length of time in which a complete event can happen on a modern computer; a 5GHz CPU will execute 5 cycles in this time, and a 1Gb/s network will transmit one bit. It is, however, a much finer granularity than is likely to be needed for simulating large-scale autonomic systems. If finer granularity is needed then it is more likely that an instruction-accurate CPU simulator would make sense as a simulation environment, since the inaccuracies introduced by the kind of high-level abstractions made by SimEAC are likely to introduce far more errors at this scale than the lack of temporal detail. At this level, the system will behave in a manner similar to a *real time simulation*.

On top of this system is built a discrete event system, where all independent components communicate via message passing. This approach has been designed to meet the goal of allowing the user to trade simulation accuracy for run time. The discrete time model allows for varying detail, simply by adjusting the size of the time interval. By mapping discrete events into discrete time slices, it is possible to use the same high-level simulation at different granularities. Discrete events within the system are generated at the finest temporal granularity, giving an approximation of a continuous-time model, but are then propagated and handled at the user-defined discrete time slice granularity.

Each component maintains an event queue, as in any parallel discrete event simulation. Unlike pure discrete event simulations, where the events in the queue would be handled individually, discrete time slices are allocated to them in which they may handle the events. These slices may be long enough to process multiple events, or too short to process an entire event. This may also vary depending on the size of the event. For example, an event encapsulating the transmission of some data over an interconnect might be completed in a single time slice if it represented a small packet, but take multiple slices for larger packets. It is up to the individual components to track the completion of partial events.

7.6.1 Messages

Most interaction between components in the system takes place through a message passing system. Accesses to storage devices and communication between tasks, for example, are both accomplished via the same message passing mechanism.

Each message has an associated size, used to determine how long it takes to travel along each interconnect on its path, and a set of properties. The properties are in the form of an associative array, and are interpreted by the recipient.

Messages are delivered along interconnects, handled at the hardware layer. The same mechanism is used both for interacting with hardware and for (distributed) IPC, allowing accurate modelling of data flow.

Messages are guaranteed delivery in most cases, although unreliable mechanisms can be implemented easily by randomly dropping messages. The delivery semantics guarantee sequence between two parties. This is not guaranteed between more than two processes, however. The exception to the rule of guaranteed delivery is the case in which no connection between two tasks (or other message-sending components) exists. Note that this disconnection may occur after the message is originally sent, if a hardware component failure is simulated between the message being sent and arriving.

Consider the situation of two producer tasks, A and B, both send a sequence of messages (A_1, A_2, $\ldots$ and B_1, B_2, $\ldots$) to a third task, C. In this case, A_1 will always arrive before A_2, and B_1 will always arrive before B_2. The order in which A_1 and B_1 will arrive, however, is undefined unless other factors come into play. If the connection between A and C has a higher latency than the connection between B and C, for example, the messages from A will arrive consistently later than those from B, assuming that they are sent at the same time.

For simulating packet-switched networks, the resource manager can split a message into a sequence of messages representing packets and then have them re-assembled by the receiving resource manager before being passed to the task.

The simulation engine also includes support for sending link-local broadcast messages. A broadcast message is delivered to every container on the network segment. The routing of these beyond that is handled by the resource manager associated with the container.

The combination of point to point transmission and broadcast allows multicast routing to be implemented at the resource manager level if required. Since the simulator's transmission mechanisms do not enforce strict layer separation at the OSI network layers, there is no semantic difference between having it implemented at the hardware and the resource manager layers. Adding support at the hardware layer for multicast was considered, however it was decided that it was not possible to anticipate all possible routing strategies.

Optimal routing in a multicast network requires knowledge of the structure of the network. An autonomic routing algorithm would dynamically determine the structure of the network and determine the best paths to the various endpoints in the multicast group, while a simulation running on an existing multicast-capable network could assume that this knowledge had been gathered before the simulation ran. If this were implemented at the hardware level of the simulator, it would be harder for users to replace with their own behavior.

By moving it in to the resource manager, users can add their own multicast capability easily by adding secondary destinations in the message dictionary. This also makes it possible for more complex routing systems, such as a system which uses a different multicast strategy depending on message types, which would be much harder on a simulation environment which supported multicast as a “native” ability.

Messages are defined by a pair of C structures and a set of C functions. These are shown in Listing 7.6. The Message type represents a message in transit. The QueuedMessage type is used for convenience when a message has arrived, and simply associates a time stamp with a message.

Listing 7.6: Structures representing messages

```
typedef struct
{
    NSString * type;
    NSDictionary * body;
    double size;
    id sender;
    BOOL isBroadcast;
} Message;

Message * newMessage(void);
Message * copyMessage(Message* aMessage);
void freeMessage(Message* aMessage);

typedef struct
{
    Message * message;
    sim_time_t timestamp;
}
QueuedMessage;

QueuedMessage* newQueuedMessage(void);
void freeQueuedMessage(QueuedMessage* queuedMessage);
```

The functions for creating and destroying these structures pool freed ones. Since they typically have a relatively short lifespan, the overhead of calling malloc() and free() found to be a performance problem in earlier versions of the system.

Most of the fields in the Message structure should be self-explanatory. The isBroadcast attribute determines how the message is delivered when passed over an interconnect. If this is set, the interconnect will pass the message to all connected endpoints. This allows the implementation of multicast and broadcast networks on bus networks. Their implementation on switched networks can be handled by using a custom resource manager at the switch to implement the required delivery strategy.

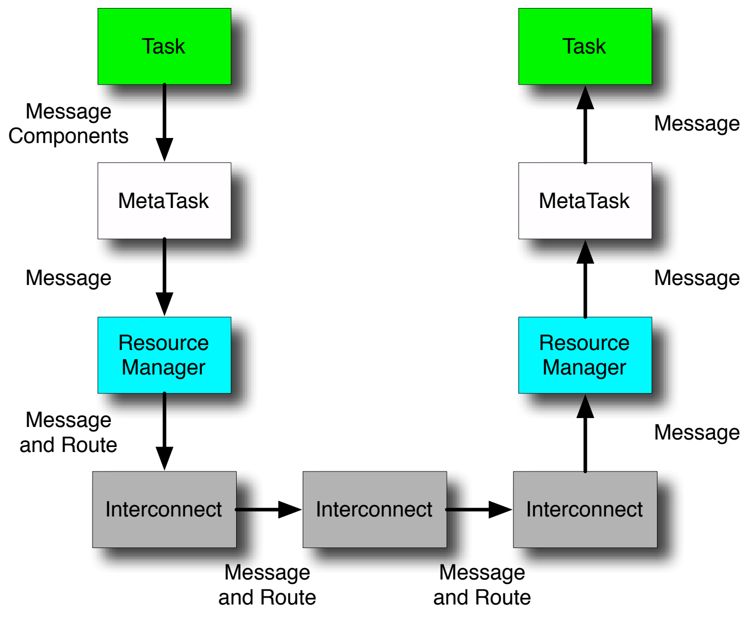


Figure 7.2: The lifecycle of a message in the SimEAC system.

Figure 7.2 shows the life of a typical messages in the system. The first actor is the task sending the message. This invokes a method in its delegate which assembles a message from its principle components — the type, body, size and origin — and then begins the process of sending it.

The MetaTask class shown here is the glue layer which mediates interactions between the task layer and the resource manager layer of the system. It contains a default routing algorithm for cases when the resource manager chooses not to implement one. The existence of the MetaTask is transparent to users of the system.

The route and the message are then passed to the first interconnect in the sequence, and then on to each subsequent one until they arrive at their destination. By default, they are then added to the receiving task's message queue. The message queue itself is implemented in the Task class from which it is suggested that new tasks inherit. This allows custom message queue implementations to be created if required.

For simulating packet-switched networks, the resource manager can split a message into a sequence of messages representing packets and then have them re-assembled by the receiving resource manager before being passed to the task.

The simulation engine also includes support for sending link-local broadcast messages. A broadcast message is delivered to every container on the network segment. The routing of these beyond that is handled by the resource manager associated with the container.

The combination of point to point transmission and broadcast allows multicast routing to be implemented at the resource manager level if required. This decision was made since the relatively poor support for multicast on existing networks, particularly the Internet, means that multicast is generally implemented at the application level, if it is used at all.

7.6.2 Time

The simulation engine guarantees that all events that happen in tick n will be processed before events in tick $n + 1$. The order of two events at times t_1 and t_2 is not guaranteed if both t_1 and t_2 fall within the same tick.

Time is represented in the system by the sim_time_t scalar quantity. This stores a 64-bit value representing a number of nanoseconds, allowing a simulation to run for just under 600 years of simulation time before an overflow occurs.

Every time-dependent event has an associated time stamp. When a task is run, for example, it is told the start time and the duration of the time slice. Any message it sends or receives will, likewise, have a time stamp.

While time stamps provide a mechanism for dealing with time on a fine-grained level, ticks exist at a coarser granularity. Each tick is an integer number of milliseconds. During each tick, every single component in the system is guaranteed to be run a minimum of once, unless disabled or not active for some other reason. Each resource manager will be run once, and should then allocate the time give to it amongst the tasks for which it is responsible. The tasks, in turn, should process any pending messages, or run their idle loop if required.

The simulation engine runs using discreet time intervals, known internally as ticks. The size of each tick is controlled by the user; shorter ticks produce a more accurate simulation at the expense of requiring more processor run time for the simulator. Every message in the system as a timestamp associated with it, as does each component. Incrementing of time passes down the grid via the –tick: method, which is implemented by all hardware nodes. Each container, including the root container, will call each child in turn.

The argument to this method is the time at the end of the next tick. For convenience, the –tick: method in the HardwareNode class sets two instance variables, last and now, containing the start and end of the current tick respectively.

Three of the standard hardware components have special behaviour when they receive a tick: message; interconnects, containers, and storage units. Interconnects will pass as many of the messages that they have enqueued as possible across their connection in the available time. The container, as well forwarding the tick: message to all of its children, sends a runFor: message to its resource manager. The resource manager then shares time on processing units among running tasks.

Storage units process their message queues when they receive the tick: message. All communication with storage units happens via the message passing interface, over the same interconnects as messages between tasks.

It is worth noting that increasing the tick time has only a minor effect on system throughput, and so a small interval is only required when measuring the latency of various components.

7.6.3 Output

As SimEAC runs, it displays summary usage for each component. This allows trouble spots — places where resources are distributed unevenly — to be identified relatively easily.

If more detailed analysis is required, it is possible to turn on logging for any hardware component or task. This produces a log file, with a usage entry for each tick. The log format is compatible with gnuplot, allowing graphs such as those shown in the next chapter to be generated directly by the system.

Additionally, any task or resource manager has access to the full C and OpenStep standard libraries and so can perform extra logging independent of the main system, if this is required. This mechanism is used in our first case study to produce logs of the structure of the overlay grid at various time intervals.

Chapter 8

SimEAC Design and Use

Contents

8.1 Introduction

This chapter describes the implementation of the SimEAC system, and gives an overview of how it can be used. The system is built in Objective-C, using the OpenStep frameworks, and familiarity with both the language and library are assumed.

SimEAC is split into three components, from a build perspective. The SimEAC framework contains classes that are used by the simulation engine and by externally developed resource manager and task classes; the public interface to SimEAC. The simulation engine is contained within the SimEAC application. Finally, a set of example resource managers and tasks are contained within a loadable bundle.

It is envisaged that users of the system will create additional bundles for their own components, and thus not need to recompile the system at any point. The example bundle should be used as an example for this. Additional bundles placed in the `Application Support/SimEAC/PlugIns` folder in any of the Library paths on the system will be detected and loaded. No additional glue code is required; all task and resource manager classes implemented within these bundles will be automatically exposed via the user interface and can be referenced by name from XML grid descriptions.

8.2 Requirements and Implementation

In the last chapter, the requirements for SimEAC were enumerated. This section, describes how they have been addressed and provides an overview of some of the specific features of the system.

8.2.1 Abstract Operation

The design of SimEAC is intended to allow simulations to be conducted at different levels of abstraction. At the highest level, a SimEAC simulation can be only slightly more concrete than a CSP specification; sequential processes communicating via an abstract message-passing mechanism.

Once a simulation at this level of abstraction has been shown to be functional, it is possible to progressively refine it until it closely mirrors the operation of a real system. The messages being sent, for example, can be split into packets. These packets might then take different routes through a grid, giving a more accurate simulation of throughput.

Similarly, it is possible to begin by running each task on a separate execution unit, then gradually refine the model so that each will be pre-empted by other (unrelated) activities for varying amounts of time, or will have to contend for other resources.

8.2.2 Easy Re-Configuration

Once the tasks have been created, they are assembled into a grid by the simulator. The recipe for constructing the grid is an XML file. These files can be created from within the simulator, or by an external tool.

XML was chosen because there already exist a large number of tools for manipulating it. When evaluating an algorithm in a wide variety of settings, it might be convenient to construct a set of building blocks (cluster nodes, workstations, etc.) in the SimEAC user interface, and then combine them into a wide variety of permutations using a custom program. XML allows this easily.

The schema (discussed in detail later) was designed to be human-readable and editable in order to facilitate the rapid creation of programs to manipulate it. The publication of the schema makes it possible for a schema validator to check the output of such a program without requiring the simulator to load it.

8.2.3 Support for Large Systems

There are no artificial limitations imposed on the simulator. The size of the grid which can be supported is limited only by the power of the machine running the simulation.

In cases where the grid is particularly large, it is common that only a fairly coarse-grained simulation is required. To this end, the granularity of the system has been made adjustable. Each tick represents a configurable number of nanoseconds. When the tick size is large, the speed of the simulation increases dramatically at the expense of some accuracy. It is likely that many uses of the system will involve simulating a large grid with a large tick size, and then simulating smaller sub-grids with a smaller tick-size.

The system has been designed in such a way that the trade between the resource requirements and the accuracy of the simulation can be tuned by the user.

8.2.4 Simple Development

Development of components for SimEAC is done according to the object oriented model. A set of base classes are provided which handle all of the book-keeping work required for interacting with the system transparently to the developer. By sub-classing one of these, it is possible to quickly implement the required functionality, without needing to implement anything more. For example, implementing a task can be as simple as subclassing SimpleTask and implementing one method for each kind of message the task can process.

Objective-C was chosen as an implementation language since it is simple to learn for anyone familiar with C and provides a dynamic runtime framework well suited to fast development. The first commercial Rapid Application Development tool, developed by NeXT, used Objective-C, and the strengths of the language that made it appropriate for this also apply in a simulation environment.

8.2.5 Extensible Design

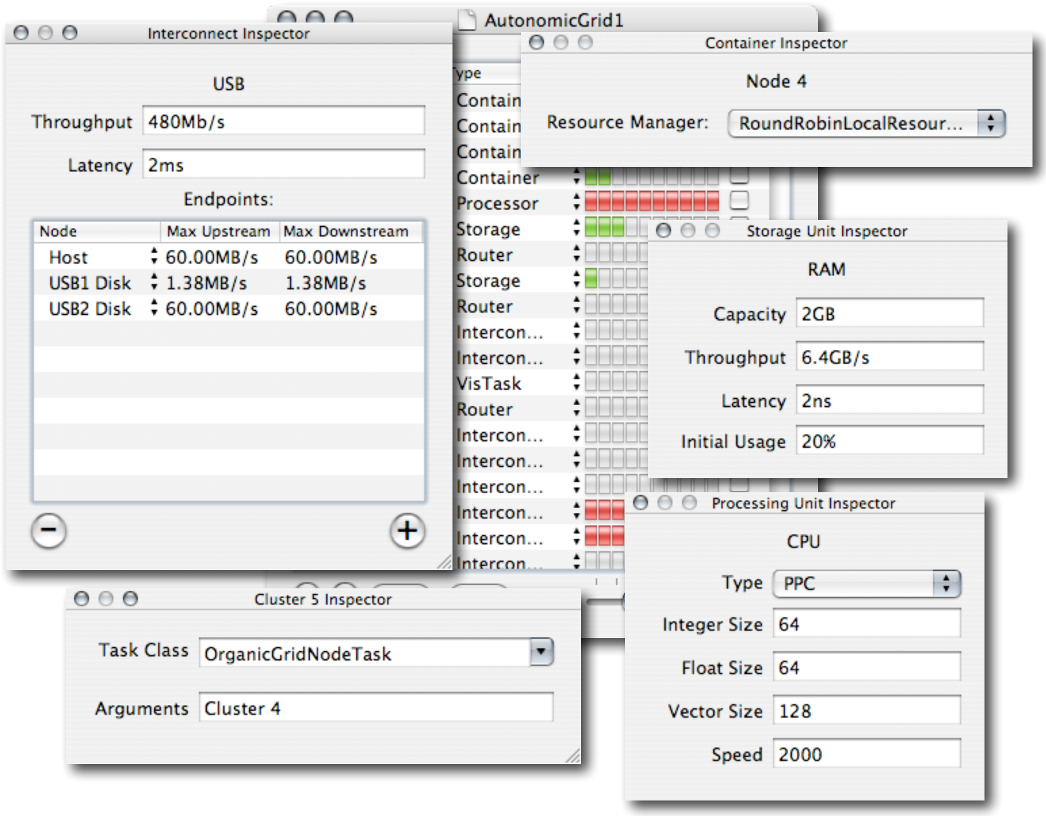
It is hoped that the core components of SimEAC will be sufficient for their task. It may be, however, that there at a future date some hardware that is not adequately representable will be developed. In this case, it is relatively easy to add components to the system simply by sub-classing the HardwareNode or InterconnectableNode class. This would, however, require a modification to the XML schema used to describe grids.

More likely is the need to add new resource managers or tasks to the system. Both of these are possible using the plug-in interface, and so can be done without having to re-compile the simulator.

8.3 Application Programming Interfaces

This section describes the interfaces exposed to developers using the system and how they are implemented.

8.3.1 Hardware Simulation


Interconnect Inspector

USB

Throughput: 480Mb/s

Latency: 2ms

Endpoints:

Node	Max Upstream	Max Downstream
Host	60.00MB/s	60.00MB/s
USB1 Disk	1.38MB/s	1.38MB/s
USB2 Disk	60.00MB/s	60.00MB/s

AutonomicGrid1

ype
Contain
Contain
Contain
Container
Processor
Storage
Router
Storage
Router
Intercon…
Intercon…
VisTask
Router
Intercon…
Intercon…
Intercon…
Intercon…
Intercon…
Intercon…

Container Inspector

Node 4

Resource Manager: RoundRobinLocalResour…

Storage Unit Inspector

RAM

Capacity: 2GB

Throughput: 6.4GB/s

Latency: 2ns

Initial Usage: 20%

Processing Unit Inspector

CPU

Type: PPC

Integer Size: 64

Float Size: 64

Vector Size: 128

Speed: 2000

Cluster 5 Inspector

Task Class: OrganicGridNodeTask

Arguments: Cluster 4

Figure 8.1: SimEAC user interface

The layout of a simulated infrastructure, which can range from a large heterogeneous network of computers to a single device, is defined in XML. In SimEAC, such a layout is referred to as a grid. XML is particularly well suited to this task since it naturally represents hierarchical structures. These XML grids can either be created manually or via the SimEAC user interface.

Listing 8.1 shows a snippet of XML defining a handheld computer. The majority of the listing outlines the hardware; a moderate speed ARM CPU and a small amount of RAM and Flash storage. The container node specifies a PDA-oriented resource manager. This resource manager has a simple round-robin scheduler, and does not provide virtual memory. The PDAResourceManager is one of a small number included in SimEAC.

At the bottom of the listing is a task node. This instructs the simulator to create the specified task running in this container. Any arguments specified in by the args attribute would be processed by the task on start-up.

Each of the XML stanzas in this short snippet corresponds to an object in the system and will cause it to be instantiated when the file is loaded.

