VISUALIZING THE PERFORMANCE OF PARALLEL PROGRAMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

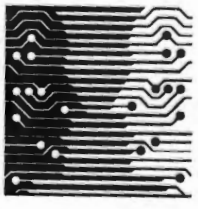
By

Peter Hinz

September 1995

Supervised by

H.A. Goosen



Abstract

The performance analysis of parallel programs is a complex task, particularly if the program has to be efficient over a wide range of parallel machines. We have designed a performance analysis system called Chiron that uses scientific visualization techniques to guide and help the user in performance analysis activities. The aim of Chiron is to give the user full control over what section of the data he/she wants to investigate in detail. Chiron uses interactive three-dimensional graphics techniques to display large amounts of data in a compact and easy to understand/conceptualize way. The system assists in the tracking of performance bottlenecks by showing data in 10 different views and allowing the user to interact with the data.

In this thesis the design and implementation of Chiron are described, and its effectiveness illustrated by means of three case studies.

Acknowledgements

I would like to thank Philip Machanick for using Chiron to do performance analysis on his SpaceLib memory system library and for suggesting many improvements on the user interface and the analysis tools.

Thanks go to Dieter Polzin for introducing me to the Chiron project and for writing and documenting the first prototype system.

Thanks also go to Jack Veenstra for writing a fast and easy-to-use tracing system, Mint.

Special thanks go to Aleks Strez for writing Chiron's symbol table tracking system.

Many thanks go to my supervisor Henk Goosen for his time and effort in producing this thesis

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel programs are hard to write, and harder to optimize. The shared-memory paradigm of programming multiprocessor applications has emerged as that preferred to the message passing model, because in most cases it makes it possible to parallelize existing sequential algorithms with reasonable effort [31]. Unfortunately the shared-memory paradigm hides many of a machine's architectural details from the programmer by presenting the abstraction of a large shared address space. The automatic data communication in shared-memory machines hides the actual data communication and migration cost from the programmer, and complicates the performance optimization of shared-memory programs. To obtain high performance from a parallel machine a program has to show high spatial, temporal, and processor locality [1]. Unless proper care is taken in data layout and algorithm design, a parallel program may spend a large portion of its execution time blocked on memory system stalls.

Two factors are exacerbating the performance problems on parallel machines. First, processor speeds have increased by more than two orders of magnitude over the last decade, but main memory speeds have barely increased by a factor of two [21]. To close this performance gap, high–performance computer systems are designed with complicated memory systems that include multi-level caches and interleaved main memory. A consequence of these complex memory systems is that cache miss latencies have become extremely costly when counted in processor clock cycles. Thus a program can spend a large portion of its

execution time waiting for the memory system to be put into a coherent state or for data to be loaded into caches. The problem of keeping the contents of the memory system coherent is however a major obstacle, which has been the focus of much research [5] [7].

Second, scientific applications are written more often using object–oriented programming techniques [12]. To resolve run–time information about objects, object–oriented programs, especially C++ programs, generate large numbers of extra memory references [8]. Unless these extra memory references can be offset by better cache behavior, the performance of object-oriented code will suffer.

To improve parallel program performance the programmer has to reduce the memory system stall time by ensuring that a program's data has high spatial, temporal, and processor locality. This is a daunting task for the application programmers as they are often expert in whatever problem area their programs are solving, but not necessarily experts in computer architecture and performance tunning. To this end programmers use tools that aid in writing efficient parallel programs, for example performance analysis and debugging tools, in addition to tools which are used to ensure correct functional behavior.

1.1 Performance optimization tools

Performance analysis tools have advanced greatly since the time when *prof* [45] was the standard profiling tool. The complexity and capabilities of analysis tools have increased to such an extent that performance analysis tools, like *SM-prof* [6], have become an essential part of the parallel program design cycle. We believe that a typical software development cycle should contain the following three steps: design, coding, and performance analysis.

1.2 Scientific visualization techniques

Even though advanced performance analysis tools, like *MemSpy* [17], *ParaView* [49], and *SM-prof* [6], use graphics to display the performance data these tools have not yet fully exploited the use of scientific visualization techniques to display large amounts of data. We considered a range of visualization tools, like IRIS Explorer [46], Data Explorer [3], Khoros

[41], AVS [50], Glyphmaker [42], and Obliq-3D [36], to determine what type of visualization techniques would be applicable to performance data.

Visual programming techniques based on the dataflow model form the basis of many visualization systems [3] [55] [41] [50]. These systems were made for flexibility and general use, but our aim was to design a system specificaly tailored to performance analysis. The basic criterion in our search of a visualization system was user-graph interactivity. The user had to be able to easily select any part of an image and find out more about the data represented by that part. Of all the systems we investigated only AVS provided a high level of user-graph interactivity, but the system's interaction capabilities were not flexible enough.

1.3 The Chiron system

The Chiron parallel program performance visualization system, which is presented in this thesis, is based on a fixed rendering pipeline and uses presentation techniques from the scientific visualiztion field to help the programmer understand the behavior of a program, and to find and eliminate performance bottlenecks. Chiron uses interactive three-dimensional graphics to display large amounts of performance-related data. The aim of these graphics is to enable the human visual system to identify performance problems visually and to gain insight from the visual information presented by these graphs.

The user has full control over the orientation, level of detail, and zoom factor of the graphics. This enables the user to explore the data set and possibly identify performance problem areas. Chiron allows for full user-graphics interaction, i.e., the user can select a point on a graph and find out more about the data which is represented by that point. This high degree of user interaction is aimed at enabling the user to quickly find performance bottlenecks.

The Chiron system tries to display as much of the performance data simultaneously as is possible. The data is not preprocessed into a more manageble size because the problem identification techniques rely on the visual patterns which specific performance problems have.

Chiron has a variety of standard ways, called views, of looking at performance data. All views occupy a common three-dimensional space and can be placed anywhere in that space. The views are divided into three areas: Global views, correlation views, and temporal views. Global views give general performance statistics. Correlation views show more detail about specific performance aspects and are used to form data correlations between two or more data sets. Temporal views give detailed information about the behavior of architectual or program constructs.

1.4 How is performance data obtained?

To be able to analyse a program's performance it was necessary to obtain information about its execution behavior. In particular, we were interested in obtaining a detailed account of a program's memory accesses, the resultant cache activity, and process synchronization behavior. The problem at hand was how to obtain that information without changing the actual aspects we wanted to measure.

We opted for a full software simulation of a parallel processor target machine's architecture. This allowed us to closely monitor the behavior of the memory system and to make a detailed record of all process synchronization activity. The *Mint* [52] system simulator was used for this purpose. The Mint simulator interprets the target program's instructions. Since the target program is interpreted and the whole simualtion is controlled by a timed event-wheel, the simulator can record relevant information about instruction execution without changing the program's behavior.

The ParaDiGM [16] memory system simulator was used to obtain information about the memory system behavior of a program's execution. ParaDiGM simulates a multi-level memory system architecture with configurable cache parameters. The Mint simulator communicates all memory references to the memory system simulator, which in turn records memory system activity. We were particularly interested in first-level and second-level cache movement.

Other components of the Mint simulator are used to record events of interest to performance analysis. A memory allocation tracking system is used to associate useful names

with allocated portions of memory. Chiron users can use this information to quickly identify memory locations and dynamically allocated data structures. A stack frame tracking system records all function entries and exits. This allows the Chiron user to identify all functions that were executed at any given time during a program's execution. A process synchronization tracker records all synchronization activity. The source line tracker maps a source code line number with every location in a program's executable.

1.5 Aim of the investigation

The objective of the investigation reported in this thesis was to show that with the aid of Chiron the visual analysis of shared-memory parallel-program performance data is effective and that performance bottlenecks can easily be identified. The visual analysis techniques are based on the premise that the human visual system can identify patterns in an image and draw conclusions about behavior represented by those patterns. It is postulated that Chiron permits the analyst to identify *visual signatures*, instantly obvious patterns, in the visualizations and map these directly to specific performance problems.

1.6 The thesis layout

The Chiron performance visualization system consists of two components, the trace generator, and the visualization package. Chapter 2 reviews existing systems, a range of performance analysis tools and visualization systems being described, to gain insight into what the field had to offer and on what basis our approach to the problem at hand was structured.

In chapter 3 the Chiron system is presented, including the user interface, the different views, their interactions, and their interpretations. Each view is described in detail and if applicable what visual signatures could be identified.

In chapter 4 the execution trace generation is described.

In the next three chapters the experiences gained in visual performance analysis are presented by means of three examples. First, a simple parallel vector multiplication program

shows how quickly synchronization problems and false-sharing can be identified. Second, the *Barnes-Hut* program from the SPLASH [48] benchmark suite is used to show how correlations can yield a high amount of extra information. The *Barnes-Hut* example also shows the power of visualization in identifying true- and false-sharing memory behavior. The third test case is a highly optimized version of the *WATER* simulation, also from the SPLASH benchmark suite, which has been choosen to show what kind of subtle behavior Chiron can easily expose.

Chapter 6 presents the conclusions drawn from the study and suggestions for future work.

Chapter 2

Related work

This chapter outlines two areas of computer science research that have been combined to help create the Chiron performance visualization system. First, performance debugging tools were investigated to determine what are the current state of the art performance debuggers. Second, a study of data visualization systems was made to determine what types of visualizations exist and what visualization techniques could be applied to performance data analysis.

2.1 Performance Debuggers

Debugging tools have been around for quite a few years now, the main thrust behind the development of these tools being the idea of correctness debugging. In recent years a step beyond simple correctness debugging has been taken, this step led to application performance debugging. Many performance debuggers now exist, covering points along a spectrum from very high–level, low–overhead tools to very detailed, high–overhead tools. At the low end of this spectrum are tools like *prof* and *pixie* which are intended to produce simple, high–level statistics with minimal performance overhead. At the high end tools like *SM–prof* and *Chiron* are intended to give more detailed information with an increase in overhead. This section discusses a selection of tools covering selected points in this spectrum.

2.1.1 Prof

Prof [45] is a commonly used execution profiler. It gives a hierarchically arranged profile of the execution time of a program's most expensive procedures. *Prof* can handle programs that use **sproc** to create threads within a job. By looking at *prof*'s high-level view we can find the procedures which would have the greatest potential for optimization.

However, in contrast to Chiron, *prof* does not distinguish between CPU computation time and memory system time, and therefore does not help in locating memory system bottlenecks. *Prof* can only be used to get an overview of where a program spends most of its CPU time.

2.1.2 Pixie

Pixie [44] is only slightly more complex than *prof*. *Pixie* partitions a program into basic blocks and counts the execution time of each basic block. Analysis tools, like *pixstats*, give a hierarchically arranged profile of the execution time of each basic block in a program. By looking at the most expensive basic blocks we can identify that small region within a procedure which might have the greatest potential for optimization.

The purely code oriented nature of *Pixie* also does not provide enough insight for the analyst in finding memory system bottlenecks. *Pixie* does handle parallel programs, provided they use **sproc** to handle the threads with in a job.

The clear advantage of using *prof* and *Pixie* is, that these systems allow for a quick analysis of programs.

2.1.3 MTOOL

MTOOL [13] [14] is a system specifically designed to detect memory system bottlenecks in both sequential and parallel programs. *MTOOL*'s basic performance metric is the difference between a program's ideal execution time, with an ideal memory system, and the actual execution time, with non–ideal memory system behavior. This difference is the amount of

execution time for which the processor was stalled due to memory delays. This performance information is presented for loops and procedures within the program.

While *MTOOL* is a useful tool for focusing attention on the primary memory bottlenecks in the code, since it is procedure and loop oriented, it most often does not provide insight into which data objects were responsible for the bottlenecks. Chiron performance data analysis is not limited to simple code constructs, but also allows detailed analysis of individual lines of source code.

2.1.4 MemSpy

MemSpy [17] is a tool that helps programmers identify and fix memory system bottlenecks in both sequential and parallel programs. A new concept introduced in *MemSpy* is the notion of data-oriented, in addition to code-oriented, performance tuning. For both source code and data objects *MemSpy* provides detailed performance information such as cache miss rate, cause of cache misses, information on cache validation and local versus remote memory misses. In addition to this information *MemSpy* helps in pinpointing memory system bottleneck causes, such as poor spatial locality and interference among data structures. *MemSpy* lets the user view both code- and data-oriented statistics in a concise way.

However, *MemSpy* uses mainly two–dimensional graphics to display performance data, rather than the three–dimensional graphics used by Chiron. Locating memory bottlenecks in *MemSpy* is a process of refinement, the analyst starts with some high–level program information which will focus attention on problem areas in the application. Then *MemSpy* guides the analyst to the application areas which cause the memory bottlenecks. What sets *MemSpy*, and Chiron, apart from other performance debuggers are the data-oriented statistics. The code-i and data-oriented statistics provide orthogonal views of program performance.

2.1.5 Upshot

The *Upshot* [22] performance analysis tool can be used to visualize a *logfile* of significant events in the order in which they have occurred during the execution of a parallel program.

Upshot, like Chiron, provides a graphical view of events aligned on the parallel time lines of individual processes. The state of individual processes can be defined and displayed with respect to the events in the *logfile*. *Upshot* has a very limited use, mainly due to only one view being used to display the events, as opposed to Chiron's multitude of views and extensive viewing flexibility. The main feature of *Upshot* is the graphical representation of the program behavior.

2.1.6 PIE

The *PIE* [28] system makes a program's functional and performance behavior visible through automated system support. *PIE* shows performance data in histogram and time-line formats for quick detection of problem areas. The main view shows the computational components of a program against a time line. The user can select the components in this graph and find out which computational component was active at the given time.

PIE only maps performance data onto program constructs, Chiron on the other hand also maps performance data onto data constructs. PIE has a limited data source and presentation facility, a user can only determine if a specific program construct was active or inactive at a given time. Chiron, on the other hand, has a wider source data base and more and more flexible data visualization techniques.

2.1.7 ParaView

ParaView [49] is a performance debugger designed to assist programmers in identifying and correcting the performance problems in parallel programs. *ParaView* has an easy–to–use X Windows environment, and provides methods for detecting problems with poor cache performance, false sharing, synchronization and load balancing. *ParaView* users can freely adjust the granularity of the displayed trace data, which is displayed in five views.

2.1.8 SM-prof

The *SM-prof* [6] performance visualization tool can be used to identify and analyze performance bottlenecks related to cache coherence and shared data access patterns. *SM-prof*,

like Chiron, links source code lines causing performance degrading access patterns to the visualization, which enables the user to follow the execution of a program and possibly track the exact code responsible for a performance bottleneck. *SM-prof* focuses on the time–varying behavior of the program, dividing the program execution into time slots, and uses two–dimensional graphics to display various counts of memory reference events during these time slots.

2.1.9 Other performance debuggers

Much like Chiron some of the latest performance debuggers enable a user to eliminate excess synchronization in shared-memory parallel programs [40]. The work of Alva L. Couch and David W. Krumme [10] has shown that portable execution trace debuggers are important, and with the inclusion of Mint in the Chiron project we have followed this trend.

2.1.10 Summary

In summary, many points of the tradeoff scale are covered by current performance debugging tools. This thesis builds upon the *Chiron* performance visualization tool, which occupies the same tradeoff spot as *MemSpy*, *ParaView* and *SM-prof*, but has the added advantage of using 3D abstract data visualization techniques to guide the analyst to a variety of possible performance bottlenecks.

2.2 Visualization Systems

A range of visualization systems have been investigated and categorized into two general areas, according to whether or not the systems base their rendering pipeline on the dataflow model.

Visualization systems that use the dataflow model are:

- IRIS Explorer [46] [3]
- Data Explorer [3]

- Khoros [55] [56] [41] [51]
- Application Visualization System [50] [2] [4]

Visualization systems that do not use the dataflow model are:

- Glyphmaker [42]
- Obliq-3D [36]

2.2.1 The dataflow model

The dataflow model is a compromise between the flexibility of a graphics library and the ease of use of an off-the-shelf visualization package. While off-the-shelf visualization packages are easy to use, they are often specialized and therefore often impose limits on the visualizations. The greatest limitation is that they usually only allow the user to *visualize* the data but not *interact* or *interrogate* it [54].

A graphics library, on the other hand, provides the flexibility and power to create almost any visualization with its accompanying interaction and interrogation techniques. However, one of the disadvantages in using a graphics library is that there is a significant time delay between when the visualization is conceived and when it can be analysed and evaluated. The creator of a visualization is also required to have programming and computer graphics knowledge as libraries mainly deal with primitives and not in high-level abstractions.

The dataflow model is based on visual programming techniques. The user defines a visual program by creating a network of interacting modules. A module is understood to be a black box that accepts input data, processes it, and produces some output. The interconnecting network is responsible for providing the data exchange among modules.

2.2.2 IRIS Explorer

IRIS Explorer is a visualization system based on the dataflow model. The types of modules which form the basis of the visualization can be divided into four groups [54].

The **input** modules read data from a file, pipe or another application which is producing data, and make this data available to the connection network.

The **filter** modules perform some form of data filtering or modification.

The **transform** modules transform the input data into a geometric representation.

The **output** module is most often the renderer which produces some form of graphic on the output device.

The IRIS Explorer visualization system consists of three main sections:

Map Editor: The Map Editor is used to create and modify maps and module connection networks.

DataScribe: DataScribe is a conversion utility that allows data to be converted between Explorer's format and a number of other data formats.

Module Builder: The Module Builder allows the user to create custom modules.

The creation of a visualization network is done through selecting a number of modules from a module library and placing them on the work area using a simple drag and drop interface. The user then defines the data paths by either creating or destroying connections between the modules.

Each module can have a number of knobs, dials, sliders, etc. These allow the user to modify parameters which control, for example, a *filter* or *transform* module's actions. They also form the user interface of the visual program. While this interface is suitable for many forms of visualizations, it does not provide an easy facility for directly interacting with and interrogating the visualization. Each user interaction results in the firing of modules which can result in the visualization being rerendered.

The lack of interrogation facilities in IRIS Explorer limits its usefulness in some application areas.

2.2.3 Data Explorer

Data Explorer is a visual application builder. It is based on a dataflow model, with applications being written by connecting modules by means of a network [3].

Unlike other dataflow model systems, modules do not have a default graphical user interface, but instead use type-ins to change parameter values. Graphical widgets can be placed on a separate control panel and connected to parameters to create a custom user interface.

The visualization system allows for an application to be run without the need to use the visual program editor. In such a case only the control panel and the viewport are displayed. This provides a method of creating custom applications where the end user need never interact with the visual program editor.

Camera movement within the viewport is done via the mouse, however scaling, rotation, and transformation have to be done using corresponding interaction modules. Data Explorer does not provide comprehensive facilities for picking and direct data manipulation, as all interaction takes place though the control panel and modules.

2.2.4 Khoros

Khoros is a data visualization system that is partially based on the dataflow model and partially on an event-driven system. The system makes use of the Cantata visual programming language.

Cantata

Cantata is a visual programming language which is programmed by placing *glyphs* or *operators* in a *workspace*. The glyphs are connected to form a network that defines the visual program.

Unlike visual programming systems used in other visualization systems, the Cantata network does not only indicate the data flow but is also used to control the execution. A control connection can be established between two glyphs to transfer synchronization information.

This facility can be used where the order of module execution is not dictated by the data flow, for example in the case of parallel data paths in the connection network [55].

Cantata includes all the programming constructs of a textual programming language, such as allowing users to create loops using counters or conditionals, conditional branching, and switches, as well as merging and splitting of data flow.

Workspace can be encapsulated to form an application that can be run without displaying the visual programming glyphs [55, 56]. This is comparable to creating a "stand alone" application in AVS 2.2.5 and IRIS Explorer. These encapsulated workspaces can also be used as procedural glyphs.

Visualization Tools

While Khoros has been used extensively for the visualization and teaching of image processing [41] it does have the tools to create 3D visualizations, namely the geometry toolbox. However, the Khoros system has been more focused on image processing and DSP visualization and as a result these tools are still in the development phase [51].

Khoros provides facilities to create simple 3D geometric shapes, such as isosurfaces and spheres. Tools for manipulating color maps and creating slices through the data set are also provided. The user has control over camera parameters, a number of different shaders, and object transformations.

Khoros has focused more on 2D visulizations with 3D, and higher dimensions, having been added only recently. The system does not support easy to use 3D building blocks which could be used to create complex visualizations.

The interaction in a 3D visualization is limited to controlling the inputs to the visualization and does not allow for direct interaction with the 3D graph itself.

2.2.5 Application Visualization System

AVS is a visualization system that is widely available on many platforms ranging from work stations to supercomputers. Its aim is to provide a relatively simple method of allowing

non-experts to make use of complex 3D graphics [50]. AVS is based on the dataflow model with some extensions that provide the ability to make stand alone applications.

AVS Components

AVS consists of five interactive parts, namely the Geometry Viewer, the Image Viewer, the Graph Viewer, the Data Viewer, and the Network Editor [3].

- **The Geometry Viewer:** The geometry viewer is used for rendering views of 3D geometric primitives. These primitive include point, lines, polygons, spheres, etc. [2]

  These primitives are used in conjunction with lights, cameras and other property information to create 3D scenes which can be rendered into either a single or multiple windows.

- **The Image Viewer:** The image viewer is designed for use in image processing applications: it provides functionality for displaying and arranging images, as well as a limited set of image processing functions.

- **The Graph Viewer:** The graph viewer is a 2D graph viewer.

- **The Data Viewer:** The data viewer is used to create visualization tutorials.

- **The Network Viewer:** The network viewer is equivalent to IRIS Explorer's map editor. It is used to modify and create networks of nodes that define the dataflow model.

The Graphical User Interface

AVS uses X11 as the basis for all its widgets. AVS, like IRIS Explorer, has buttons, sliders, dials, etc., widgets which can be used to scale, rotate, or otherwise modify objects which appear in an AVS viewer.

Each viewer has an interaction window that displays the options associated with that viewer. The viewers also use interactive picking, i.e. selection of objects in the viewport. This

eliminates the need for moving the mouse pointer out of the viewport, selecting an object by using some widget or list, and then moving back to the viewport. This, combined with the ability to select options directly from the interaction window, simplifies the use of the viewers [3].

AVS allows a user to change the layout of the graphical interface of a visual program. It also allows for the grouping of a number of interface widgets, usually the most important ones, on a page and to hide others, thereby simplifying the overall user interface. This feature is also present in IRIS Explorer where a number of modules can be grouped into a single module.

AVS summary

As AVS is based on the dataflow model there are naturally many similarities to other dataflow systems, such as IRIS Explorer and Khoros. AVS, however, does provide better data interaction by means of directly interrogating and interacting with a viewer. The interaction flexibility and the build in *Command Language* make up a very power full and effective visualization system.

2.2.6 Glyphmaker

Glyphmaker [42] is a visualization system for visualizing multi-variate data sets. Instead of prescribing a visualization method that would be applicable to all data sets, Glyphmaker uses glyphs and allows the user to bind properties of these glyphs to fields in a data set.

Glyphs can be viewed as 3D icons that have a number of properties which define their appearance: these properties include size, shape, orientation, color, position, etc. Glyphmaker allows the user to bind these properties to a field in the data set, and view the data in 3D space.

Glyphmaker was designed to be used by non-experts with little or no programming experience to create their own custom visualizations of data. It was built on top of IRIS Explorer, using the underlying dataflow model as well as specifically written custom modules. The visual programming is done via a point and click interface.

Glyphmaker was designed as an exploration tool. The focus was on users who do not fully understand their data and who do not always know which visualization technique would increase their understanding of it.

Design issues

The following is a list of objectives that the designers of Glyphmaker took into account while designing the system.

Detailed control: Being able to bind individual data elements to a single glyph gives the user detailed control over the visulization.

Highly responsive controls: The responsiveness of interface controls does not only refer to the time lag between adjusting a control and seeing the result in the rendered visualization, but also includes the time taken to find and adjust the control. In Glyphmaker the controls are prioritised, with those controls of a high priority being conveniently placed for the user.

Interactive visualization: For a visualization to be interactive it has to respond to user control in a timely fashion.

Visualizing large data sets has a negative impact on performance. To increase performance Glyphmaker provides a facility to reduce glyphs to points or not to render them at all.

Focus and conditions: Statistical graphics research provides a number of guidelines as to how to approach visualizing multi-variate data sets. Many of these guidelines focus on reducing the data set to a more pertinent subset. Glyphmaker users can use the conditional box to eliminate large portions of a visualization.

Correlative linking: Correlative linking is a process whereby a number of views of complex data sets are linked. The correlation can be shown by means of animation or by highlighting linked features in glyphs.

Glyphmaker modules

The visualization system consists out of four modules.

Read Module: The problem with many visualization systems is that they need to use data from a variety of different sources. This data has to be massaged into some common format which can then be used in a visualization system.

Glyphmaker uses a system of self-describing files. Each file has a header that gives the description of each variable. The description for a variable includes the name, primitive type, number of instances, maximum and minimum value.

The Read Module will accept data files in straight ASCII form, ASCII header and binary data or straight binary data.

Glyph Editor: The glyph editor is a simple 3D editor that allows the user to create and edit glyphs. The editor can also be used to combine glyphs into compound glyphs.

To construct custom compound glyphs the user is provided with some geometric primitives. these include points, lines, spheres, cuboids, cylinders, cones and arrows.

Glyph Binder: The binder presents the user with a list of active elements as well as a list of all variables. The user can then use a simple point and click interface to link variables to active elements.

In addition a text widget is associated with every active element and variable. This widget displays the maximum or minimum values for the variable or active element, initially these contain default values but they can be modified by the user.