Figure 8.2 shows the inheritance hierarchy of the hardware related classes. These classes all inherit from the HardwareNode class, which defines the basic functionality of all hard-

Listing 8.1: An XML description of a PDA

```xml
<container
     ResourceManager='PDAResourceManager'
     name='PDA'>
     <processor
          name='CPU'
          type='ARM'
          int='32'
          float='32'
          vector='0'
          speed='200' />
     <storage
          name='RAM'
          size='64MB'
          access='5ns'
          persistent='NO'
          rate='6.4GB/s' />
     <storage
          name='FLASH'
          size='64MB'
          access='5ms'
          persistent='NO'
          rate='40MB/s' />
     <interconnect
          throughput='6.4GB/s'
          latency='0ns'>
          <node>CPU</node>
          <node>RAM</node>
     </interconnect>
     <interconnect
          throughput='100MB/s'
          latency='4ms'>
          <node>CPU</node>
          <node>FLASH</node>
     </interconnect>
     <task
          class='RemoteDisplayTask'
          name='display'
          args='' />
</container>
```

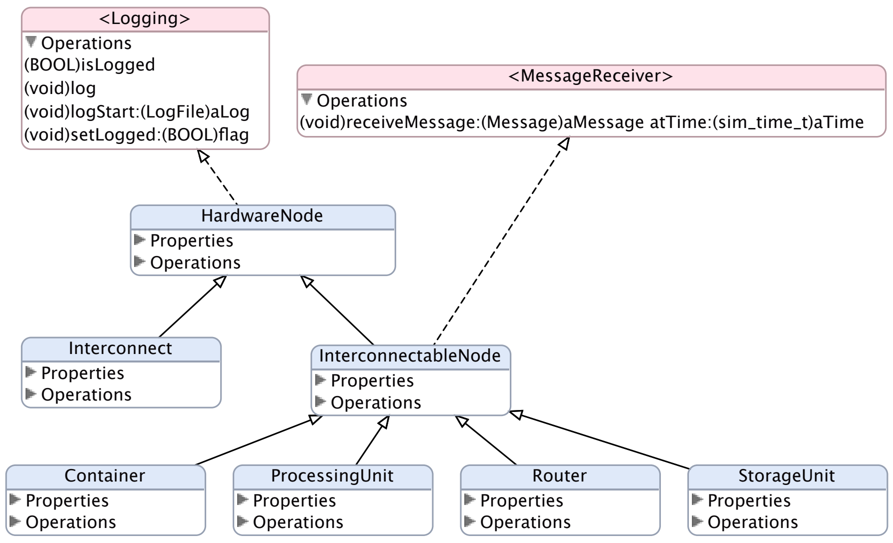


Figure 8.2: Abridged class hierarchy diagram for hardware nodes.

ware nodes. Classes are then specialised into nodes that can be connected together and interconnects. The full interfaces for these classes are given in Appendix B.

Only the leaf classes in this hierarchy should be instantiated. The various kinds of interconnectable nodes all implement the MessageReceiver protocol. This protocol is also implemented by all task classes, allowing the message delivery mechanism of the simulator to work without knowing the type of the receiver.

8.3.2 Software Simulation

Resource manager and tasks are implemented in Objective-C. Each one is implemented as a class conforming to a formal protocol. To aid implementation a number of classes have been provided which perform the basic functionality required for a number of common uses. These include:

- SimpleTask is a task which does nothing in the context of the simulation and is intended for subclassing. This class places received messages in a queue which can be easily accessed by subclasses. When the task is allocated run time, methods of the class whose names correspond to the enqueued messages are invoked. This provides a simple way of implementing event-driven tasks.
- LocalResourceManager handles the book-keeping tasks required for a local resource manager.

- DistributedResourceManager performs the equivalent functionality for a distributed resource manager.
- LocalResourceManagerWithSimplePaging includes a simple virtual memory implementation. Creating a functional local resource manager from this requires nothing more than implementing a process scheduling algorithm and (optionally) an I/O scheduler. A small number of concrete subclasses of this are included with various scheduling algorithms implemented.

Objective-C, being a dynamic language, allows these classes to be enumerated at run time and so no changes outside of the file in which a new task or resource manager is defined are necessary. Adding a new task to the system requires nothing more than writing and compiling the class - it can then be added to a grid description file either manually or via the SimEAC user interface (Figure 8.1). Since the resource managers and tasks are written in a well-supported existing language, a range of debugging tools are available for use during testing and calibration of simulated environments.

This process requires some calls to the underlying Objective-C runtime methods. The process for doing so is shown in Listing 8.2.

Listing 8.2: The process for enumerating all resource managers.

```
//Find out how many classes are registered with the runtime system
int classes = objc_getClassList(NULL, 0);
//Allocate enough space for the list
Class * classList = malloc(classes * sizeof(Class));
//Create an array of all classes
objc_getClassList(classList, classes);
//Find the ones we want
NSMutableArray * resourceManagers = [NSMutableArray
    arrayWithCapacity:classes];
for(int i=0 ; i<classes ; i++)
{
    Class class = classList[i];
    if(class_getInstanceMethod(class,@selector(conformsToProtocol
        :)) != NULL)
    {
        NS_DURING
            if([class conformsToProtocol:@protocol(ResourceManager
                )])
            {
                [resourceManagers addObject:[class className]];
            }
        NS_HANDLER
        NS_ENDHANDLER
    }
}
```

This process involves requesting a list of all loaded classes from the runtime, then iterating through them to determine which conform to the correct protocol. The names of those that do are cached in an array for efficiency reasons. There are typically several thousand classes known to the runtime library while a program is executing, and enumerating them

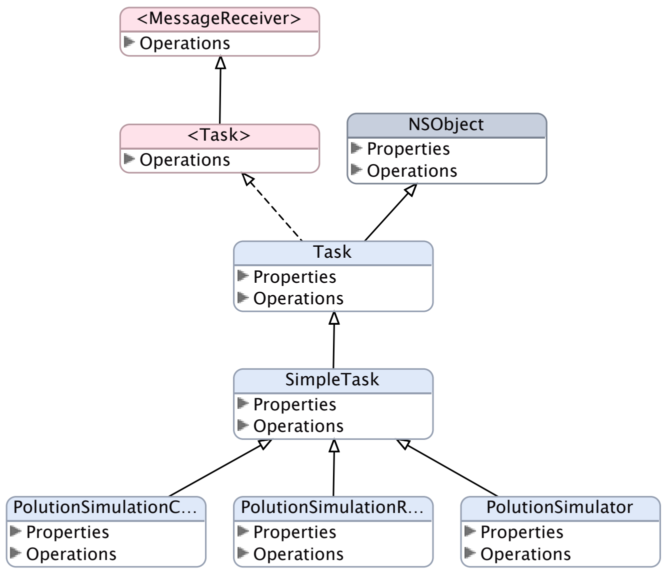


Figure 8.3: Class hierarchy for Pollution Simulator example tasks.

every time the list of loaded resource managers or tasks is required would be very time-consuming.

8.3.3 Tasks

Figure 8.3 shows the class hierarchy of the pollution simulator case study from the next chapter. All of the classes implemented for this example inherit from the SimpleTask class. This, in turn, inherits from the Task class.

The Task class implements the basic functionality for any task. Although any class that conform to the Task protocol can be used as a class, it is recommended that subclasses of the Task class be implemented, rather than duplicating existing functionality.

Listing 8.3: Two methods from the Task protocol

```
- (sim_time_t) runFor:(sim_time_t)nanoseconds
          onProcessors:(NSArray*)processors
                atTime:(sim_time_t)now;
- (BOOL) runsOn:(CPUType)anInstructionSet;
- (unsigned int) threads;
- (float) usage;
```

Listing 8.3 shows the most important methods that a task must implement. At the simplest level, a task is a consumer of CPU time. The first three of these methods all relate to this; the –threads method returns the degree of parallelism supported by the task. This is used by the underlying resource manager to decide how many processing units to assign to the task. The –runsOn: method serves a similar process; it is used to determine what architectures the task supports.

When a resource manager assigns run time to a task, it does so using the –runFor:onProcessors:atTime: method. This tells the task which processing units to run on, and for how long. The implementation of this method in the Task class does not consume any CPU time.

The Task class has fairly simple handling of received messages. It simply adds them to a queue, and allows subclasses to access them in a simple manner. The SimpleTask class makes use of these methods.

Listing 8.4: Message handling code from the SimpleTask class.

```
SEL messageSelector = NSSelectorFromString([aMessage->message->type
    stringByAppendingString:@":"]);
if([self respondsToSelector:messageSelector])
{
    sim_time_t timeTaken = ((msg64)objc_msgSend)(self, messageSelector
        , aMessage);
    if(timeTaken > 0)
    {
        timeTaken = MIN(timeTaken,spareRuntime);
        currentTime += timeTaken;
        spareRuntime -= timeTaken;
    }
}
```

Listing 8.4 shows part of the run loop from the SimpleTask class. The aMessage variable is a C structure containing the enqueued message. This is used to store the message in a queue, and contains the message and the time of arrival. The message itself is another C structure, containing an Objective-C string as the type, and some other attributes, such as the sender, size and a dictionary of other arbitrary values.

The type of the received message is used to generate a selector with the same name as the message, and one parameter. This the message type @"foo" will be translated to @selector (foo:). If the class responds to this selector, then it is invoked with the message as the argument. The return value is taken to be the amount of time spent executing. The called method can check that it doesn't exceed the amount of available run time by reading the spareRuntime instance variable. Several other instance variables of this nature are set by the run loop, including currentTime and currentProcessorSet. If the called method completes handling the message, it should remove it from the queue.

8.3.3.1 Adding Tasks

The complexity of adding new tasks scales with the complexity of the task being implemented. It is intended that a simple task should be easy to implement. This is important in the context of Autonomic Computing, since many autonomic systems are comprised of simple autonomous components. These components would each be implemented as a task within the framework of the simulation.

Listing 8.5: A simple task implementation

```
@interface ExampleTask : SimpleTask {
}
@end

@implementation ExampleTask
- (sim_time_t) Frame:(QueuedMessage*)aMessage
{
    //Work out time to decode a frame on the current processing unit
    sim_time_t decodeTime = 10000000
    decodeTime /= [[currentProcessorSet objectAtIndex:0] throughput];
    //Decode the frame
    sim_time_t timeTaken = MIN(decodeTime,spareRuntime);
    if(timeTaken == decodeTime)
    {
        [delegate sendMessage:nil
                     withType:@"DecodedFrame"
                        sized:640*480*4
                           to:[aMessage->message->body objectForKey:@"
                               next"
                       atTime:now + timeTaken];
        //Remove the current message from the message queue
        [self removeMessageAtIndex:0];
    }
    return timeTaken;
}
@end
```

Listing 8.5 shows the full code required for implementing a simple task. This task understands one message type, "Frame."[1] When it receives a message of this type, it consumes some processing time and then passes on the decoded frame to the next task.

When a task receives a message, the –receiveMessage:atTime: method is called. The Task base class provides a default implementation of this, which places the message in to a queue. This can be overridden by subclasses if some other behaviour is required. Alternatively, the queue can be directly inspected. Most tasks, however, simply process messages in sequence. Since Objective-C inherits the Smalltalk object model, that of simplified computers communicating by message passing, it fits this model well. SimEAC messages are mapped to Objective-C messages by the SimpleTask class, which inherits from Task. Implementors thus create one message handler (method) for each kind of message they expect to receive.

A subclass of SimpleTask may also implement a –handleUnrecognisedMessage: method. If it chooses to do so, then all messages not otherwise handled will be passed to this method, otherwise they will be ignored and a warning logged to the console.

The parameter passed to each of these methods is a QueuedMessage structure, which contains a Message structure and a timestamp indicating when the message arrived. The message itself contains a dictionary (associative array), which can contain any informa-

[1]The superclass uses the dynamic dispatch capabilities of the Objective-C language to invoke the subclass's Frame: method when it receives a message with the type set to "Frame."

tion set by the sender, as well as some standard information such as the message type, size and the sender.

The task in Listing 8.5 is a very simple example. It consumes CPU time, but does not interact with the system in any other way (such as by using memory or disk space). Listing 8.6 shows the interface through which a task interacts with the system.

Each of these methods is implemented by a proxy class which provides a default implementation. This proxy delegates much of the functionality described here to a resource manager, as long as the underlying resource manager implements it. Several features of the resource manager, such as the ability to define a routing algorithm, are optional. In these cases, the system will test for the implementation of a method in a resource manager class, and substitute a default if one is not provided.

Listing 8.6: A task's interface to the rest of the system

```
@protocol TaskDelegate<NSObject>
- (StorageUnit*) newFileNamed:(NSString*)fileName
                     WithSize:(double) bytes;
- (BOOL) resizeFile:(NSString*)fileName
                 on:(StorageUnit*)storage
                 to:(double) bytes;
- (BOOL) store:(double)bytes
        toFile:(NSString*)fileName
            on:(StorageUnit*)storage
        atTime:(sim_time_t)now;
- (BOOL) load:(double)bytes
     fromFile:(NSString*)fileName
           on:(StorageUnit*)storage
       atTime:(sim_time_t)now;
- (void) sendMessage:(NSDictionary*)aMessage
            withType:(NSString*)aType
               sized:(double)bytes
                  to:(id<MessageReceiver>)process
              atTime:(sim_time_t)now;
- (BOOL) setVMSize:(double) bytes;
- (double) vmSize;
- (double) inCoreMemorySize;
- (double) pageFault:(double) bytes;
- (BOOL) isComponentEnabled:(NSString*)aComponent;
- (void) setComponent:(NSString*)aComponent isEnabled:(BOOL)aFlag;
- (id) retainProcessNamed:(NSString*)name
                   ofType:(NSString*)className
                 withArgs:(NSString*)args
                   create:(BOOL)flag;
- (void) releaseProcess:(id)process;
- (void) terminate:(id<VisualisationTask>)process;
@end
```

8.3.4 Creating New Resource Managers

Implementing a custom resource manager means creating a class which implements the ResourceManager protocol. This is similar to the TaskDelegate protocol outlined in the previous section, since it has similar functions. The principle difference is that each method described by this protocol has an extra argument specifying the task to which it applies.

Listing 8.7: Methods from the ResourceManager protocol related to process migration

```
- (double) canRun:(id<VisualisationTask>)_newTask;
- (void) underscheduledProcesses:(NSArray**)processes
                         weighted:(NSArray**)weights;
- (void) addTask:(MetaTask*)task;
- (void) removeTask:(MetaTask*)task;
```

Listing 8.7 shows some of the additional methods that can be implemented, related to process migration. The first is called by its parent in the resource manager hierarchy to enquire whether it is able to accept a currently running task. The return value of this indicates how willing the resource manager is to accept it, and can be zero if it does not support process migration or if it is not responsible for any hardware on which the running task can execute.

The second method is the converse operation; it is used to retrieve a list of tasks that are not receiving enough runtime. These are processes that the resource manager would like to have migrated away from it. The weights argument should be used to return a pointer to a weight for each returned process.

A resource manager written be sub-classing one of the standard classes will have default implementations for these methods. The implementations of the last two methods add and remove tasks from an internal list and assign the correct proxy objects to them. This is the simplest way of creating a new resource manager, since the superclass will handle all of the book-keeping tasks and allow the developer to focus on the algorithms they wish to test.

In addition to its other rôles, the proxy class stores some metadata about the task with which they are associated, such as the current set of processors on which the task is running. This metadata can be accessed and used by the resource manager, making implementing things like processor affinity easier.

The interface that must be implemented by all resource managers is given in full in Appendix B. In addition to this, resource managers may also implement a pathFrom:to: method. This takes two interconnectable nodes within the grid, and should return an array of the nodes on the path. Support for this method will be determined at runtime. If it does exist, then it will be called by the system to determine how to route messages. This can be used to override the simple shortest path routing that is normally performed, for example to provide some load balancing between links.

8.3.5 Simulating Failure

One of the key components of an autonomic system is that it should be self-healing. In order for a system to heal, part of it must break. In general, any failure in a system can be categorised as either a hardware or a software failure. Simulating software failure is trivial; the simulated task simply terminates itself at a random (or specific, but undesired) point in time. Alternatively, a task can enter a failure mode, consuming all CPU time available to it, for example, or leaking memory, without doing any useful work.

In order to simulate hardware failures, SimEAC provides tasks with a pair of methods for determining whether a particular piece of hardware is enabled in the system and for enabling or disabling it. Any hardware component in the system can be disabled in this way.

To make life easier for users of the system, two special tasks are provided; DeterministicFailure and NonDeterministicFailure. The first of these allows the simulation of hardware failures at specific time intervals. It takes the name of a component and a list of times as arguments. At each time interval, it toggles the enabled state of a specified component.

The second class takes a name of a component and a probability of failure per second as arguments. It will randomly cause the named component to fail with the specified probability. In general, this task is more useful for simulating highly unreliable components. When using this task, it is a good idea to log it. Each of the failure tasks will log a value of 1 while the component they are responsible for is active and a value of 0 when it is not. This makes it easier, when examining the simulation results, to determine whether a particular behaviour was correlated with (and thus likely to be caused by) a component failure.

Note that this mechanism can also be used for simulating power management scenarios, for example the shutting down and restarting of nodes in a datacenter or cluster at times of reduced load. It can also be used for simulating dynamic networks; individual interconnects can be enabled and disabled at run time allowing nodes to leave and join networks dynamically.

8.3.6 Output

As SimEAC runs, it displays summary usage for each component. This allows trouble spots — places where resources are distributed unevenly — to be identified relatively easily.

This is done via the –usage method. Classes that require logging should implement this method, returning a float. The system will call it periodically, expecting a value between 0 and 1 indicating the load on a particular component.

The SimEAC user interface is updated periodically with load values for each component. Any component marked for logging will have the value returned at the end of every tick logged to a file, which can be used to generate graphs of system performance.

Since the −usage method can be called either as a result of logging or because the user interface requires updating, it should not modify the component's internal state. The implementation in the SimpleTask class simply returns the usage instance variable. The recommended way of using the logging facility from a custom task is to set this to a meaningful value when the class receives run time.

If more advanced logging is required, the OpenStep and C standard libraries are available. Mechanisms such as NSLog can be used to produce detailed output. In the Organic Grid case study, the structure of the overlay network was written to the console in this way.

The mechanism by which components report their output to the system is described in the previous chapter. Once the system has these results, it uses a C interface, described in Listing lst:logfile .

Listing 8.8: The log file interface

```
typedef struct _LogFile LogFile;
// SimEAC log file interaction

LogFile* newLog(NSString * fileName);
unsigned int addLogObjectWithName(LogFile * logFile, NSObject* object,
    NSString * Name);
void tickLog(LogFile * logFile, sim_time_t timeStamp);
void logValueForObject(LogFile * logFile, float value, unsigned int
    object);
void endLogging(LogFile * logFile);
```

This interface can be used to implement new logging back-ends, if required. The current log file format is compatible with gnuplot, and allows quick generation of graphs in a variety of formats.

The newLog() and endLogging() functions are used to open and close log files. The addLogObjectWithName() function is called by an object that wishes to have its values logged. In the default implementation, it reserves a column for the logged object and prints its name in a comment at the top of the log file. The logValueForObject() function sets an object's value for the current tick, while the tickLog() function begins a new tick.

8.4 SimEAC API Details

Listings 8.9 and 8.10 show the interfaces that must be implemented by Resource Managers and Tasks respectively. Classes providing implementations of these methods are included; implementing a new task or resource manager is usually a matter of subclassing one of these and overriding a subset of the listed methods to provide new functionality.