Conditional Box: The conditional box allows the user to create a condition that will isolate a certain spatial region for closer examination.

2.2.7 Obliq-3D

Obliq-3D is a 3D animation system that allows for animations to be created and modified easily and quickly. This is achieved by using the Obliq interpreted language in conjunction with the Anim3D animation library [36].

The problem in creating animations of abstract information is that it requires not only animation skill but artistic and communicative skills as well. Couple this with an iterative design process and the need for an animation system with a fast turnaround time is evident.

Obliq-3D is based upon two basic concepts: graphical objects and properties.

Graphical Objects

Graphical objects are geometric shapes, such as cubes, spheres, cones, etc., lights, cameras, groups, which group together a number of graphical objects, and root nodes, which form the base of an object tree.

These objects can be linked into a directed acyclic graph to describe a scene. The scene, or object tree, is created by making a graphical object the child of another. The root of the tree forms the starting point of the rendering process.

The render process in Obliq-3D is based on a *damage-repair* model. Whenever the scene graph is modified, it is flagged to be damaged, as the scene represented in the view port no longer corresponds to the scene graph. The animation thread detects the damaged scene graph and repairs it by rendering the scene to the view port.

Properties

Every graphical object has associated with it a number of properties, which include color, location, scale factor, etc. A property consists of a name and a value. Properties do not only affect the graphical objects they are attached to, but also all descendant objects.

Properties are also time-variant. A property can be defined to have an initial value, a final value, and a time frame during which the transition is to take place. As properties include such values as position and size, this facility make animation very easy to define.

Not all property values can change with time. To deal with this problem Obliq-3D defines four types of properties, namely constant, asynchronous, synchronous and dependent. Constant property values are time invariant, while asynchronous property values change irrespective of other property values, synchronous property values change when signalled, and dependent property values change depending on another property value.

2.2.8 Summary

The Chiron performance data visualization system is not based on the dataflow model. It has a fixed render pipeline and a range of fixed data visualization methods. The system could be best compared to Vis-5D [23], as being an application area-specific visualization tool.

2.3 Related Debugging and Visualization Research

Much research has gone into the field of combining execution trace generation and performance debugging. The work of [47], [10], [33], [32], [37] and [43] has produced a variety of performance debugging and visualization environments. Much like Chiron these systems combine a whole range of techniques and tools to create a useful environment.

A substantial amount of work has been done in the field of performance debugging visualization . [19], [53], [18], [25], [34], [38], [29], [11] and [24] all have build systems to analyse and visualize the performance of parallel programs. These systems represent a major step away from adhoc performance analysis by supplying an execution trace generator and a data visualization and analysis tool. Like Chiron these systems fall into a new type of debugging category, that of visual performance debuggers.

Chapter 3

Chiron

Chiron is a performance data visualization tool. What sets Chiron apart from other performance analysis tools (as mentioned in Chapter 2) is the use of interactive three– dimensional (3D) graphics to display large amount of performance data. Chiron also tracks both memory system and synchronization inefficiencies, and relates the performance data to source code lines and data objects, making it easy for the user to focus on modifications to the source code and data structures. [15]

Traditional performance evaluation tools usually process large amounts of performance data to obtain a managable data set, which would then be analysed. Unfortunately, processing the data (by calculating averages, for example) also removes useful information from the data. Chiron presents performance data visually, and can therefore display more data than is feasible with other techniques.

The display techniques used draw from experience gained in the field of scientific visualization and are designed to enable the human visual system to derive insights from the spatial, color, and texture information in the images. Chiron's data representation is based on the premise that the human visual system can pick up patterns in an image faster than in a set of numerical data. The user can identify *visual signatures* (instantly obvious patterns) in the images and map these directly to a performance problem.

These *visual signatures* are the key idea behind Chiron, and the system is designed to give the user the ability to see as much of the relevant performance data in one glance as is

possible. Chiron provides different views on the same data, thus enabling the user to detect patterns and correlations visually.

To display data about each object or each source code line is difficult, given that a typical display contains some one million pixels. Displaying a multi–million point data set on such a display becomes an interesting mapping excersise. Chiron provides several overview displays that can give a programmer a quick feel for the most important areas of performance behavior on which to focus. The analyst can then interactively select fine–grain displays to focus on the most interesting features.

Chiron incorporates a variety of standard ways of looking at the data, called *views*. Each view is implemented and displayed as a separate object in a common 3D space, and each view can be interactively manipulated, for examples, scaled, rotated, translated, or level–of–detail toggled.

The use of three dimensions is important for two reasons. First, it enables us to display a much larger amount of data than would be possible in two dimensions. Second, it provides an extra dimension in which we can display interesting correlations. The interactive nature of Chiron's user interface complements the 3D aspect by allowing the user to explore interesting aspects of the displayed images, while ignoring irrelevant detail, further improving our ability to display large amounts of data.

There are three types of views: global views, correlation views, and temporal views. The global views serve as roadmaps to performance problems by presenting an overview of a particular metric of interest. The analyst can refer to the global views to decide whether a particular visual feature represents program behavior that is likely to have a large impact on the performance.

The correlation views are more detailed than the global views, typically focusing on one object instance or program statement. Suitable correlation views are selected by the user after noticing an interesting feature on a global view. Chiron also provides temporal views that provide detailed information about the behavior in time of an object or code fragment.

This chapter outlines the Chiron user interface and the different visual signatures that can be identified on each view.

3.1 Chiron's user interface

The Chiron performance visualization system presents the user with a view into a three-dimensional world with tools and short cut buttons and gadgets surrounding the main display window. Chiron only ever presents the user with a maximum of two windows at a time.

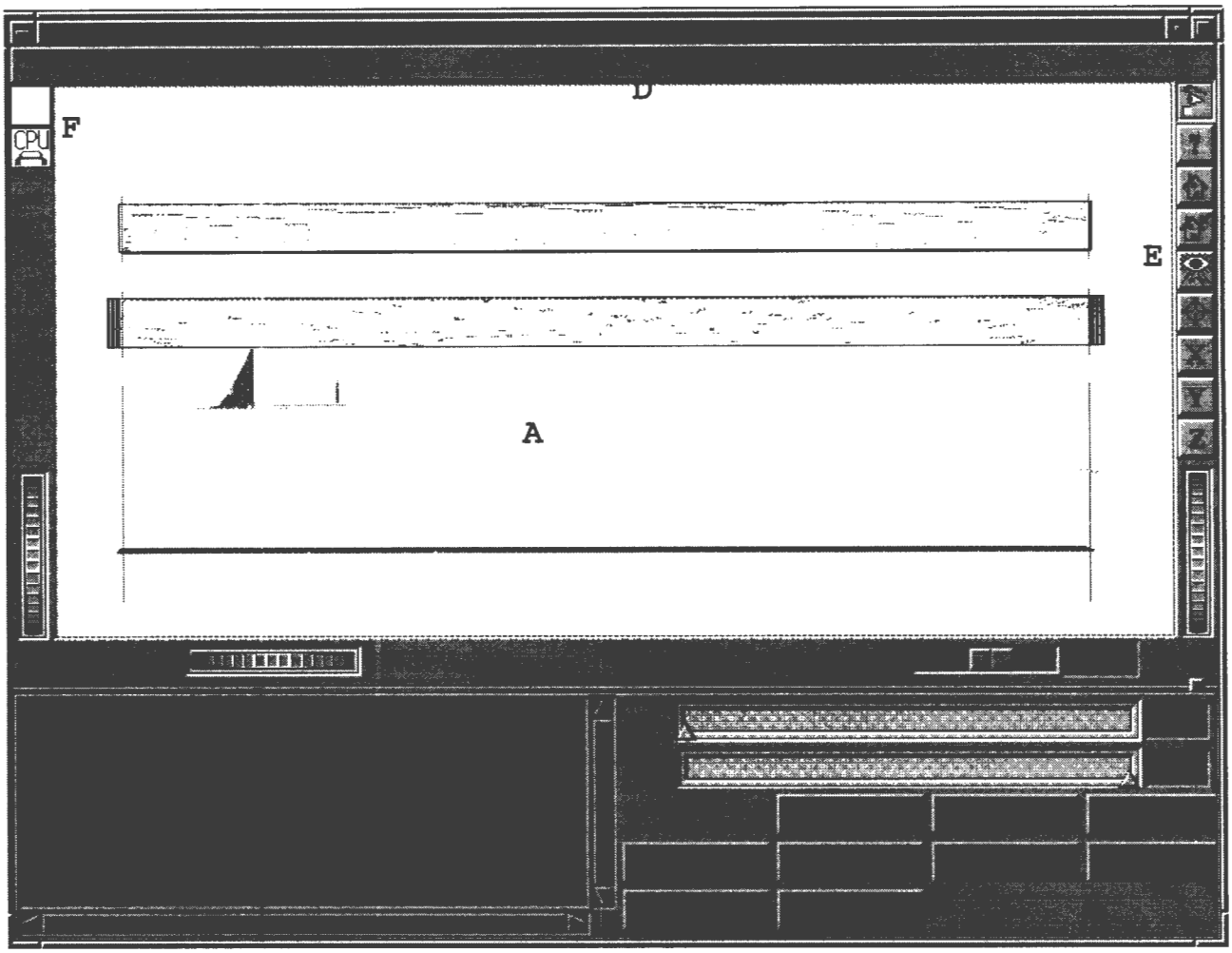


Figure 1: Chiron's main display window with all tools and short cuts visible.

Figure 1 shows the first window. The part labeled **A** shows the 3D view with the surrounding tools and short cut buttons and gadgets around it. All views on data are shows in the 3D area (**A**). The user interaction with these views also takes place in this area. Henceforth

this area will be refered to as *the 3D window.*

The window labeled **B** is used to present the user with some textual feedback regarding specific interactions in the 3D window or of a general nature. Henceforth this area will be refered to as *the text window.*

The area labeled **C** contains 9 buttons which form short cuts to bringing a specific view into the centre of the screen, and two granularity sliders. The buttons are an important tool for fast access to different views, because it is easy to lose one's orientation in the 3D space. The level of detail sliders can be used to change the data granularity, and in temporal views the actual time span, that the views represent.

The area labeled **D** gives access to ten menu categories. The options relating specificly to Chiron are located on the right hand side, *Views*, *LineViews*, *MeshViews*, and *SyncViews.*

The areas labeled **E** and **F** contain buttons which perform operations on the overall 3D window. For example the buttons labeled **X**, **Y**, and **Z** can be used to view the 3D views along the specifid axis.

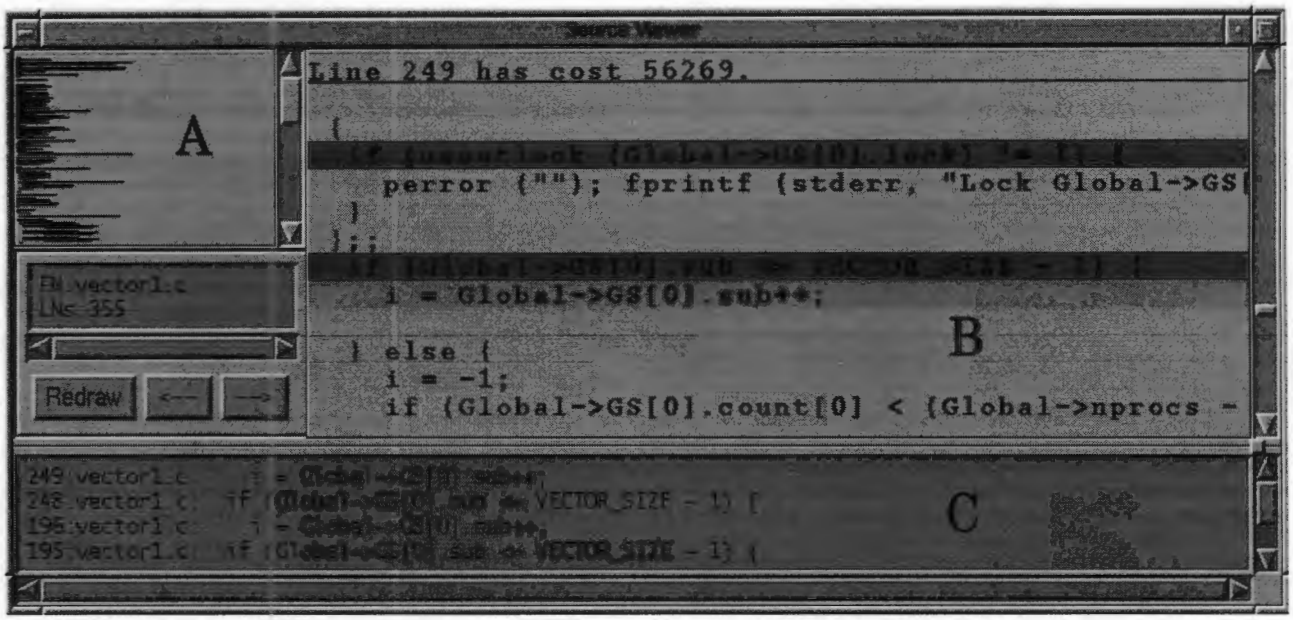


Figure 2: Source code reference window

Figure 2 shows the second window that Chiron uses. The source code window is used to show source line and source file references. The area labeled **A** shows a concise version of a

source file, with areas of interest marked in red. Area **B** shows the actual source lines with specific lines highlighted. Area **C** is used for data correlations between the *Source* and em Object views.

3.2 3D views overview

The views that Chiron can display are divided into three categories: *Global information views*, *Temporal views*, and *Correlation views.*

Global information views serve as road maps to performance bottlenecks and also as analysis starting points. Each view presents an overview of a particular metric of interest. The task of analysing a program's performance starts by looking at the *global information views* and determining whether a particular visual feature represents a possible performance bottleneck. *Global information views* also provide high-level correlation information when an additional metric is projected onto them.

Temporal views provide detailed information about the temporal behavior of specific components in a program or a part of the target machine's architecture. The visual representation of these graphs form the basis of the visual signature identification process in Chiron, and thus form an integral part of the performance analysis and algorithm/architecture understanding process.

Correlation views form mappings between two or more *Global information views* and/or *Temporal views.* A correlation is created between a set of views by projecting a set of metrics onto a target metric. The mapping can be shown in the form of a color, a texture, or a perturbation being applied to the target view.

3.3 Global information views

Chiron presents global information in five views: *Performance Overview*, *Source Line View*, *Object View*, *Function Call Relation View*, and *Process Synchronization Overview.*

A global view shows an overview of a specific performance metric. The data presented by these views represents object instances, classes, or program statements in the target

program. Although the metric is usually averaged over the run-time of the program, some views, i.e., the *Source Line View* and *Object View*, show a metric at a fine object granularity.

3.3.1 Performance Overview

This view shows an overview of five performance-influencing metrics: Active CPU time, barrier synchronization time, semaphore synchronization time, lock synchronization time, and memory miss wait time.

The *Active CPU time* is the total time during which a processor was doing some real work, that means not waiting on synchronization or the memory system. The *Barrier synchronization time* is the amount of time a processor spends waiting for barrier synchronization to complete. The *Semaphore synchronization time* is the amount of time a processor spends waiting for a *P()* operation on a semaphore to be completed. The *Lock synchronization time* is the amount of time a processor waits for a lock to be acquired. The *Memory miss wait time* is the total amount of time a processor spends stalled waiting for the memory system to be updated or brought into a consistent state.

For each processor on which the program ran, the view shows a bar consisting of the five performance categories. For easy identification each category is color-coded. The user can query any part of the view by selecting the category in a bar.

Figure 3 shows an example view: the text annotations detail the different bar components.

A textual overview of the performance data is also shown. The example below shows the textual data from a program trace.

```
Trace overview
CPU 0 : TCT:190914  ST:0       MT:201   (ST = bwt(0)+swt(0)+lwt(0)
CPU 1 : TCT:183492  ST:102085  MT:32933 (ST = bwt(0)+swt(682)+lwt(101403)
CPU 2 : TCT:182958  ST:92702   MT:41835 (ST = bwt(0)+swt(0)+lwt(92702)
CPU 3 : TCT:183790  ST:99546   MT:35739 (ST = bwt(0)+swt(854)+lwt(98692)
CPU 4 : TCT:184162  ST:102118  MT:33474 (ST = bwt(0)+swt(1123)+lwt(100995)
CPU 5 : TCT:183785  ST:95396   MT:39649 (ST = bwt(0)+swt(822)+lwt(94574)
```

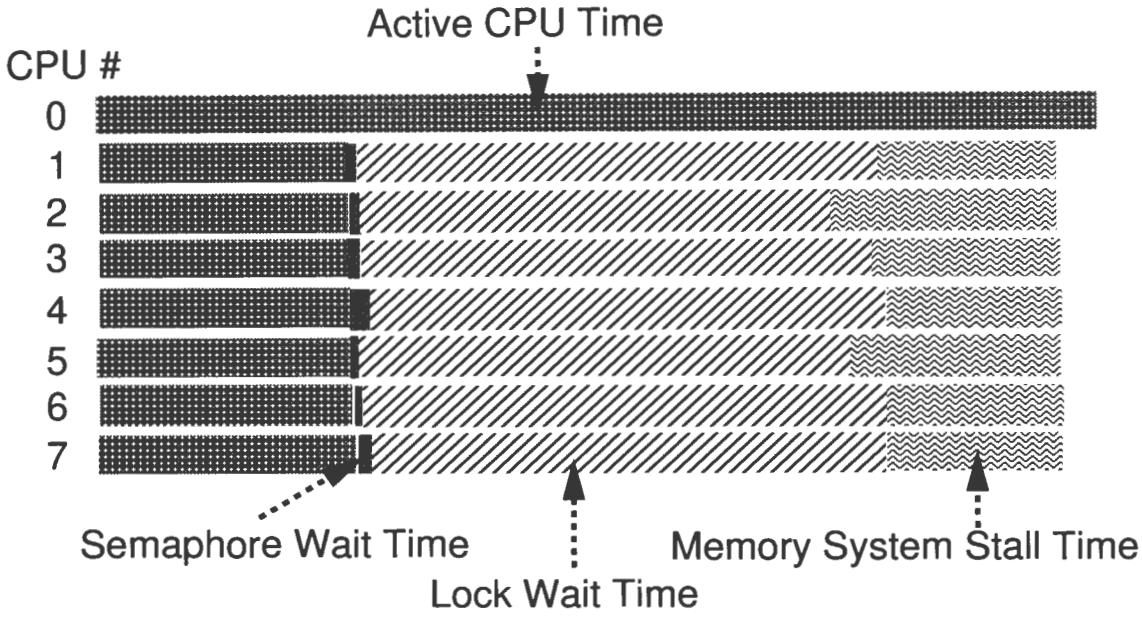


Figure 3: A example Performance Overview Graph

```
CPU 6 : TCT:184194  ST:101790  MT:33514 (ST = bwt(0)+swt(1152)+lwt(100638)
CPU 7 : TCT:183436  ST:101643  MT:32686 (ST = bwt(0)+swt(667)+lwt(100976)
TCT = total CPU Time:ST = Synchronization Time:MT = Cache Miss Time
bwt = Barrier Wait Time:swt = Semaphore Wait Time:lwt = Lock Wait Time
Percentages:
CPU 0  : Active 99.89%  Sync  0.00%  Miss  0.11%
CPU 1  : Active 26.42%  Sync 55.63%  Miss 17.95%
CPU 2  : Active 26.47%  Sync 50.67%  Miss 22.87%
CPU 3  : Active 26.39%  Sync 54.16%  Miss 19.45%
CPU 4  : Active 26.37%  Sync 55.45%  Miss 18.18%
CPU 5  : Active 26.52%  Sync 51.91%  Miss 21.57%
CPU 6  : Active 26.54%  Sync 55.26%  Miss 18.19%
CPU 7  : Active 26.77%  Sync 55.41%  Miss 17.82%
Cache layout
# of groups: 2: 4 4
Each L1 cache has 2048 lines and is 4 way set associative
```

```
Each L1 cache line is 32 bytes wide.
L1 cache size: 262144 (256KB)

Each L2 cache has 2048 lines and is 4 way set associative
Each L2 cache line is 128 bytes wide.
L2 cache size: 1048576 (1024KB)
```

Interpretation

The user can quickly identify which components of the bar graph dominate. If the process synchronization section dominates the graph then it would be a good starting point to further investigate the performance problems in the synchronization section. Similarly when the memory system stall time dominates the graph, the problem area could be excessive cache interference, and therefore we would start by investigating the memory system behavior.

Interaction

The view has six degrees of freedom, so it can be placed anywhere in the 3D space and its orientation can be changed. The user can select any bar component of the view and Chiron will display that bar's information, such as CPU number, metric name and represented time span.

3.3.2 Implementing the Folded Graphs

The *Source Line View*, the *Object View*, and the *Process Synchronization Overview* graphs are all implemented as *folded graphs*. The data these views show is inherently two-dimensional, object or some other criterion versus cost. However, because of the large number of nodes in a typical graph of this nature, we display these graphs in a three-dimensional *folded graph* as shown in Figure 4.

The object cost values, which will be represented by the folded graph, are sorted according to some criterion, either decreasing cost or increasing memory address, before they are

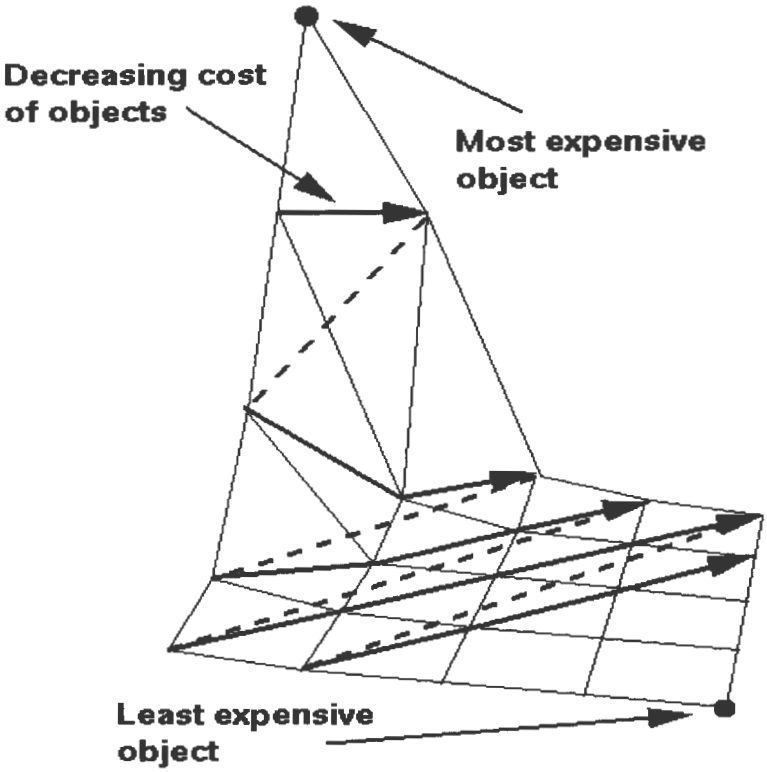


Figure 4: A folded graph concisely displays large amounts of data.

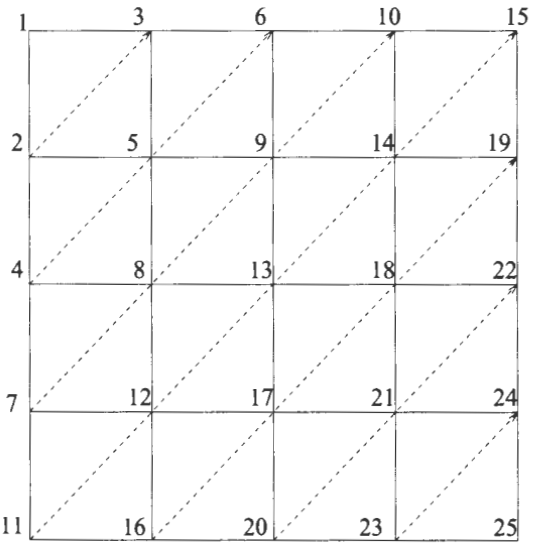


Figure 5: Top down view of a grid showing the node assignment order.

assigned to positions on the folded graph. The first cost value is assigned to one corner of the grid. Then the rest of the sorted data items are assigned along the diagonals of the grid. Figure 5 shows the order in which the nodes are assigned to the grid positions.

The result is a surface that summarizes a large two-dimensional graph as a small three-dimensional graph. By laying the data out in this way, we are able to get a feel for $N \times N$ pieces of information in one glance, something which is difficult to accomplish with a linear 2D display. In addition, the folded graph makes it easier to pick a particular node by using the mouse, and it also makes it easier to see visual correlations applied to the folded graph surface, as described in section 3.5.

3.3.3 Source Line View

The *Source Line View* shows a graph on which every node represents a specific line in a source code file. The height of a node represents the cost of the source line, as counted in the number of clock cycles which a processor was stalled for due to a cache miss caused by code from that source line line. Section 4.3.2 described the process of generating a source line reference from a cache miss.

Figure 6 shows an example of a *Source Line View*. The graph labeled **A** shows the view as it would appear on the screen. The highlighted spot indicates that the user selected a point in the view and information pertaining to that point is shown in the *Source Code Window*. The graph labeled **B** shows a wireframe representation of view **A**. This clearly shows the individual nodes on the graph.

Interpretation

The height of a node in the graph shows the relative cost of the corresponding data item. The peak in the graph does not necessarily represent an expensive object, but just the object with the highest cost.

Under normal conditions some source lines will cause cache misses and thus incur a cost. The performance analyst would start the investigation with the most expensive source lines, which lie closest to the peak of the graph.