Listing 8.9: The interface for SimEAC resource managers

```
@protocol ResourceManager<NSObject>
- (double) canRun:(id<Task>)_newTask;
- (void) underscheduledProcesses:(NSArray**)processes
                        weighted:(NSArray**)weights;
- (void) addTask:(MetaTask*)task;
- (void) removeTask:(MetaTask*)task;
- (sim_time_t) processingUnitTimeUsedBy:(id<Task>)task;
- (BOOL) isDistributed;
+ (id) resourceManagerForContainer:(id)aContainer
                        withParent:(id<ResourceManager>)
                            aResourceManager;
- (id) initForContainer:(id)aContainer
             withParent:(id<ResourceManager>)aResourceManager;
- (BOOL) addChild:(id<ResourceManager>)aChild;
- (void) runFor:(sim_time_t)nanoseconds;

- (BOOL) setVMSizeForProcess:(MetaTask*)task to:(double)bytes;
- (void) pageFault:(double) bytes forTask:(MetaTask*)aTask;
- (StorageUnit*) newFileNamed:(NSString*)fileName
                     WithSize:(double) bytes;
- (BOOL) resizeFile:(NSString*)fileName
                 on:(StorageUnit*)storage
                 to:(double) bytes;
- (BOOL) store:(double)bytes
        toFile:(NSString*)fileName
      onDevice:storage
       forTask:(MetaTask*)task
        atTime:(sim_time_t)now;
- (BOOL) load:(double)bytes
     fromFile:(NSString*)fileName
     onDevice:storage
      forTask:(MetaTask*)task
       atTime:(sim_time_t)now;
- (void) log;
@end
```

These functions can be grouped into a few broad categories:

Process migration methods let the resource manager decide whether it should accept a running task for migration, and whether it wishes to have any of its own tasks migrated away. The isDistributed: method falls into this category, since it is used to determine if the resource manager should treat resource managers associated with child containers as children, and thus permit migration between them. If a resource manager returns NO to this method, then it must also return NO to any call to its addChild: method.

Storage management methods include writing to files and consuming memory. A simple model is used for the contents of storage devices; a task may create named files, which have a size and a location associated with them. Writing data simply causes some of the storage device's transfer bandwidth to be used up. A task can not create a file on a specific device (although if it has received a filename and a pointer to the storage device by some other mechanism, it may right to the file). It is the responsibility of the resource manager to decide which devices are used for persistent storage.

The top parts of the storage hierarchy contain main memory (and potentially caches, if they are being modeled). The task may set its virtual memory requirements using the –setVMSizeForProcess:to: method (indirectly, via its delegate). The resource manager must then attempt to allocate this much space in some storage units. If this changes the amount of in-core (processing-unit local) space allocated to the task, then the resource manager must call the task's –setInCoreMemorySize: method. The task may then issue page faults to the resource manager. When this occurs, the resource manager should not schedule the task again until the page fault has been handled.

Processing time allocation is handled by a single message; –runFor:, which tells the resource manager how many nanoseconds it should schedule tasks for.

Logging is handled by the –log method. This must pass a –log message to all tasks in a local resource manager. It may also use the logging interface to output its own usage in some way, if this is relevant.

Listing 8.10: The interface for SimEAC tasks

```
@protocol Task<NSObject, MessageReceiver>
- (id) initWithArguments:(NSString*)arguments Name:(NSString*)aName;
- (void) setDelegate:(id)aDelegate;
- (id) delegate;
- (sim_time_t) runFor:(sim_time_t)nanoseconds
          onProcessors:(NSArray*)processors
                atTime:(sim_time_t)now;
- (BOOL) runsOn:(CPUType)anInstructionSet;
- (unsigned int) threads;
- (void) retainTask;
- (void) releaseTask;
- (float) usage;
@end
```

Listing 8.10 shows the interface for tasks in the system. This is designed to be simple, since it is imagined that most uses for the system will require the development of several tasks. Most of the methods listed are handled by the Task class. The default implementation, in several cases makes use of an instance variable that should be set by a custom –init method, if the default is not acceptable. Table 8.1 shows these values. The last one is implemented in the SimpleTask class, although this is generally more useful to subclass than Task. The defaults for these are all supported general purpose instruction sets, one and zero respectively.

If a task needs to parse its arguments, the –initWithArguments:Name: method should be overridden. Tasks wishing to implement their own message queues should override the methods in the <MessageReveiver> protocol.

Most of the task implementation is likely to be within the –runFor:onProcessors:atTime: method. This is adequate for trivial tasks, however for more complex tasks it is recommended that the dynamic dispatch provided by SimpleTask be used, as described in the previous chapter.

Method	Instance Variable	Description
−runsOn:	instructionSets	Bitwise-OR’d set of instruction sets supported by the task
−threads:	threads	Maximum number of processing units the task can make use of.
−usage:	usage	Utilisation of the task, for logging purposes.

Table 8.1: Variables used by convenience methods in the Task class.

The XML schema used for describing grids is given in Appendix B. This can be used to validate an XML grid file for use in SimEAC. It is not possibly to capture all of the required semantics for such a grid in XML Schema, however. Restrictions such as the fact that a task or resource manager name must correspond to a class implementing the correct interfaces that has been loaded into the system will not be checked when validating a grid file against the schema.

Chapter 9

SimEAC Case Studies

Contents

One of the design goals of SimEAC was that it should be easy to use. The most common use of SimEAC will involve creating one or more new tasks and running them in a simulated grid environment.

This last chapter gave a brief overview of how new tasks and resource managers can be added to the system. This chapter will describe two scenarios in which SimEAC has been used.

Part of the purpose of the case studies was to demonstrate the usability of the system. Table 9.1 gives an overview of the parts that were tested by each study.

Both made use of the core engine, including all of the different types of hardware node, in a variety of configurations. Both extended the standard task classes and used the standard resource managers. The second case study introduced a new resource manager, although this has since been moved into the core system.

Compnent	Organic Grid	Autonomic Visualisation
Core Engine	√	√
Standard Tasks	√	√
Standard Resource Managers	√	√
Custom Tasks	√	√
Custom Resource Managers	×	√
Matching Published Results	√	×
Matching Experimental Results	×	√

Table 9.1: A summary of the components tested by each case study.

The first case study was an implementation of a published algorithm. Attempts were made to match the published results as closely as possible. It was not possible to be entirely sure that this was successful, since the published description of the test system was quite vague, grouping CPU and network speeds into a small number of categories. Within these bounds, however, the simulation was judged to be a success.

The second case study involved the use of a the simulator in conjunction with a prototype system. Results from the prototype were used to callibrate the simulation, then modifications were made to both in an attempt to judge the accuracy of the simulation. In the evaluated cases, it was shown that the simulation was able to correctly model a real system.

9.1 Case Study: The “Organic Grid”

Our first case study demonstrates the implementation of an existing algorithm within the context of the simulator, and shows some results from this process.

The “Organic Grid” [56] describes an organizational system for distributed computation. The principle behind the design is an autonomic algorithm that allows a hierarchy of nodes to evolve as the system runs, allowing the easy distribution of a parallel computation over a large number of nodes in an ad-hoc network.

The nodes are configured into an *overlay network*, a logical hierarchy used for task distribution. This is an abstract layer on top of the physical hierarchy. The “Organic Grid” automatically promotes fast nodes to near the top of this tree, and demotes slow nodes towards the leaves.

We implemented a version of the described algorithm and evaluated its performance in a number of different initial configurations, two of which are described in the results section.

For this simulation a single additional task was written. The task took the name of another task as an argument, and used this as its initial parent in the overlay network. Work units are sent down the overlay network from the root node and distributed by branch nodes. Both branch and leaf nodes perform the computations.

Each node in the overlay network was a single instance of the newly created task class, assembled into a tree structure on initialization. On launch, each node acquires a handle to the parent node in the tree. This is done via the task delegate, which implements a basic name server for the simulator, allowing connections to be bootstrapped. It then sends an initial join message and a work request message to the parent.

When the parent node receives a work request message, it forwards it up the tree until it receives a block of data. This is then passed down to the original requesting node, which consumes CPU time until it is finished. The amount of CPU time to consume is determined by dividing a constant by the speed of the processing units available to the task (supplied by the simulator in an argument to the run loop method, as described earlier).

The speed of a node was determined by the time taken for a job to be processed added to the time taken to transmit a single work unit from a parent to its child. Each node logs the time it sends a work unit to a child and the time at which it receives the processed reply. This means that the speed of a node is dependent on its position in the overlay network; moving a node to become a child of a node with a slow Internet connection will make it slower.

This task used the native message passing facilities of the simulator. Each received message was implemented as a separate method and called automatically by the code in the superclass.

Since only a single task, the "Organic Grid" task, was running on each hardware node, a simple resource manager was all that was required. We chose a simple round-robin scheduler without preemption, since this placed a light computational load on the simulator without compromising accuracy.

9.1.1 Configurations

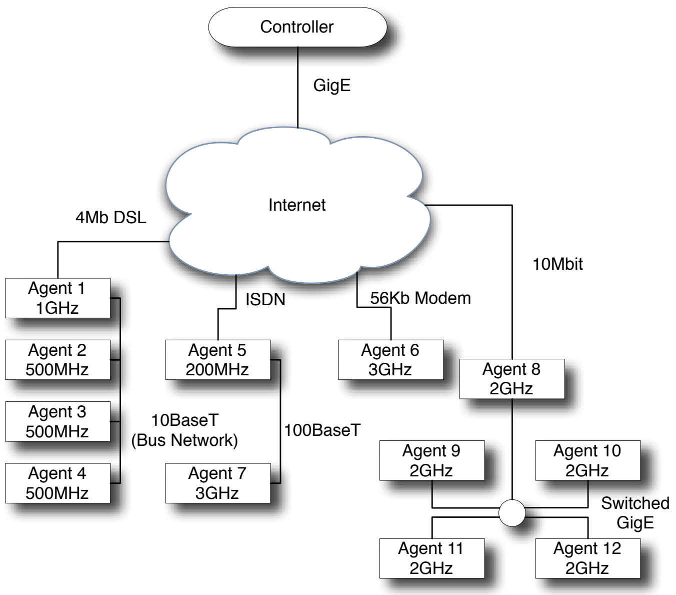


Figure 9.1: Physical topology of the first "Organic Grid" test

The results of testing the algorithm in two configurations will be presented in the next section. The first configuration is a simple WAN configuration, while the second is a more unusual topology.

The first physical configuration we will describe is shown in Figure 9.1. This is a fairly standard arrangement for a relatively small grid. Two small clusters are connected via relatively fast Internet connections, and two individual nodes are connected to comparatively slow links. The controller is connected via a very high-speed link — all of the data must flow from here originally, and so we can not effectively evaluate the algorithm if this causes a bottleneck.

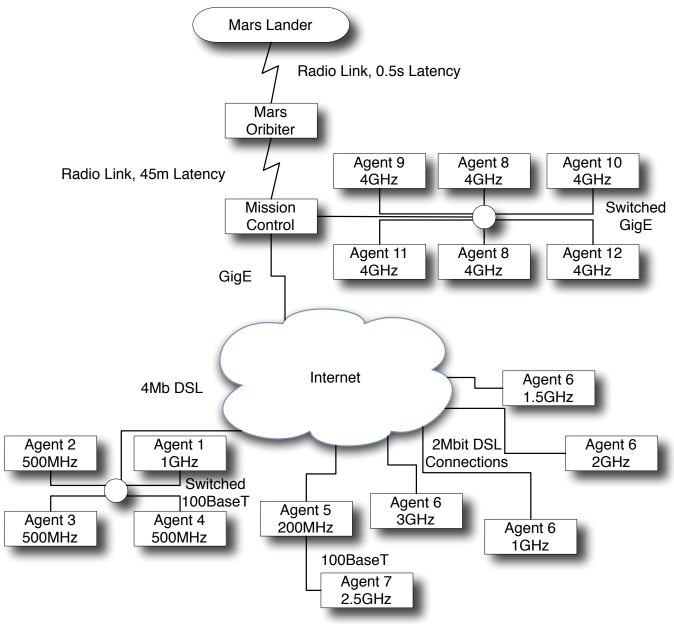


Figure 9.2: Physical topology of the second "Organic Grid" test

The second example we will demonstrate used a somewhat more unusual configuration. This example involves a hypothetical Autonomic Martian Explorer. Such a system would be required to take samples of the Martian surface and make decisions about the direction to explore based on the results. Since the analysis of these samples would require a considerable amount of processing power, the lander would be unlikely to be able to do this itself.

The main feature of interest in this configuration is the very high latency between the Mars Orbiter and the Terran resources. It is assumed that the orbiter will have some computational resources, and that mission control will have significantly more (including a cluster of high-speed machines). In addition, machines connected to the Internet are able to donate run-time to the program. The physical topology of this hardware grid is shown in Figure 9.2.

9.1.2 Results

Figs. 9.3 and 9.4 show the initial and final configurations of the networks. It is worth noting that the simulation task itself logs the tree structures in a form that can be rendered by a LaTeXpackage, allowing easy visualization of the results. The overlay networks shown in this section were created by pasting the simulation output into the paper source. The code for generating this output was periodically invoked by the standard logging mechanism.

Agent 7 remains a leaf node in this configuration, in spite of the fact that it is running on

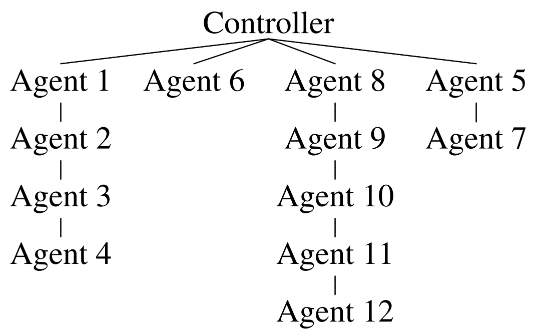


Figure 9.3: Initial overlay network for the first “Organic Grid” test.

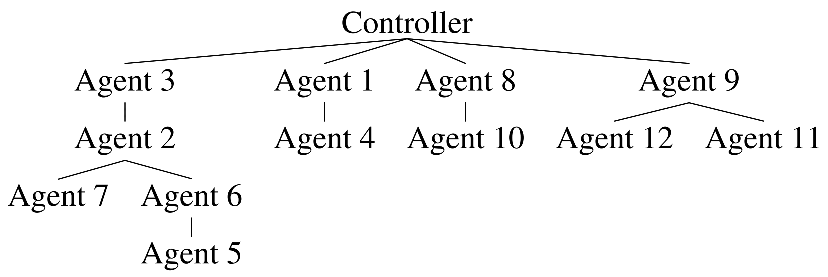


Figure 9.4: Stable overlay network for the first “Organic Grid” test.

one of the fastest machine in the network. This is due to the fact that it has a comparatively slow Internet connection, which causes a bottleneck.

The initial and final configurations of the overlay network for this configuration are shown in Figs. 9.5 and 9.6. As can be seen from this example, the algorithm scales well to systems containing very high latencies. The smallest latency link in this configuration is a few milliseconds, while the largest is almost an hour; a difference of six orders of magnitude between the smallest and largest latency. In spite of this, the algorithm gives a sensible configuration.

Note in particular how the arrangement of the terrestrial nodes is heavily dependent on their CPU speed, since this becomes a limiting factor once the interplanetary bottleneck has been surpassed. This can be seen from the overlay network layout in the stable configuration, where the slowest resources are grouped below Agent 7 while the fast cluster nodes on the high-speed network have migrated up the tree.

This test highlights the advantage of using a simulator to test autonomic systems. Launching a Mars mission simply to provide test data for an autonomic algorithm would most certainly not be economically feasible. Our system allowed simulated periods of several days to be run in a few minutes. The results could then be used to refine the algorithm before real-world deployment. Using the simulator as a refinement tool is covered more in Section 9.2.

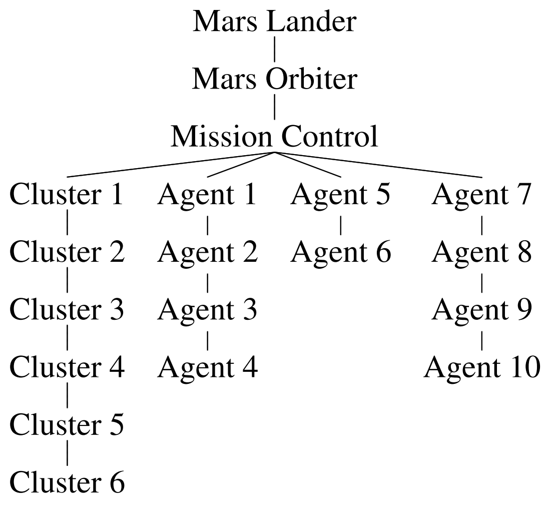


Figure 9.5: Initial overlay network for the second "Organic Grid" test.

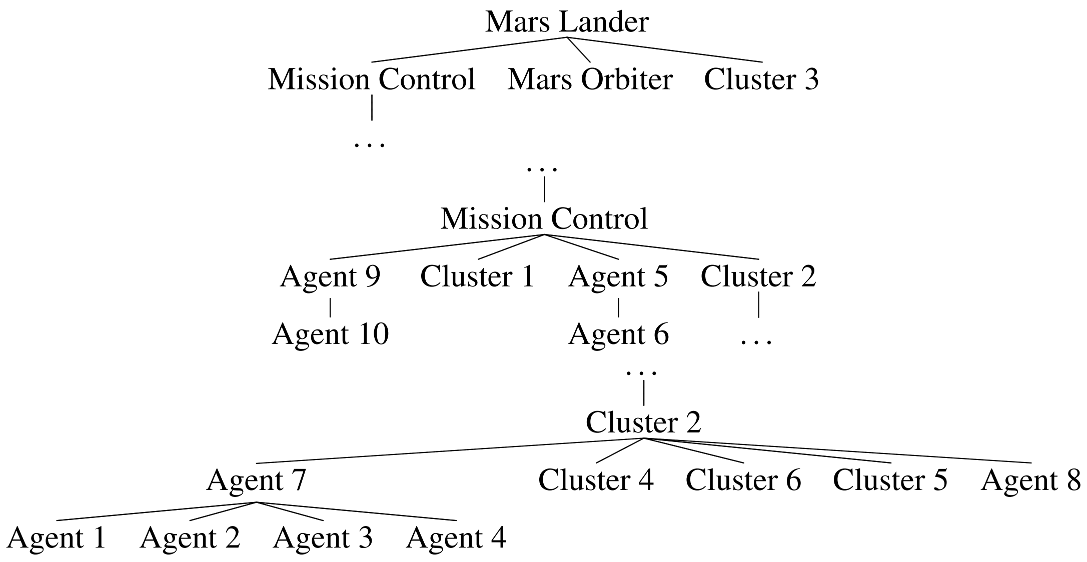


Figure 9.6: Stable overlay network for the second "Organic Grid" test.

9.2 Case Study: An Autonomic Visualization System

Our second case study was an agent-driven distributed visualization system. Autonomic computing has an important role in managing large-scale visualization systems [41]. Work on this simulation was conducted as part of the e-Viz project, which aims to develop a fully autonomic visualization environment. The e-Viz system [261, 263] is designed to be self-healing, self-optimizing, and self-configuring system for scientific visualization. SimEAC was used to simulate several components of the system. This section will describe two of them.

The first component of the system was designed to transparently handle the creation of a visualization pipeline and allow a user to navigate a volume dataset. The system should automatically optimize itself by distributing the rendering tasks to the nodes with the highest performance. It should also be self-healing—if a renderer failed then the load should be shifted elsewhere. SimEAC was initially calibrated against a prototype of the system running on a small number of machines and then run on a larger, simulated, scale.

The second component of the system simulated was a tightly-coupled scientific computation and visualization application. This application consists of three discrete components. The computation application feeds a 3D data structure to the renderer component, which feeds rendered frames to the client. Each component was implemented as a visualization task in SimEAC. The objective of the simulation was to determine how the system would function in different network configurations.

9.2.1 Simulated Tasks

A small number of custom tasks were written for these simulations, three for each. In the first simulation, one task represented a user of the system. This task would send messages to a controller requesting an image for a given viewpoint. The controller would, on receipt of this message, attempt to start renderers to fulfill the request. If renderers had already been created on all available cluster nodes then the requests were dispatched to the existing ones. Each cluster node would run an agent which could create a renderer, and the renderer itself. The agent was responsible for automatically restarting the renderer itself.

The three tasks in the second simulation represented the three components of the system. The computation task fed data to the renderer task, which fed frames to the client task. The latter two tasks also provided feedback messages telling the earlier stages in the pipeline when they had completed processing (either rendering or displaying) a step.