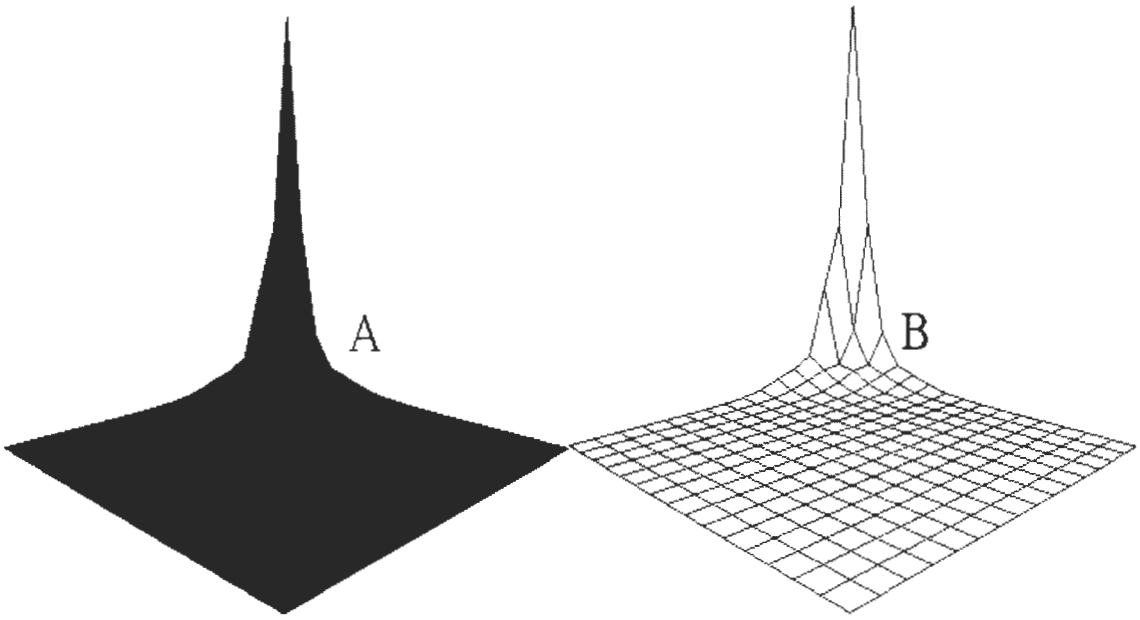


Figure 6: An example of a *Source Line* View.

Interaction

The user can select a node by moving the mouse pointer on to the node and clicking the left mouse button. The selected node is highlighted in red and Chiron displays information about that node in the *Source Code Window*: the source line which corresponds to the selected node is highlighted in purple. The user can use a scroll bar to scroll through the source code. The file name, line number, and cost of the selected line are shown at the top of the section labeled **B** in figure 2.

The view can also be rotated and zoomed in on, giving the user the ability to focus on specific areas of interest. Being able to rotate the view in 3D gives the user the ability to interpret different parts of the view more easily. By looking at the view edge–on the user might get more insight that by just looking at the view from the front (as shown in figure 6).

3.3.4 Object View

The *Object View* summarizes memory traffic information. It has a node for each identified object and data structure which was responsible for one or more cache block misses. Chiron keeps track of all these nodes' memory location and cache behavior. Every node has a *misses caused* counter, which is incremented whenever a memory access to the corresponding object or data structure caused another object or data structure to move out of a cache.

The default view is created by sorting all nodes by decreasing *number of misses caused* and then mapping the objects onto a folded graph. Each node on the surface now represents one object and the height is proportional to the number of misses which were caused. (See section 3.3.2 for a detailed description of this process.)

Not all objects or data structures have the same size; therefore if one object is much larger than another one, it is conceivable that the larger object might have a higher miss count. To ensure that the user does not misinterpret the graph, Chiron has the option to represent an object as individual memory locations in the view, so that each node on the graph would represent 4 bytes in memory. This node-switching option gives the user the ability to identify problematic objects or data structures, and also allows the user to identify exactly which component of an object or data structure represents the problem area.

The difference between these two representations is shown by the following example: If we have an instance, call it *one_array*, of class *Array*.

Array *one_array = new Array;
The array class definition is:

```
class Array {
public:
  int nodes[100];
};
```

On default the view will show the *one_array* object as one node. All misses which were caused due to the elements in *one_array* will be added to the miss counter of *one_array*. If

all 100 node elements of the array were accessed and each access caused one miss, then the cost of *one_array* would be 100 misses.

In the alternative view representation each element inside the *one_array* object will be considered to be an object. In the above example we would have 100 objects, each with a cost of 1 miss, and all belonging to the class *Array*. This view shows the objects on a fine-grained level. The advantage of this display method is, if one specific element of an object is responsible for a large number of misses, the analyst will be able to see this very clearly on the graph. The disadvantage of this method is the large number of nodes on the graph. Depending on the workstation, performance might suffer with a very large number of nodes in the graph.

Figure 7 shows different views of an object view which contains 6796 data nodes. **A** shows the graph when viewed along the x axis and having the objects sorted by cost. **B** shows the same data viewed along the z axis. These two views clearly show the data layout. Since, the data is sorted on cost we can see what relative cost the nodes have, but this does not provide any information on where in memory these data objects are located.

The data items in the view can also be sorted by increasing memory location. **C** shows memory-sorted data viewed along the x axis of the graph. With all the high peaks obscuring other data in the graph it is very difficult to identify specific nodes. To improve this situation, the user can view the graph along the y axis, as was done in **D**. The dark patches in the graph **D** represent cost peaks.

Interpretation

The *Object View* gives an overview of the target program's memory behavior, and thus enables the user to identify possible performance problems at a glance.

In graph **A** in figure 7 the most expensive object is clearly shown as the peak of the graph. The analyst has a good chance of optimizing the target program when the cost associated with that object can be reduced. The visual analysis process is based on identifying peaks in the graph. From the graph it is also easy to see that the object cost goes up in steps, which means that a specific section of the target program's data could be responsible for

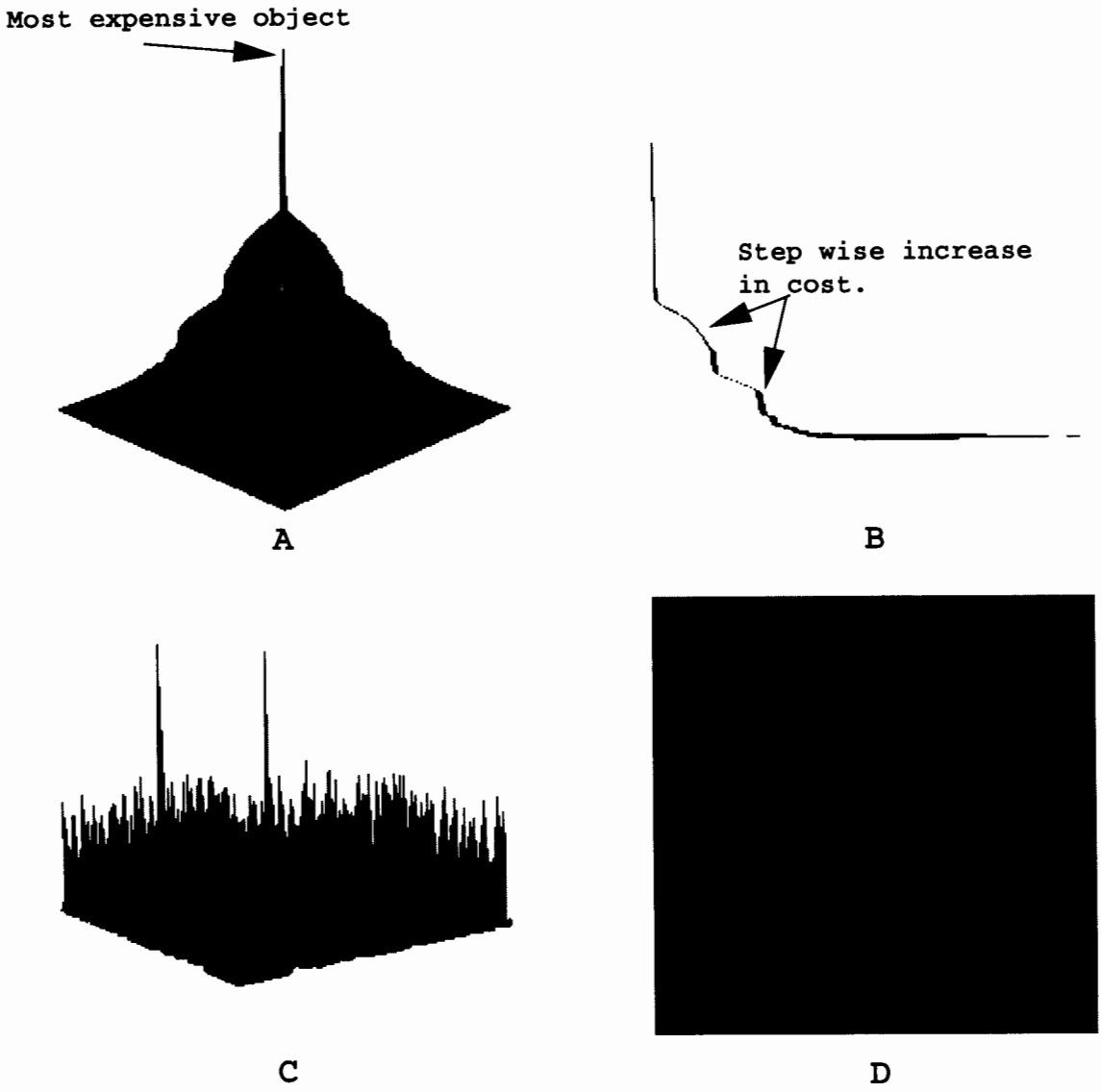


Figure 7: Example object view graphs

the performance bottlenecks.

The step wise cost increase is easily identified on graph **B**. The steps might mean that a specifc part of the data items in the application are used more ofter than others, or might cause more misses due to multiple processors accessing them and causing replacement or coherence misses.

To find out more about the cost of objects, the user can change the sort order of objects in the graph. Instead of sorting objects by decreasing cost they can be sorted by increasing memory location. This will provide the user with more insight into the relative memory location of expensive nodes in the program. Expensive nodes are clearly visible as peaks in graph **C** and as dark areas in graph **D**. The dark shades occures due to the steap slopes of the graph at those points.

Sometimes an object is moved out of the cache due to false sharing or due to some other memory location related problem, Graph **A** and **B** in figure 7 do not help in analyzing this problem, but graph **C** and **D** do. Graph **C** and **D** show the same data set as **A** and **B**, but the user can now see how two closely located spickes on the graph interact. This means that the analyst can find possible false sharing problems by just looking for two or more spikes located near each other.

By looking closely at graph **D** the user can quickly identify memory regions with relatively high cost, by simply looking for dark regions in the graph.

The dark areas in graph **D** enable the user to quickly identify memory regions with relatively high cost. The steap gradient of the graph at these locations, thus the dark shading, corresponds to a substantial change in cost, thus a substantial change in memory access counts. If, for example, a high spicke is found next to a low cost object, are these two objects reside in the same cache line, a clear performance bottleneck has been identified.

Interaction

By moving the mouse pointer onto the view and clicking the left mouse button, the analyst can select a node in the view. The selected node is highlighted in yellow and detailed information about the node is displayed in the text window.

The information about a node contains:

Object name	This is the name which the program uses to refer to a specific object. In the above example it would be *one_array*.
Rank	Indicates the relative cost of the object.
Memory address	Shows the object's location in memory.
Cost	Indicates how many cache misses this object has caused.

3.3.5 The Function Call Relation View

The *Function Call Relation* view shows an overview of functions and the calls made between them. Each function which was called at least once during a program's execution is represented by a cube and a function call is represented by a cone. The cone originates from the caller and ends in the called function. The cost of a function is calculated as the amount of time all processors spent executing the function.

Each function call that a program executes is recorded and reflected in this view. Since a program might have a very large number of functions and function calls, the layout of the cubes and cones was a problem. We wanted to layout the cubes and cones in such a way that it would be easy both to view the data and to select any cube or cone.

The layout method which suited the problem at hand best was to arrange the cubes in a spiral pattern, with the cones connecting the cubes. This layout method reduced the problem of intersecting cones and still gives the user easy access to every cube and cone. If a problem with access to a cone arises the user has the ability to increase the number of turns on the spiral, which should solve the access problem.

Figure 8 shows three views on a *Function Call Relation* view: **A**, **B**, and **C** show the spiral viewed along the **x**, **y**, and **z** axis respectively. It can be clearly seen that the spiral representation gives easy access to the cubes and cones, and still keeps the image at a managable size.

The *Function Call View* is an experimental view. We believe that much more information can be gained from this view by futher exploring the graphical layout and data contents of the view. As the cone structure is hard to identify if the graph is viewed from a distance the user will have to zoom into a smaller region of the graph. A cone structure can easily

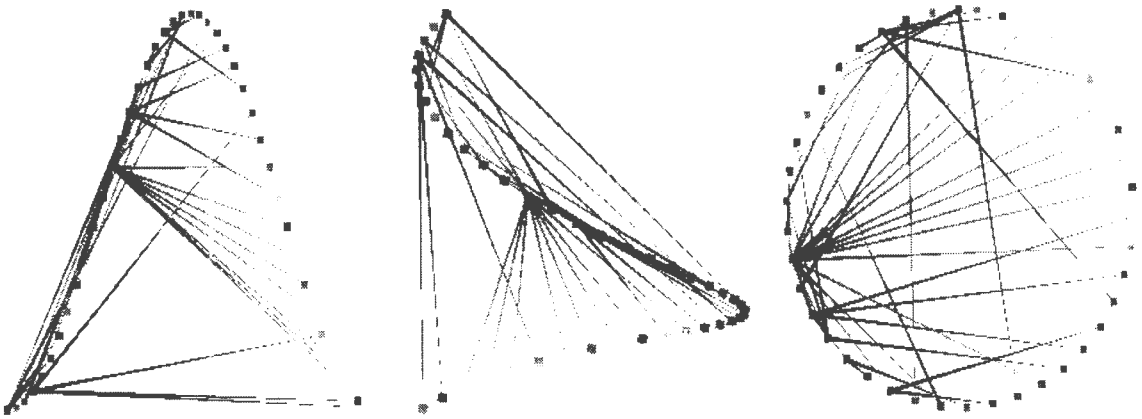


Figure 8: Three views of a spiral representing the Function Call Relation View

be identified at the top of graph **B** in figure 8. The reader can see a cone originating from the top most cube and stretching out to the bottom right hand side.

Interpretation

The view can be used as a basic profile of a target program. When the view is seen along the **y** axis the topmost cube represents the most expensive function. The cones show the direction of a function call, and it is easy to determine the call relationship between functions.

Interaction

The user can select a cube or cone to find out more detail about the function or function call respectively. By moving the mouse pointer onto a cube or cone and pressing the left mouse button the user selects the target object.

If a cube is selected the name and the cost of the function it represents is displayed in the text window. Similarly, if a cone is selected the source and the destination functions of the call, and the number of calls along the cone are displayed.

The user can rotate the view in any direction to gain easy access to any cube or cone. The

number of turns the spiral makes can also be set by the user. This option allows the user to interactively reduce the number of cone intersections.

3.3.6 The Process Synchronization Overview Graph

The *Process Synchronization Overview Graph* shows an overview of barrier, semaphore and lock behavior. Like the *Source Line* and *Object* views this graph is also a folded graph with each node representing an object instance.

For each instance of a barrier, semaphore or a lock there is one node in the graph. The height of the node is the cost of the object, where the cost is calculated as the amount of time all processors were stalled for trying to access that particular synchronization object. For barriers the cost is the time between the first *B_ATTEMPT* and the last *B_ACQUIRE* message. For semaphores the cost is the time between the *P_ATTEMPT* and the *P_SUCCESS* message, similarly for locks, the cost is the time between the *LOCK_ATTEMPT* and the *LOCK_ACQUIRE* message.

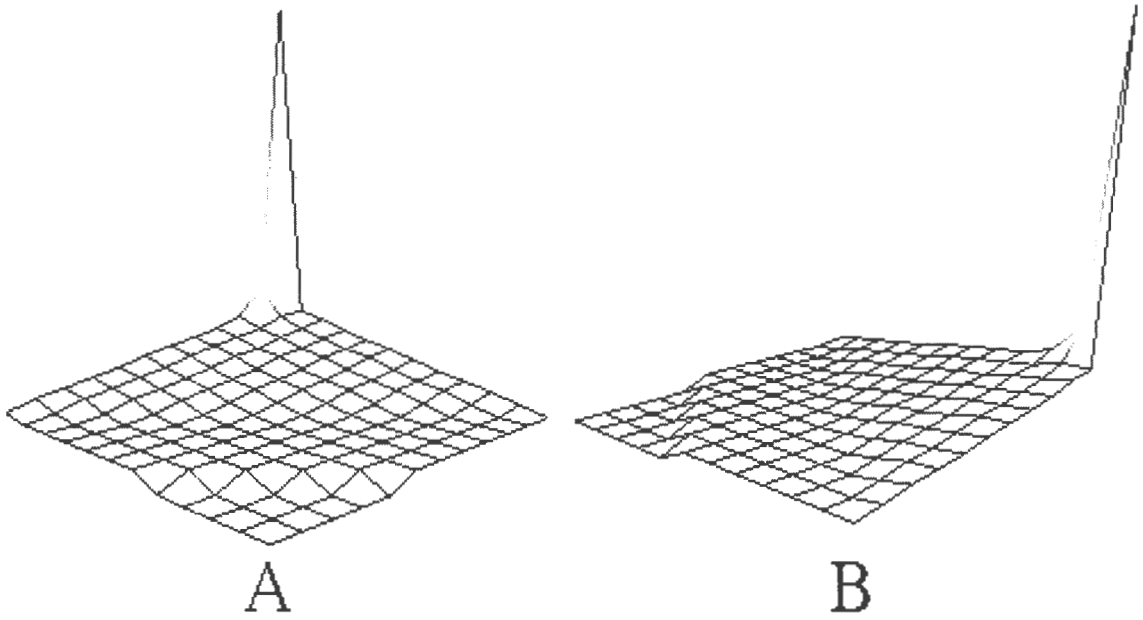


Figure 9: An example synchronization overview graph

Figure 9 shows a *Process Synchronization Overview* graph. Graph **A** is a front on view and graph **B** views the graph from the side.

Interpretation

The folded graph shows the relative costs of the locks, semaphores, and barriers. The most expensive object is easily identified and queried. The user can quickly see how many synchronization objects are responsible for a performance bottleneck, and how these objects relate to one another. Since the folded graph gives a quick overview of a large number of data points, the user can see at a glance what could have been a very large two-dimensional graph.

Interaction

The user can query any node on the graph by selecting it with the mouse pointer. For each selected node Chiron displays the barrier's, semaphore's or lock's memory address, cost, name, and relative position on the graph. A node at position 1 will be more expensive than a node at position 2. The cost order indicates what position in a decreasing cost array the object occupies. This information is useful to determine whether we are looking at the most expensive or the second most expensive object in the graph.

The piece of text below shows the information contents of a selected node:

```
Position 2 selected:
addr = 0x40020328 cost = 1643
Name = array_lock_data_6
```

3.4 Temporal Views

Summary statistics, like the *Performance Overview*, can be misleading. Temporal views on the other hand show detailed behavior of program and architecture components. By inspecting the temporal views, the user can visually identify brief but significant anomalities in program behavior, such as hot spots of activity, or regions where a working set is too large for a cache.

There are five temporal views: *First-level and second-level cache views*, *Cache miss overview*, *Call stack view*, and *Process synchronization view*.

3.4.1 First-level and second-level cache views

The *first-level and second-level cache views* show the cache block movement of the first-level and second-level caches respectively. The second-level cache view (hereafter referred to as L2) is very similar to the first-level cache view (hereafter referred to as L1), and therefore the L1 view will be described in detail, and where applicable differences in the views will be pointed out.

The L1 cache view visualizes the dynamic behavior of the cache occupancy of the first-level cache. The data is shown by drawing a cube at location (x y z), if processor x accesses cache block y at time z. The cube ends at position (x y z1), where $z1$ is the time that processor x moves cache block y out of its cache.

Along the x axis of the graph we have the CPU index. If we run a program on 8 CPUs then the graph will have 8 distinct sections, each representing the data for the entire CPU.

Along the y axis of the graph we have cache blocks. As a processor typically accesses a few hundred cache blocks during the execution of a reasonably sized program, we limit the L1 view to the *100 most expensive* cache blocks. The cost of a cache block is calculated as the total number of CPU cycles spent by all processors waiting for that cache block to be moved into the cache. The cache blocks are sorted according to decreasing cost. At the top of the graph we have the most expensive block.

Along the z axis of the graph we have time. A program starts executing at time $(z) = 0$. The first entry into the graph will occur at time $z = N$, where N is the time the first cache block is accessed. Since a program normally starts with a read, N is mostly 0.

Data in the cache block view is presented in planes. For example, if the most expensive cache block has virtual address 0x1000A000 (in hexadecimal) at time t, and it has the greatest cost, then all blocks in the corresponding xz plane have the same virtual address or map to the same cache block.

All cache blocks accessed by processor x will be displayed with the same x value, so all blocks accessed by this processor will lie in the same yz plane. If more than one processor was active on a program at any given time, then there will be one yz plane for each processor.

So if a program was executed on eight processors then there will be eight *yz* planes in the graph.

For the sake of clarity and ease of identification we have given all blocks belonging to the same processor (in the same *yz* plane) the same colour. This means that the user can identify the processor to which a block belongs by looking at the colour of the block.

In figure 10 the axes show how each of the images have been rotated in order to obtain the view. Along the x-axis the graph shows processors and along the y-axis the graph shows cache blocks.

Figure 10 shows three views of a L1 and L2 cache view. Graph **A** shows all the L1 and L2 cache data to which Chiron has access. Time progresses from left to right. Every element in the graph consists of a cube which represents the amount of time for which an object was located in the cache. Two distinct cache movement activity times can be distinguished from the graph.

Graph **B** shows the L1 cache graph viewed along the z axis. This graph clearly shows the different processors and the blocks associated with each. The aim of this image is to clearly show that the cache view graph represents a range a caches on a number of processors. The frames surrounding each processor's plane clearly identifies which cache blocks belong to which processor.

Graph **C** shows a closeup view of excessive cache block movement in the first-level cache. The graph covers a short time span of a program's execution and shows the L1 cache behavior of 4 processors. The processors can be destinguished by the color of the cubes, as seen in figure 30 **C**. Excessive cache block movement is identified by the short blocks along the time exis of the graph.

Interpretation

In my experience the temporal first-level and second-level cache views are the most informative views that Chiron has to offer. These views give detailed information about the memory behavior of all performance critical objects and data structures. From this information the user can identify a range of performance problems relating to improper data

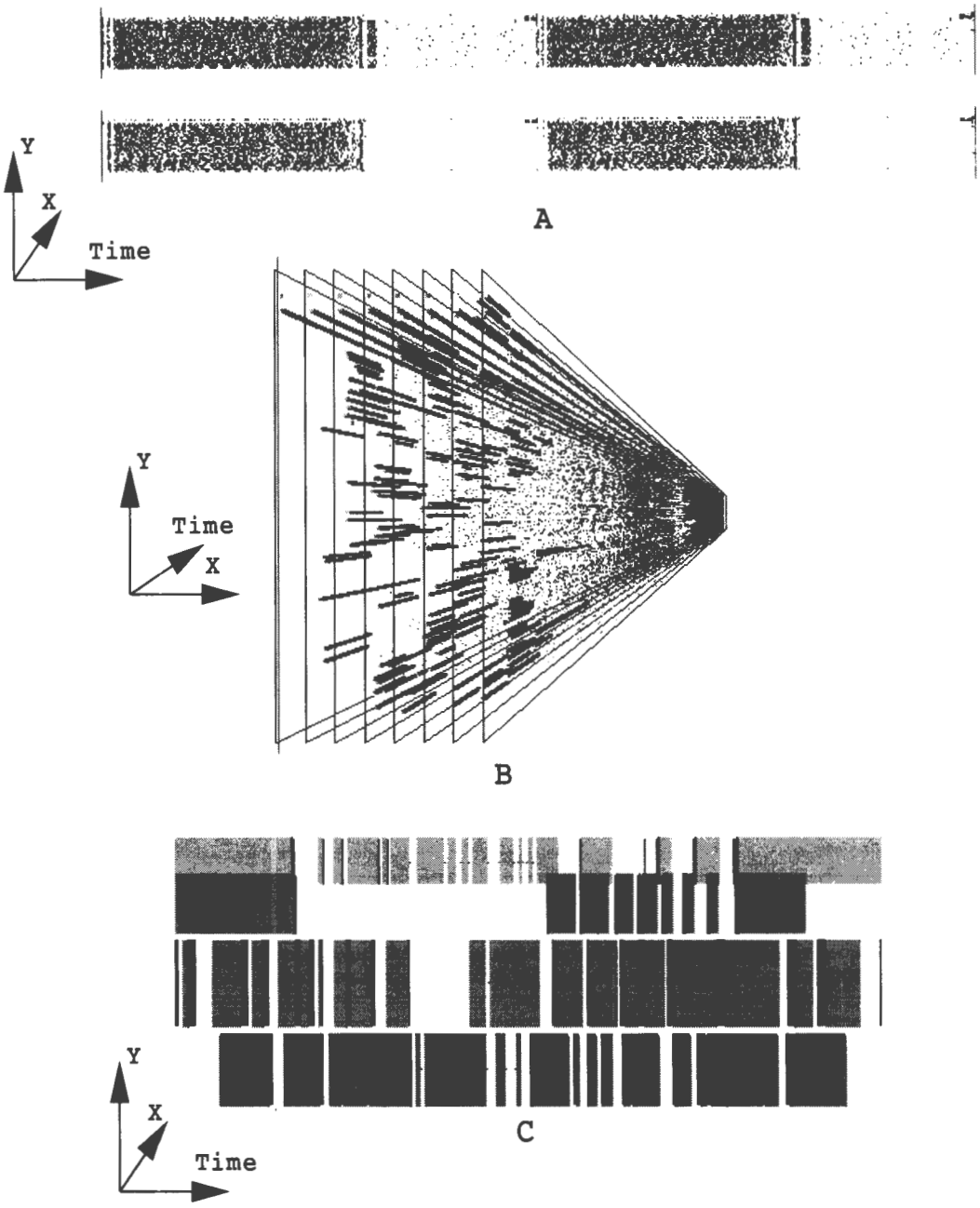


Figure 10: Example L1 and L2 cache views

layout and excessive data interference.