9.2.2 Resource Managers

No new resource managers needed to be written for this simulation. The existing *GPU Resource Manager* would, if given a task which could run on a GPU or a general purpose CPU, attempt to run it on the GPU. This allowed us to write a single task representing both the hardware and software renderers.

The same render task could be started on all cluster nodes, and would run on the GPU on those in which a GPU was present. This is a fairly reasonable choice to make in the general case, since a GPU processing path is usually only provided for algorithms that are well suited to running on large vector processors and will therefore run faster on the GPU than any CPU of equivalent age.

9.2.3 Hardware

We simulated a small rendering cluster of thirty two nodes, eight of which were equipped with GPUs. The cluster was internally connected with a switched gigabit Ethernet, and connected to the outside world via a 100Mbit/s uplink. This configuration was selected as a reasonable size for a departmental cluster.

Outside the cluster was a small network of six machines running the client task. Each client was connected by a 100Mbit/s network connection to a switch, which was connected to the visualization cluster.

For the second simulation, a variety of different network conditions between the three components were evaluated, modifying both the speed and latency of the link. The initial configuration, present in our lab setting (100Mbit networks between each component), was used as a base-line for calibration. Different combinations of fast and slow links between the pollution modeler, renderer, and client were tried.

9.2.4 Results

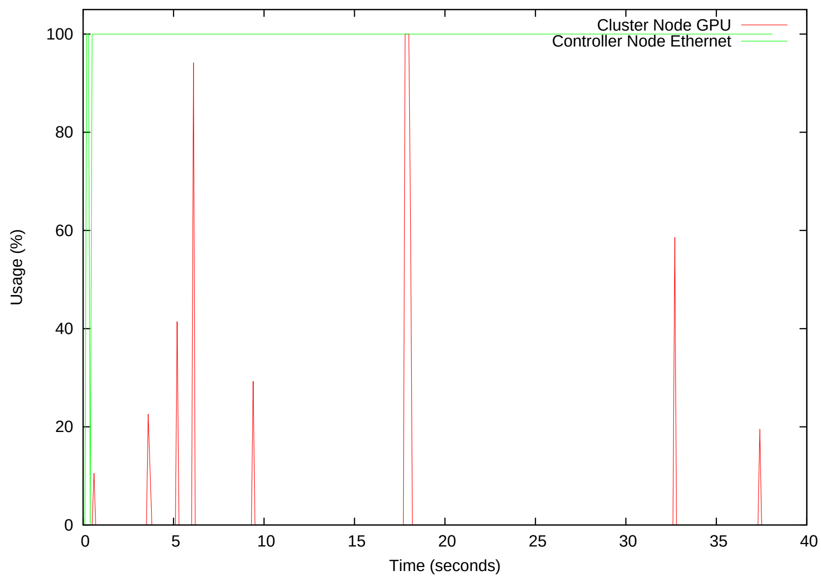


Figure 9.7: Network and GPU usage for the first test run.

Figure 9.7 shows an obvious bottleneck in our original configuration for the first simulation. All rendered frames are assembled by the controller node, and this completely saturates the node's network connection. As the graph shows, the GPU usage on the nodes spikes periodically as a frame is rendered, but this does not happen very often. The reason for this is that the messages requesting new frames are being delayed by the rendered frames being relayed through the controller node. This problem had not been exhibited by our small-scale real-world tests, since we had only a single client and fewer renderers.

A solution to this is to instruct each renderer to send the resulting image[1] directly to

[1]Note that images in this context are simply messages with their type set to the string "image".

the client to assemble. Doing this moved the bottleneck to the external network. Our next strategy was to compress the images and ‘upgrade’ the cluster’s external Ethernet connection to gigabit Ethernet. We determined that a compression ratio of approximately 80:1 was required to prevent network congestion is causing dropped frames with our system. A fully autonomic system should automatically increase the compression ratio as congestion increases. This was noted for inclusion into our final system.

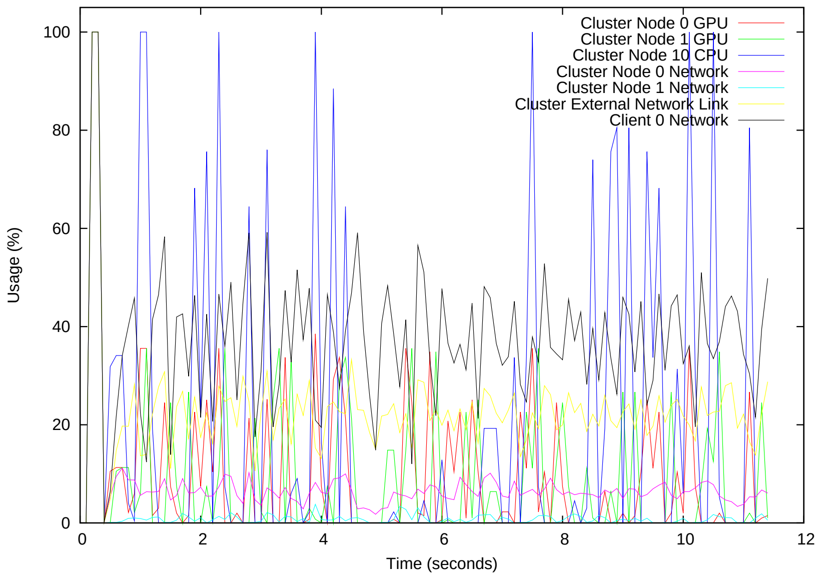


Figure 9.8: Selected component usage for the final test run.

Figure 9.8 shows the results after the described modifications. It can be seen from these results that no single component causing a bottleneck. In the first plot, the controller node Ethernet connection is saturated at 100% for the entire run while the cluster node GPU is idle for most of the time. In the second plot, no single component is at full utilization for more than an instant. If any of the lines on this plot were the 100% mark, this would represent a bottleneck. As such, we can observe that no single component is providing a bottleneck and the controller is distributing the rendering work amongst the cluster nodes in such a way that none is overloaded.

Note that not all components are plotted. During the simulation run, the summary display in the simulator was observed to decide which to display in more detail. During the first run the level indicator next to the controller node Ethernet was displaying full usage (and colored red) indicating that it was the trouble spot, while all of the other components showed little or no use. The cluster node GPU was selected as an example of another component which we would expect to be doing some work. In the second test, no specific components showed a bottleneck and so a representative sample were chosen.

For the second simulation, we began by conducting a primary simulation running in near-optimal conditions. Figure 9.9 shows some results collected from this simulation. Each component of the system was run on a different (simulated) machine, connected with a

100Mbit/s network connection. It can be seen from this graph that the application is able to meet the target frame rate without saturating either network; something we already knew experimentally.

This test was used to calibrate the simulation model. Instrumentation was added to the application to determine the CPU load, simulation and rendering times, and network usage in this configuration. Once these values had been collected, they were applied to the simulation, and validated by modifying other values such as the granularity of the simulation.

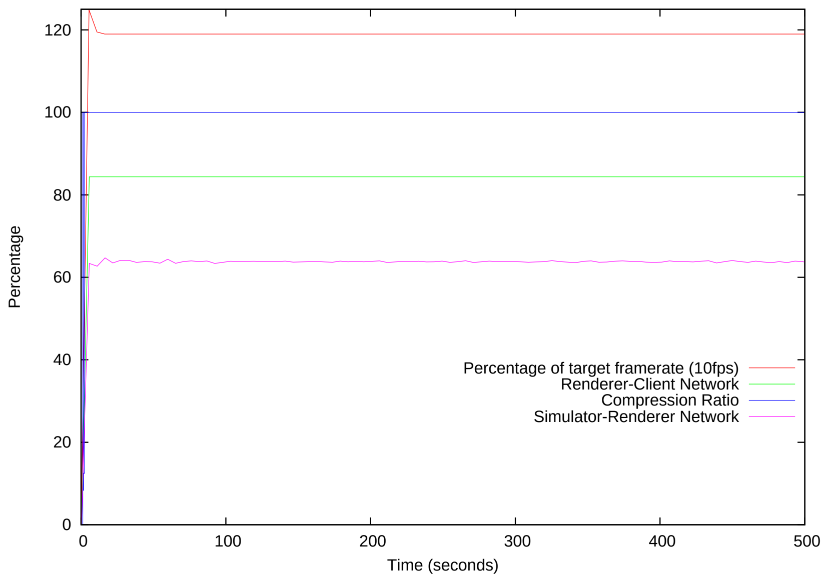


Figure 9.9: Calibration run for the simulated scientific visualization system.

Based on the calibrated simulation model, we evaluated the performance of the system working under different conditions by modifying various parts of the specification, resulting in a performance profile for the distributed system. Figure 9.10 shows the result of one such simulation. In this example, the renderer and the client are separated by a comparatively low-bandwidth Internet link.

In this case, there is not enough bandwidth available to deliver the required frame rate. To overcome this problem, we added a new autonomic feature to the simulated model. The simulated renderer task detects a long delay between sending a frame and receiving an acknowledgment, and increases the amount of compression applied (indicated by the blue line on the graph). This autonomic feature is adaptive, and the compression ratio stabilizes relatively quickly to a value which allows both the desired frame rate and an effectively utilized network.

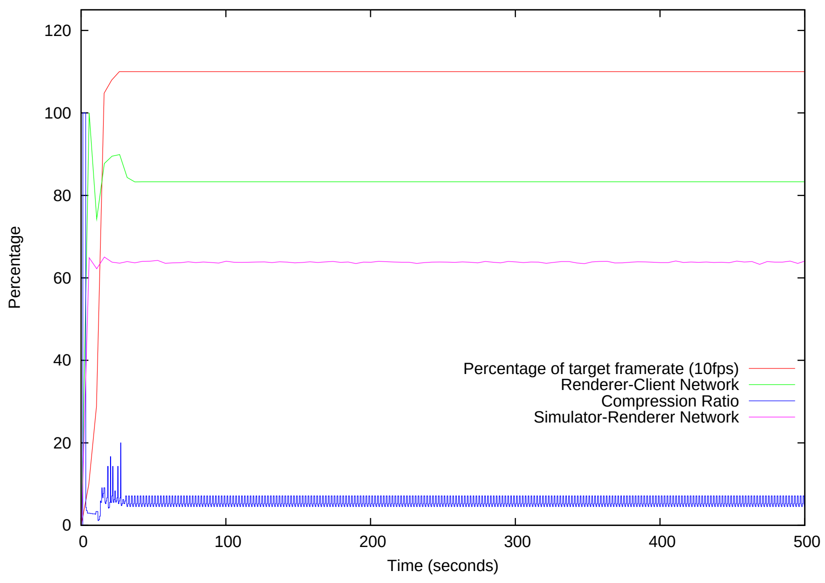


Figure 9.10: Simulated scientific visualization system with an Internet link.

9.3 Summary and Conclusions

SimEAC is a versatile platform for simulating autonomic systems. It can be used to prototype and evaluate algorithms and designs for autonomic systems without the expense, in terms of both money and time, of developing a fully working system on real hardware.

The implemented technical features of SimEAC have addressed most of the requirements outlined in Section 3. In particular, the provision of multi-granularity specifications for simulation is a novel design feature of SimEAC and is critical for cost-effective modeling of autonomic components and their operational environments. The provision of a variety of specification methods (e.g., XML, programmatic, and user interface) and many built-features (e.g., reconfiguration, failures and viruses) also makes the simulation specifically suited for autonomic computing.

Through our case studies, we have shown that SimEAC is capable of simulating widely different systems within the same framework. The ease of use of our system is such that simulations of comparable or greater complexity than those demonstrated in the case studies could be constructed with very little effort.

The most important lesson that we have learned through the development of SimEAC is that one needs to constantly review the design decisions for a software system to address an emerging and changing topic such as Autonomic Computing. The system has been revised for many times since the commencing of its development in 2004 inspired by [164]. The case studies, the publication of [96] and the reviews of the original submission all made important contributions to the enhancement of SimEAC.

While the core simulator is stable, the current SimEAC is only the first chapter of its development. We have plans to add many important features in part of the future work. We would like to build a large collection of reusable tasks and resource managers in the form of domain-specific libraries (e.g., agents, networking, Grid, visualization). A considerable amount of work has been performed in recent years on scheduling techniques for large and heterogeneous systems. We would like to introduce a more sophisticated mechanism for programming tasks and resource managers, and integrating such programs through parameter passing. We would like to investigate into a more flexible mechanism for reconfiguration allowing the autonomic components to create arbitrary new configuration. We would like to introduce some basic optimization functions for easy calibration with real world data, and for supporting users' needs for self-optimization.

Part IV

Closing Remarks

Chapter 10

Conclusions

Contents

Building a fully autonomic system for visualisation in a distributed environment is a challenging proposition. Creating a fully autonomic infrastructure for visualisation is likely to take many years, if not decades, for large numbers of people. This thesis has begun the process, and it is hoped that others will continue to work towards this goal.

At the simplest level, it has been shown that an autonomic approach to problems can yield useful results. Part II discussed a specific visualisation problem, namely that of rendering point data. This application was chosen as a typical example of a problem for out-of-core rendering, since it had some fairly complex data access patterns.

It was shown that this rendering strategy benefited from a custom data management strategy for point sets whose octrees did not fit into memory, rather than relying on the operating system's hardware. An algorithm-specific strategy was developed, giving better performance, and this was then evaluated in comparison with two adaptive strategies. Aim 1[1], to create a prefetching strategy for this application, was met, and it was shown in Chapter 5 that this strategy provided a significant performance and scalability improvement over delegating the management of external memory to the operating system.

The two adaptive strategies were intended to exemplify the *self-optimising* part of autonomic computing. They both worked by acquiring data about the access patterns of a run, and then use this captured knowledge to make predictions. Aim 2 was to determine if such a strategy could achieve results commensurate with the algorithm-specific strategy. This was the first attempt to apply knowledge-based strategies to out-of-core prefetching, and showed the potential of autonomic data management strategies, as well as providing algorithms that can be used now. The knowledge-based approach was evaluated and found to give equivalent or better performance than an algorithm-based approach with a smaller investment in development time. The results of this quantitative testing were pre-

[1]The aims were enumerated in Section 1.3.

sented in Chapter 6, indicating that a knowledge-based approach is feasible and provides benefits over a more traditional approach.

Some preliminary work was presented in Chapter 4 on extending this concept to the distributed context, and further work in this area is expected to yield beneficial results.

A new framework for implementing out-of-core algorithms was presented. The framework allowed the evaluation of out-of-core algorithms in a fair setting. While the clean isolation between the layers meant that the framework did not give the best possible performance, it could still be used in real-world applications.

The results of testing within the framework showed that knowledge-based approaches could provide equivalent or superior performance to those relying on a priori information. This confirmed the hypothesis stated earlier. Future work will include expanding this to the multi-user context, where there is a much richer source of potential knowledge that can be acquired. Each user of a particular dataset can have their access patterns recorded, and used to infer patterns for others.

In addition to the framework for out-of-core applications, a remote data server and client interface was discussed. Both the local framework and client-server implementation act as examples of the proposed five-layer model for out-of-core applications (although the local implementation only includes a subset of the layers, since it does not support networked access).

The five-layer model for out-of-core applications is, itself, another contribution of this thesis. The five layers provide the first systematic model for designing out-of-core systems, in accordance with Aim 3. This promotes code-reuse, since the boundaries between the layers are well-defined and different implementations of a given layer for a different application can be used. This was demonstrated in the case of the bottom (block) layer with the sample implementation, where a number of different (interchangeable) versions were provided. External memory systems to date have been developed in an ad hoc manner, designed to solve a specific problem. The new model proposed in this thesis will permit the re-use of concepts and algorithms from one implementation in another by providing a clear layering that allows better reasoning about out-of-core systems.

Aim 4 was to demonstrate the feasibility of this model by providing an implementation of it. This was done, and the results in Chapters 5 and 6 were collected by using this implementation. The improvement in performance given over a pure in-core approach showed that this implementation, while intended purely for experimental purposes, achieved good performance in real data management tasks.

Data management is an important problem, but is only a small part of building an autonomic infrastructure for visualisation. In order to facilitate the development of these ideas further, the first simulation environment aimed at the rapid evaluation of autonomic algorithms was developed. This environment, SimEAC, is based on a hybrid discrete-time and discrete-event model designed for the needs of autonomic system simulation, and fulfills Aim 5.

The system makes it easy to model new tasks and resource managers in a variety of situations. Some case studies were presented in Chapter 9, highlighting the flexibility of the system. In addition to demonstrating the use of the system, and providing some results that were used in the development of the e-Viz system, these studies contributed to the evaluation of the system (Aim 6), and demonstrated that it met the stated design goals.

In summary:

1. An out-of-core prefetching algorithm tailored to point-based volume ray tracing was designed, and shown to faster and more scalable than relying on the operating system. (Aim 1)
2. Two general knowledge-based out-of-core prefetching algorithms were created. (Aim 2)
3. The three prefetching algorithms were evaluated in comparison with each other and with a pure demand paging approach. The results showed that all three algorithms performed better than demand paging. (Aims 1 and 2)
4. A five-layer model for out-of-core systems was proposed. This is the first proposed model for systematic design and classification of external memory systems. (Aim 3)
5. The five-layer model was shown to be feasible with two prototype implementations. (Aim 4)
6. It was shown that a knowledge-based approach can achieve equivalent performance to a tailored approach. These results indicate a significant amount of potential in knowledge-based algorithms for prefetching. (Aim 2)
7. The first simulation architecture tailored to the demands of autonomic computing was designed. (Aim 5)
8. The proposed simulation environment was developed. (Aim 6)
9. The simulation environment was evaluated in a variety of uses and shown to be sufficiently accurate and flexible to meet the requirements. (Aim 6)

10.1 Future Work

The field of autonomic computing in general is still very young, and very little work has been done at all in the context of visualisation applications. There is a great deal of scope for further work in this area.

Large scale remote visualisations are particularly important in the context of virtual worlds. The out-of-core techniques described in this thesis could be extended to the distributed context, and work has begun in this area. The five layer model includes support for remote access, and a prototype client-server implementation has been developed. There is scope for future work adapting this to existing problem domains.

Virtual worlds are an obvious example, since they already have the requirement that the rendering be performed on the client and the world stored on the server. To achieve interactive framerates, pre-fetching of data is required. A number of heuristics are applied to these in existing systems to perform this prefetching, but a self-optimising approach could benefit the development of such systems considerably.

On the simulation side, there is always room for further work. Modern operating systems use some quite complicated strategies for scheduling (particularly in multiprocessor contexts), and memory management. The simulator could be improved by detailed re-implementation of some more of these algorithms in resource managers, allowing different operating systems to be simulated. This is less important in the context of autonomic computing in the setting of ubiquitous computing, since micro-scheduling concerns are much less important than macro-scheduling, but it would allow the simulator to be used in some other contexts.

Beyond this, the simulator should be used to design additional components of an autonomic visualisation infrastructure.

Part V

Appendices

Appendix A

The SimEAC User Interface

Contents

SimEAC is designed to be able to take advantage of existing development tools. Users can use their favourite text editor or IDE to write tasks and their favourite debugger to inspect their operation in detail. The SimEAC application is designed to run either in a debugger, for developing new tasks, or outside for performing simulations.