From the graph labeled **A** in figure 10 we can clearly distinguish two types of regions: regions with a high density of points, and regions with only a few data points. The points in the images represent cache block movement, and therefore regions with low point density can be discarded because only a few cache block movements occur. An investigation should concentrate on the high-density areas.

Experince with Chiron has shown that excessive cache coherence activity and block replacement interference each have a distinct visual pattern when viewed as a time graph. These visual signatures, high-density areas, have many high-frequency components, and are easily identified and thus enable the user to quickly find performance bottlenecks in a very large data set. In the case of coherence interference, these blocks all correspond to the same cache block moving between caches, and in the temporal cache view all these blocks therefore appear at the same vertical position. In the case of replacement interference, the interfering blocks will be at different vertical positions.

Figure 10 **A** shows the complete cache activity data set for both the first-level and second-level caches. The user can see from this view that the program execution was completed in four stages, two stages of high cache activity and two low activity stages. Clearly the user can dismiss the two low activity stages as insignificant and concentrate the performance analysis on the high activity stages.

Graph **B** shows that all 8 processors were active from the start of the program and that the cache shows a substantial amount of block movement right from the outset. Processors can be identified by the reference lines surrounding the graph. The black lines on the left hand side of the graph indicate cache block occupency. In this case the program is building up a complex data structure and all eight processors are used for this. The high block movement activity shows that the user might have to reorganize the data structures or the work load partitioning algorithm.

Graph **C** shows block movement which is due to coherence misses. The graph shows short blocks, which indicates that data is shared between processors. The user can select these blocks and obtain more detail about the objects which occupy the cache block.

The temporal cache view also allows the user to identify if a set of block movements related to coherence activity is due to false sharing. False sharing occures if unrelated data used by two or more processors is located within the same cache block. Access to the data values by different processors causes the valid copy of the block to move from cache to cache, which reduces program performance.

Interaction

The temporal cache view can be interactively manipulated. Three viewing directions are of particular interest. Looking along the x axis the user can see the yz plane, which means that that all cache blocks and their behavior in time can be seen in one glance. Looking along the y axis the user can see the xz plane, which means that the most expensive cache block over all N processors can be seen in one glance. Looking along the z axis the user can see the xy plane, which shows the starting conditions of all cache blocks.

Since an average program produces millions of memory references and thousands of cache block movements, the L1 cache view can become very cluttered with blocks. This means that some blocks would obscure others and possible important information might be obscured from the user. A very large number of blocks also puts a burden on the polygon engine of a graphics workstation, which would reduce the interactivity of the application.

The user can select any block in the L1 cache view by moving the mouse pointer onto the block and clicking the left mouse button. Information about the selected block is displayed in the text window. The following data is displayed: A block cost index, the processor number to which the block belongs, the start address of the cache block, the cost of the block movement, and the cause of the miss. The address of the reference which moved the block into the cache, and the block entry and exit times are also displayed. When available, Chiron displays the names of the objects which fall into the selected cache block.

The following is an example from Chiron's text window:

```
Block 0 from cpu 6 - addr 0x40003000 - cost 136808 - coherence miss
Reference was 0x40003004
```

```
Entry/Exit (2082541/2082616)

Objects in selected block:
nprocs 0x40003000
GS 0x40003004
```

The selected block occupies position 0 in the cost array, and therefore it is the most expensive cache block, in processor 6. The block start address is 0x40003000 and the reference which caused the block to be moved into the cache was at address 0x40003004. The cost of the movement was 136808 clock cycles. The cause of the block movement was a cache coherence miss. The block moved into the cache at time 2082541 and moved out of the cache at time 2082616. Finally Chiron was able to resolve the addresses which the cache block covers and maped it onto two names. The variables **nproc** and **GS** fall into the selected cache block.

3.4.2 Cache Miss Rate Overview

The *Cache Miss Rate Overview* shows a summary of the cache miss behavior of the L1 and L2 cache views. The yz plane, cache blocks vs. time, is divided into 100×10 equal regions. Each region counts the number of misses that fall into the time span vs. cache block region. The number of misses is then normalized, and the normalized value is used as a brightness factor in the graphical representation.

The graph is a plane of 100 by 10 squares, where the intensity of each square indicates the number of misses that fall into a the corresponding region. The graph is placed behind the L1 and the L2 cache block views.

This view can be used as a fast problem area indicator. The user can easily zoom in on problem areas by looking for bright red squares in the *Cache Miss Rate Overview*. Once a problem area has been identified the user can switch off the overview graph and concentrate on the L1 and L2 views.

Figure 11 shows a small section of a *Cache Miss Rate Overview*. The intensity of the red regions indicates the number of cache block movements in that region.

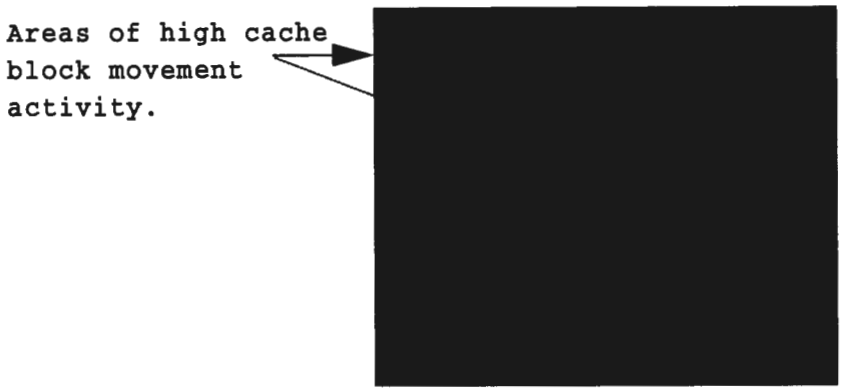


Figure 11: Example L1 and L2 cache miss rate overview

3.4.3 Call Stack View

The *Call Stack View* shows which functions were being executed by which processor at any given time. The user can query the state of any processor and find out which function was being executed.

This view helps the analyst track down logical problems in an algorithm, and it can help in the identification of load balancing problems. For example if a particular processor was assigned to much work, then this would be clearly visible on the graph, as all other processors would have already finished their work and would be waiting on a barrier, for example, and the last processor could still be seen to execute the work functions.

3.4.4 Process Synchronization View

The *Process Synchronization View* shows detailed information about barrier, semaphore and lock accesses behavior during program execution. Three separate views are presented. One for each of the basic synchronization types.

The following sections describe the three graphs and their basic interaction, following this, we describe the interaction which effects all graphs.

Barrier Synchronization View

The *Barrier Synchronization View* shows the dynamic behavior of all barriers during a program's execution. The data is shown by drawing a cube at location (x y z), if processor x enters barrier y at time z. The cube ends at position (x y z1), where $z1$ is the time that processor x exits from the barrier.

Along the x axis of the graph we have the CPU index. If we run a program on 8 CPUs and all 8 CPUs access a barrier structure, then the graph will have 8 distinct sections, each representing the data for the entire CPU. Along the y axis of the graph we have barrier objects. Barrier objects are shown in order of creation, cost or memory address. The barrier cost is calculated as the number of CPU cycles spent by a processor inside the barrier structure.

Along the z axis of the graph we have time. A program starts executing at time $z = 0$. The first entry into the graph will occur at time $z = N$, where N is the time the first processors enters a barrier.

For every *B_ATTEMPT* and *B_ACQUIRE* message pair there is one cube in the barrier graph. The cube starts at the time the *B_ATTEMPT* message is issued and ends when the *B_ACQUIRE* message is received. After entering a barrier a processor blocks until a barrier condition is satisfied. The time that the processor is blocked is taken to be the cost of the barrier.

Interpretation

Figure 12 shows a typical barrier synchronization view. From the view it is easy to see when each of the processors enter the barrier and when they all leave.

Since barrier synchronizations occure relatively seldom, compared to other operations in a program, a typical graph only has a few cubes on it. In such a case it is easy to identify when a barrier synchronization occures.

The true power of the visual representation is shown when the user compares the synchronization view with the first and second level cache views. These views can be time-aligned

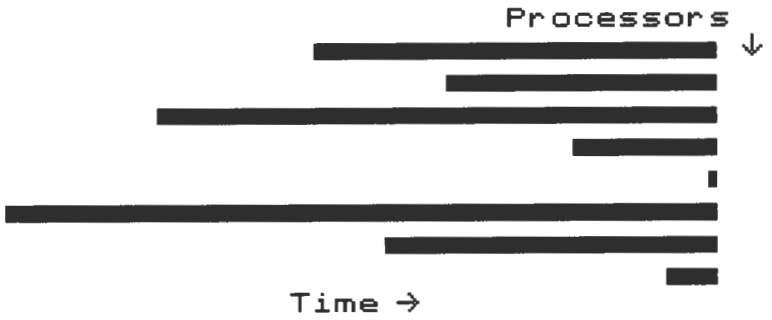


Figure 12: An example barrier synchronization view

so that the start and end times of the views correspond and that they cover the save distance on the screen. When the two views are overlaid the user can easily identify what cache behavior occures when a processor enters a barrier.

Interaction

The user can select any barrier cube to find out more details about it. The information contains data such as the barrier's memory location, the waiting processor's number, and the barrier name. Additionally the barrier entry and exit times, the number of blocked clock cycles, and the percentage of total barrier wait time this barrier contributes, is displayed.

The following is an example from Chiron's text window:

```
Barrier Addr:0x2000a0f0 CPU:1 Name = MainBarrier
BARRIER_ATTEMPT 1002789
BARRIER_ACQUIRE 1005020: wait time 2231 (18.12%)
```

Processor 1 entered the *MainBarrier* barrier, which is located at memory location 0x2000a0f0, at time 1002789, and spent 2231 clock cycles blocked in it. At time 1005020 the processor was unblocked and continued executing code. The 2231 clock cycles wasted due to the block represent 18.12% of the total barrier block time.

Semaphore Synchronization view

The *Semaphore Synchronization view* shows the dynamic behavior of all semaphores during a program's execution. The data is shown by drawing a cube at location (x y z), if processor

x performs a P() operation on semaphore y at time z. The cube ends at position (x y z1), where $z1$ is the time the P() operation succeeded.

Along the x axis of the graph we have the CPU index. If we run a program on 8 CPUs and all 8 CPUs access a semaphore object, then the graph will have 8 distinct sections, each representing the data for the entire CPU. Along the y axis of the graph we have semaphore objects. A program can have more than one semaphore object. Semaphores are shown in order of creation, cost or memory address. A semaphore's cost is calculated as the number of CPU cycles spent by a processor blocked on the P() operation. Along the z axis of the graph we have time.

For every *P_ATTEMPT* and *P_SUCCESS* message pair there is one cube in the semaphore graph. The cube starts at the time the *P_ATTEMPT* message is issued and ends when the *P_SUCCESS* message is received. The time that the processor is for blocked is taken to be the cost of the semaphore.

Interpretation

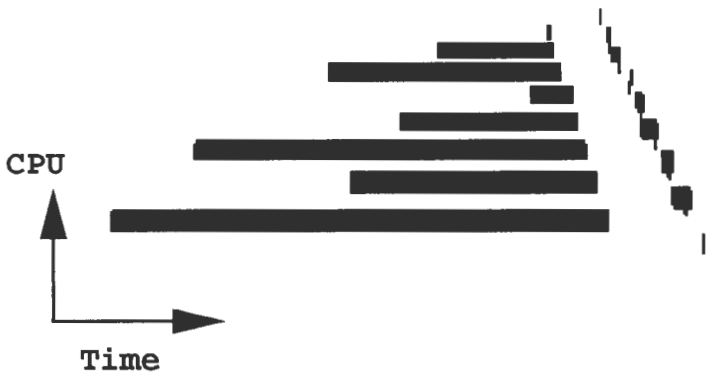


Figure 13: An example semaphore synchronization view

Figure 13 shows a typical semaphore synchronization view. From the view it is easy to see when each of the processors issue a P() operation on a semaphore and when the P() operation succeded.

Similar to the barrier synchronization view, this view is best used in conjunction with the first and second level cache views.

The view can also yield important information about the synchronization behavior of programs. Since semaphores are used more often than barriers, the view contains more cubes, and thus more information. When the user identifies a pattern of excessive synchronization, or very long P() service times, he/she can further investigate the behavior by, for example, looking at the *Call Stack View* to see which functions where active at a given time.

Interaction

The user can select any semaphore cube to find out more details about it. The information contains data such as the semaphore's memory location, the waiting processor's number, and the semaphore name. Additionally the P() operation attempt and success times, the number of blocked clock cycles, and the percentage of total semaphore wait time this semaphore contributes, is displayed.

The following is an example from Chiron's text window:

```
Semaphore Addr:0x40003018 CPU:7 Name = SyncSema1
PSEMA_ATTEMPT 2221418
PSEMA_ACQUIRE 2222085: wait time 667 (12.15%)
```

The selected semaphore **SyncSema1** at memory location 0x40003018 is accessed by CPU 7. The P() operation was started at time 2221418 and completed at time 2222085. It took 667 clock cycles to service the P() operation, which is 12.15% of the total time the program spent waiting on P() operations.

Lock Synchronization view

The *Lock Synchronization view* shows the dynamic behavior of all locks during a program's execution. The data is shown by drawing a cube at location (x y z), if processor x issues a *LOCK_ATTEMPT* trying to access lock y at time z. The cube ends at position (x y z1), where $z1$ is the time the lock has been acquired. A second cube is drawn from (x y z1) to (x y z2), where $z2$ is the time the lock has been released.

Along the x axis of the graph we have the CPU index. If we run a program on 8 CPUs and all 8 CPUs access a lock object, then the graph will have 8 distinct sections, each representing the data for the entire CPU. Along the y axis of the graph we have lock objects, of which a program can have more than one. Locks are shown in order of creation, cost or memory address. A lock's cost is calculated as the number of blocked CPU cycles a processor spent between the time a lock accesses was attempted and the time a lock was acquired. Along the z axis of the graph we have time.

For every *LOCK_ATTEMPT* and *LOCK_ACQUIRE* message pair there is one cube in the lock graph, similarly, for every *LOCK_ACQUIRE* and *LOCK_RELEASE* message pair. The time that the processor is blocked for is taken to be the cost of the lock.

Interpretation

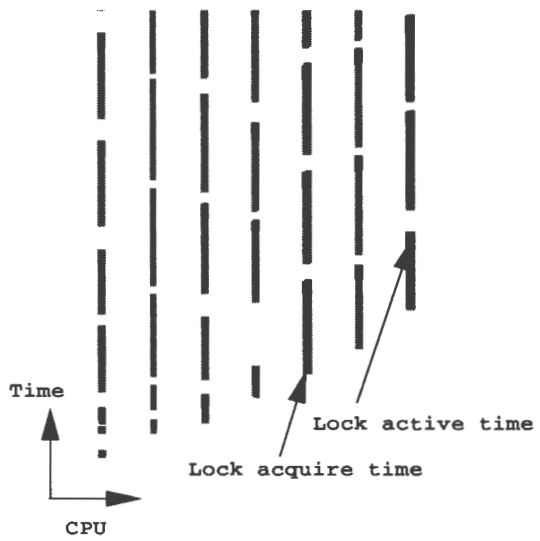


Figure 14: An example lock synchronization view

Figure 14 shows a typical lock synchronization view. From the view it is easy to see when each of the processors enter the lock acquire stage and when the lock is finaly available. Under perfect conditions the cube representing the *lock acquire* time should be very short, compared to the cube representing the *lock active* time.

Locks are used often, compared to barriers and semaphores, and form an important part

of a parallel program's synchronization. When ever a parallel program wants to update a shared datum it has to lock it before it can write to it. This locking process can be, as experience has shown, a very time-consuming operation, when for example more than one processor tries to access the locked datum then a bottleneck occurs. If, processor A has acquired the lock and the lock is in the active state, then processors B and C try to acquire the lock, but both of them block until A releases the lock again. This behavior can be clearly seen in figure 14, where seven processors contend for the same lock.

The visual representation of the lock behavior allows the user to quickly identify bottlenecks, the visual signature of lock access bottlenecks is very distinct. When the *Lock Synchronization View* is placed on top of of the first and second level cache views then lock bottlenecks can also be identified as areas of high cache block movement. Since every processor wants to write to the lock structure, the cache block containing the lock will move from processor to processor, further increasing the number of wasted clock cycles.

Interaction

The user can select any lock cube to find out more details about it. The information contains data such as the lock's memory location, the waiting processor's number, and the lock name. Additionally the lock attempt, lock acquire and lock release times, the number of blocked clock cycles, and the percentage of total lock wait time to which this lock contributes, is displayed.

The following is an example from Chiron's text window:

```
Lock Addr:0x40003030 CPU:1 Name = SyncLock
LOCK_ATTEMPT @2219787
LOCK_ACQUIRE @2220534: wait time 747 (0.11%)
LOCK_RELEASE @2220633: active time 99 (0.07%)
```

The selected lock cube 'SyncLock' at memory location 0x40003030 is accessed by CPU 1. The processor tries to access the lock at time 2219787 and blocked for 747 clock cycles until at time 2220534 the lock became available. The program held the lock for 99 clock cycles and released the lock at time 2220633. The 747 clock cycles the processor was blocked for contributed 0.11time.

General Synchronization Interaction

The *SyncViews* menu gives the user a wide range of interaction options. The menu is divided into functional groups, each of which are described in detail below.

Lens On/Off

For performance reasons the data in the three *Process Synchronization Views* is shown as lines instead of cubes. The default setting allows the user to identify problem areas from an overview and then to zoom into specific problem areas. Once a problem area has been located detailed information views, in the form of cubes, can be enabled. To facilitate this detail suppression the graph has a lens which works like a normal magnifying glass. The lens can be moved over any area of the graph, the covered area will be shown in detail.

All Detail

Using the lens is not the only way to see the data in full detail. Chiron provides an option to enable *full detail* displays for each of the synchronization views. When this option is enabled all lines are converted to cubes and information can be extracted from the graph. When a program shows substantial synchronization behavior it is advisable not to use this option, as the rendering time of the view increases to intolerable proportions.

Graph On/Off

If the user wishes to concentrate on one particular synchronization graph, he/she can do so by switching any other graph off. When a graph is switched off no trace of it is left on the screen. To gain accesses to a graph which is switched off, the user has to enable the graph again.

Overview

Chiron can shows some synchronization overview data in the text window.

```
Synchronization Overview
TotalWaitSyncTime =    695468
Barr WST =          0 (0.00%)
Sema WST =       5488 (0.79%)
Lock WST =     689980 (99.21%)
Lock AST =     138147
```

The *TotalWaitSyncTime* is the total amount of time a processor spent waiting on synchronization. The *Barr WST*, *Sema WST* and *Lock WST* indicate the amount of time that was spent waiting for each of the synchronization types. *Lock AST* is the time all processors spent holding an active lock.

Sort order

Each of the three views can be sorted according to three criteria: Sorted on Cost, Sorted on Address, and Sorted on Creation Order. When the data is displayed sorted on cost, then the most expensive synchronization object is displayed at the top of the view. When data is displayed sorted on address, then the synchronization object with the lowest memory location is displayed at the top of the view. The last option shows the synchronization objects in the order they were created in during program execution.

An average program execution will take a few million clock cycles; this means that there is a potentially very large number of data elements in the synchronization graph. On startup the user is presented with an overview graph which contains all the information. The user is advised to use the two time sliders to limit the time span which will be used for closer inspection.

3.5 Correlation Views

Each of the views discussed so far can convey a large amount of information about a particular metric, but sometimes that it not enough: the user might want to know how two metrics correlate. Chiron supports a range of correlation views, which are implemented as some form of visual data change. The aim of the correlation views is to change the target view enough to show the new metric, but without destroying the underlying metric data, thus producing a new set of visual signatures, which the experienced user can identify.

The following sections describe the correlations that Chiron supports.

3.5.1 Source Lines – Objects

Chiron can display a correlation between source lines and objects in a program. When a node of the *Source Line View* is selected all objects that where touched by that source line are highlighted in the *Object View*. The object nodes are highlighted in yellow and the selected source line in red.

Once the object nodes are highlighted the user can select each of the nodes to obtain more information. Chiron displays node information in the text window. (See section 3.3.4 for more information.)

The mapping from source line to touched objects can be used to determine if the source line can be optimized or not. If a source line touches a few very expensive objects then the chance of optimizing the source line by changing the access patterns to those objects is very high. If on the other hand the source line touches many very cheap objects then there is very little chance of optimizing the source line.

3.5.2 Objects – Source Lines

When a node in the *Object View* is selected all source lines that touched any part of that object are highlighted in the *Source Line View*. The source nodes are highlighted in yellow and the object node in red.

The mapping from objects to source lines can be used to determine where the cost of the object comes from. If an object is expensive and many source lines access it then the cost might be mainly due to infrequent accesses. In a case like this it would be difficult to optimize one particular source line. If an object is very expensive and only a few source lines access it then optimizing those few source lines is advisable. In the ideal case the most expensive object will only be touched by one line.

The *Object – Source Lines* correlation can also be used to find false sharing problems. This is done by selecting an object in the *Object View*, and then investigating each of the source lines that touch that object. If a source line does not contain an access to the selected object, then the object might have been touched due to false sharing.

3.5.3 Cache Block View – Source Line View

When a block in the first or second level cache block view was selected then the node in the *Source Line View* which was responsible for the block movement is highlighted. This information can be used to determine if the movement is a false sharing problem or if the cache block movement is due to a capacity miss.

The user can selected the highlighted node in the source line view and look at the corresponding source line. From the source line the user should be able to decide if the cache movement was due to false sharing or if the movement represents a problem.

3.5.4 Source Line – Cache Block View

Chiron can form a correlation between a source line and specific cache behavior. When a node in the *Source Line View* is selected Chiron can highlight all blocks in the first and second level caches that were touched by the selected source line. The user can investigate the selected cache blocks and make performance deductions from the cache block movement patterns.

If the highlighted cache region has high block movement traffic then the user's chance of optimizing the program by changing the code in the selected source line is good. The visual signature of the block movement pattern might point the user to investigate further on the grounds of coherence or replacement misses.

If, on the other hand, the highlighted cache block region does not have a high block movement traffic then the chance of optimizing the selected source line is low. In such a case the high cost of the source line might be due to the source line being called very often and touching a large amount of data. If the data does not collide with other data but must be brought into the cache, due to some startup cost, Chiron still registers the source line as expensive.

3.5.5 Object – Cache Block View

When a node in the *Object View* is selected all cache blocks that intersect with the object are highlighted. The highlighting criterion is: If the address space which is covered by a cache block intersects with the selected object's address space then the cache block is highlighted.

This view can be used to find out why an object has a high cost. The user can select an object and then switch to the cache views and dicover the source of the cost. The highlighted cache blocks can be queried for coherence or replacement misses and for which other objects or source lines were responsible for the high cost.

Chapter 4

Execution trace generation

To find the performance bottlenecks in a program we have to have detailed information about the program's execution and the resultant behavior of specific machine components. This chapter outlines our endeavour in obtaining this performance data.

The process of obtaining the performance and machine specific data is called *trace generation*. A *trace* is a collection of program events and architecture states.

4.1 Trace generation metrics

Two metrics had to be considered before we could decide on a trace generation system. These are disk space and the type of event monitor used.

If disk space is at a premium, writing the event information, also called trace data or trace, to disk is not feasible, as these trace files easily exceed 100MB for a medium size problem. In such a case, analysis tools could obtain the trace data directly from an execution or hardware monitor. Disk space would be saved, but the events would have to be regenerated the next time we wanted to analyze the data. Additionally, since not all events in a parallel program occur in a deterministic fashion, analysing a particularly interesting point in the event data in two different sessions would be very difficult.

If disk space can be spared, the events can be written to a trace file, which would then be analyzed. The advantage here is that all the interesting events are kept at hand and the

trace data can be reused during multiple analysis sessions. The non-deterministic nature of parallel programs does warrant the recording of the trace information.

4.2 Monitoring Techniques

It is reasonably difficult to monitor the execution of a program on a system without effecting the target program's performance and behavior. There are three basic ways in which event information can be obtained, hardware monitoring, software monitoring, or simulation.