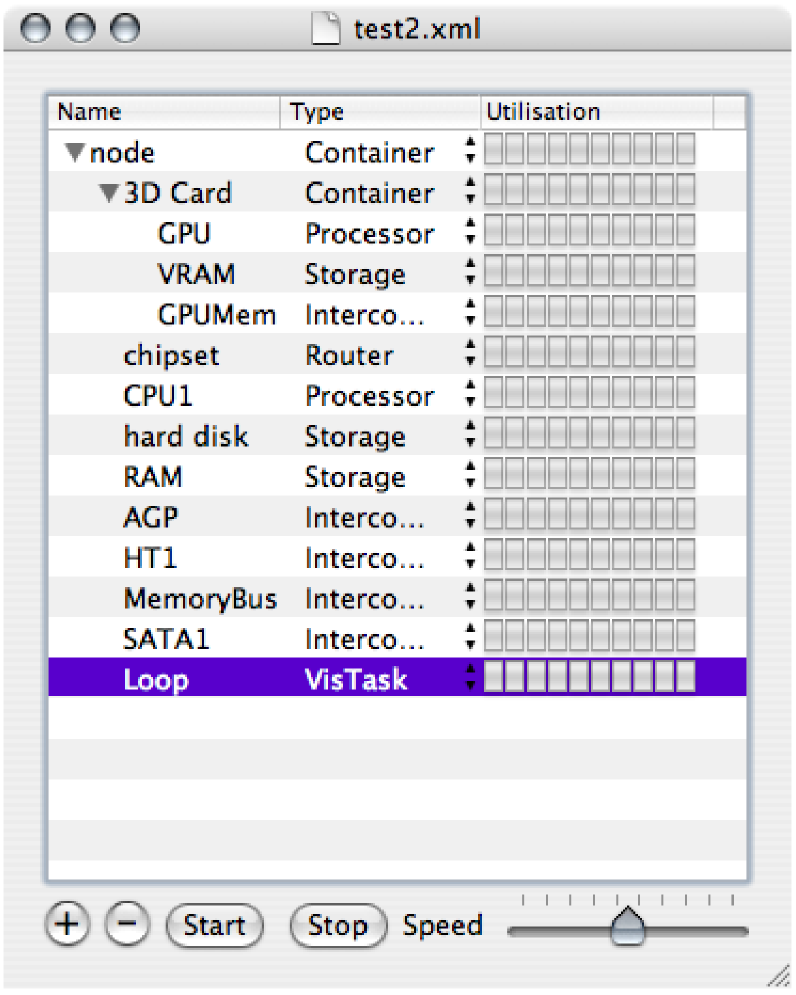
A.1 Loading Bundles

While the core system comes with a number of tasks and resource managers, it is anticipated that any use of the system will involve writing more. In order to implement new tasks and resource managers, a user of the system will need access to the header files and object code used to define the interfaces. These are in the SimEAC framework, which is linked to by the application and should also be linked to by any bundles containing user code.

Bundles placed in the `Application Support/SimEAC/PlugIns` sub-folder of the any of the `Library` folders on the system will be automatically loaded when the system runs.

As mentioned previously, no glue code is required. The classes can be accessed by name once they have been loaded from the user interface, and will be used to populate various components in the system interface.

A.2 User Interface Overview


test2.xml

Name	Type	Utilisation
▼node	Container	
▼3D Card	Container	
GPU	Processor	
VRAM	Storage	
GPUMem	Interco...	
chipset	Router	
CPU1	Processor	
hard disk	Storage	
RAM	Storage	
AGP	Interco...	
HT1	Interco...	
MemoryBus	Interco...	
SATA1	Interco...	
Loop	VisTask	

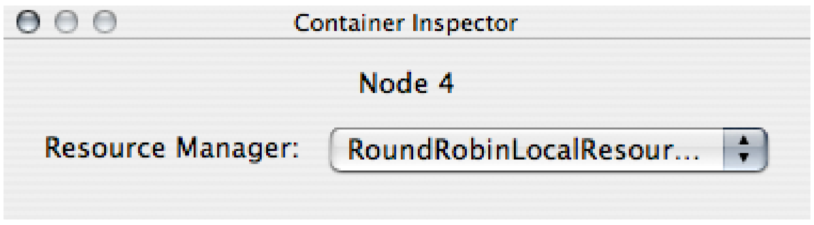
Figure A.1: SimEAC grid construction view.

Figure A.1 shows a grid being constructed. The main view shows the hierarchy of components. Each container in the system can hae children. Pressing the button with a + icon will add a new component within the currently selected container (if the currently selected object is not a container, the new component will be created in the parent container). Components can, similarly, be removed by selecting them and pressing the − button.

The start and stop buttons, and the speed control are used to control the simulation. The speed control alters the size of each tick, which controls the granularity of the simulation. Moving it to the right will make the simulation run faster at the expense of some accuracy.

Each component in the system can be further configured via its inspector. The inspector can be accessed by either selecting 'Get Info' from the menu, or hitting meta-I. The inspectors for the various components will be discussed in the remainder of this section.

A.2.1 Containers


Container Inspector

Node 4

Resource Manager: RoundRobinLocalResour...

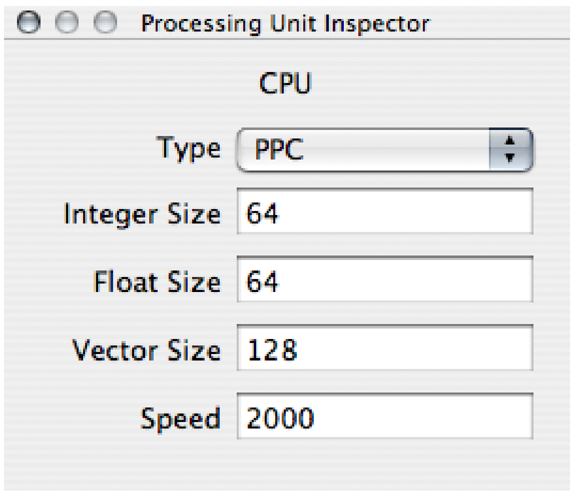
Figure A.2: Container Inspector.

The container inspector, shown in Figure A.2 is used to set the resource manager associated with a given container.

The drop-down list box in this inspector is populated with the resource manager classes found by the system. Any class which implements the correct protocol will be displayed in this list. If you have added a bundle which contains a resource manager then you can check it was loaded correctly by looking in this list.

Containers play an important rôle in the system, but they are not particularly customisable. Their job is simply to act as groupings of other objects.

A.2.2 Processing Units


Processing Unit Inspector

CPU

Type	PPC
Integer Size	64
Float Size	64
Vector Size	128
Speed	2000

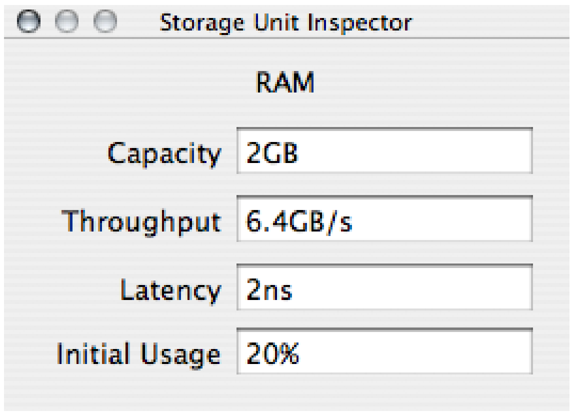
Figure A.3: Processing Unit Inspector.

Processing units are the components of the system which represent anything that can run tasks. Their run time is consumed by tasks and (optionally) resource managers. Figure A.3 shows the inspector displayed when one of these components is inspected.

The most fundamental attribute of a processing unit is the instruction set. The system understands PowerPC (PPC), Itanium (IA64), SPARC and x86 as general purpose CPU types, as well as DSPs and GPUs as more specialised functions. The primary purpose of these is processor-affinity; a process will not be migrated from one CPU architecture to another once it has been started by default (although individual tasks can override this behaviour).

The next three options define the register sizes of the processing resource. These are used by tasks to provide scaling factors allowing them to match the speed attribute to their specific workload.

A.2.3 Storage Units



RAM

Capacity	2GB
Throughput	6.4GB/s
Latency	2ns
Initial Usage	20%

Figure A.4: Storage Unit Inspector.

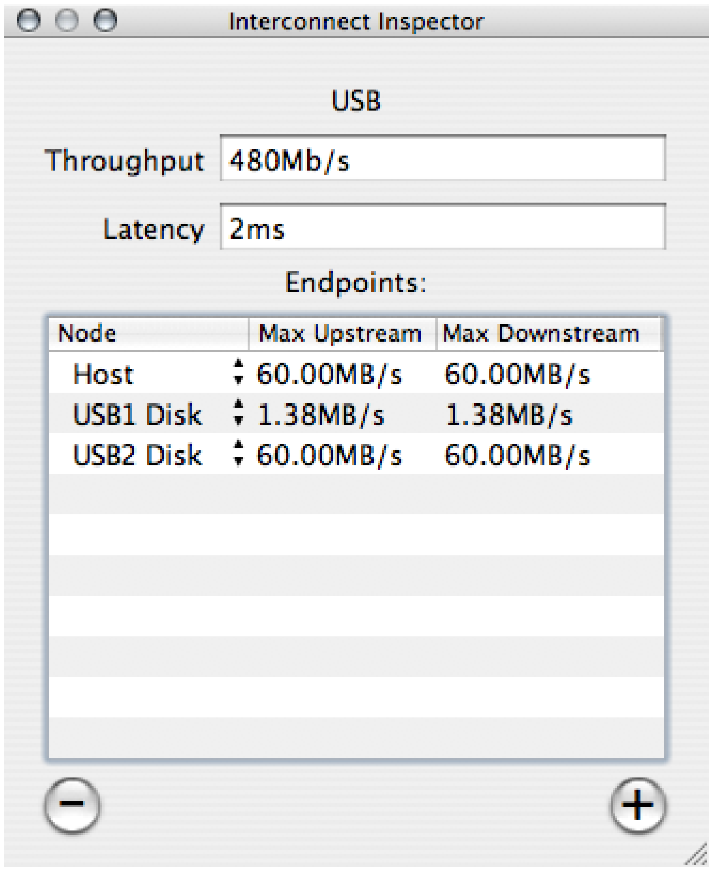
Storage units are repositories of data. They can represent anything that stores data, from high-speed RAM to tape drives. The primary attribute is their capacity, which defines the amount of data that can be stored on them.

The throughput and latency are used when passing messages to and from the storage units. Latency adds a delay to messages, while throughput gives a maximum amount of data that can be moved to and from the storage unit in a given time period. The discrete time simulation calculates available bandwidth every tick.

The final attribute is the amount of the storage unit that is being used when the system starts. It is common, for example, for a hard disk to have some data stored on it by system files. This can be configured here. This has no real effect on the system that could not be simulated by simply reducing the size of the storage unit, but it does allow more user-friendly output from a simulation to be generated.

A.2.4 Interconnects

Interconnects are used to join up components. Each interconnect is modelled as a bus, with the option of imposing an extra limit on the amount of data that can be transferred in each direction by each component.



Throughput: 480Mb/s

Latency: 2ms

Node	Max Upstream	Max Downstream
Host	60.00MB/s	60.00MB/s
USB1 Disk	1.38MB/s	1.38MB/s
USB2 Disk	60.00MB/s	60.00MB/s

Figure A.5: Interconnect Inspector.

The inspector, shown in Figure A.5, provides two global parameters, throughput and latency. These affect the delivery of messages across the Interconnect, by imposing a delay (latency) and a limit on the amount of data that can be transferred in a single tick (calculated from the throughput).

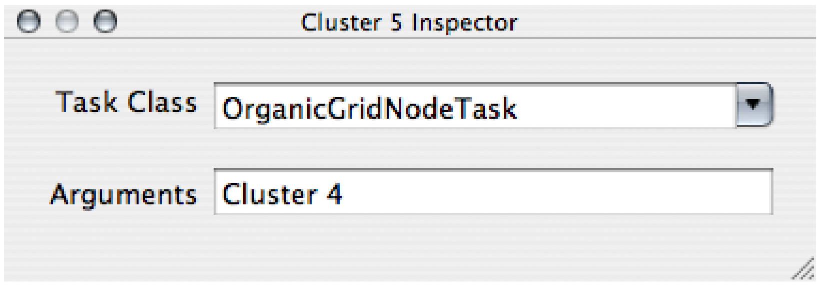
The endpoints display contains a list of nodes that are connected to this Interconnect. They can be added and removed with the + and − buttons at the bottom of the inspector. The node column contains the name of the endpoint, and if clicked will display a drop-down box of all possible endpoints[1].

Each of these can have limits on the amount of data they transfer imposed in each direction. In the example shown, the Interconnect is modelling a USB 2 bus. One device on this Interconnect only supports USB 1, and so is limited to 12Mb/s.

A.2.5 Tasks

Tasks can be added to the system in the same way as any other object; by adding a new node and selecting ‘Task’ as its type in the main view.

[1]Recall that only nodes in the same container as the Interconnect can be connected


Cluster 5 Inspector

Task Class: OrganicGridNodeTask

Arguments: Cluster 4

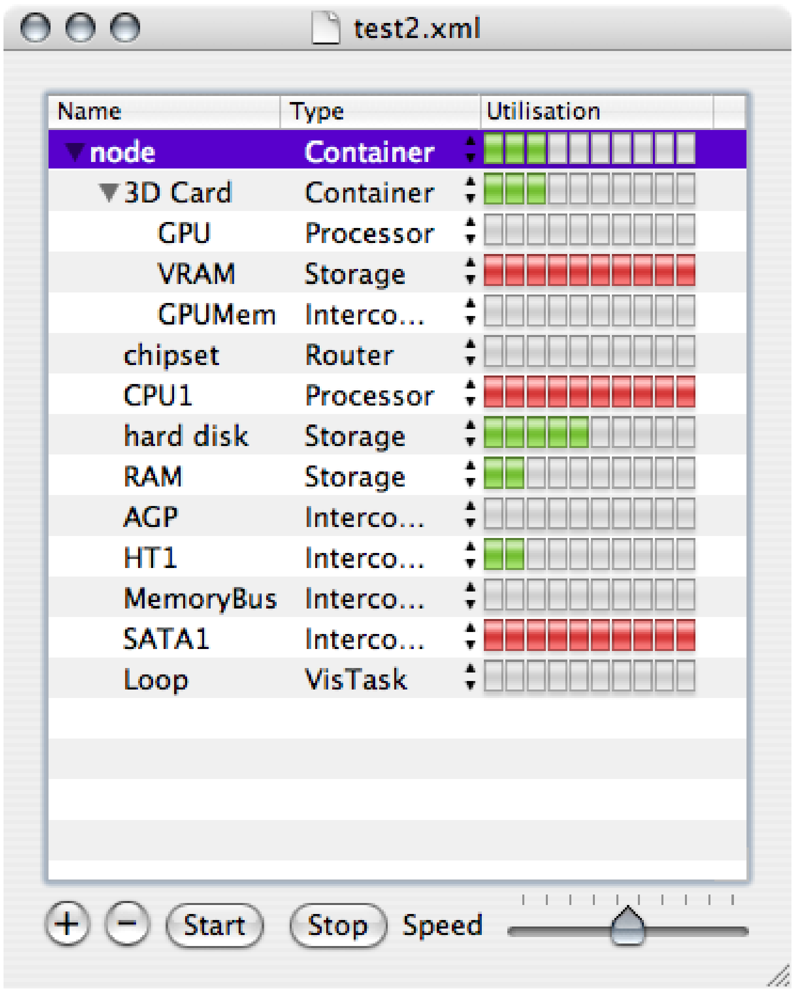
Figure A.6: Task Inspector.

When inspected, something like Figure A.6 will be displayed. As with the resource manager inspector, this gives a drop down list box which contains all tasks in the system, populated at run time.

In addition to this, arguments can be provided for each task running in the system. When the simulation runs, the string entered here will be passed to the task, and can be parsed in any way the class desires. This screen shot is from the Organic Grid case study, where each task of this class takes the name of a node as the argument. This node is then used as the parent node in the overlay network.

A.3 Running The Simulation

When the simulation is started (by pressing the start button), the display updates as shown in Figure A.7, to show the instantaneous usage of each component.



test2.xml

Name	Type	Utilisation
▼ node	Container	🟩🟩🟩⬜⬜⬜⬜⬜⬜⬜
▼ 3D Card	Container	🟩🟩🟩⬜⬜⬜⬜⬜⬜⬜
GPU	Processor	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
VRAM	Storage	🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥
GPUMem	Interco...	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
chipset	Router	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
CPU1	Processor	🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥
hard disk	Storage	🟩🟩🟩🟩🟩⬜⬜⬜⬜⬜
RAM	Storage	🟩🟩⬜⬜⬜⬜⬜⬜⬜⬜
AGP	Interco...	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
HT1	Interco...	🟩🟩⬜⬜⬜⬜⬜⬜⬜⬜
MemoryBus	Interco...	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
SATA1	Interco...	🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥
Loop	VisTask	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

Figure A.7: SimEAC running grid view.

Appendix B

SimEAC Interface Summary

The XML schema used to describe SimEAC grids is shown in Listing B.1. XML Schema is not expressive enough to describe all of the restrictions on a valid grid file. In addition to the syntax described here, the following conditions must be upheld:

- Names of elements must be unique within their parent container.
- Resource manager and task classes must reference classes available to the system.

Listing B.1: The XML schema used for describing SimEAC grids.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Grid description schema for SimEAC.
        </xsd:documentation>
    </xsd:annotation>

    <!-- Root element must be a container.  Only one allowed. -->
    <xsd:element name="container" type="containerType" />


    <xsd:complexType name="containerType">
        <!--
        Any hardware node may be in a container, including nested
        containers.  The order does not matter.
        Tasks may also be found in containers.
        -->
        <xsd:sequence minOccurs="0" maxOccurs="1" >
            <xsd:sequence minOccurs="0" maxOccurs="unbounded" >
                <xsd:element name="container" type="containerType"
                    minOccurs="0" />
                <xsd:element name="processor" type="processorType"
                    minOccurs="0" />
                <xsd:element name="storage" type="storageType"
                    minOccurs="0" />
                <xsd:element name="router" type="routerType" minOccurs
                    ="0" />
                <xsd:element name=”reference” type=”referenceType” minOccurs=”0” />
                <xsd:element name=”task” type=”taskType” minOccurs=”0” />
            </xsd:sequence>
            <xsd:sequence minOccurs=”0” maxOccurs=”unbounded” >
                <!--
                Interconnects must be declared after the elements they connect.
                Tasks are declared anywhere, but are usually put at the end for clarity
                -->
                <xsd:element name=”interconnect” type=”interconnectType” minOccurs=”0” />
                <xsd:element name=”task” type=”taskType” minOccurs=”0” />
            </xsd:sequence>
        </xsd:sequence>
        <xsd:attribute name=”name” type=”xsd:string” />
        <!--
        The resource manager defaults to ResourceManager, a null implementation
        -->
        <xsd:attribute name=”ResourceManager” type=”xsd:string” use=”optional” />
    </xsd:complexType>

    <xsd:complexType name=”processorType”>
        <xsd:attribute name=”name” type=”xsd:string” />
        <xsd:attribute name=”type” type=”processorInstructionSetType” />
        <xsd:attribute name=”int” type=”xsd:nonNegativeInteger” />
        <xsd:attribute name=”float” type=”xsd:nonNegativeInteger” />
        <xsd:attribute name=”vector” type=”xsd:nonNegativeInteger” />
        <xsd:attribute name=”speed” type=”xsd:nonNegativeInteger” />
    </xsd:complexType>


    <xsd:complexType name=”storageType”>
        <xsd:attribute name=”name” type=”xsd:string” />
        <xsd:attribute name=”size” type=”capacityType” />
        <xsd:attribute name=”access” type=”timeType” />
        <xsd:attribute name=”rate” type=”speedType” />
        <xsd:attribute name=”persistent” type=”boolType” use=”optional” />
        <xsd:attribute name=”usage” type=”percentageType” use=”optional” />
    </xsd:complexType>

    <xsd:complexType name=”interconnectType”>
        <xsd:choice>
            <xsd:element name=”node” type=”interconnectNodeType” maxOccurs=”unbounded” />
        </xsd:choice>
        <xsd:attribute name=”name” type=”xsd:string” />
        <xsd:attribute name="throughput" type="speedType" />
        <xsd:attribute name="latency" type="timeType" />
    </xsd:complexType>


    <xsd:complexType name="routerType">
        <xsd:attribute name="name" type="xsd:string" />
    </xsd:complexType>

    <xsd:complexType name="taskType">
        <!--
        The class attribute must correspond to the name of a class
            which
        implements the Task protocol.
        -->
        <xsd:attribute name="name" type="xsd:string" />
        <xsd:attribute name="class" type="xsd:string" />
        <xsd:attribute name="args" type="xsd:string" />
    </xsd:complexType>

    <xsd:complexType name="interconnectNodeType">
        <!--
        Each node may have optional restrictions placed on its
            connection
        speed in both directions.  This is ignored if larger than the
        interconnect's total bandwidth.
        -->
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="name" type="xsd:string" />
                <xsd:attribute name="upstream" type="speedType" use="
                    optional" />
                <xsd:attribute name="downstream" type="speedType" use=
                    "optional" />
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="referenceType">
        <!--
        Must refer to the name attribute in a node in the current
            level
        -->
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="name" type="xsd:string" />
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:simpleType name="capacityType">
        <xsd:restriction base="xsd:string">
            <!--
            A capacity in bits or Bytes supporting standard SI
                prefixes
            -->
            <xsd:pattern value="\d*(.\d*)?(K|M|G|T|P|E)?(B|b)"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="timeType">
        <xsd:restriction base="xsd:string">
            <!--
            A time specified in nano-, milli-, or Mega-seconds.
            -->
            <xsd:pattern value="\d*(.\d*)?(n|m|M)?s"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="speedType">
        <xsd:restriction base="xsd:string">
            <!--
            Any speed of the form 12.34B/s, supporting both b(its) and B(ytes)
            and standard SI prefixes and an arbitrary numeric precision
            -->
            <xsd:pattern value="\d*(.\d*)?(K|M|G|T|P|E)?(B|b)/(n|m|M)?s"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="percentageType">
        <xsd:restriction base="xsd:string">
            <!--
            Any number of the form x.y%, with arbitrary precision
            -->
            <xsd:pattern value="\d*(.\d*)?%"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="processorInstructionSetType">
        <xsd:restriction base="xsd:string">
            <!--
            The current implementation treats these as being case insensitive.
            -->
            <xsd:enumeration value="x86"/>
            <xsd:enumeration value="gpu"/>
            <xsd:enumeration value="ppc"/>
            <xsd:enumeration value="sparc"/>
            <xsd:enumeration value="dsp"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="boolType">
        <!--
        The standard xsd:boolean type was not used because this is more
        consistent with the Objective-C BOOL type, which allows values
             of YES
        and NO, making it easier for users dealing with both the XML
            and
        Objective-C values.
        -->
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="YES"/>
            <xsd:enumeration value="NO"/>
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

The interfaces that must be implemented by task and resource manager classes are shown in Listings B.3 and B.2 respectively. Once classes that implement these interfaces have been loaded into the system, they can be referenced from XML grids.