Hardware monitoring typically requires specialized circuitry to gather and process traces from target machines. The hardware does not interfere with a program's execution; it is non–intrusive, but it is difficult to relate events to a specific program, which makes it difficult for the analyst to gain insight into the behavior of a program. Hardware monitors are very expensive and not always available for a target machine. The advantages of hardware monitors are that they are non–intrusive and permit real time performance debugging. Often these monitors record only counts of events rather than actual events themselves, because they have limited memory. [35]

Software monitoring tools make minor additions and modifications to source or executable files. These changes add small bits of instrumentation code to record the execution of program events or to collect and record data. The instrumentation code can be added at any stages of the compilation process.

At first a source-to-source transformer can add measurement code directly to a program's source code. The *m4* macro processor can be used in conjunction with the *ANL macros* [30] to parallelize a program.

The second opportunity would be to let a modified compiler insert instrumentation code while compiling a program. An early version of *AE* [26] was a modified GNU C compiler that inserted tracing code while compiling a C program.

A third possibility would be to augment the assembly language produced by a compiler before passing it to an assembler. The *tango* [20] tool augments a C or FORTRAN compiler's output by inserting recording and tracing code.

A fourth possibility would be to insert instrumentation code by rewriting an executable file: *pixie* [44] and *QP/QPT* [27], for example, do this with great success.

All of the above mentioned instrumentation techniques have their advantages and disadvantages, but the fact that adding extra code to a program changes the execution behavior of a program cannot be ignored. First, the extra calls to routines to handle tracing and profiling slow down the program, and second, the memory system and process synchronization behavior may be affected. When performance debugging a program, the analyst would like to have an accurate as possible event trace.

A third monitoring technique is software simulation. In this case the whole machine architecture is simulated, CPU, memory hierarchy and OS interaction. *Mint* [52] is such a simulator. Clear advantages of simulation are that any target architecture can be simulated, simply by changing the simulation software and/or parameters, and that simulators are effectively non–intrusive. A disadvantage of simulators is the time penalty incurred by simulating the machine architecture.

The *Mint* [52] simulator reads an executable file and converts the machine code instructions into simulation commands. To run a program, an interpreter *executes* each of the simulation commands. Since the code is interpreted, the simulator can record any event of interest to the performance analyst. The recording of events does not interfere with a program's behavior, as the original executable file is not changed and the whole simulation is controlled by a timed event–wheel.

While software simulation is the slowest method of event generation it is also the cheapest and most versatile. No special hardware is required and a program's behavior is not altered by the event recording, which makes the software solution price effective.

It was decided to use a software simulation event monitor to generate the system events for the Chiron system. The following section describes the event creation system in detail.

4.3 Tracing System Architecture

The tracing system is divided into six modules, the *Mint Kernel*, the *ParaDiGM Memory System Simulator*, the *Memory Allocation Tracker*, the *Stack Frame Tracker*, the *Process*

Synchronization Tracker, and the *Source Line Tracker*.

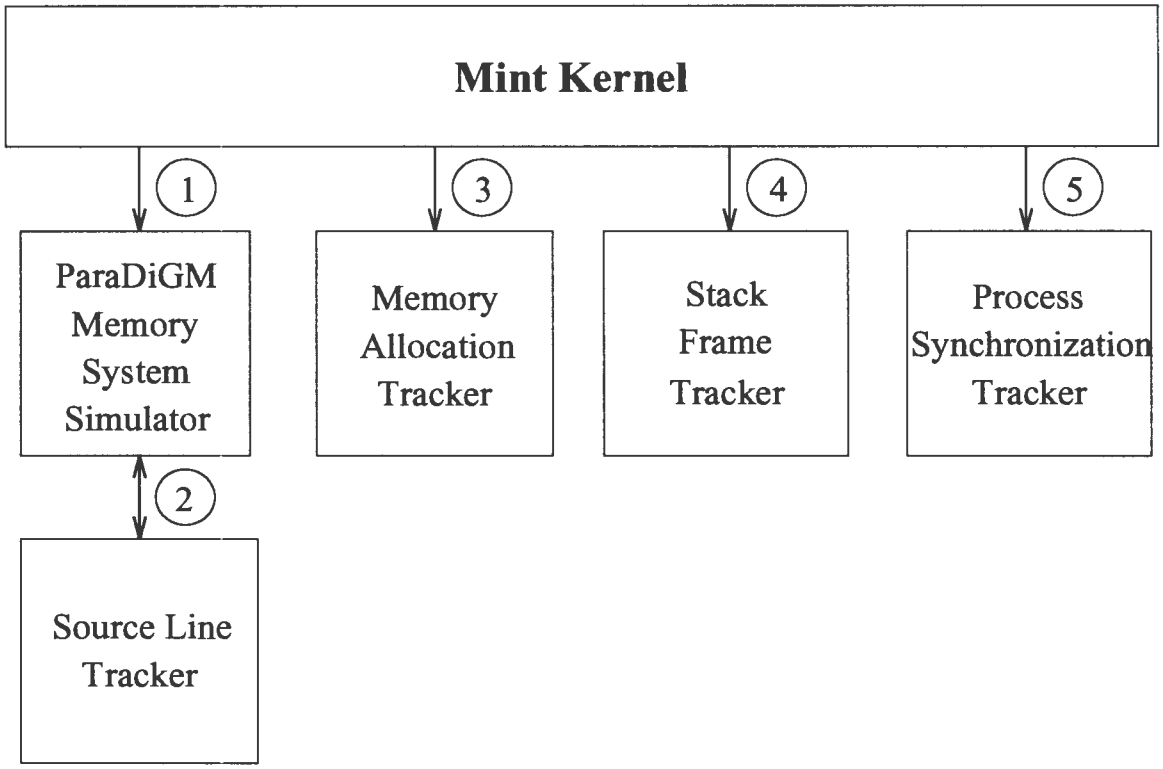


Figure 15: The Mint based tracing architecture

Figure 15 shows the tracing system architecture and the inter–module communication paths, labelled *1–5*.

4.3.1 Tracing System Module Description

The whole tracing system is based on the Mint [52] system simulator. Mint is a tool for generating memory reference traces of sequential and parallel programs. Mint reads a standard executable file and simulates each instruction. As part of the instruction simulation, Mint can optionally call a user–defined function. This allows an arbitrary action to be associated with any instruction. The ability to hook user–defined functions was used to transfer event information to the memory system simulator, and other event tracking modules.

The first module that the tracing system communicates with is the ParaDiGM [9] [16] Memory System Simulator (*MSS*). The simulator is event-driven and models the memory system

in great detail. Components such as the L1 and L2 cache and the buses are implemented as functional modules. This allows one to model concurrent accesses to caches, for example, cache accesses can continue while the cache is waiting for misses to be serviced.

The second module is the Memory Allocation Tracker (*MAT*). This module keeps a list of all memory blocks that where allocated and deallocated by the simulated program. Each memory block (called *object*) is given a unique name by which the performance analyst can later refer to it.

The third module is the Stack Frame Tracker (*SFT*). This module builds a function call list for each running process. The list contains information about what function and when it was called, and when the function returned to the caller.

The fourth module is the Process Synchronization Tracker (*PST*). All synchronization calls that a process makes are registered by this module. We register what kind of synchronization call it was, when it occurred, when it was serviced, and how long a process held a resource. The *PST*'s information forms the basis of many analysis techniques used in Chiron.

The fifth module in the Source Line Tracker (*SLT*). This module can be sent a request to map a particular code location in the executable back to a source code file and line number.

4.3.2 Tracing System Module Interaction

The inter module communication paths show how information is passed through the simulator.

Since we are simulating a RISC-based machine the memory hierarchy of the machine can only be accessed by the simple *read* and *write* instructions. The *Kernel–Memory System Simulator* date path (numbered 1 in figure 15) communicates every **read** and **write** instruction's information to the *MSS*. The memory location, the processor doing the operation, and the time of the operation are communicated.

The *MSS* logs a set of events that may happen during the course of the simulation. For some events we record the source line number which caused it. To map a memory location to a source line number, the *MSS* and the *SLT* communicate via the data path numbered

2 in figure 15. The *MSS* sends a memory location to the *SLT*, which returns a possibly correctly mapped *source line* and *source file* mapping. An error message is returned when the mapping fails and is also logged and later analyzed by Chiron.

To facilitate the easy identification of memory locations the tracing system associates a *name* with every data object that is allocated by the executing program. Every time dynamic memory is allocated via one of the standard techniques, i.e., **malloc** or **new**, the *Mint Kernel* informs the *MAT* via the *Kernel–Memory Allocation Tracker* data path (numbered 3 in figure 15) of this. The starting location in memory, the length of the data block, and a source line reference are communicated to the *MAT*. When memory is deallocated via one of the standard techniques, i.e., **free** or **delete**, the starting location is again passed to the *MAT*. The *MAT* builds up a list of active memory objects and tries to generate meaningful names to the allocated memory.

Memory object names can be generated in two ways. In the first technique the programmer augments the source code with a specific set of instructions, which are put into a comment block, so as not to influence the actual program. This allows the programmer to associate any name with a memory object allocation process. The name can contain *variables* which are then filled in by the *MAT*.

The second naming technique is used when the *MAT* does not find a comment block within a range of line numbers above the memory allocation function call. In this case the name is made up of the variable name to which the memory location pointer is assigned, the line number, and the source file name of the originating function call.

To easily identify what part of a program is executing at any given time, the Mint Kernel communicates with the *SFT* via the data path numbered 4 in figure 15. Every function *entry* and *exit* is communicated to the *SFT*; it records the function reference numbers and time of entry and exit. Chiron uses this information to draw a time-based program execution graph.

The data path numbered 5 in figure 15 is used to inform the *PST* of every call that a program makes to one of the standard process synchronization functions. The type of synchronization construct, *semaphore*, *lock*, or *barrier*, the time of the function call, the processor issuing the call, and the memory location of the construct are communicated.

The *PST* logs all these events for use by Chiron.

4.4 Generating a trace file

It is a four-step process to generate a set of trace files which can be analyzed and visualized by Chiron. The first step is to link the target serial or parallel program with a *Mint stub library*. The library contains a set of empty *system call* stub routines. The stub routines are replaced with calls to internal simulator routines as the target program is analyzed by *Mint*. We link the program with the library to reduce the size and analysis time of the target program, but the step is optional.

The second step is to set an environment variable which will form the base name of the trace system log files. For example, if the **BASE_NAME** variables is set to **trace.1**, then the trace log files would be named as in table 1:

trace.1	For the memory system log data
trace.1.back_ref	A list of all source files that make up the target program.
trace.1.proc_ref	A list of all functions that make up the target program.
trace.1.struct_ref	A list of all structures that where declared in the target program.
trace.1.line.data	The stack frame trace log.
trace.1.sync.data	All process synchronization event logs.
trace.1.sim.run	Summary information about the trace.

Table 1: Trace log file names and descriptions

The third step is to configure the ParaDiGM memory system simulator. The simulator has to build the memory system architecture in its internal tables before it can simulate it. For flexibility the authors of the simulator opted for a file-based configuration technique. Appendix A gives more detail about this.

The fourth step is to let *Mint* execute the target program. This will produce the files as listed in table 1. These files are read and analyzed when the Chiron performance analysis tool is run.

Appendix B gives detailed descriptions of the file formats that are written out by the tracing systems.

Chapter 5

Case Studies

To test the usefulness of Chiron three case studies have been used. The first program is a very simple example of a parallel program, the multiplication of two vectors on eight processors. This example is simple enough to illustrate the use of Chiron without getting bogged down in understanding complex dynamic program behavior.

The final two case studies are examples of real programs executing small but realistic workloads, and show the use of Chiron in a more typical performance optimization environment. Both these programs are taken from the widely available SPLASH benchmark suite [48].

5.1 Vector multiplication

5.1.1 The Vector Multiplication Program

The vector multiplication program uses the ANL macros for its multitasking constructs. [30]

In the initial version of the program processor zero initialized two arrays with numbers, and the seven other processors multiplied the two arrays (element by element) and stored the results in a third array. Processor zero did the work distribution. After all the elements were added, processor zero printed out the result array.

Each array contains 1024 elements, with each element being 4 bytes in size.

The code segment below shows the element multiplication routine:

```
void Multiply()
{
  int i;

  GETSUB(Global->GS, i, VECTOR_SIZE - 1, Global->nprocs)

  while (i >= 0) {
    Global->c[i] = Global->a[i] * Global->b[i];
    GETSUB(Global->GS, i, VECTOR_SIZE - 1, Global->nprocs)
  }
}
```

5.1.2 The Target Machine Architecture

The program was run on a shared-cache multiprocessor machine. Figure 16 shows the machine architecture.

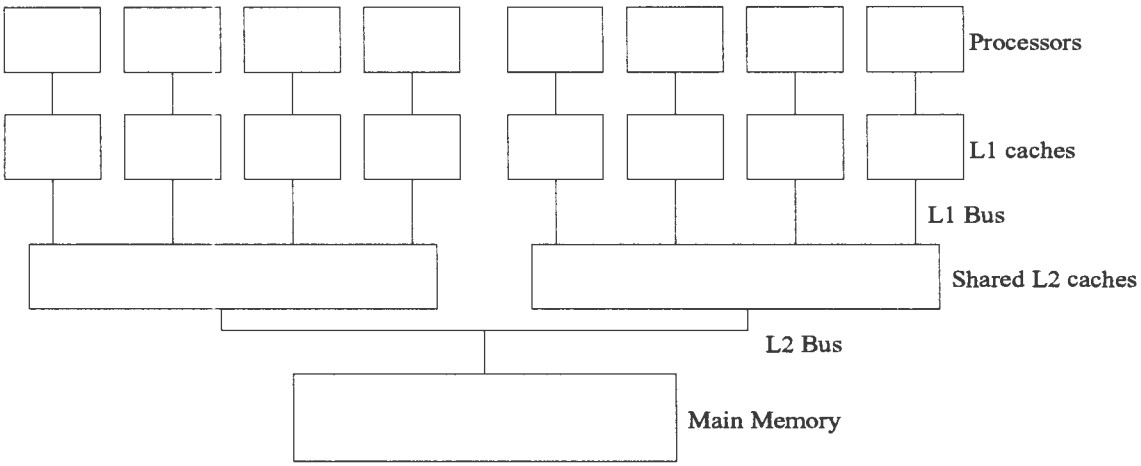


Figure 16: The Paradigm architecture

The architecture parameters are: The first-level cache is 4 way set associative with 2048 lines each 32 bytes wide, so 256Kbytes big. The second-level cache is 4 way set associative

with 2048 lines each 128 bytes wide, so 1024Kbytes big.

Four processors and first-level caches share one second-level cache.

5.1.3 Using the visual analysis techniques

Chiron was used to analyse the performance of the parallel program. Figure 17 shows pictures taken during the analysis process. We analysed three versions of the program. The base line version was a naive implementation, with poor parallel performance. The more optimized version achieved a 102% and the final optimized version achieved a 169% performance increase.

The base line vector multiplier

The program trace shows the following overall results:

```
Elapsed simulated cycles: 1108752

Private: 18868 reads, 6513 writes
Shared:  8285 reads, 4116 writes
```

The *Performance Overview* (figure 17 A) clearly shows that the lock synchronization time (the blue bar component) dominates the graph. In this case the blue bar is associated with a specific set of performance problems, namely lock synchronization, thus quickly identifying a next step in the performance analysis. Closer inspection shows that the *total* synchronization time for each of the work processors, ranging from 92702 to 102118 clock cycles, is 2.6 times *higher* than the total memory wait time, which ranges from 32686 to 41835 clock cycles. Since the graph clearly shows the high synchronization wait time, the next step was to look at the lock behavior.

The *Lock Synchronization view* (figure 17 B) shows the lock access behavior. The yellow bar indicates when a processor is stalled waiting for access to a lock, and the blue bar indicates the amount of time the lock is active for. This view clearly shows that the processors are

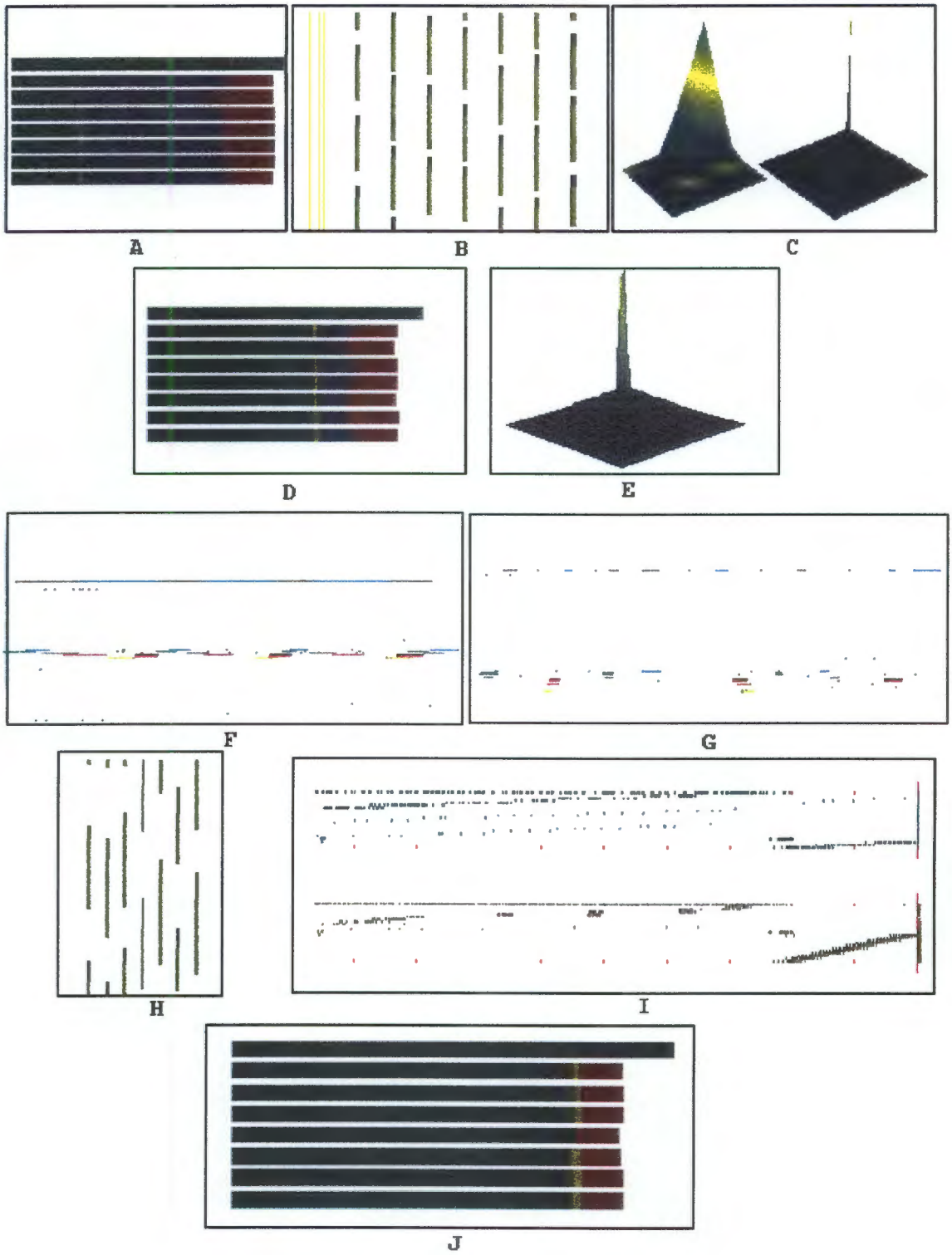


Figure 17: Array multiplication analysis process

contending for access to the lock; this can be seen from the overlapping yellow bars. The total synchronization wait time of all eight processors was 695280 clock cycles. Only one lock was used and 99.21% (689980 clock cycles) of the synchronization time was spent on this one lock. In comparison, the lock was active only for 138147 clock cycles. These facts indicate that a synchronization problem causes a performance bottleneck.

To find the source of the bottleneck we looked at a combination of the *Source Line* and the *Object* views. Figure 17 C shows these two images, the source line view on the left and the object view on the right. The object view is nearly totally flat, with a very high peak at the top of the graph. On closer inspection of the peak we found that it represents the work partition variable, with 2605 caused misses. The source line view shows the lines which access this work partition variable. The lock structure, with 1053 misses, is the second most expensive object in the object view.

From this we saw that access to the work partitioning variable and the associated lock structure, which protects access to this variable, represents the main performance bottleneck. To improve the performance of the parallel program, we had to find a better work partitioning algorithm. The original algorithm stated that each processor uses the work variable to determine which element in the vector array to calculate. The performance problem lies in the granularity of the dispatched work. Each processor gets one work index number for every calculation it has to perform, but since accesses to locks are very expensive, we want to dispatch a set of calculations to a processor at a time.

We restructured the work partition algorithm, by taking into account that a 32 byte first-level cache block spans eight consecutive vector elements, so that each work index now represents eight vector indexes. In addition to this we also aligned the data to a first-level cache block.

An optimized vector multiplier

The program trace shows the following overall results:

```
Elapsed simulated cycles: 548687
```

```
Private: 15541 reads, 6770 writes
Shared:  2909 reads, 3220 writes
```

The *Performance Overview* (figure 17 D) now shows that the lock synchronization time has dropped to between 7591 and 10506 clock cycles, a reduction by a factor of 10. In comparison the memory system wait times are now higher than the lock synchronization times, ranging from 11022 to 14199 clock cycles, but still significantly lower than in the first version of the program. The work processor's active time has increased from 26% to 66%. The overall performance improvement was 560065 clock cycles, which means that the program showed a 102% performance increase.

The high memory system miss time is the next optimization target. From the *Performance Overview* we move to the *Object View* (figure 17 E). The peak of the graph again represents the work partition variable and the associated lock, but in this case the number of misses these two objects have caused were much lower, 360 and 341 misses respectively. The performance bottleneck still seemed to be represented by these two objects; the third most expensive object only caused 24 misses.

To find the cause of the misses, we used the *Object View* to *Cache Block View* correlation. By selecting the most expensive object in the *Object View* and mapping its address space onto the cache block views, we can clearly see the cache behavior of the work partition variable. Figure 17 F shows the first-level and second-level cache behavior of this object, and the top section shows the second-level cache behavior. We can clearly see that the second-level time bars are relatively long compared to the first-level time bars, which suggests that the work variable has good second-level cache locality. The first-level cache behavior clearly shows which processor currently accesses the variable. No performance bottleneck is clearly shown by the views.

By selecting the second most expensive object in the *Object View* and mapping its address space onto the cache block views, we can clearly see the cache behavior of the lock structure. Figure 17 G shows the first-level and second-level cache behavior, and again the top section shows the second-level cache behavior. In contrast to figure F, this figure clearly shows the short time bars, which represent a high number of cache misses, and thus a performance

bottleneck. To confirm the performance bottleneck, we looked at the *Lock Synchronization View* (figure 17 H) which clearly shows multiple processors contending for the lock structure.

The high memory wait time does not just come from the cache movement of these two objects, and therefore to find possible memory performance problems we looked at the high–frequency components of the first-level and second-level cache views. This was done by representing the cache movement time bar start and end times as points. Figure 17 I shows that the second-level cache shows much block movement. Further investigation showed that false-sharing between the two second-level caches is occurring. Two processors not connected via the same second-level cache access the same cache block, which caused the block to move from one second-level cache to another.

To further improve the performance of the program we increased the work load of each processor to encompass one second-level cache block. This would decrease the number of accesses to the lock structure and the work variable, and improve the second-level cache behavior of the vector data.

The final optimized vector multiplier

The program trace shows the following overall results:

```
Elapsed simulated cycles: 412313

Private: 14965 reads, 6578 writes
Shared:  2333 reads, 3124 writes
```

The *Performance Overview* figure 17 J clearly shows that the synchronization time has dropped to insignificant levels, ranging from 67 to 1281 clock cycles, and that the memory wait time has also decreased, ranging from 5256 to 5684 clock cycles. The work processor's active time has increased from 26% to 87%. The overall performance improved by 169% compared to the first vector multiplier, and by 33% compared to the second vector multiplier.

5.1.4 Conclusions

As the first case study we were able to improve the execution time of the parallel vector multiplication program by 169%, effectively decreasing the number of execution clock cycles from 1108752 to 412313.

The visual representation of the performance data (totalling 1.2MB of raw data) enabled us to quickly identify performance bottlenecks and to find the cause of these bottlenecks. There were two major problems in the application. First, excessive synchronization was caused by incorrect work partitioning. Second, the second-level cache false-sharing problem was caused by two processors, connected to the same second-level cache, accessing consecutive first-level cache blocks which did not map into the same second-level cache block.

The pictures in figure 17 show the images from which our analysis decisions were made. It is acknowledged that the static two-dimensional images shown here do not really convey the dynamic nature of the interactive three-dimensional graphic produced by Chiron.

This example shows that shared memory parallel program performance data can be analysed using visual signature recognition techniques and that these techniques enable the analyst to quickly identify performance bottlenecks and their causes.

5.2 Barnes–Hut

5.2.1 A brief description of Barnes–Hut

The Barnes–Hut application simulates the evolution of a system of bodies under the influence of gravitational forces. Every body is modeled as a point of mass and exerts a force on all other bodies in the system. The simulation progresses in time–steps, each step computing the net force on every body and thereby updating the body's position. The largest part of the program's execution time is spent in the force computation phase. The naive implementation of the program has a time complexity that is $O(n^2)$ in the number of bodies. The Barnes–Hut algorithm has a time complexity of $O(nlogn)$.

The Barnes–Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The root of this tree represents a space cell containing all bodies in the system.

The tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The result is a tree whose internal nodes are cells and whose leaves are individual bodies. The tree is adaptive in that it extends to more levels in regions that have high particle densities.