Listing B.2: Interface for resource managers.

```
@protocol routingResourceManager
- (NSArray*) pathFrom:(InterconnectableNode*)source to:(
   InterconnectableNode*)destination;
@end

@protocol ResourceManager<NSObject>
- (double) canRun:(id<Task>)_newTask;
- (void) underscheduledProcesses:(NSArray**)processes
                        weighted:(NSArray**)weights;
- (void) addTask:(MetaTask*)task;
- (void) removeTask:(MetaTask*)task;
- (sim_time_t) processingUnitTimeUsedBy:(id<Task>)task;
- (BOOL) isDistributed;
+ (id) resourceManagerForContainer:(id)aContainer
                        withParent:(id<ResourceManager>)
                            aResourceManager;
- (id) initForContainer:(id)aContainer
             withParent:(id<ResourceManager>)aResourceManager;
- (BOOL) addChild:(id<ResourceManager>)aChild;
- (void) runFor:(sim_time_t)nanoseconds;

- (BOOL) setVMSizeForProcess:(MetaTask*)task to:(double)bytes;
- (void) pageFault:(double) bytes forTask:(MetaTask*)aTask;
- (StorageUnit*) newFileNamed:(NSString*)fileName
                     WithSize:(double) bytes;
- (BOOL) resizeFile:(NSString*)fileName
                 on:(StorageUnit*)storage
                 to:(double) bytes;
- (BOOL) store:(double)bytes
        toFile:(NSString*)fileName
      onDevice:storage
       forTask:(MetaTask*)task
        atTime:(sim_time_t)now;
- (BOOL) load:(double)bytes
     fromFile:(NSString*)fileName
     onDevice:storage
      forTask:(MetaTask*)task
       atTime:(sim_time_t)now;
- (void) log;
@end
```

Listing B.3: Interface to be implemented by tasks.

```
@protocol Task<NSObject,MessageReceiver>
- (id) initWithArguments:(NSString*)arguments Name:(NSString*)aName;
- (void) setDelegate:(id)aDelegate;
- (id) delegate;
- (sim_time_t) runFor:(sim_time_t)nanoseconds
           onProcessors:(NSArray*)processors
                 atTime:(sim_time_t)now;
- (BOOL) runsOn:(CPUType)anInstructionSet;
- (unsigned int) threads;
- (void) retainTask;
- (void) releaseTask;
- (float) usage;
@end
```

Listings B.4 to B.10 show the interfaces to the hardware components within the system.

Listing B.4: Interface to the HardwareNode class.

```
@interface HardwareNode : NSObject<ETXMLParserDelegate,Logging> {
    sim_time_t now;
    sim_time_t last;
    NSString * name;
    id parent;
    id parser;
    BOOL isLogged;
    BOOL isEnabled;
    unsigned int logIndex;
    LogFile * logFile;
}
- (id) initWithXMLParser:(id)parser parent:(id<NSObject,
   ETXMLParserDelegate>)parent;
- (id) initWithXML:(ETXMLNode*)_xml withParent:(HardwareNode*)_parent;
- (NSString*) name;
- (void) name:(NSString*)_name;
- (ETXMLNode*) toXML;
- (BOOL) canHaveChildren;
- (void) setContainer:(id)_parent;
- (id) container;
- (BOOL) isValidChild:(id)_child;
- (void) tick:(sim_time_t)_newTime;
- (float) usage;
- (void) start;
- (BOOL) isEnabled;
- (void) setEnabled:(BOOL)aFlag;
- (NSComparisonResult)compare:(HardwareNode*)aNode;
@end
```

Listing B.5: Interface to the InterconnectableNode class.

```
@interface InterconnectableNode : HardwareNode<MessageReceiver> {
     NSMutableArray * connections;
     NSMutableDictionary * knownRoutes;
     NSMutableSet * knownAbsensesOfRoutes;
}
- (NSArray*) pathTo:(InterconnectableNode*)destination alongRoute:(
    NSArray*)path;
- (NSArray*) pathTo:(InterconnectableNode*)destination from:(
    InterconnectableNode*)origin alongRoute:(NSArray*)path;
- (void) addConnection:(Interconnect*)path;
- (void) removeConnection:(Interconnect*)path;
@end
```

Listing B.6: Interface to the Interconnect class.

```
@interface Interconnect : HardwareNode {
     double throughput;
     double latency;
     double spareBandwidthLastQuantum;
     double spareBandwidthThisQuantum;
     double bandwidthThisQuantum;
     double bandwidthLastQuantum;
     double parsedUpstream;
     double parsedDownstream;
     double dataTransferredSinceLog;
     float usage;
     sim_time_t lastLog;
     NSMutableSet * connectedObjects;
     NSArray * sortedEndpoints;
     TRArray packets;
}

+ (id) interconnectWithLatency:(double)_latency throughput:(double)
    _throughput;
- (id) initWithLatency:(double)_latency throughput:(double)_throughput
    ;
+ (id) interconnectWithLatencyFromString:(NSString*)_latency
    throughput:(NSString*)_throughput;
- (id) initWithLatencyFromString:(NSString*)_latency throughput:(
    NSString*)_throughput;
+ (id) interconnectWithXML:(ETXMLNode*)_xml withParent:(HardwareNode*)
    _container;
- (BOOL) connect;
- (double) throughput;
- (NSString*) throughputString;
- (void) throughput:(double)_throughput;
- (void) throughputFromString:(NSString*)_throughput;
- (double) latency;
- (NSString*) latencyString;
- (void) latency:(double)_latency;
- (void) latencyFromString:(NSString*)_latency;
- (void) addEndPointNamed:(NSString*)aName;
- (void) addEndPoint:(id)node;
- (void) addEndPoint:(id)node withUpstream:(double)upstream downstream
    :(double)downstream;
- (void) removeEndPoint:(id)node;
- (void) send:(Message*)aMessage to:(id<MessageReceiver>)destination
    alongRoute:(NSArray*)aRoute atTime:(sim_time_t)startTime;
- (void) broadcastMessage:(Message*)aMessage from:(HardwareNode*)
    anOrigin atTime:(sim_time_t)startTime;
- (NSSet*) endPoints;

- (unsigned int) endPointCount;
- (id) endpointAtIndex:(unsigned int) index;
@end
```

Listing B.7: Interface to the Container class.

```
@interface Container : InterconnectableNode {
    NSMutableDictionary * storageUnits;
    NSMutableDictionary * processingUnits;
    NSMutableDictionary * interconnects;
    NSMutableDictionary * interconnectableNodes;
    NSMutableDictionary * containers;
    NSArray * allInterconnectableNodes;
    NSArray * allInterconnects;
    NSMutableSet * upstreamRouters;
    NSMutableSet * names;
    id resourceManager;
    NSString * resourceManagerName;
    NSMutableArray * tasks;
    NSMutableDictionary * tasksByName;
}
+ (id) containerFromFile:(NSString*)_file;
+ (id) containerWithXML:(ETXMLNode*)_xml withParent:(id)_grid;
- (InterconnectableNode*) nodeNamed:(NSString*)_name;
- (id) childAtIndex:(unsigned int)index;
- (void) addProcessor:(id)aProcessingUnit;
- (void) addStorage:(id)aStorageUnit;
- (void) addContainer:(id)aContainer;
- (void) addRouter:(id)aRouter;
- (void) addInterconnect:(id)anInterconnect;
- (void) removeChild:(id)_child;
- (unsigned int) childCount;
- (unsigned int) interconnectCount;
- (unsigned int) taskCount;
- (NSString*) getUnusedName;
- (Interconnect*) interconnectAtIndex:(unsigned int)index;
- (MetaTask*) taskAtIndex:(unsigned int)_index;
- (BOOL) addChild:(id)_child;
- (id) initWithXML:(ETXMLNode*)_xml withParent:(HardwareNode*)_parent;
- (HardwareNode*) componentNamed:(NSString*)aName;
- (void) addDefaultChild;
- (void) runFor:(sim_time_t) nanoseconds;
- (BOOL) containsInterconnectableNode:(InterconnectableNode*)aNode;
- (NSDictionary*) processingUnitsIncludingChildren:(BOOL)flag;
- (NSDictionary*) storageUnitsIncludingChildren:(BOOL)flag;
- (HardwareNode*) componentNamed:(NSString*)aName;
- (void) start;
- (void) addTask:(MetaTask*)task;
- (void) removeTask:(MetaTask*)task;
- (MetaTask*) taskNamed:(NSString*)taskName;
- (id)resourceManager;
- (void) setResourceManager:(NSString*)aResourceManagerName;
- (NSString*)resourceManagerName;
- (BOOL) contains:(InterconnectableNode*)node;
- (NSArray*) interconnectableNodeNames;
- (MetaTask*) getTaskNamed:(NSString*)taskName;
- (MetaTask*) getTaskNamed:(NSString*)taskName checkParent:(BOOL)flag;
@end
```

Listing B.8: Interface to the ProcessingUnit class.

```
@interface ProcessingUnit : InterconnectableNode {
    CPUType instructionSet;
    int integerSize;
    int floatSize;
    int vectorSize;
    int throughput;
    float usage;
}
+ (id) processingUnitWithInstructionsSet:(CPUType)_instructionSet
    integerSize:(int)_integerSize floatSize:(int)_floatSize vectorSize
    :(int)_vectorSize throughput:(int)_throughput;
- (id) initUnitWithInstructionsSet:(CPUType)_instructionSet
    integerSize:(int)_integerSize floatSize:(int)_floatSize vectorSize
    :(int)_vectorSize throughput:(int)_throughput;
+ (id) processingUnitWithXML:(ETXMLNode*)_xml withParent:(HardwareNode
    *)_parent;
- (CPUType) instructionSet;
- (void) instructionSet:(CPUType)_instructionSet;
- (int) integerSize;
- (void) integerSize:(int)_integerSize;
- (int) floatSize;
- (void) floatSize:(int)_floatSize;
- (int) vectorSize;
- (void) vectorSize:(int)_vectorSize;
- (int) throughput;
- (void) throughput:(int)_throughput;
- (float) usage;
- (void) setUsage:(float)newUsage;
@end
```

Listing B.9: Interface to the StorageUnit class.

```
@interface StorageUnit : InterconnectableNode {
    double size; //Capacity in bytes
    double access; //Access time in ns
    double rate; //Transfer rate in B/s
    double spareCapacity; //Unused storage capacity
    double spareBandwidth; //Unused interface bandwidth.
    double bytesOfCurrentMessageSent;
    double initialUsage;
    BOOL persistent; //is the storage device volatile?
    NSMutableDictionary * files;
    TRArray messageQueue;
}
    + (StorageUnit*) storageUnitWithSize:(double)_size access:(double)
    _access rate:(double)_rate persistent:(BOOL)flag;
    - (id) initWithSize:(double)_size access:(double)_access rate:(double)
    _rate persistent:(BOOL)flag;
    + (StorageUnit*) storageUnitWithSizeFromString:(NSString*)_size access
    :(NSString*)_access rate:(NSString*)_rate persistent:(NSString*)
    flag;
    - (id) initWithSizeFromString:(NSString*)_size access:(NSString*)
    _access rate:(NSString*)_rate persistent:(NSString*)flag;
    + (id) storageUnitWithXML:(ETXMLNode*)_xml withParent:(HardwareNode*)
    _parent;
    - (void) start;
    - (double) size;
    - (NSString*) sizeString;
    - (void) size:(double)_size;
    - (void) sizeFromString:(NSString*)_size;
    - (double) access;
    - (NSString*) accessString;
    - (void) access:(double)_access;
    - (void) accessFromString:(NSString*)_access;
    - (double) rate;
    - (NSString*) rateString;
    - (void) rate:(double)_rate;
    - (void) rateFromString:(NSString*)_rate;
    - (BOOL) isPersistent;
    - (void) persistent:(BOOL)flag;
    - (void) free:(double)bytes;
    - (void) setUsageFromString:(NSString*)usageString;
    - (BOOL) setSize:(double)bytes forFile:(NSString*)fileName;
    - (double) sizeOfFile:(NSString*)fileName;
    - (double) initialUsage;
    - (double) freeSpace;
    @end
```

Listing B.10: Interface to the Router class.

```
}
+ (Router*) routerWithXML:(ETXMLNode*)_xml withParent:(HardwareNode*)
    _parent;
- (BOOL) isUplink;
- (void) setUplink:(BOOL)flag;
@end
```

Bibliography

[1] Ensight gold. http://www.ensight.com/ensight-gold.html. last accessed: January 2008.

[2] G. Abram and L. Treinish. An extended dataflow architecture for data analysis and visualization. *ACM SIGGRAPH Computer Graphics*, 29(2):17–21, 1995.

[3] Access Grid documentation. http://www.accessgrid.org. last accessed: January 2008.

[4] M. Agarwal and M. Parashar. Enabling autonomic compositions in Grid environments. In *Proc. 4th Int. Workshop on Grid Computing*, pages 34–41, 2003.

[5] P. K. Agarwal, L. Arge, T. M. Murali, K. R. Varadarajan, and J. S. Vitter. I/o-efficient algorithms for contour-line extraction and planar graph blocking (extended abstract). In *Proc. Symposium on Discrete Algorithms*, pages 117–126, 1998.

[6] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *IEEE CG&A*, 21(4):34–41, 2001.

[7] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 206–213, 2004.

[8] K. Akeley. The silicon graphics 4D/240GTX superworkstation. *IEEE CG&A*, 9(4):71–83, 1989.

[9] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, and C. Silva. Point set surfaces. In *Proc. IEEE Visualization*, pages 29–36, 2001.

[10] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Cactus tools for Grid applications. *Cluster Computing*, 4(3):179–188, 2001.

[11] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM '04: Proceedings of the 16th Int. Conf. on Scientific and Statistical Database Management (SSDBM'04)*, pages 21–23, Washington, DC, USA, 2004. IEEE Computer Society.

[12] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc. AFIPS Conf.*, volume 30, pages 483–485, 1967.

[13] D. S. Anderson, L. Stoller, M. Hibler, T. Stack, and J. Lepreau. Automatic online validation of network configuration in the emulab network testbed. *Int. Conf. on Autonomic Computing*, 2006.

[14] S. Anderson, M. Hartswood, R. Procter, M. Rouncefield, R. Slack, J. Soutter, and A. Voss. Making autonomic computing systems accountable: the problem of human computer interaction. In *Proc. 14th Int. Workshop on Database and Expert Systems Applications*, pages 718–724, 2003.

[15] W. Anderson. An overview of motorola's powerpc simulator family. *Commun. ACM*, 37(6):64–69, 1994.

[16] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, 1993.

[17] C. S. Ang, D. C. Martin, and M. D. Doyle. Integrated control of distributed volume visualization through the world wide web. In *Proc. IEEE Visualization*, pages 13–20, 1994.

[18] M. Ankerst, D. A. Keim, and H. P. Kriegel. Circle segments: a technique for visual exploring large multidimensional data sets. In *Proc. IEEE Visualization*, page Hot Topic Session, 1996.

[19] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 540–548, New York, NY, USA, 1997. ACM Press.

[20] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of i/o-efficient algorithms for multidimensional batched searching problems (extended abstract). In *Proc. Symposium on Discrete Algorithms*, pages 685–694, 1998.

[21] L. Arge, L. Toma, and N. Zeh. I/o-efficient topological sorting of planar dags. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 85–93, New York, NY, USA, 2003. ACM Press.

[22] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47(1):1–25, 2007.

[23] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[24] O. Babaoglu, H. Meling, and A. Montresor. A framework for the development of agentbased peer-to-peer systems. In *Proc. 22nd Int. Conf. on Distributed Computing Systems*, pages 15–22. IEEE Computer Socity Press, 2002.

[25] S. Baker. *CORBA Distributed Objects*. Addison Wesley, 1997.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[27] D. Bartz and C. Silva. Rendering and visualization in parallel environments. In *Eurographics Tutorials*, 2001.

[28] D. Beckett. Resource description framework. http://www.w3.org/TR/rdf-schema/, 2004.

[29] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX*, 2005:41–46, 2005.

[30] S. D. Benford, J. Bowers, S. Gray, T. R. Rodden, M. Rygol, and V. Stanger. The VirtuOsi project. In *Proc. London Virtual Reality Expo*, 1994.

[31] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[32] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *Proc. IEEE Symp. Parallel and Large Data Visualization and Graphics*, pages 41–50, 2003.

[33] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 2002.

[34] J. Blinn. A generalized algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[35] J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometry Design*, 5(4):341–355, 1988.

[36] B. G. Blundell and A. J. Schwarz. The classification of volumetric display systems: characteristics and predictability of the image space. *IEEE Trans. Visualization and Computer Graphics*, 8(1):66–75, 2002.

[37] D. Brickley and R. V. Guha. Vocabulary description language. http://www.w3.org/RDF/, 2004.

[38] K. Brodlie, L. Brankin, G. Banecki, A. Gay, A. Poon, and H. Wright. GRASPARC–a problem solving environment integrating computation and visualization. In *Proc. IEEE Visualization*, pages 102–109, 1993.

[39] K. Brodlie, J. Brooke, M. Chen, D. Chisnall, A. Fewings, C. Hughes, N. W. John, M. W. Jones, M. Riding, and N. Roard. Visual supercomputing - technologies, applications and challenges. (24):217–245, 2005.

[40] K. Brodlie, J. Wood, D. Duce, J. Gallop, and J. Walton. Distributed and collaborative visualization. In *Eurographics 2003 State of the Art Report*, pages 223–251, 2003.

[41] K. W. Brodlie, J. Brooke, M. Chen, D. Chisnall, A. Fewings, C. Hughes, N. W. John, M. W. Jones, M. Riding, and N. Roard. Visual supercomputing–technologies, applications and challenges. *Computer Graphics Forum*, 24(2):217–245, 2005.

[42] K. W. Brodlie, J. Brooke, M. Chen, D. Chisnall, C. J. Hughes, N. W. John, M. W. Jones, M. Riding, N. Roard, M. Turner, and J. D. Wood. Adaptive infrastructure for visual computing. In *Theory and Practice of Computer Graphics*, pages 147–156, 2007.

[43] K. W. Brodlie, J. Wood, D. Duce, J. R. Gallop, D. Gavaghan, M. Giles, S. Hague, J. Walton, M. Rudgyard, B. Collins, J. Ibbotson, and A. Knox. XML for visualization. In *Euroweb*, 2002.

[44] W. Broll. Interacting in distributed collaborative VE. In *Proc. VRAIS*, pages 148–155, 1995.

[45] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational steering in RealityGrid. In *Proc. UK e-Science All Hands Meeting*, 2003.

[46] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[47] J. D. Bruijn, H. Lausen, A. Polleres, and D. Fensel. The wsml rule languages for the semantic web. In *W3C Workshop on Rule Languages for Interoperability*, 2005.

[48] M. Bsoul. *Economic scheduling in Grid computing using Tender model.* PhD thesis, Loughborough University, 2007.

[49] G. Burdea and P. Coiffet. *Virtual Reality Technology*. Wiley, 2nd edition, 2003.

[50] R. Buyya, editor. *High Performance Cluster Computing–Volume 1: Architectures and Systems*. Prentice Hall, 1999.

[51] R. Buyya, editor. *High Performance Cluster Computing–Volume 2: Programming and Applications*. Prentice Hall, 1999.

[52] Z. Cai, V. Kumar, B. F. Cooper, G. S. Eisenhauer, K. Schwan, and R. E. Strom. Utility-driven availability-management in enterprise-scale information flows. Technical report, Georgia Institute of Technology, 2006.

[53] C. Carlson and O. Hagsand. DIVE: a platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–669, 1993.

[54] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. Extending distortion viewing from 2D to 3D. *IEEE CG&A*, 17(4):42–51, 1997.

[55] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1986.

[56] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: Self-organizing computation on a peer-to-peer network. *Autonomic Computing*, 1(1):96–103, 2004.

[57] X. Chang. Network simulations with opnet. In *WSC '99: 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM Press.

[58] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion resource management system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

[59] M. Chen. Combining point clouds and volume objects in volume scene graphs. In *Proc. Volume Graphics*, pages 127–135, New York, 2005.

[60] M. Chen and J. V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, 2000.

[61] M. Chen, J. V. Tucker, R. H. Clayton, and A. V. Holden. Constructive volume geometry applied to visualization of cardiac anatomy and electrophysiology. *Int. J. Bifurcation and Chaos*, 13(12):3591–3604, 2003.

[62] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 36–43, 2004.

[63] D. M. Chess, C. C. Palmer, and S. R. Andwhite. Security in an autonomic computing environment. *IBM Systems journal*, 42(1):107–118, 2003.

[64] Y.-J. Chiang, R. Farias, C. T. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc. IEEE Symposium on Parallel and Visualization and Graphics*, pages 59–66, 2003.

[65] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[66] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[67] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[68] Y.-J. Chiang and C. T. Silva. Interactive isosurface extraction. In *Proc. IEEE Visualization '98*, pages 167–174, 1998.

[69] C. H. Chien and J. K. Aggarwal. Volume/surface octrees for the representation of three-dimensional objects. *Computer Vision, Graphics and Image Processing*, 36(1):100–113, 1986.

[70] H. Childs. VisIt. http://www.graphics.stanford.edu/˜mhouston/VisWorkshop04/, 2004. last accessed: January 2008.

[71] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[72] D. Chisnall and M. Chen. The making of simeac. In *Int. Conf. on Autonomic Computing*, pages 301–302, 2006.

[73] D. Chisnall, M. Chen, and C. Hansen. Knowledge-based out-of-core algorithms for data management in visualization. In *EuroVis*, pages 107–114, 2006.

[74] D. Chisnall, M. Chen, and C. Hansen. Ray-driven dynamic working set rendering. *The Visual Computer*, 23(3):167–179, March 2007.

[75] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proc. IEEE Symp. Volume Visualization*, pages 31–38, 1996.

[76] M. Clavel, F. D. zn, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[77] P. E. Clements, T. Papaioannou, and J. Edwards. Aglets: Enabling the virtual enterprise. In *Managing Enterprises–Stakeholders, Engineering, Logistics and Achievement (ME-SELA'97)*, Loughborough University, UK, 1997.

[78] J. C. Collis and L. C. Lee. Building electronic marketplaces with the ZEUS agent tool-kit. *Lecture Notes in Computer Science*, 1571, 1999.

[79] Condor. http://www.cs.wisc.edu/condor. last accessed: January 2008.

[80] Z. Constantinescu. Towards an autonomic distributed computing environment. In *Proc. 14th Int. Workshop on Database and Expert Systems Applications*, pages 699–703, 2003.

[81] F. Corbato. A paging experiment with the multics system. Technical report, M.I.T. Press, 1968.

[82] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pages 2–8, 2003.

[83] COVISE. http://www.visenso.de/?id=14. last accessed: January 2008.

[84] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proc. IEEE Visualization*, pages 235–244, 1997.

[85] R. Crawfis and N. Max. Texture splats for 3d scalar and vector field visualization. In *Proc. IEEE Visualization*, pages 261–266, 1993.

[86] B. D'Amora and F. Bernardini. Pervasive 3D viewing for product data management. *IEEE CG&A*, 23(2):14–19, 2003.

[87] G. W. Daniel and M. Chen. Video visualization. In *Proc. IEEE Visualization*, pages 409–416, 2003.

[88] G. Deen, T. Lehman, and J. Jaufman. The almaden optimalgrid project. In *Autonomic Computing Workshop–Proc. 5th Int. Workshop on Active Middleware Services*, pages 14–21, 2003.

[89] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed interactive ray tracing for large volume visualization. In *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pages 88–94, 2003.

[90] H. B. Demuth. *Electronic Data Sorting*. PhD thesis, Stanford University, 1956.

[91] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[92] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, Semptember 1970.

[93] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated execution driven alpha 21264 simulator, 2001.

[94] P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors. *Parallel Processing for Computer Vision and Display*. Addison-Wesley, 1989.

[95] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS '91: Proceedings of the first Int. Conf. on Parallel and distributed information systems*, pages 280–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[96] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.

[97] L. J. Doctor and J. G. Torborg. Display techniques for octree-encoded objects. *IEEE CG&A*, 3(1):29–38, 1981.

[98] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *Proc. IEEE Int. Conf. on Performance, Computing, and Communications*, pages 61–68, 2003.

[99] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and P. Winterbottom. The inferno^{TM} operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[100] D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):973–980, 2006.

[101] J. Dunagan, R. Roussev, B. Daniels, A. J. C. Verbowski, and Y. Wang. Towards self-managing software patching process using black-box persistent-state manifestes. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 106–113, 2004.

[102] D. S. Dyer. A dataflow toolkit for visualization. *IEEE CG&A*, 10(4):60–69, 1990.

[103] R. Elwald and L. Mass. A high performance graphics system for the Cray-1. *ACM SIGGRAPH Computer Graphics*, pages 82–86, 1978.

[104] K. Engel and T. Ertl. Interactive high-quality volume rendering with flexible consumer graphics hardware. In *Eurographics State of the Art Reports*, pages 25–48, 2002.

[105] K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proc. IEEE Visualization*, pages 139–146, 1999.

[106] D. W. Erwin and D. F. Snelling. UNICORE: a Grid computing environment. *Lecture Notes in Computer Science*, 2150:825–834, 2001.

[107] EUROGRID. http://www.eurogrid.org. last accessed: January 2008.

[108] T. Eymann, M. Reinicke, O. Ardaiz, P. Artigas, F. Freitag, and L. Navarro. Self-organizing resource allocation for autonomic network. In *Proc. 14th Int. Workshop on Database and Expert Systems Applications*, pages 656–660, 2003.

[109] R. Farias and C. T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE CG&A*, 21(4):42–50, 2001.

[110] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[111] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, 1972.

[112] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. ACM Symp. Theory of Computing*, pages 114–118, 1978.

[113] I. Foster, J. N. C. Kesselman, and S. Tuecke. The physiology of the Grid: an open Grid services architecture for distributed system integration. In *Open Grid Services Infrastructure WG*, 2002.

[114] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. J. Supercomputer Applications*, 11(2):115–128, 1997.

[115] I. Foster and C. Kesselman. The Globus project: a status report. In *Proc. Heterogeneous Computing Workshop*, pages 4–18, 1998.

[116] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[117] D. Foulser. IRIS Explorer: A framework for investigation. *ACM SIGGRAPH Computer Graphics*, 29(2):13–16, 1995.

[118] R. M. Fujimoto. Advanced tutorials: Parallel and distributed simulation systems. In *Proc. 33nd Conf. on Winter Simulation*, pages 147–157, 2001.

[119] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computional Science and Engineering*, 1(2):11–23, 1994.

[120] M. Gardner. *Magic Show*, chapter 7. Knopf, 1977.

[121] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[122] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[123] G. Germain. Concurrency oriented programming in termite scheme. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 20–20, New York, NY, USA, 2006. ACM.

[124] S. Goil and S. Ranka. Dynamic load balancing for ray-traced volume rendering on distributed memory machines. In *Proc. Int. Conf. on High Performance Computing*, 1995.

[125] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[126] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):12–21, 1993.

[127] S. Green. *Parallel Processing for Computer Graphics*. MIT Press, 1991.

[128] C. M. Greenhalgh and S. D. Benford. MASSIVE: a virtual reality system for teleconferencing. *ACM Trans. Computer Human Interfaces*, 2(3):239–261, 1995.

[129] J. Grimes, L. Kohn, and R. Bharadhwaj. The intel i860 64-bit processor: a general-purpose cpu with 3d graphics capabilities. *IEEE CG&A*, 9(4):85–94, 1989.

[130] GRIP. http://www.grid-interoperability.eu/. last accessed: January 2008.

[131] M. Gross. The utility of points as primitives for visualization. In *Keynote Speech in IEEE Visualization*, 2005.

[132] J. P. Grossman and W. J. Dally. Point sample rendering. In *Proc. Eurographics Workshop on Rendering*, pages 181–192, 1998.

[133] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024 hypercubes. *SIAM J. Scientific and Statistical Computing*, 9(4):609–638, 1988.

[134] M. Hadwiger, C. Berger, and H. Hauser. High-quality two level volume rendering of segmented data sets on consumer graphics hardware. In *Proc. IEEE Visualization*, pages 301–308, 2003.

[135] C. Hansen. Known and potential high performance computing applications in computer graphics and visualization. In *Proc. Int. Workshop on High Performance Computing for Computer Graphics and Visualisation*, pages 23–29, 1996.

[136] J. G. Hansen, E. Christiansen, and E. Jul. The laundromat model for autonomic cluster computing. In *Proc. Int. Conf. on Autonomic Computing*, pages 114–123, June 2006.

[137] P. Hartling, A. Bierbaum, and C. Cruz-Neira. Virtual reality interfaces using tweek. In *SIGGRAPH '02: ACM SIGGRAPH 2002 inproceedings abstracts and applications*, pages 278–278, New York, NY, USA, 2002. ACM.

[138] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS-V: Proceedings of the fifth international inproceedings on Architectural support for programming languages and operating systems*, pages 187–197, New York, NY, USA, 1992. ACM Press.

[139] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *Computer*, 32(8):51–58, 1999.

[140] J.-H. Her and R. S. Ramakrishna. An external-memory depth-first search algorithm for general grid graphs. *Theor. Comput. Sci.*, 374(1-3):170–180, 2007.

[141] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[142] A. Hinneburg, D. A. Keim, and M. Wawryniuk. Hd-eye: Visual mining of high-dimensional data. *IEEE CG&A*, 19(5):22–31, 1999.

[143] T. Hoare, R. Milner, M. Thomas, and A. Bundy. Criteria for a grand challenge. http://www.cra.org/Activities/grand.challenges/hoare.pdf, 2002. last accessed: January 2008.

[144] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. "http://www-306.ibm.com/autonomic/pdfs/autonomic_computing.pdf", October 2001. last accessed: January 2008.

[145] J. Hua and H. Qin. Haptics-based dynamic implicit solid modeling. *IEEE Transactions on Visualization and Computer Graphics*, 10(5):574–586, 2004.

[146] C. Hughes and N. John. A flexible approach to high performance visualization enabled augmented reality. In *Theory and Practice of Computer Graphics*, pages 181–186, 2007.

[147] C. Hughes, N. John, and M. Riding. A generic approach to high performance visualization enabled augmented reality. In *UK e-Science Programme All Hands Meeting*, pages 441–444, September 2006.

[148] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual inproceedings on Computer graphics and interactive techniques*, pages 129–140, New York, NY, USA, 2001. ACM.

[149] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002.

[150] IBM. Autonomic deployment model. http://www-306.ibm.com/autonomic/levels.shtml, 2004. last accessed: January 2008.

[151] S. Izadi, P. Coutinho, T. Rodden, and G. Smith. The FUSE platform: supporting ubiquitous collaboration within diverse mobile environments. *Automated Software Engineering*, 9(2):167–186, 2002.

[152] H. Jeon, C. J. Petrie, and M. R. Cutkosky. Jatlite: A java agent infrastructure with message routing. *IEEE Internet Computing*, 4(2):87–96, 2000.

[153] H. Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM Press.

[154] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Trans. Parallel and Distributed Systems*, 11(6):589–603, 2000.

[155] M. W. Jones. *The Visualisation of Regular Three Dimensional Data*. PhD thesis, University of Wales, 1995.

[156] M. W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, 1996.

[157] D. Kalra and A. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics (Proc. SIGGRAPH 89)*, 23(3):297–306, 1989.

[158] N. Kandasamy, S. Abdelwahed, and J. Hayes. Self-optimization in computer systems via on-line control: applications to power management. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 54–61, 2004.

[159] U. Kanus, G. Wetekam, and J. Hirche. Voxelcache: a cache-based memory architecture for volume graphics. In *Proc. SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 76–83, 2003.

[160] D. A. Keim, H. P. Kriegel, and M. Ankerst. Recursive pattern: a technique for visualizing very large amounts of data. In *Proc. IEEE Visualization*, pages 279–286, 1995.

[161] D. A. Keim, W. Muller, and H. Schumann. Visual data mining. In *Eurographics State of the Art Reports*, 2002.

[162] J. Kelso, L. E. Arsenault, S. G. Satterfield, and R. D. Kriz. DIVERSE: a framework for building extensible and reconfigurable device independent virtual environments. In *Proc. IEEE Virtual Reality*, pages 183–190, 2002.

[163] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Dec. 2005.

[164] J. O. Kephart. Research challenges of autonomic computing. In *ICSE '05: 27th Int. Conf. on Software engineering*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[165] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, 2003.

[166] G. Kindlmann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proc. IEEE Symp. Volume Visualization*, pages 79–86, 1998.

[167] J. T. Klosowski, P. D. Kirchner, J. Valuyeva, G. Abram, C. J. Morris, R. H. Wolfe, and T. Jackman. Deep view: high-resolution reality. *IEEE CG&A*, 22(3):12–15, 2002.

[168] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. Visualization and Computer Graphics*, 6(2):108–123, 2000.

[169] D. E. Knuth. *Sorting and Searching*, volume 3. Addison-Wesley.

[170] T. Kohonen. *Self-Organizing Maps*. Springer, 2nd edition, 1997.

[171] M. Koutek. *Scientific Visualisation in Virtual Reality: Interaction Techniques and Application Development*. PhD thesis, Delft University of Technology, 2003.

[172] E. E. Koutsofios, S. C. North, R. Truscott, and D. A. Keim. Visualizing large-scale telecommunication networks and services (case study). In *VIS '99: Proceedings of the inproceedings on Visualization '99*, pages 457–461, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[173] E. V. Krishnamurthy. *Parallel Processing: Principles and Practice*. Addison-Wesley, 1989.

[174] C. J. Kuehner and B. Randell. Demand paging in perspective. In *Proc. Fall Joint Computer Conference*, pages 1011–1018, 1968.

[175] T. Kurc, Ü. Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Visualization of large data sets with the active data repository. *IEEE CG&A*, 21(4):24–33, 2001.

[176] T. Kure, Ü. Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Visualization of large data sets with the active data repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, 2001.

[177] Y. Kurzion and R. Yagel. Space deformation using ray deflectors. In *Proc. 6th Eurographics Workshop on Rendering*, pages 21–32, 1995.

[178] P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorisation. *IEEE Trans. Visualization and Computer Graphics*, 2(3):218–231, 1996.

[179] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proc. IEEE Visualization*, pages 355–362, 1999.

[180] E. LaMar, B. Hamann, and K. I. Joy. A magnification lens for interactive volume visualization. In *Proc. IEEE Pacific Conf. on Computer Graphics and Applications*, pages 223–231, 2001.

[181] F. Lamberti, C. Zunino, A. Sanna, A. Fiume, and M. Maniezzo. An accelerated remote graphics architecture for PDAS. In *Proc. 8th Int. Conf. on 3D Web Technology*, pages 55–ff, 2003.

[182] D. LaRose. A fast, affordable system for augmented reality. Master's thesis, Robotics Institute, Carnegie Mellon University, April 1998.

[183] M. LaRow. The five engines of e-CRM. *Computer Technology Review*, August 2000.

[184] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. *ACM SIGGRAPH Computer Graphics*, 25(4):285–288, 1991.

[185] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es):7, 1996.

[186] J. L.Bentley. Multidimensional binary trees used for associative searching. *Communications of ACM*, 18(9):509–517, 1975.

[187] A. Lee and M. Ibrahim. Emotional attributes in autonomic computing systems. In *Proc. 14th Int. Workshop on Database and Expert Systems Applications*, pages 681–685, 2003.

[188] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Trans. Visualization and Graphics*, 2(3):202–217, 1996.

[189] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers (extended abstract). In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 704–713, 1993.

[190] P. Leutenegger and K.-L. Ma. R-tree retrieval of unstructured volume data for visualization. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, volume 50. DIMACS Book Series, American Mathematical Society, 1999.

[191] M. Levoy. Display of surfaces from volume data. *IEEE CG&A*, 8(5):29–37, 1988.

[192] G.-S. Li, U. D. Bordoloi, and H.-W. Shen. Chameleon: an interactive texture-based rendering framework for visualizing three-dimensional vector fields. In *Proc. IEEE Visualization*, pages 241–248, 2003.

[193] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proc. ACM SIGGRAPH*, pages 259–262, 2000.

[194] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Trans. Visualization and Computer Graphics*, 8(3):239–254, 2002.

[195] C. Linn. Semantic reliability in distributed systems: ontology issues and system engineering. In *Proc. IEEE/WIC Int. Conf. on Web Intelligence*, pages 292–300, 2003.

[196] M. Litoiu. A performance analysis method for autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 2(1):3, 2007.

[197] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Proc. IEEE Visualization*, pages 175–180, 1998.

[198] Y. Livnat and X. Tricoche. Interactive point-based isosurface extraction. In *Proc. IEEE Visualization*, pages 457–464, 2004.

[199] W. Lorensen and H. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[200] W. E. Lorensen. Marching through the visible man. In *Proc. Visualization*, pages 368–373, 1995.

[201] K. Ma, J. S. Painter, and M. F. Krogh. Parallel volume rendering using binary swap composition. *IEEE CG&A*, 14(4):59–67, 1994.

[202] K.-L. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE CG&A*, 21(4):72–83, 2001.

[203] P. Mackerras and B. Corrie. Exploiting data coherence to improve parallel volume rendering. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(2):8–16, 1994.