To compute the net force acting on a body, the tree is traversed once per body. The force–calculation algorithm for a body starts at the root of the octree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is sufficiently far removed from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of that cell, and the force this center of mass exerts on the body is computed. If the center of the mass is not sufficiently far away, all sub-cells of the cell have to be visited. In this way, a body traverses deeper down those parts of the tree which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales.

5.2.2 The C++ version of Barnes–Hut

We did not use the SPLASH version of Barnes–But, but decided to try to optimize a version written in C++ by Philip Machanick. This version of Barnes–Hut uses the SpaceLib [39] memory management library to optimize the memory layout in a parallel program. The aim here was to investigate an already optimized C++ program and see how it compares to an equally optimized C program.

The differences between the C and the C++ implementations of Barnes–Hut are detailed below:

- In the C++ implementation all global variables have been removed and are now members of classes.
- The synchronization is done through class constructors and destructors instead of explicitly locking data regions.
- The whole program is located in shared memory. We use the *sproc* semantics to allocate the program and data space in shared memory.

- ANL Macros were replaced by C++ parallelization constructs.
- A general *array* class is used to pad and align array elements.
- Inheritance and virtual functions are used instead of similar looking structures for cells and bodies, eliminating the need in most cases to check a flag to take different actions for bodies and cells.
- Critical regions are protected by declaring a variable of class *Lock*. This holds a lock as long as the variable is in scope.
- Barriers are implemented similarly. A barrier is entered when the object is constructed and exited in the barrier destructor.

The main thrust behind the C++ conversion was to make the program more maintainable and to maintain or improve the performance.

5.2.3 The Target Machine Architecture

Figure 18 shows the architecture of the simulated machine on which Barnes–Hut was run on.

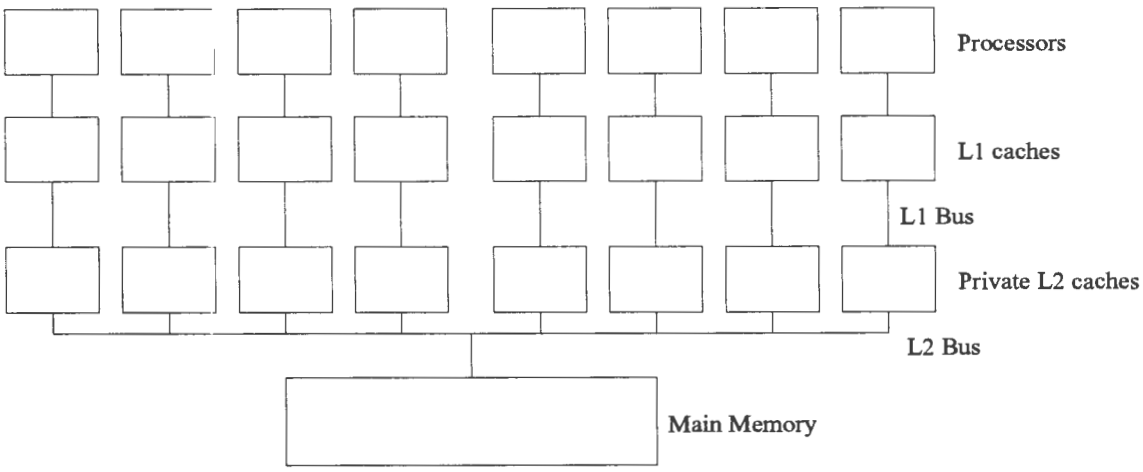


Figure 18: A sample architecture with 8 first and 8 second-level caches

The architecture parameters are: The first-level cache is 4 way set associative with 1024 32 byte wide lines, so 128Kbytes big. The L2 cache is 4 way set associative with 2048 128 byte wide lines, so 1Mbyte big. Each processor has its own L1 and L2 cache.

Barnes–Hut was run for four time steps and with 256 particles.

5.2.4 Using the visual analysis techniques

Chiron was used to analyse the performance of the parallel program. Figure 19 shows pictures taken during the analysis process.

The first version of Barnes-Hut

The execution trace of the original optimized version of Barnes–Hut shows the following overall results:

```
Elapsed simulated cycles: 54796492

Private: 2858519 reads, 255940 writes
Shared:  3980387 reads, 3504833 writes
```

The *Performance Overview* (figure 19 A) shows that all eight processors are active for an average of 81% of the program's execution time. The synchronization and memory miss wait times average at 11% and 7% respectively. Because the synchronization time dominates the memory miss time, it is a good starting point for optimization.

Synchronization behavior On closer inspection of the *Synchronization View's* textual overview we found that only 3% of the synchronization time is spent on locks, the rest of the time being spent blocked on semaphores. The *Semaphore Synchronization View* (figure 19 B) clearly shows that the first time step accounts for the bulk of wasted cycles, and that all processors are waiting for one processor to finish, clearly indicating a load balancing problem.

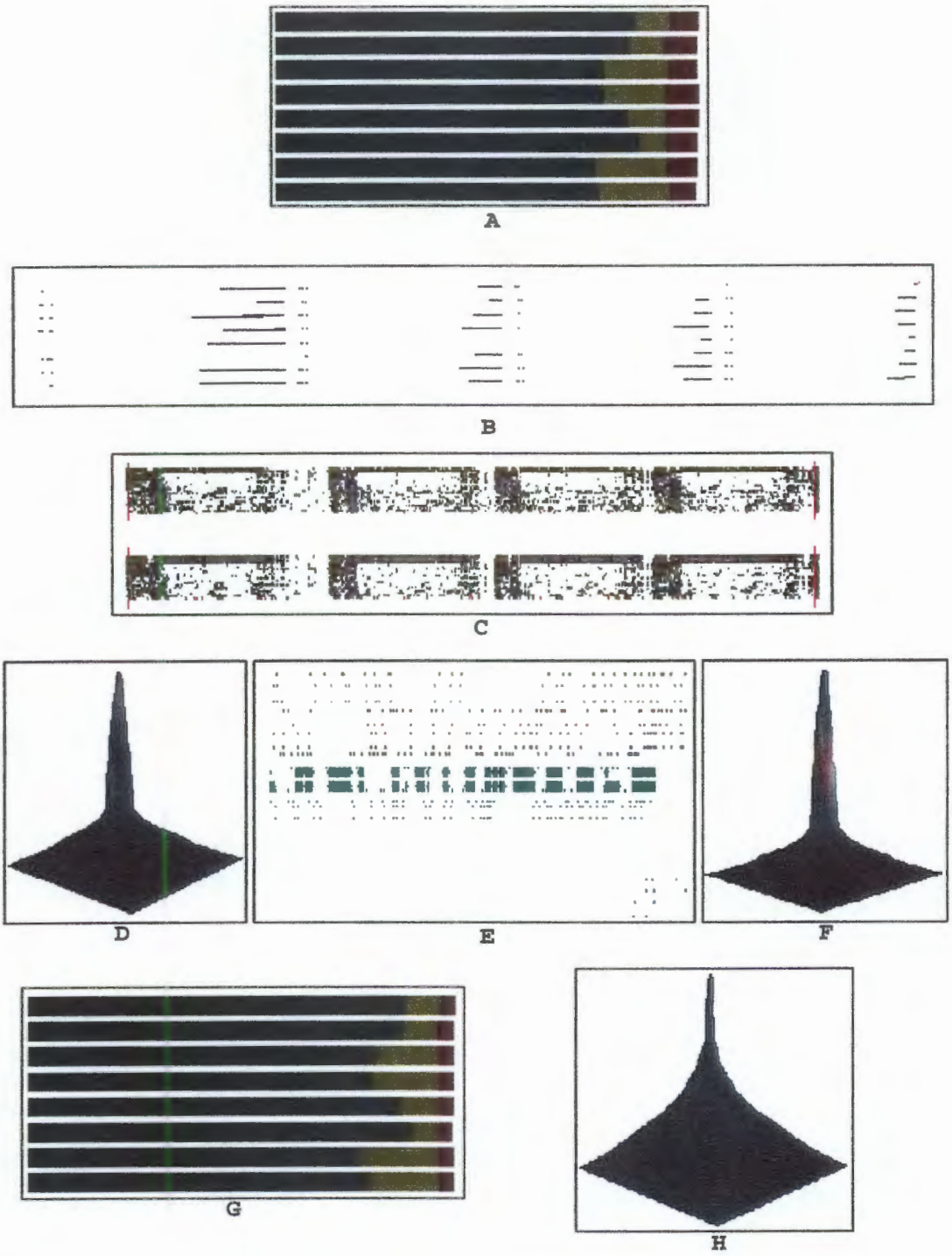


Figure 19: The visual Barnes-Hut analysis process

The *minor_out_synch* semaphore is used to simulate a barrier at the end of the force computation step. All processors have to complete their calculations before the particles can be moved inside the octree space structure. The work load assigned to a processor is directly related to the amount of time it takes to finish one computation step. Which means that not every particle force calculation takes an equal amount of time. To start off the simulation the program assumes a uniform particle distribution, and that every particle will cause the same amount of work. Only *after* the first time step, incorporating a force calculation and a resultant movement on each particle, can the algorithm calculate a good work load distribution.

Since we only simulated four time steps, the load imbalance of the first time step has a very large adverse effect on the average semaphore block time. Once the load balancing has been done, the average block time is reduced to 144724 clock cycles. We decided not to rearrange the barrier synchronization, because in a normal program run many more time steps are simulated, and the initial inefficiency will not be very important.

The prompt elimination of the synchronization time as an optimization target is a good example of the power of a visual performance analysis system. In this case the eye quickly identified a group of long time bars near the beginning of the view and a set of shorter bars later on. This example has shown, that phases in a computation show up clearly in the temporal views of a time-stepped application (see figure 19 C) leading naturally to deductive reasoning based on application knowledge.

Cache behaviour The *Object View* (figure 19 D) shows the most expensive objects, *first_page*, which caused a total of 20261 cache misses. The eight *first_page* objects, one per processor, control access to the *B_cell* objects implementing the octree space structure in Barnes–Hut.

Using the *Object to Cache Block* correlation and mapping a *first_page* object's address space onto the first-level and second-level caches we found that 99% of the misses happened during the octree building phase. The true-sharing coherence misses occur because all eight processors need write access to information stored in the *first_page* objects. The small percentage of cache movement that is not due to coherence misses is caused by replacement

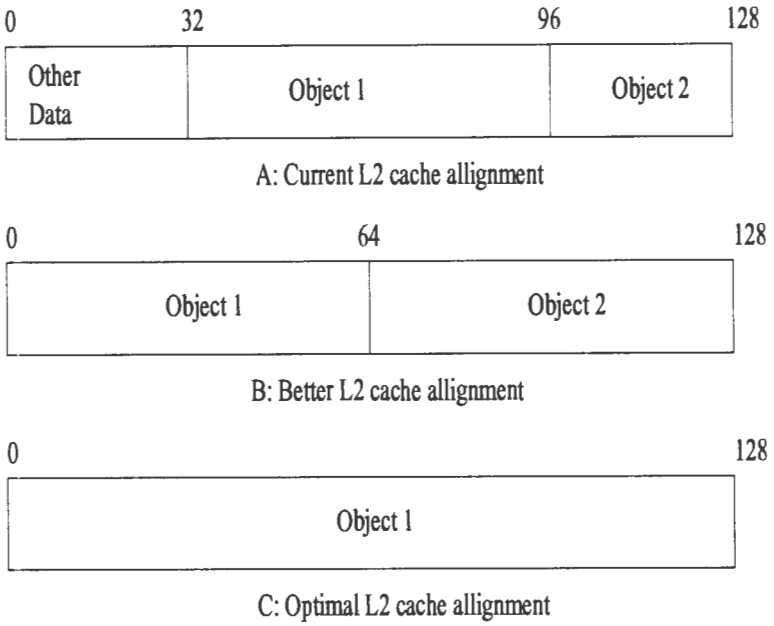


Figure 20: The second-level cache layout problem

misses. In this case the *precinct* object collides with the *first_page* object.

We improved the performance problem by moving the space control elements out of the *first_page* objects and into the *cell* objects.

The *precinct* objects were the second most expensive objects in the program, causing a total of 12702 cache misses. When a *precinct* object's address space was mapped onto the first-level and second-level cache views, Chiron clearly showed that the *precinct* object covers two cache lines in the first-level cache (see figure 19 E). The *precinct* object is 44 bytes big and a first-level cache line is 32 bytes wide. With some foresight we had already padded the *precinct* objects to 64 bytes and had aligned them to 32 bytes, so that they fitted perfectly into the first-level cache. Chiron also showed that two *precinct* objects occupied the same second-level cache block. The 32-byte alignment of the *precinct* objects worked very well for the first-level cache but not for the second-level cache.

Figure 20 *A* shows the layout problem that we found. The first *precinct* object is aligned on a 32-byte boundary which solves the first-level cache false-sharing problem. In the second-level cache the block alignment does not fall at the beginning of the cache block, so that what is block aligned in the first-level cache is not block aligned in the second-level cache.

A better solution is to block align the object to the second-level cache, which will ensure first-level and second-level cache alignment. The second-level cache alignment as shown in figure 20 *B* is optimal as long as only one processor accesses the two objects. As soon as more than one processor accesses the objects in the second-level cache block false-sharing will occur. Figure 20 *C* shows the optimal object alignment and padding for the *precinct* object. We pad and align the object to fill the second-level cache.

We solved the false-sharing problem on the *precinct* object by padding and aligning the data to the second-level cache size. In this case the problem was easily solved by padding and aligning.

5.2.5 The optimized version

The execution trace of the optimized version of Barnes–Hut shows the following overall results:

```
Elapsed simulated cycles: 53008705

Private: 2874646 reads, 255872 writes
Shared:  3978236 reads, 3502662 writes
```

The *Performance Overview* (figure 19 G) clearly shows that the memory miss time has been improved. The eight processors are now active for an average of 84% of the program's execution time. The synchronization and memory miss wait times average at 11% and 3% respectively.

The improved memory system behavior of the *first_page* and *precinct* objects is shown by comparison of figures 19 F and H. Figure F shows the placement of the object in the initial version of the program. The objects clearly occupy the peak of the graph, thus indicating their high cost. Figure H shows the position of the objects after the optimizations were done. All objects have moved out of the bottleneck region and now every instance of the objects only caused 8 misses, 2 per time step.

The overall program has shown a 3% performance increase.

5.2.6 Performance differences between the new and the old version

We set out to analyse the performance of the C++ Barnes–Hut program, but the results we obtained mean little when they are not seen in context. Performance measurements from the original SPLASH version and those from the new C++ version are compared in table 2.

	SPLASH	C++
Number of Reads	5633399	6809387
Number of Writes	3196679	3741164
CPU cycles	22321604	22050080
L1 Cache info		
Cache Misses	43041	29825
Cache Protects	12433	9389
Clean Replaces	258	666
Dirty Replaces	210	2932
Clean Invalidates	30989	18788
Dirty Invalidates	2644	261
BMI Shared	32719	21349
BMI Private	5077	6875
L2 Cache info		
Cache Misses	22289	14663
Cache Protects	0	0
Clean Replaces	0	0
Dirty Replaces	132	237
Clean Invalidates	16405	8545
Dirty Invalidates	0	0
BMI Shared	15735	9824
BMI Private	2111	3659

Table 2: Barnes-Hut, SPLASH, and C++ performance figures.

5.2.7 Conclusions

The aim of the C++ version was to improve maintainability and speed of the program. We believe that the C++ version is more maintainable than the original C version. All the data have been moved into classes and the control structure has been moved into methods. The optimization changes we made to the algorithms and data structures were easier to implement in C++ than C.

The C++ conversion brought a higher program overhead in reads and writes with it, but the overall program performance was increased. Analysing the comparison table 2 we found that the first-level and second-level *cache miss rate* and the first-level's *cache protects* show that the C++ version has a better data layout than the C version. The lower cache misses and cache protects mean that the data is better localized and that different processors do not interfere with each other as much. The low number of *invalidates* show that the false-sharing problem has been solved. In the C version the large numbers of *clean invalidates* can be attributed to false-sharing. The high number of *BMI's (Block Move In)* in the C version underlines our findings that the data layout was not optimal.

The visual techniques used in Chiron again enabled us to quickly find performance problems in a parallel program. The prompt elimination of the synchronization time as an optimization target was a good example of the power of the visual analysis techniques. The quick identification of the *first_page* and *precinct* object false and true-sharing bahavior, respectively, enabled us to improve the performance of the program by 3%.

The Chiron performance analysis tool was a valuable aid in our search for improved program performance. The performance bottlenecks were clearly visible and easily identified. The object view gave a very good indication of the problem areas in the program's data structures. The synchronization and cache behavior views were invaluable in the analysis of the time stepping behaviour and the overall program performance.

5.3 Water

5.3.1 A brief description of WATER

WATER is an N–body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a specified number of time–steps, which should allow the system to reach a steady state. Every time–step involves setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions. The total potential is computed as the sum of intra– and inter–molecular potentials. To avoid computing all the

$\frac{n^2}{2}$ pair-wise interactions among molecules a spherical cutoff range is used with radius equal to half the box length. The box length is computed by the program to be large enough to hold all the molecules. The simulation can be used to predict a variety of static and dynamic properties of water.

5.3.2 The Target Machine Architecture

The program was run on a shared memory multiprocessor machine. Figure 21 shows the memory architecture of the machine.

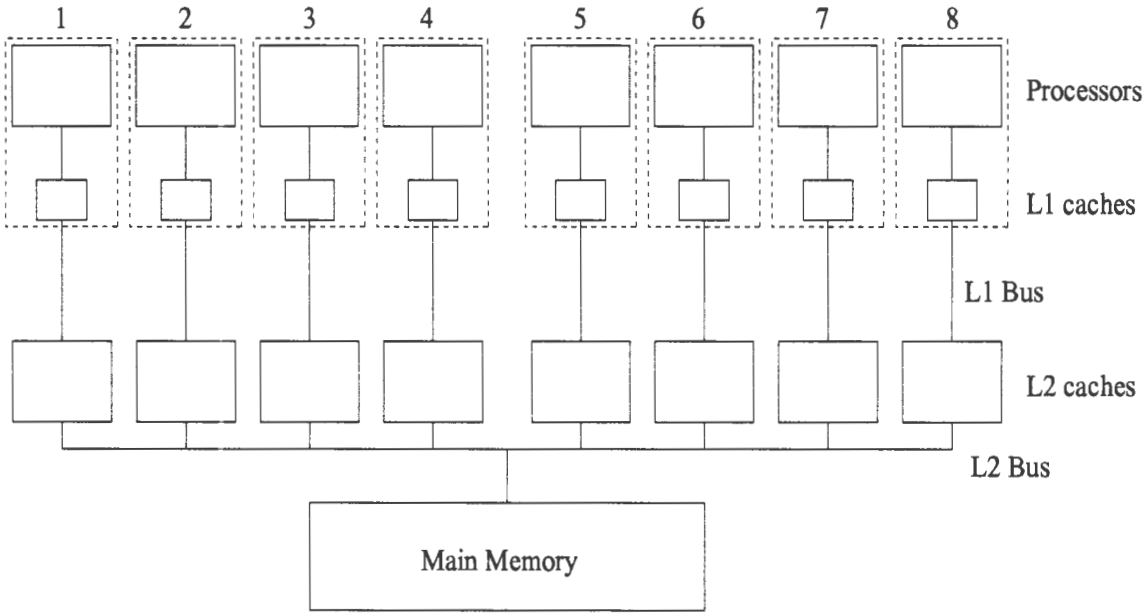


Figure 21: Memory architecture of a common shared memory multiprocessor

The architecture parameters are: The first-level cache is 128Kbytes big and is 4 way set associative with 1024 32 byte wide lines. The second-level cache is 1Mbyte big and is 4 way set associative with 2048 128 byte wide lines. Each of the 8 processors has its own first-level and second-level cache.

5.3.3 Using the visual analysis techniques

Chiron was used to analyse the performance of the parallel program. Figure 22 shows pictures taken during the analysis process.

Analysis results

The simulation was run on a simulated 8 processor machine and with a data set of 125 molecules for 2 time steps. The program trace shows the following overall results:

```
Elapsed simulated cycles: 376649778

Private: 15201554 reads, 755290 writes
Shared:  17251178 reads, 10634095 writes
```

The *Performance Overview* (figure 22 A) clearly shows that the eight processors have very little synchronization or memory wait time cost. The color bars on the right are very short compared to the green bars on the left.

The average memory miss wait time is only 0.98% of the total execution time of the program. When compared to the previous two examples in chapter 5.1 and 5.2 the WATER application shows very good memory system behavior characteristics.

The average synchronization wait time is slightly higher, at 1.46% of the total execution time of the program. A glance at the *Synchronization View* shows that from the total of 5501661 wasted synchronization clock cycles 5435087 (or 98.79%) were wasted in semaphore constructs. The *start_delay* semaphore was responsible for 3512479, the *interf_delay* semaphore for 1906312, and the *poteng_delay* for 16296 wasted clock cycles.

On closer inspection of the *start_delay* semaphore behavior we found that the WATER simulation has a load balancing problem. This semaphore is used to simulate a barrier, all processors have to wait on the semaphore before they can continue with the simulation. The reason why some processors have to wait for up to 445858 clock cycles is that the work load assigned to each processor is not balanced. The number of inter-molecule and

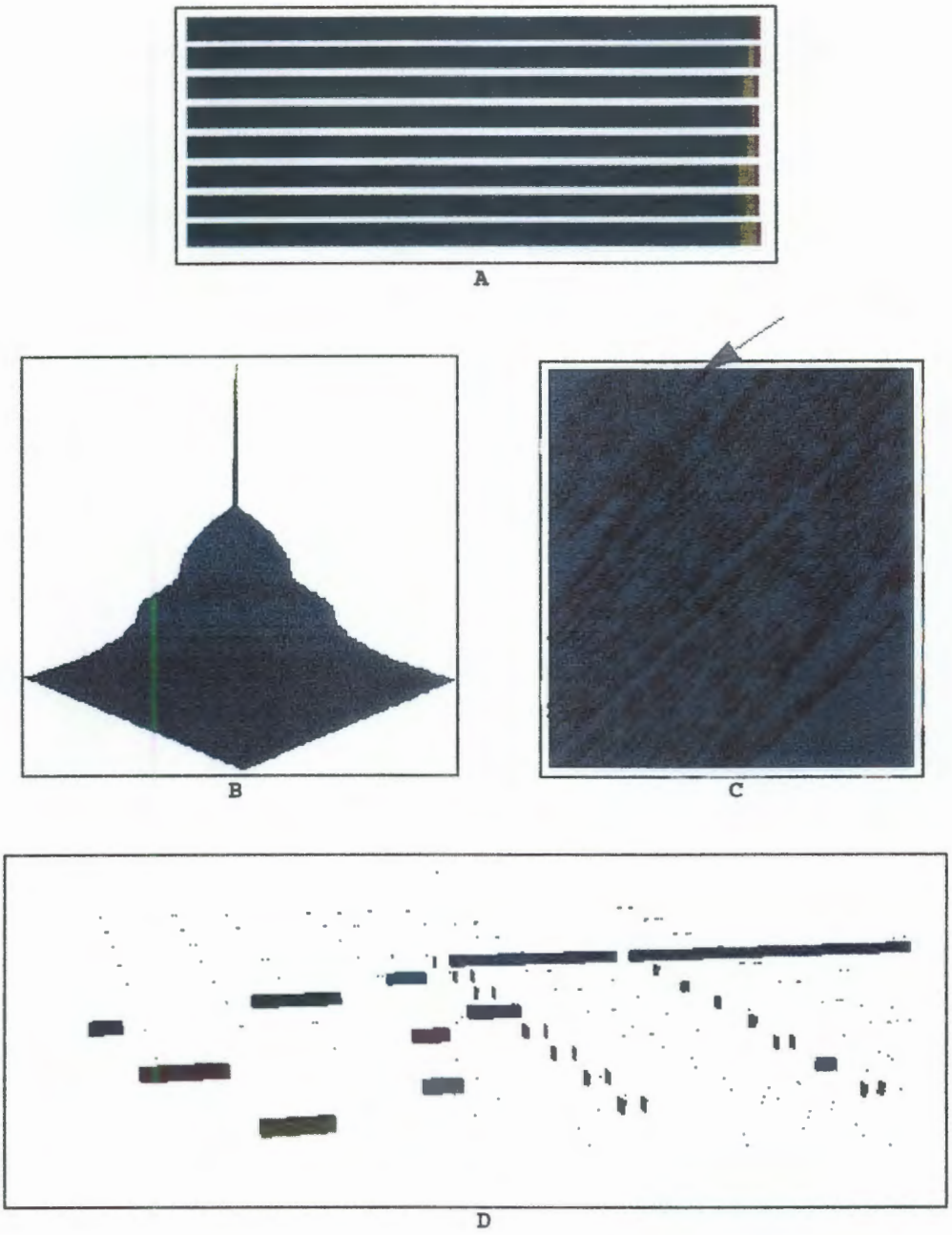


Figure 22: The WATER analysis process

boundary collisions is an indication of the work load. Before the first time step of the simulation the application assumes that all molecules are distributed evenly through space and that each processor will have the same amount of work. During a time step the number of inter-molecule and boundary collisions are counted and these counts are used to calculate the actual work load for the next time step.

The general assumption is that a given set of molecules will produce the same amount of work at each time step. The first time step is an exception, however, since work load data is not available. Since our simulation only runs for two time steps, the first time step's wait time dominates and skews the results. We decided not to change the work load balancing algorithm.

With the synchronization problem solved the next step was to investigate the memory system performance.

The memory system wait time is very small, but our aim was to identify problems in the memory system behavior. As a starting point we looked at the *Object View* and identified the most expensive objects and their relative position in memory. Figures 22 B and C show the *Object View* sorted by **cost** and **memory location** respectively. Figure 22 B clearly shows the cost peak. Figure 22 C shows a top down view of the graph, which clearly shows that the most expensive object (as indicated by the arrow) is not surrounded by high cost objects, the dark areas, thus indicating no false-sharing on this object.

The most expensive object is the *start_delay* lock, which caused 475 cache misses. After the object's address space has been mapped onto the first-level and second-level cache views the cache movement behavior becomes apparent. Figure 22 D shows the second-level cache behavior of the *start_delay* object. In the figure time progresses from left to right, and along this time line **three** synchronizations are clearly visible. The first one shows relatively long time bars, clearly indicating that processors are waiting for access to the synchronization constructs. The other two synchronizations show very short time bars (an average of 50 clock cycles) which indicates that not much work is done while the lock is held.

The *start_delay* object shows true-sharing behavior. Every time a processor needs to write to the *start_delay* object and it does not have write access to the object, a cache coherence miss occurs and the object is moved out of its current cache and into the requesting cache. Since

the object is already 128 bytes large, and thus second-level cache block size padded (and cache block aligned), no simple structural change will improve the performance problem.

Other expensive objects, including other lock access structures, did not show a clear pattern which we were able to identify via visual signatures.

5.3.4 Conclusions

Examination of the memory and synchronization performance shows that the most expensive objects, as seen from the *Object View* and the *Synchronization View*, are those containing the locks used in process synchronization. From the address space to cache block mapping of these objects we saw that most (99%) of the misses caused on these objects were caused by *true-sharing* coherence misses, occurring synchronously with the acquisition and release of locks. The fact that these misses were caused by the synchronization behavior means that the misses cannot be eliminated without changing this behavior. This suggests that the most promising approach for further optimizing WATER would be to design a different load balancing and synchronization strategy that would cause less sharing.

Although we did not manage to improve the performance of WATER, we included it here to demonstrate the type of subtle behavior that Chiron can easily expose.

Chapter 6

Conclusions

Chiron is a visualization system for displaying the memory system and synchronization performance of shared-memory multiprocessor applications. The system uses three dimensional graphics to display large amounts of both code-oriented and data-oriented performance information. Chiron goes beyond previous performance debuggers and visualizers in providing a greater variety of information and new types of visualizations. The system is designed to isolate problems such as low cache block utilization, improper layout of data in memory resulting in excessive replacement interference, improper partitioning of work among the processors resulting in excessive coherence interference, and synchronization inefficiencies. The user interface facilitates a variety of interactions that help the user identify performance problems and understand complex data correlations.

Although Chiron does not suggest solutions to performance problems, we believe that the intuitive and direct display methods contribute to a deeper understanding of a program's memory and synchronization behavior. Chiron maps source lines, data structures, function calls, and hardware components, like cache lines, directly to graphic displays.

Data correlation views provide additional insight into a program's behavior that would be difficult to achieve by analyzing statistical data.

6.1 Summary of performance analysis results

The Chiron system was used to identify performance problems in three parallel applications: Vector Multiplication, Barnes-Hut, and WATER. For all three of the applications we were able to easily identify memory and synchronization bottlenecks, but it was possible to redesign and improve the performance of the algorithms and data structures only for two of the examples. Vector Multiplication and Barnes-Hut.

6.1.1 Vector Multiplication

The performance of the *Vector Multiplication* program was improved by 169%. This high performance increase was achieved by using Chiron's *Synchronization View* to identify the extent of the problem, and by using a source line and object view correlation to find the actual source code lines which caused the performance problem. From the source code and the data in Chiron's views it was clear that access to the work partitioning variable is lock controlled and that the work allocation algorithm was inefficient. Changing the algorithm improved the performance.

The performance was further improved by resolving a second-level cache false-sharing problem. Chiron was used to project an object's address space onto the first-level and second-level caches, which allowed the user to identify some high frequency components in these views. The cause of the high frequency components was a second-level cache false-sharing problem. A further change to the work allocation algorithm solved this problem.

Figure 17 shows images taken during the analysis process.

6.1.2 Barnes-Hut

The performance of the *Barnes-Hut* application was improved by 3%. Chiron's *Performance Overview* quickly drew attention to the relatively high synchronization cost of the application. The *Synchronization View* clearly showed that improper load balancing for the first time step caused the performance problem. Since the first time step's results dominated the

short simulation. we promptly eliminated the synchronization bottleneck as an optimization target.

By investigating the object view, first-level and second-level cache views we identified true-sharing coherence misses on the *first_page* objects, and second-level cache false-sharing problems on the *precinct* objects. The performance problems were solved by modifying the *first_page* data structure and by padding and aligning the *precinct* objects to fit into a second-level cache block.

From the time-based views, *Synchronization View* and *first-level* and *second-level Cache Views*, we were able to clearly identify the different time steps and execution stages of the application. This helped in the understanding of the application's algorithms and synchronization behavior.

Figure 19 shows images taken during the analysis process.

6.1.3 WATER

WATER was the only application for which Chiron failed to improve the performance. Visual examination of the memory and synchronization performance showed that the most expensive objects are those containing the lock structures used in process synchronization. To improve the performance we would have had to design a better synchronization algorithm, but due to the overall low synchronization cost we decided not to proceed with this.

However the quick identification of the load balancing problem on the semaphore in the *Synchronization View* is a good example of Chiron's power. The true-sharing behavior on the *start_delay* object became apparent after its address space was mapped onto the *first-level* and *second-level cache views* and the block movement behavior was inspected.

Figure 22 shows images taken during the analysis process.

6.2 Conclusions on the Chiron visualization system

The aim of the Chiron system was to enable the user to identify specific behavioral patterns in images produced from a program's synchronization and memory system access data. It was postulated that the Chiron user would be able to identify *visual signatures* from the images and map their behavior directly to a specific performance problem.

In this thesis it has been have shown that a Chiron user can identify patterns in a shared-memory parallel program visualization and map these behavioral patterns (*visual signatures*) to specific performance problems. The analysis of the three example programs demonstrates the power of the visualization techniques used and the ease with which performance problems were identified.

The Chiron system is not without problems: visualizing a large data set can take a significant amount of time. To recrease rendering time detail suppression techniques were used to simplify an image's data representation. The user has control over three levels of detail which can be used to quickly move through a large data set until the user has found an area he/she would like to investigate in more detail.

Chiron should also be interfaced to more runtime data collecting and tracing systems, in addition to the simulation interface (*Mint* with *ParaDiGM*) that is currently supported.

6.3 Future work

To transform Chiron from a prototype performance visualization system into a truly useful tool, we need to interface Chiron with more tracing systems and runtime data collection systems. A wider range of data sources would help in identifying performance problems and make sure that a specific performance problem is not due to a data collection tool.

Furthermore a detailed study of a whole range of algorithms should be done to identify a large set of performance problems and their corresponding visual signatures. We would like to build up a library of visual signatures and descriptions of what causes these signatures, so that analysts could use the library as a reference guide when analyzing a program.

Currently only a few people have used Chiron and have supplied us with feedback about its usefulness and ease-of-use. We should like to increase the number of Chiron users, and correlate all user responses, so that we can evaluate the usefulness of each view. We also aim to find better or more elegant ways of showing performance metrics so that performance bottlenecks can be identified faster. A further abjective is to increase the number of data correlations views, i.e. to correlate synchronization data to cache behavior or source line information.

Another area in which we should like to work is automatic detail suppression. It is necessary to use detail suppression techniques for the user interaction with the displayed graphics to be fast. It is important that the user can move around the data set at speed and not have to wait for the workstation to draw the graphics.

Appendix A

ParaDiGM configuration description

Figure 23 shows an example memory system configuration file. Figure 24 shows a graphical representation of the memory system described in figure 23.

```
0 1 0 256 2 4 4
256 4 128
8 4 6
256 4 32
5 1 1 1
12 18 4096
16 1
32 1
```

Figure 23: An example ParaDiGM configuration file.

The first two numbers on the first line are flags indicating to the memory system simulator that it should record internal behavior. The next number flags the use of a graphical view of the caches during program execution. These flags were used for debugging purposes. The next number indicates the size of the simulation timer event queue size. The next number indicates the number of second level caches the machine is to use. The next numbers indicate the number of first level caches that are connected to each second level cache.

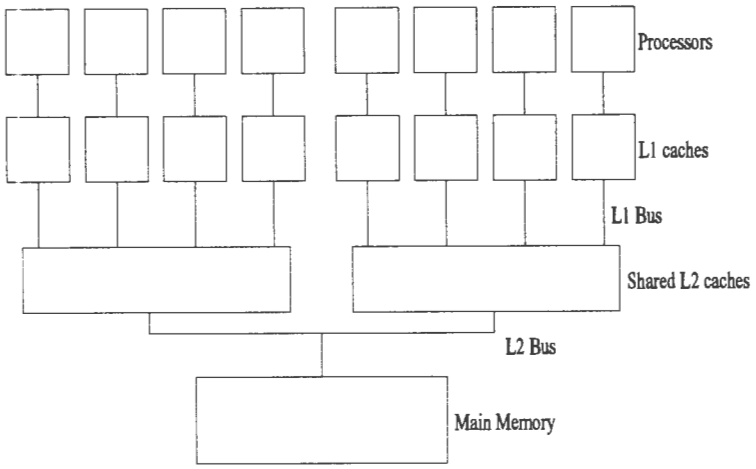


Figure 24: Modeled memory system architecture layout

The next line describes the second level caches. The three numbers indicate the number of cache lines, the cache associativity, and the block size of each cache line.

The next line describes the access times of the second level cache components. The number of clock cycles it takes to access a cache line datum, the tag field, and how long bus arbitration takes are set by the three numbers.

The next line describes the size of the first level caches. The three numbers indicate the number of cache lines, the cache associativity, and the block size of each cache line.

The next line describes the first level cache access times. The number of clock cycles it takes to identify a cache miss, to issue an interrupt, to identify a cache hit, and to replace a cache line, are indicated by these numbers.

The next line describes the main memory system behavior. The access time, the cycle time, and the page size are specified.

The next two lines describe the width and arbitration times of the memory and second level cache buses.

Appendix B

Trace file formats

The three main event logging files in table 1 are: *trace.1*, *trace.1.line.data*, and *trace.1.sync.data*. The data format in each file is described in the following sections.

B.0.1 ParaDiGM event trace file format

The ParaDiGM memory system simulator records the following information in a trace file.

- **Create Object:** Each variable, structure or object that has memory allocated by a memory allocation function is identified and written to the trace file.
  The message format is: **c** *name address length pid*
  The *name* is resolved from the source code, the *address* is the structure's location in memory, *length* is the size of the structure and *pid* is the processor id that allocated the structure.

- **Delete Object:** If an allocated structure is removed from memory, a message is sent out to signal that this has happened.
  The message format is: **d** *address pid*
  The *address* and the *pid* uniquely identify which name reference has been removed.

- **Read Cacheblock:** When a read operation is initiated, the memory location of the variable being read and the originating processor number are written to the trace file.

The message format is: **r** *address pid*

The *address* is the memory location that is being read, *pid* is the processor that issued the read operation. This message is not used by Chiron, but it was useful for debugging errors in the tracing system.

- **Write Cacheblock:** When a write operation is initiated, the memory location of the variable being written to and the originating processor number are written to the trace file.

  The message format is: **w** *address pid*

  The *address* is the memory location that is being written to, *pid* is the processor that issued the write operation. This message is not used by Chiron, but it was useful for debugging errors in the tracing system.

- **CacheMiss:** Every time a read or a write operation results in a cache miss, this message is written to the trace file.

  The message format is: **m** *address pid cache_level source_line*

  Address is the memory location and *pid* is the processor which caused the cache miss. *Cache_level* identifies the first or second level cache. *Source_line* resolved the cause of the cache miss to a specific line in the source code.

- **CacheProtect:** If a processor wants to write to a copy of a memory block in its cache it first has to have write ownership of that memory block. If it does not have write ownership then it has to request ownership. This means that all other caches have to be notified of the memory block write request and that the requesting processor has to wait for all other copies of the memory block to be invalidated.

  The message format is: p *address pid cache_level source_line*

  The elements have the same meaning as in the CacheMiss message.

- **CompleteRead:** When a read operation completes, the time the processor was stalled for is written to the trace file.

  The message format is: **R** *address pid source_line time_stalled*

  The *address* is the memory location that was read. *Source_line* is the line number of the command which issued the read operation. *Pid* is the number of the processor

which issued the read operation and *time_stalled* is the time the processor was stalled for on the read operation.

- **CompleteWrite:** When a write operation completes, the time the processor was stalled for is written to the trace file.
  The message format is: **W** *address pid source_line time_stalled*
  The *address* is the memory location that was written to. *Source_line* is the line number of the command which issued the write operation. *Pid* is the number of the processor which issued the write operation and *time_stalled* is the time the processor was stalled for on the write operation.

- **CleanReplace:** Signals a cache block moving out of the cache
  but that there is no need to write it to main memory as no
  data has changed.
  The message format is: *b address pid cache_level source_line time replace_address*
  Processor (*pid*) executes line (*source_line*) which
  wants to read from memory location (*replace_address*)
  at time (*time*). The cache block at address (*address*)
  is moved out of cache (*cache_level*).

- **DirtyReplace:** Signals a cache block moving out of the cache
  and writing it to the lower cache or main memory.
  The message format is: *a address pid cache_level source_line time replace_address*
  Processor (*pid*) executes line (*source_line*) which
  wants to read from memory location (*replace_address*)
  at time (*time*). The cache block at address (*address*)
  is moved out of cache (*cache_level*) and written back to
  main memory.

- **CleanInvalidate:** Signals for a cache block to be invalidated.
  This occurs when a shared copy of a cache block is upgraded to

a private copy and no data has to be written to a higher cache or main memory.
The message format is: *f address pid cache_level source_line time*
Processor (*pid*) is executing line (*source_line*)
which wants to write to memory location (*address*). The
data in cache (*cache_level*) is invalidated at time
(*time*).

- **DirtyInvalidate:** Signals for a cache block to be written to a higher cache or main memory, due to another processor upgrading its copy of the shared cache block to a private cache block.
  The message format is: *e address pid cache_level source_line time*
  Processor (*pid*) is executing line (*source_line*)
  which wants to write to memory location (*address*). The
  data in cache (*cache_level*) is invalidated at time
  (*time*).

- **BmiSharedMsg:** Request for a missing block to be moved into the cache. This happens when a CPU tries to read data from a cache and the cache does not contain the data. If a block is moved in *shared* it means it is a read–only copy, ie. many processors can share it.
  The message format is: *g address pid cache_level source_line time*
  Address is the start of the requested memory block which
  processor (*pid*) tried to read from cache (*cache_level*)
  executing line (*source_line*). The operation completed at
  time (*time*).

- **BmiPrivateMsg:** Request for a missing block to be moved into the cache. If the block is moved in *private* it is writeable, only one processor may have a copy, and that processor

can both read from and write to the block.
The message format is: *h address pid cache_level source_line time*
Address is the start of the requested memory block which
processor (*pid*) tried to read from cache (*cache_level*)
executing line (*source_line*). The operation completed at
time (*time*).

B.0.2 Stack Frame Tracker trace file format

For an in-depth analysis of program behavior it is useful to know what function each of the processors are in at any given time. The *Stack Frame Tracker* records each entry into and exit from a function. To uniquely identify a function call every function in the target executable is given a reference number, similarly all file names that make up the target executable are numbered. A number is generated which uniquely identifies the position in an executable file of a function call's source and a function exit's destination line.

The next section shows a few lines of a source reference file. Each line contains the path to the source code which makes up part of the target executable. The position in the file corresponds to the reference number given to the file.

```
vector1.c
/usr/include/unistd.h
/usr/include/sys/types.h
/usr/include/time.h
```

The next section shows a few lines of a function reference file. Each function that is found in an executable's symbol table is listed in this file. The position in the file corresponds to the reference number given to the function.

```
__start
_sprocmonstart
```

```
Init__Fv
Multiply__Fv
slave__Fv
Print__Fv
main
readfile__Fv
closereadfile__Fv
```

The format of the log information written out by the *Stack Frame Tracker* is:
function_number time pid line_file_number

On a function entry the *function_number* field contains the reference number of the function that is being called; on a function exit it contains a *-1*. The *time* field shows the time of entry into and exit from the function. *Pid* identifies the processor that made the function call. The *line_file_number* field identifies location in the source code that made the function call or returned from a function.

Next are a few lines of a call stack file. The time and source line fields are shown in hexadecimal notation.

```
383 b1 0 47
-1 b5 0 bf0012
295 b7 0 57
-1 b9 0 690002
296 bb 0 5e
301 112 0 6a0071
-1 13c 0 6f0026
316 177 0 6a007d
```

B.0.3 Process Synchronization Tracker trace file format

The *Process Synchronization Tracker* logs all synchronization construct activity in a trace file. Synchronization events can be divided into three categories: Locks, semaphores, and barriers. Table 3 shows which messages are associated with each category.

Locks	LOCK_ATTEMPT, LOCK_ACQUIRE, LOCK_RELEASE
Semaphores	P_ATTEMPT, P_SUCCESS, V
Barriers	B_ATTEMPT, B_ACQUIRE

Table 3: Synchronization constructs and their related messages.

Lock events

A *LOCK_ATTEMPT* message is logged when an executing program uses the **ussetlock()** function call in attempting to acquire a spinlock. If the requested lock is free then it is atomically locked, and control is returned to the caller. If the lock is already locked, the caller either spins waiting for the lock or gets queued. Once a lock has been acquired, the *LOCK_ACQUIRE* message is logged. The *LOCK_RELEASE* message is logged when a program releases a spinlock via a call to the **usunsetlock()** function.

Semaphore events

A *P_ATTEMPT* message is logged when a program attempts to perform a P() operation, via the **uspsema()** function, on a semaphore. If the count of the target semaphore becomes negative the calling process will be blocked. Once the semaphore becomes available or the P() operation is successful the *P_SUCCESS* message is logged. A *V* message is logged when the program issues a **usvsema()** instruction.

Barrier events

A *B_ATTEMPT* message is logged when a program enters a barrier via the **barrier()** function call. Once the **barrier()** function call returns the *B_ACQUIRE* message is logged.

Synchronization message file format

The synchronization message format is:

sync_msg time pid address

The *sync_msg* field contains one of the synchronization message type's identifiers. The *time* field contains the time the message was issued, *Pid* indicates what processor sent the

message. The memory location of the lock, semaphore or barrier is relayed in the *address* field.

Following are a few lines of synchronization log file.

```
5 1fc322 1 40003030
6 1fc323 1 40003030
7 1fc35d 1 40003030
5 1fc3f5 1 40003030
6 1fc3f6 1 40003030
```