[204] J. D. Mackinlay, G. G. robertson, and S. K. Card. The perspective wall: detail and context smoothly integrated. In *Proc. ACM CHI*, pages 172–180, 1991.

[205] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[206] J. Marks, B. Andalman, P. A. Beardsley, and et al. Design galleries: a general approach to setting parameters for computer graphics and animation. In *Proc. ACM SIGGRAPH*, pages 389–400, 1997.

[207] R. Marshall, J. Kempf, S. Dyer, and C. Yen. Visualization methods and simulation steering for a 3D turbulence model for Lake Erie. *ACM SIGGRAPH Computer Graphics*, 24(2):89–97, 1990.

[208] D. Martin, M. Burstein, J. Hobb, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. http://www.w3.org/Submission/OWL-S/, 2004.

[209] R. F. McCloy and N. W. John. Remote visualization of patient data in the operating theatre during helpato-pancreatic surgery. In *Computer Assisted Radiology and Surgery*, pages 53–58. Elsevier, 2003.

[210] D. L. McGuinness and F. van Harmelen. Owl web ontology language. http://www.w3.org/TR/owl-features/, 2004.

[211] M. Meissner, M. Doggett, U. Kamus, and J. Hirche. Accelerating volume rendering using an on-chip SRAM occupancy map. In *Proc. IEEE Int. Symp. Circuits and Systems*, volume 2, pages 757–760, 2001.

[212] M. Meissner, U. Hoffmann, and W. Strasser. Enabling classification and shading for 3d texture mapping based volume rendering using OpenGL and extensions. In *Proc. IEEE Visualizationn*, pages 207–214, 1999.

[213] J. Meredith, J. Conger, Y. Liu, and J. Johnson. Evaluating mobile graphics processing units (gpus) for real-time resource constrained applications. Technical Report US200618%%435, Lawrence Livermore National Laboratory, nov 2005.

[214] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[215] C. K. Michaels and M. J. Bailey. VizWiz: A Java applet for interactive 3D scientific visualization over the web. In *Proc. IEEE Visualization*, 1997.

[216] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. In *ACM Trans. Software Engineering and Methodology*, 2000.

[217] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. volume 9, pages 273–305, New York, NY, USA, 2000. ACM.

[218] ModViz Inc. http://www.modviz.com. last accessed: January 2008.

[219] A. Moerschell and J. D. Owens. Distributed texture memory in a multi-gpu environment. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 31–38, New York, NY, USA, 2006. ACM.

[220] S. Molnar, C. M, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE CG&A*, 14(4):23–32, 1994.

[221] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):82–85, 1965.

[222] P. J. Moran and C. Henze. Large field visualization with demand-driven calculation. In *Proc. IEEE Visualization '99*, pages 27–33, 1999.

[223] K. Moreland and D. Thompson. From cluster to wall with VTK. In *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pages 25–32, 2003.

[224] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by using a ray-driven approach. In *Proc. Visuzlization '96*, pages 65–72, 1996.

[225] J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Gener. Comput. Syst.*, 15(1):119–129, 1999.

[226] T. Munzner. Exploring large graphs in 3d hyperbolic space. *IEEE CG&A*, 18(4):18–23, 1998.

[227] S. Muraki, E. B. Lum, K.-L. Ma, M. Ogata, and X. Liu. A PC cluster system for simultaneous interactive volumetric modeling and visualization. In *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, pages 95–102, 2003.

[228] M. Naef, E. Lamboray, O. Staadt, and M. Gross. The blue-c distributed scene graph. In *Proc. IEEE Virtual Reality*, pages 275–276, 2003.

[229] NERSC. http://acts.nersc.gov/cumulvs/. last accessed: January 2008.

[230] G. M. Nielson. Scattered data modeling. *IEEE Computer Graphics and Applications*, 13(1):60–70, 1993.

[231] H. Nishimura, A. Hirai, T. Kawai, I. Shirakawa, and K. Omura. Object modeling by distribution function and a method of image generation. *Computer Graphics, Jounral of Papers given at the Electronics Communication Conference '85 (in Japanese)*, J68-D(4), 1985.

[232] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 222–232, New York, NY, USA, 1993. ACM.

[233] M. H. Nodine, D. P. Lopresti, and J. S. Vitter. I/o overhead and parallel vlsi architectures for lattice computations. *IEEE Trans. Comput.*, 40(7):843–852, 1991.

[234] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 120–129, New York, NY, USA, 1993. ACM.

[235] M. H. Nodine and J. S. Vitter. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM*, 42(4):919–933, 1995.

[236] V. Normand and J. Tromp. Collaborative virtual environments: the COVEN project. In *Proc. the Framework for Immersive Virtual Environments Conf.*, 1996.

[237] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: defeating spikes with a free-market application cluster. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 1198–205, 2004.

[238] A. Norton and A. Rockwood. Enabling view-dependant progressive volume visualization on the grid. *IEEE CG&A*, 23(2):22–31, 2003.

[239] H. Nwana, D. Ndumu, and L. Lee. Zeus: An advanced tool-kit for engineering distributed multi-agent systems, 1998.

[240] S. Oaks and H. Wong. *Jini in a Nutshell.* O'Reilly & Associates, 2000.

[241] S. L. Pan and J.-N. Lee. Using e-CRM for a unified view of the customer. *Communications of the ACM*, 46(4):95–99, 2003.

[242] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Proc. Symp. Interactive 3D Graphics*, pages 119–126, 1999.

[243] S. G. Parker, C. D. H. M. Miller, and C. R. Johnson. An integrated problem solving environment: the SCIRun computational steering system. In *Proc. 31st Hawaii Int. Conf. on System Sciences*, pages 147–156, 1998.

[244] J. Pascoe, N. Ryan, and D. Morse. Using while moving: HCI issues in fieldwork environments. *ACM Trans. Computer-Human Interaction*, 7(3):417–437, 2000.

[245] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *WSC '97: 29th conference on Winter simulation*, pages 1037–1044, New York, NY, USA, 1997. ACM Press.

[246] T. Peters, B. Davey, and et al. Three dimensional multimodal image guidance for neurosurgery. *IEEE Trans. Medical Imaging*, 15:121–128, 1996.

[247] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *Proc. ACM SIGGRAPH*, pages 251–260, 1999.

[248] H. Pfister and A. Kaufman. Cube-4–a scalable architecture for real-time volume rendering. In *Proc. ACM/IEEE Symp. Volume Rendering*, pages 47–54, 1996.

[249] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proc. ACM SIGGRAPH*, pages 335–342, 2000.

[250] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proc. ACM SIGGRAPH*, pages 101–108, 1997.

[251] C. Pinhanez. Augmented reality with projected interactive displays. In *Proc. Int. Symp. Virtual and Augmented Architecture*, 2001.

[252] R. Pinkham, M. Novak, and K. Guttag. Video RAM excels at fast graphics. *Electronic Design*, 31(17):161–182, 1983.

[253] pV3. http://raphael.mit.edu/pv3/pv3.html. last accessed: January 2008.

[254] J. Rasure, D. Argiro, T. Sauer, and C. Williams. A visual language and software development environment for image processing. *Int. J. Imaging Systems and Technology*, 1991.

[255] H. Ray, H. Pfister, D. Silver, and T. A. Cook. Ray casting architectures for volume visualization. *IEEE Trans. Visualization and Computer Graphics*, 5(3):210–223, 1999.

[256] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[257] J. Rekimoto and M. Saitoh. Augmented surfaces: A spatially continuous work space for hybrid computing environments. In *Proc. ACM CHI*, pages 378–385, 1999.

[258] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage-rasterization. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 109–118,147, 2000.

[259] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. Interactive exploration of volume line integral convolution based on 3d-texture mapping. In *Proc. Visualization*, pages 233–240, 1999.

[260] M. Riding, J. Wood, K. Brodlie, J. Brooke, M. Chen, D. Chisnall, C. Hughes, N. W. John, M. W. Jones, and N. Roard. e-viz: Towards an integrated framework for high performance visualization. In *Proc. UK e-Science All Hands Meeting*, pages 1026–1032, 2005.

[261] M. Riding, J. Wood, K. Brodlie, J. Brooke, M. Chen, D. Chisnall, C. Hughes, N. W. John, M. W. Jones, and N. Roard. Towards an integrated framework for high performance visualization. In *Proc. UK e-Science All Hands Meeting*, pages 1026–1032, 2005.

[262] N. Roard and M. W. Jones. Agent based visualization and strategies. In *WSCG*, pages 63–70, 2006.

[263] N. Roard and M. W. Jones. Agents based visualization and strategies. In *Full Papers Proceedings of WSCG*, pages 63–70, 2006.

[264] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[265] M. Roggenbach. A new integration of process algebra and algebraic specification. *Third AMAST Workshop on Algebraic Methods in Language Processing (AMiLP-3)*, 2003.

[266] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *Computer Graphics (Proc. SIGGRAPH 2000)*, pages 343–252, 2000.

[267] F. Saffre and H. R. Blok. ”selfservice”, - a theoretical protocol for autonomic distribution - of services in p2p communities. In *ICAC*, pages 326–327, 2004.

[268] M. Sarkar and M. Brown. Graphical FishEye views of graphs. In *Proc. ACM CHI*, pages 83–91, 1992.

[269] M. Sarkar, S. S. Snibbe, O. J. Tversky, and S. P. Reiss. Stretching the rubber sheet: a metaphor for viewing large layouts on small screens. In *UIST ’93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 81–91, New York, NY, USA, 1993. ACM.

[270] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Proc. Eurographics Workshop on Rendering Techniques*, pages 319–328, 2000.

[271] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Ari, L. Encarnac, A. Gervautz, and W. Purgathofer. The studierstube augmented reality project. Technical report, TU Wien, 2000.

[272] H. Schwetman. *CSIM: a C-based process-oriented simulation language*. ACM Press, 1986.

[273] J. Shalf and E. W. Bethel. The grid and future visualization system architectures. *IEEE Computer Graphics and Applications*, 23(2):6–9, 2003.

[274] J. Shalf and E. W. Bethel. The Grid and future visualization systems architectures. *IEEE CG&A*, 23(2):6–9, 2003.

[275] J. A. Sharp. *Data Flow Computing*. Ellis Horwood, 1985.

[276] J. A. Sharp, editor. *Data Flow Computing: Theory and Practice*. Ablex Publishing, 1992.

[277] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Proc. IEEE Visualization*, pages 371–378, 1999.

[278] W. R. Sherman and A. B. Craig. *Understanding Virtual Reality: Interface, Application and Design*. Morgan Kaufmann, 2002.

[279] B. Shneidermann. Tree visualization with treemaps: a 2D space filling approach. *ACM Trans. Graphics*, 11(1):92–99, 1992.

[280] D. B. Skillicorn. A taxonomy for computer architectures. *IEEE Computer*, 21(11):46–57, 1988.

[281] M. Slater, A. Steed, and Y. Chrysanthou. *Computer Graphics and Virtual Environments*. Addison Wesley, 2002.

[282] M. Slater, M. Usoh, S. Benford, D. Snowdon, C. Brown, T. Rodden, G. Smith, and S. Wilbur. Distributed extensible virtual reality laboratory (DEVRL): a project for co-operation in multi-participant environments. In *Proceedings of the Eurographics workshop on Virtual environments and scientific visualization '96*, pages 137–148, London, UK, 1996. Springer-Verlag.

[283] S. Smith, T. Marsh, D. Duke, and P. Wright. Drowning in immersion. In *Proc. UK-VRSIG*, pages 1–9, 1998.

[284] F. Sogandares. Stone axes and warhammers: a decade of distributed simulation in aviation research. In *Proc. 16th Workshop on Parallel and Distributed Simulation*, pages 125–132, 2002.

[285] D. Song and E. Golin. Fine-grain visualization algorithms in dataflow environments. In *Proc. IEEE Visualization*, pages 126–133, 1993.

[286] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proc. Eurographics Workshop on Rendering Techniques*, pages 151–162, 2001.

[287] T. E. Starner. Powerful change part 1: batteries and possible alternatives for the mobile market. *IEEE Pervasive Computing*, 2(4):86–88, 2003.

[288] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial space-filling hierarchy visualization. In *Proc. IEEE Information Visualization*, pages 57–65, 2000.

[289] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: Scheduled linear image compositing for parallel volume rendering. pages 33–40.

[290] R. J. Stone. Haptic feedback: A potted history from telepresence to virtual reality. In *Proc. 1st Int. Workshop on Haptic Human-Computer Interaction*, volume LNCS 2058, pages 1–7, 2000.

[291] T. Straing, C. Linhoff-Popien, and K. Frank. Cool: A context ontology language to enable contextual interoperability. *Int. Conf. on Distributed Applications and Interoperable Systems*, 2893:236– 247, 2003.

[292] A. Sulistio, G. Poduvaly, R. Buyya, , and C.-K. Tham. Constructing a grid simulation with differentiated network service using gridsim. In *Proc. of the 6th Int. Conf. on Internet Computing*, June 2005.

[293] Sun Microsystems Inc. http://www.sun.com/solutions/hpc/compute/visualgrid/. last accessed: January 2008.

[294] Sun Microsystems Inc. http://www.sun.com/solutions/infrastructure/grid. last accessed: January 2008.

[295] P. M. Sutton and C. D. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Trans. Visualization and Computer Graphics*, 6(2):98–107, 2000.

[296] J. E. Swan, II, K. Mueller, T. Müller, N. Shareef, R. Crawfis, and R. Yagel. An anti-aliasing technique for splatting. In *Proc. IEEE Visualization*, pages 197–206, 1997.

[297] A. S. Tanenbaum. *Modern Operating Systems, 2nd Ed.* Prentice Hall, 2001.

[298] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (3rd Edition).*

[299] T. B. Team. An overview of the BlueGene/L supercomputer. Technical Report 0-7695-1523-X, IBM Research and IBM Rochester and the Lawrence Livermore National Laboratory, 2002.

[300] A. Telea and J. J. van Wijk. 3D IBFV: hardware-accelerated 3D flow visualization. In *Proc. IEEE Visualization*, pages 233–240, 2003.

[301] S. Teller, C. Fowler, T. Funkhouse, and P. Hanrahan. Partitioning and ordering large radiosity computations. In *Computer Graphics (Proc. SIGGRAPH 94)*, pages 443–450, 1994.

[302] TGS. Amira. http://www.tgs.com. last accessed: January 2008.

[303] H. Theisel and M. Kreuseler. An enhanced spring model for information visualization. *Computer Graphics Forum*, 17(3):335–343, 1998.

[304] T. Theogaris. *Algorithms for Parallel Polygon Rendering*. Springer-Verlag, 1989.

[305] S. M. F. Treavett and M. Chen. Pen-and-ink rendering in volume visualization. In *Proc. IEEE Visualization 2000*, pages 203–210, 2000.

[306] N. Tsarmpopoulos, I. Kalavros, and S. Lalis. A low-cost and simple-to-deploy peer-to-peer wireless network based on open source linux routers. In *TRIDENTCOM '05: Int. Conf. on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 92–97, Washington, DC, USA, 2005. IEEE Computer Society.

[307] G. Tziallas and B. Theodoulidis. Building autonomic computing systems based on ontological component models. In *Proc. 14th Int. Workshop on Database and Expert Systems Applications*, pages 674–680, 2003.

[308] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Trans. Visualization and Computer Graphics*, 3(4):370–380, 1997.

[309] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, (Q4):14, 1999.

[310] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international inproceedings on Management of data*, pages 44–53, New York, NY, USA, 1990. ACM Press.

[311] C. Upson, T. Faulhaber Jr., D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE CG&A*, 9(4):30–42, 1989.

[312] R. van Liere, J. D. Mulder, and J. J. van Wijk. Computational steering. *Future Gener. Comput. Syst.*, 12(5):441–450, 1997.

[313] M. van Steen, A. S. Tanenbaum, A. S. Kuz, and H. J. Sips. A scalable middleware solution for advanced wire-area web services. *Distributed Systems Engineering*, 7:34–42, 1999.

[314] J. J. van Wijk and R. van Liere. An environment for computational steering. In G. M. Nielson, H. Müller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques*, pages 89–110, 1997.

[315] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric datasets. *13th IEEE Visualization 2002 (VIS'02)*, 2002.

[316] J. Veenstra and N. Ahuja. Line drawings of octree-represented objects. *ACM Trans. Graphics*, 7(1):61–75, 1988.

[317] P. Vidales, G. Mapp, F. Stajano, J. Crowcroft, and C. J. Bernardos. A practical approach for 4g systems: Deployment of overlay networks. In *TRIDENTCOM '05: Int. Conf. on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 172–181, Washington, DC, USA, 2005. IEEE Computer Society.

[318] VisIt. http://www.llnl.gov/visit/. last accessed: January 2008.

[319] J. S. Vitter. External memory algorithms. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 119–128, New York, NY, USA, 1998. ACM Press.

[320] J. S. Vitter. External memory algorithms and data structures. pages 1–38, 1999.

[321] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[322] VNC. http://www.realvnc.com/. last accessed: January 2008.

[323] B. von Rymon-Lipinski, N. Hanssen, T. Jansen, L. Ritter, and E. Keeve. Efficient point-based isosurface exploration using the span-triangle. In *Proc. IEEE Visualization*, pages 441–448, 2004.

[324] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005.

[325] I. Wald and H.-P. Seidel. Interactive ray tracing of point based models. In *Proc. Symposium on Point-based Graphics*, pages 9 – 16, 2005.

[326] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utilities functions in autonomic systems. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 70–77, 2004.

[327] S. Wang and A. Kaufman. Volume sampled voxelization of geometric primitives. In *Proc. IEEE Visualization*, pages 78–84, 1993.

[328] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[329] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics (Proc. SIGGRAPH 90)*, 24(4):367–376, August 1990.

[330] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[331] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. AK Peters, 1992.

[332] S. R. Whitman. A survey of parallel algorithms for graphics and visualization. In *Proc. Int. Workshop on High Performance Computing for Computer Graphics and Visualisation*, pages 3–22, 1996.

[333] A. Wilen, J. P. Schade, and R. Thornburg. *Introduction to PCI Express: A Hardware and Software Developer's Guide*. Intel Press, 2003.

[334] J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graphics*, 11(3):201–227, 1992.

[335] J. Wilhelms, A. van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Proc. IEEE Visualization*, pages 57–63, 1996.

[336] A. S. Winter and M. Chen. vlib: a volume graphics API. In *Proc. Volume Graphics*, pages 133–147, 2001.

[337] C. M. Wittenbrink and M. Harrington. A scalable MIMD volume rendering algorithm. In *Proc. 8th Int. Parallel Processing Symp.*, pages 916–922, 1994.

[338] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smartpointers: personalized scientific data portals in your hand. In *Proc. ACM/IEEE Conf. on Supercomputing*, pages 1–16, 2002.

[339] P. Wong. Visual data mining. *IEEE CG&A*, 19(5):20–21, 1999.

[340] J. Wood, K. Brodlie, and J. Walton. gViz–visualization and steering for the Grid. In *Proc. e-Science All Hands Meeting*, 2003.

[341] J. Wood and H. Wright. Steering via the image in local, distributed and collaborative settings. In *UK e-Science Programme All Hands Meeting*, pages 441–444, September 2005.

[342] J. D. Wood, K. W. Brodlie, and H. Wright. Visualization over the world wide web and its application to environmental data. In *Proc. IEEE Visualization*, pages 81–86, 1996.

[343] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *IEEE CG&A*, 21(4):62–69, 2001.

[344] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[345] X. Zhang and X. Qin. Performance prediction and evaluation of parallel processing on a NUMA multiprocessor. *IEEE Trans. Software Engineering*, 17(10):1059–1068, 1991.

[346] H. Zhou, M. Chen, and M. Webster. Comparative evaluation of visualization and experimental results using image comparison metrics. In *Proc. IEEE Visualization*, pages 315–322, 2002.

[347] M. Zwicker, H. Pfister, J. Baar, and M. Gross. EWA volume splatting. In *Proc. IEEE Visualization*, pages 29–36, 2001.

List of Figures

List of Tables