Appendix C

Color photos of selected images

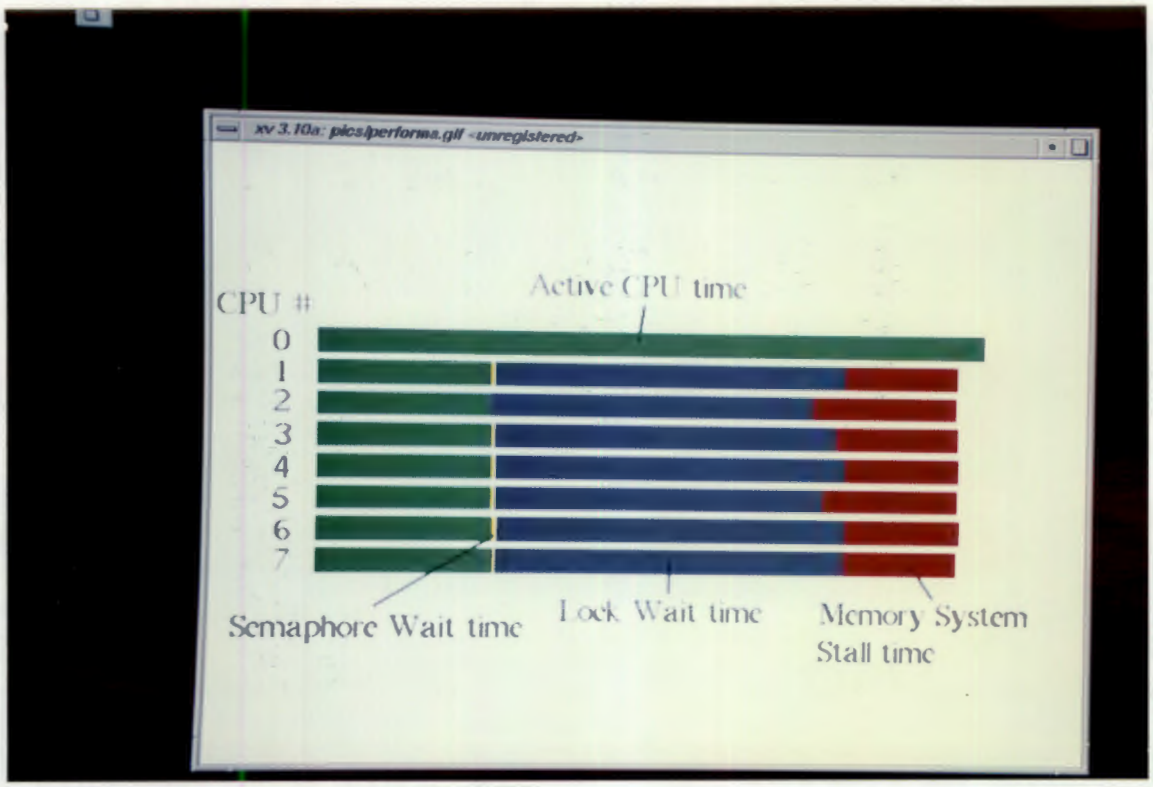


Figure 25: An example Performance Overview Graph

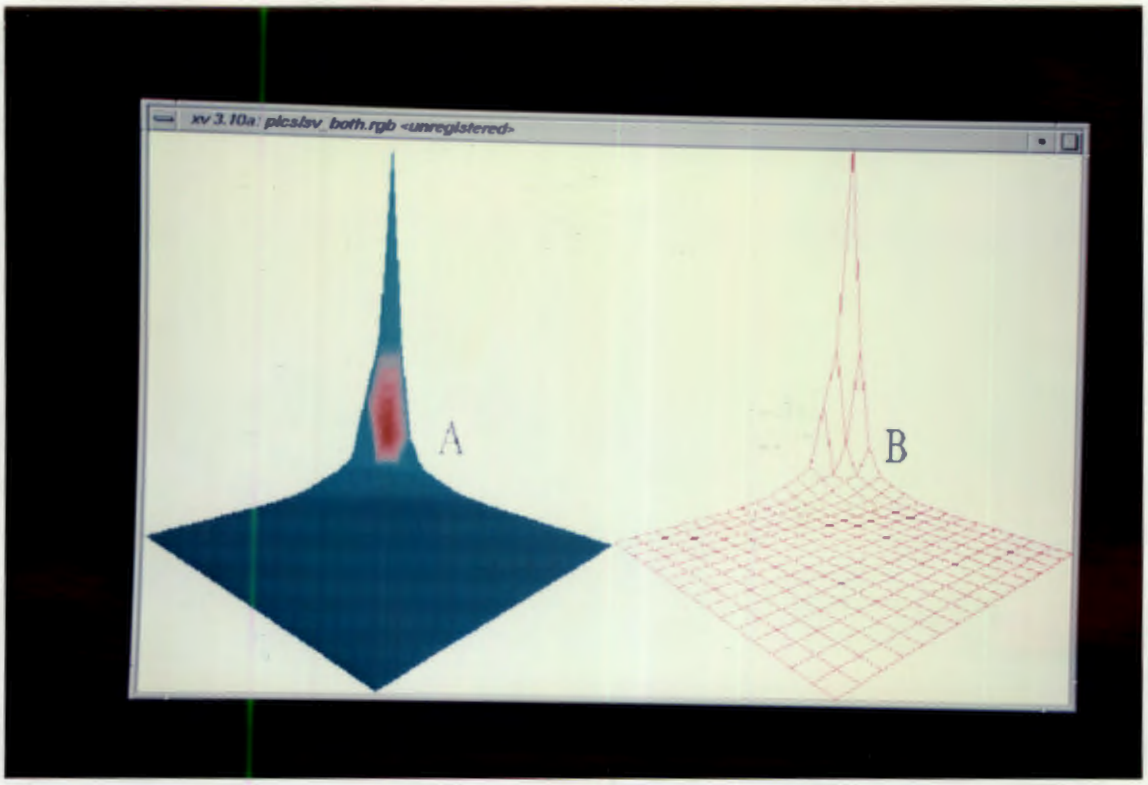


Figure 26: An example of a Source Line View

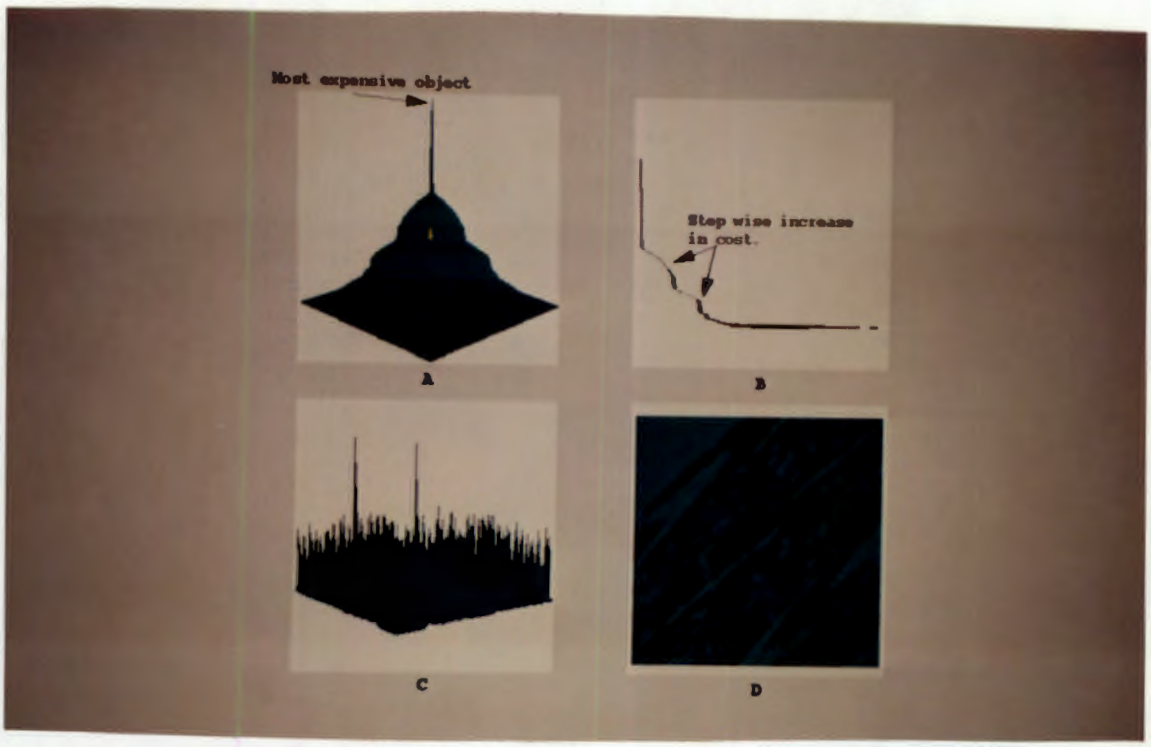


Figure 27: Example object view graphs

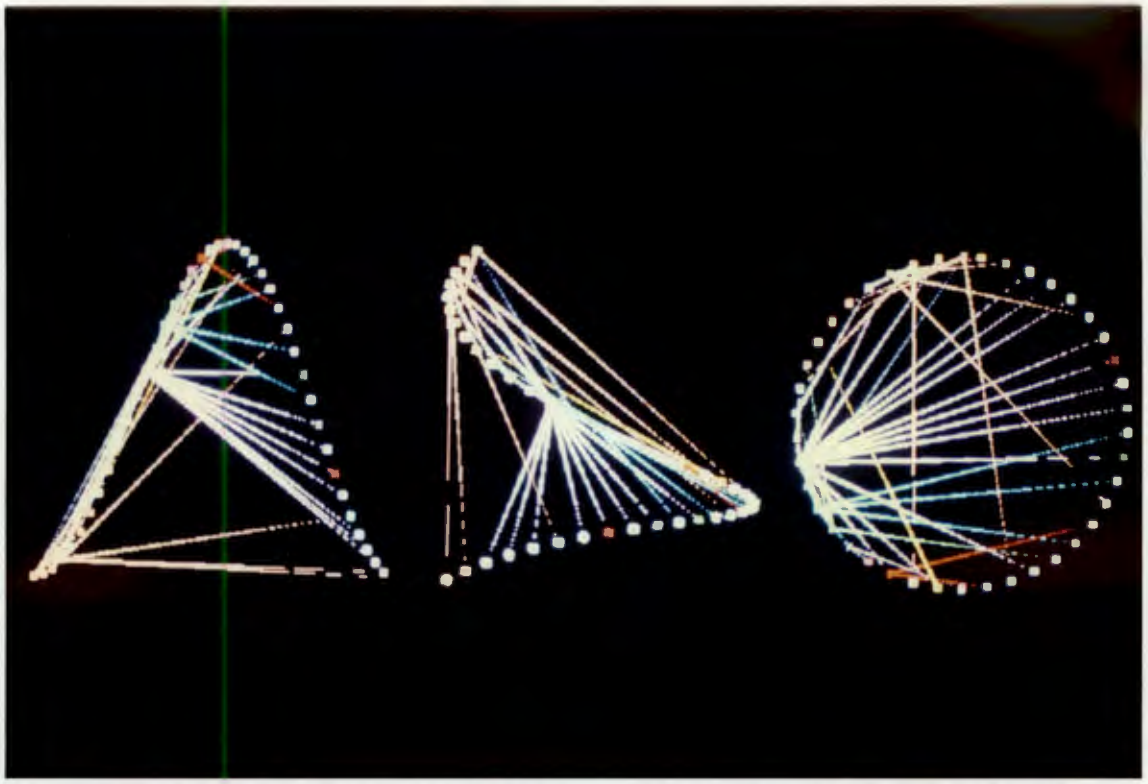


Figure 28: Three views of a spiral representing the Function Call Relation View

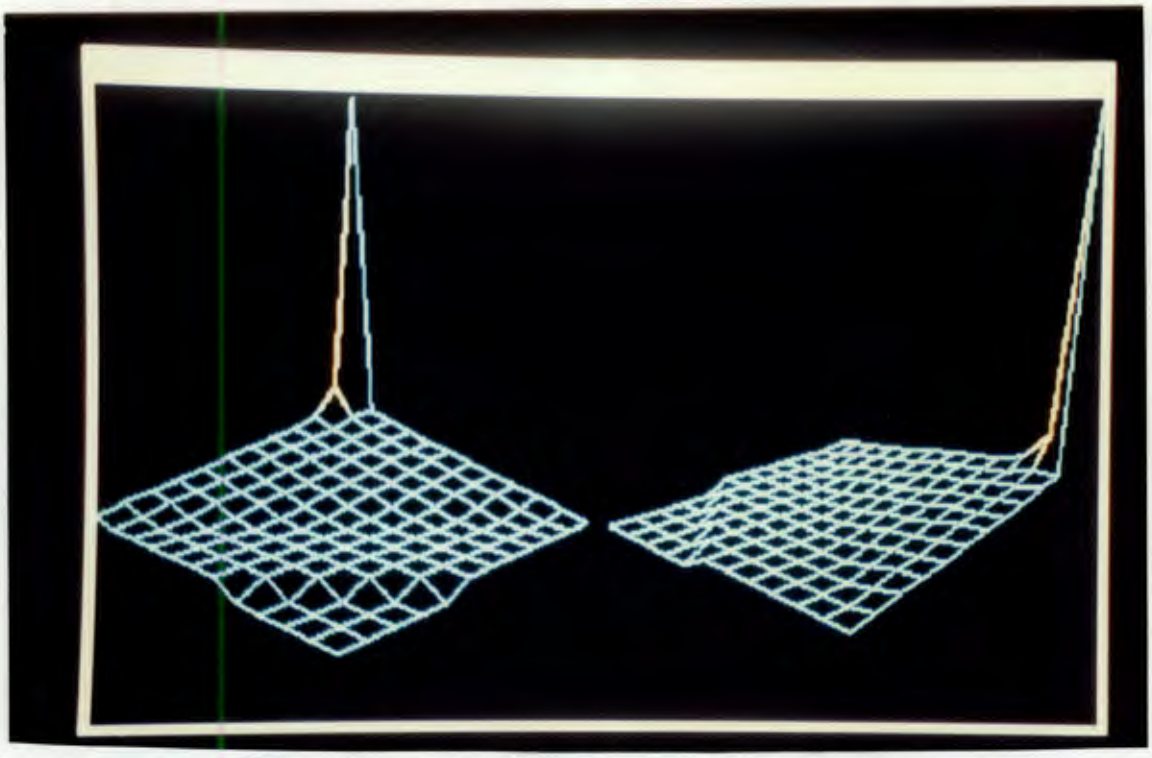


Figure 29: An example synchronization overview graph

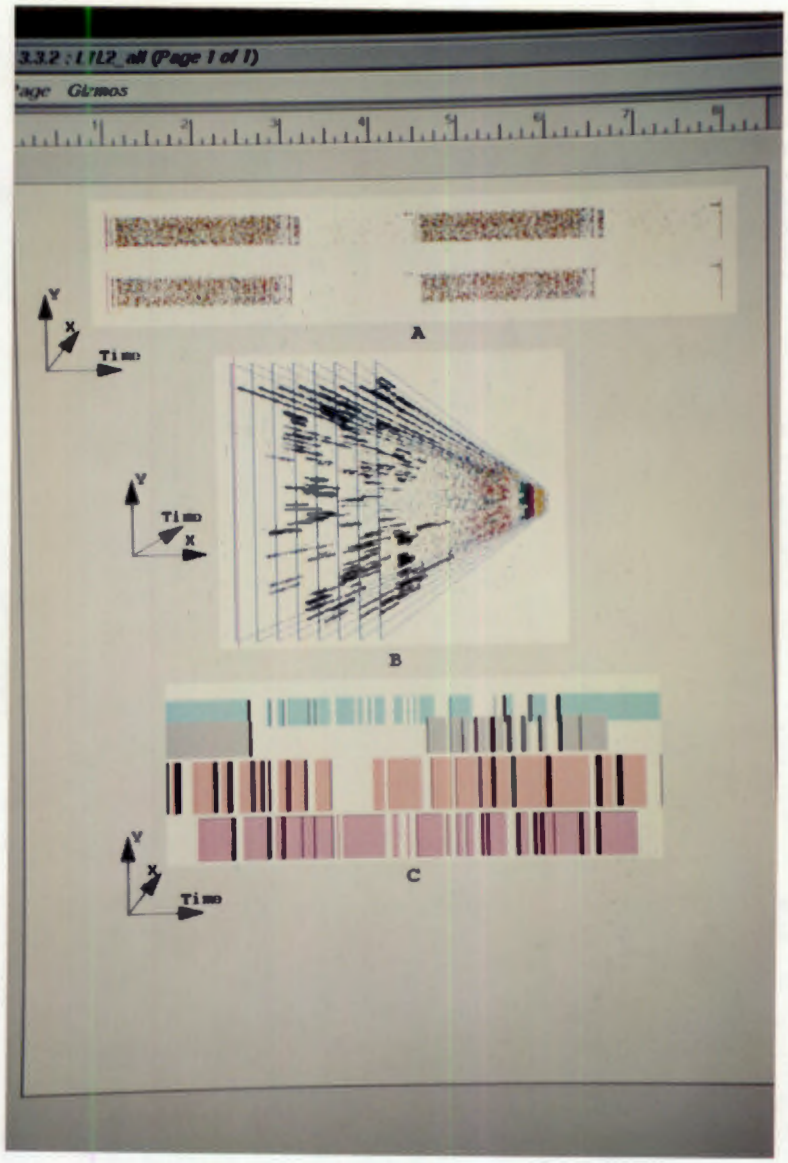


Figure 30: Example L1 and L2 cache views

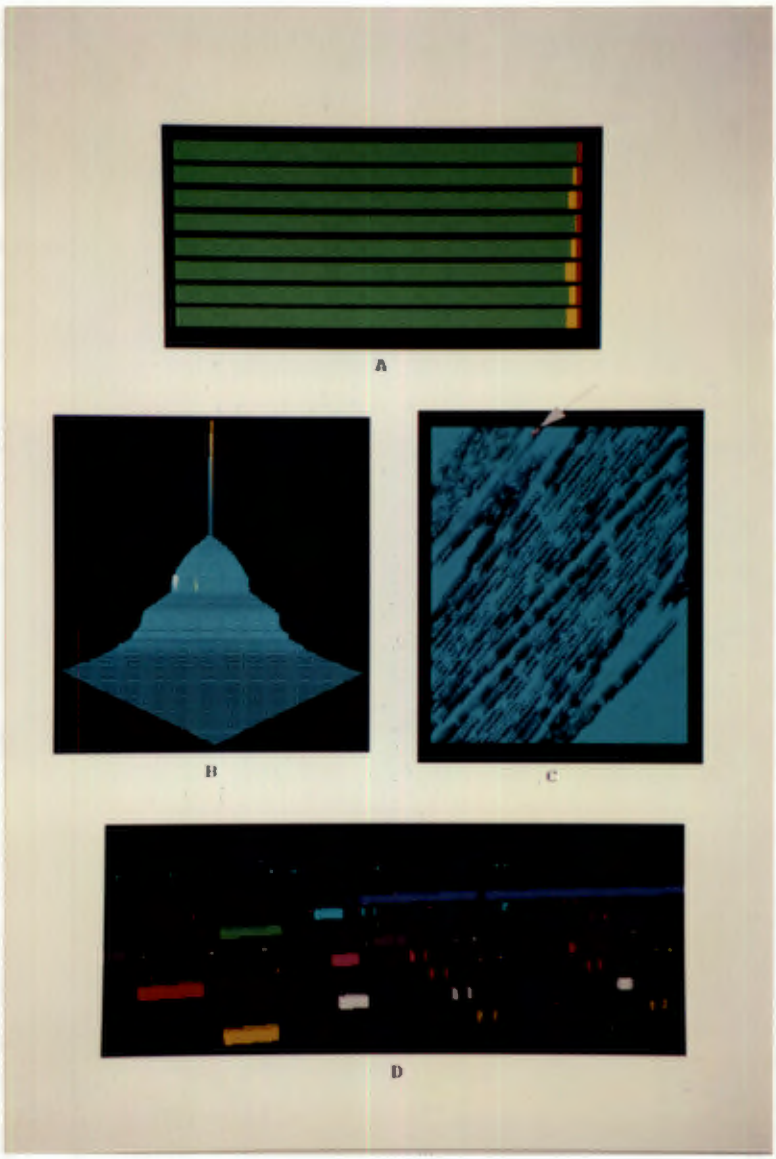


Figure 31: Array multiplication analysis process

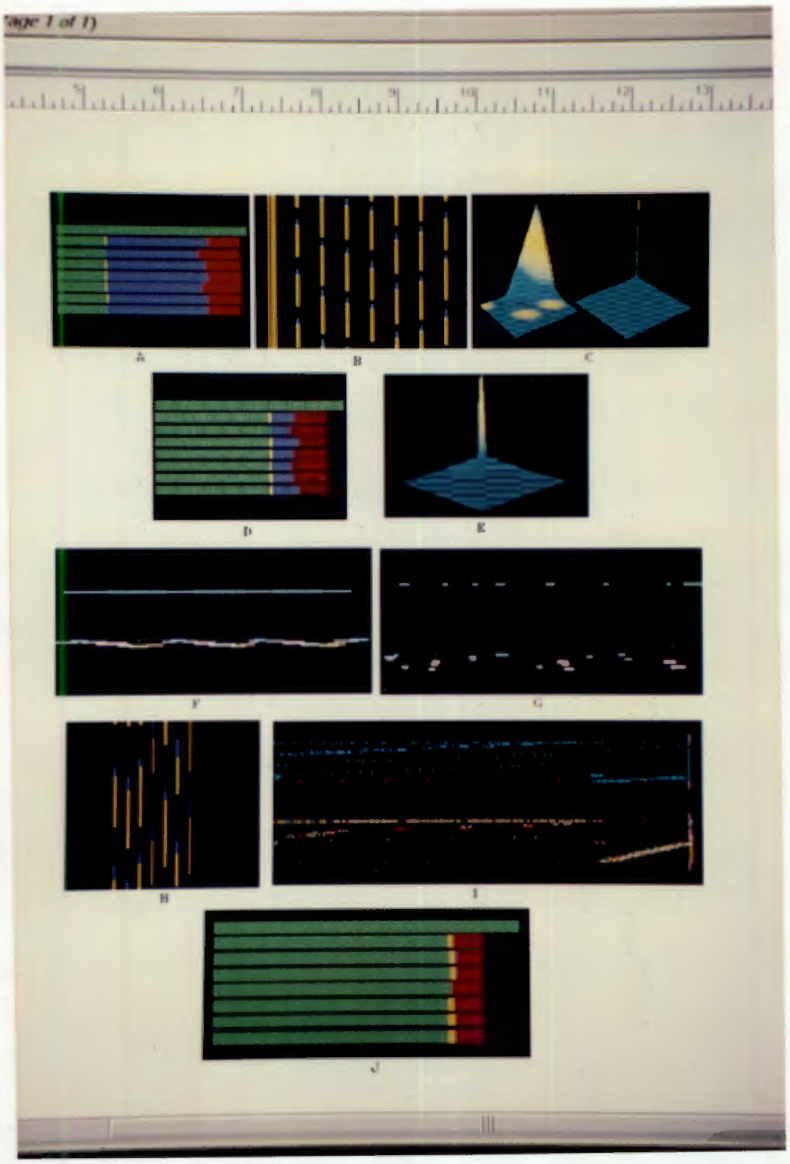


Figure 32: The visual Barnes-Hut analysis process

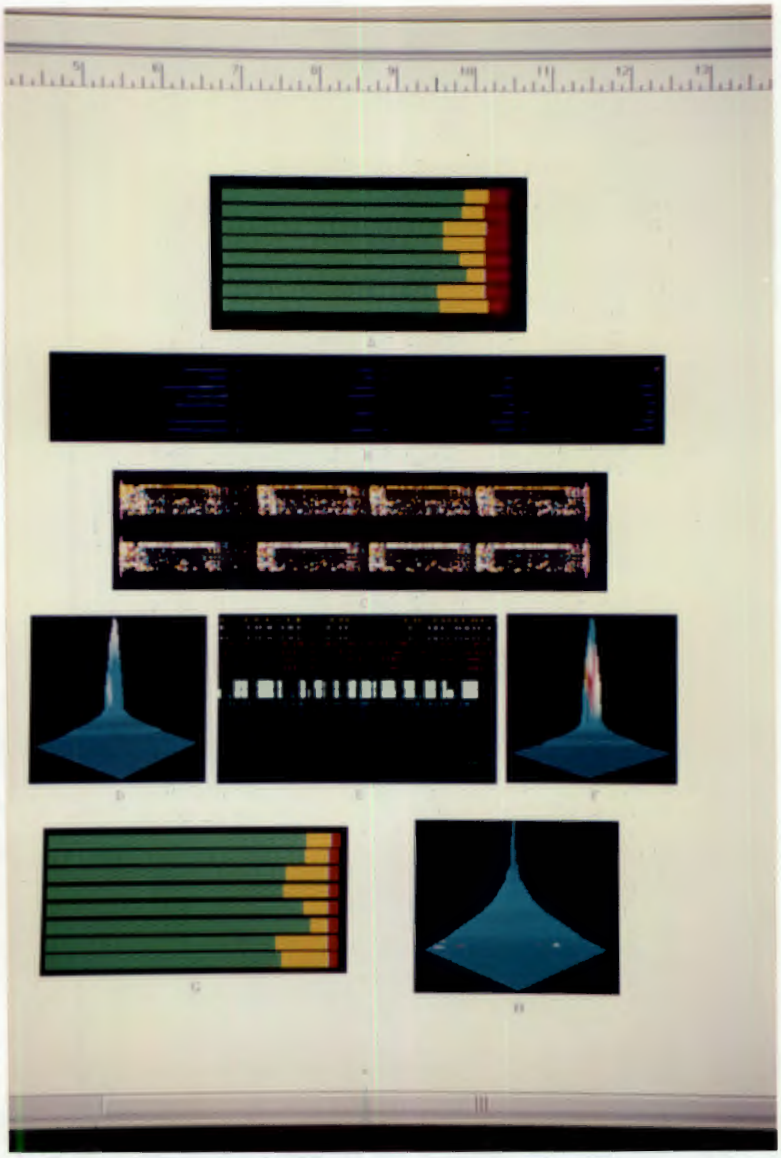


Figure 33: The WATER analysis process

Bibliography

[1] A. Agarwal and A. Gupta. Memory reference characteristics of multiprocessor applications under mach. In *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May 1988.

[2] Alan Barnum-Scrivener. How i do an avs kickstart. URL: http: //avs.ncsc.org/ HTML/ ITD/ IAC/ AVS95_www/ avs95/ netnews/ barnum/ barnum.pts1and2.html.

[3] Sander Belikn, Stefaan Poedts, Hans Goedbloed, Hans Spoelder, Ad Emmen, Jaap Hollenberg, and Rik Leenders. Comparison of visualization techniques and packages. URL: http://www.sara.nl/Consumer.Report/Report.html.

[4] Wes Bethel. Modular virtual reality visualization tools. In *Proceeding of AVS '95*, 1995.

[5] W.J. Bolosky. *Software coherence in multiprocessor memory systems.* PhD thesis, University of Rochester Computer Science Department, May 1993.

[6] Mats Brorsson. Sm-prof: A tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *1995 ACM SIGMETRICS and Performance'95*, May 1995.

[7] Mats Brorsson and Per Stenstrom. Visualisation of cache coherence bottlenecks in shared memory multiprocessor applications. *In IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 32–36, 1993.

[8] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between c and c++ programs. Technical Report CU-CS-698-94, The Department of Computer Science, University of Colorado, 1994.

[9] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2), February 1991.

[10] Alva L. Couch and David W. Krumme. Portable execution traces for parallel program debugging and performance visualization. In *Proceedings of Scalable High Performance Computing Conference 92*, pages 441–446, Williamsburg, Virginia, April 1992.

[11] Mark E. Crovella and Thomas J. LeBlanc. Performance debugging using parallel performance predicates. In *Third ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993.

[12] David W. Forslund et. al., editor. *Experiences in writing a distributed particle simulation code in C++*. USENIX C++ Conference, 1990. pages 177-190.

[13] A.J. Goldberg and J. Hennessy. Mtools: A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *International Conference on Parallel Processing*, pages 251–257, Aug 1991.

[14] A.J. Goldberg and J. Hennessy. Performance debugging shared memory multiprocessor programs with mtool. In *Supercomputing*, pages 481–490, Nov 1991.

[15] H.A. Goosen, A.R. Karlin, and D.R.Cheriton. Chiron: A system for parallel program performance visualization. In *Proceedings of the Conference on Advanced Techniques in Animation, Rendering and Visualization.* Ankara, Turkey, July 1993.

[16] Hendrik A. Goosen. *Shared Multilevel Caches for Scalable Multiprocessors.* PhD thesis, Standford University, October 1991.

[17] A. Gupta, M. Martonosi, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review*, 20(1), June 1992.

[18] S. T. Hackstadt and A. D. Malony. Next-generation parallel performance visualization: A prototyping environment for visualization development. *Lecture Notes in Computer Science*, 817:192–??, 1994.

[19] S. T. Hackstadt, A. D. Malony, and B. Mohr. Scalable performance visualization for data-parallel programs. In *Proceedings of the Conference on Scalable High-Performance Computing*, pages 342–349. IEEE Computer Society Press, May 1994.

[20] Stephen R. Goldschmidr Helen Davis and John Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, Computer Systems Laboratory, July 1990.

[21] J. Hennessy and N. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–29, September 1991.

[22] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with upshot.

[23] William L. Hibbard, Brian E. Paul, David A. Santek, Charles R. Dyer, Andre L. Battaiola, and Marie-Francoise Viodrot-Martinez. Interactive visualization of earth and space science computations. *IEEE Computer*, pages 65–72, July 1994.

[24] Jeffrey K. Hollingsworth. Finding bottlenecks in large scale parallel programs. Technical Report TR 1243, COMPUTER SCIENCES DEPARTMENT, UNIVERSITY OF WISCONSIN, MADISON, WI, September 1994.

[25] Hank Jakiela. Performance visualization of a distributed system: A case study. *Computer*, 28(11):30–36, November 1995.

[26] James R. Larus. Abstract execution: A technique for efficiently tracing programs. Technical Report 912, University of Wisconsin-Madison, Computer Science Department, May 1990.

[27] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin, Computer Science, March 1992.

[28] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L Chung, and Charles E. Fineman. Visualizing performance debugging. *Computer*, 22(10):38–51, October 1989.

[29] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing performance debugging. *Computer*, 22(10):52–61, October 1989.

[30] et al. Lusk, Overbeek. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc, 1987.

[31] M. Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs.* PhD thesis, Department of Electrical Engineering, Stanford University, January 1994.

[32] G. McDaniel. The mesa spy: An interactive tool for performance debugging. In *Performance Evaluation Review*, pages 68–76, Seattle, WA, August/September 1982. Association for Computing Machinery, Association for Computing Machinery.

[33] Barton P. Miller, Jon Cargille, R. Bruce Irvin, Tia Newhall, Mark D. Callaghan, Jeffrey K. Hollingsworth, Karen L. Karavanic, Krishna Kunchithapadam, Viresh Ratnakar, and Miron Livny. The paradyn parallel performance measurement tools RD-OPT: An efficient algorithm for optimizing DCT quantization tables. Technical Report TR 1256 TR 1257, COMPUTER SCIENCES DEPARTMENT, UNIVERSITY OF WISCONSIN COMPUTER SCIENCES DEPARTMENT, UNIVERSITY OF WISCONSIN, MADISON, WI MADISON, WI, December 1994 November 1994.

[34] L. Moore, D. A. Hensgen, D. Charley, V. Krishaswamy, D. E. Martin, T. McBrayer, and P. A. Wilsey. Graze: a tool for performance visualization and analysis. In *Proceedings of the 24th International Conference on Parallel Processing*, pages II:135–138, Oconomowoc, WI, August 1995.

[35] M.Siegle and R.Hofmann. Monitoring program behaviour on SUPRENUM. In *The 19th Annual International Symposium on COMPUTER ARCHITECTURE*, pages 332–341. Gold Coast, Australia, May 1992.

[36] Marc A. Najork and Marc H. Brown. Obliq-3d: A high-level, fast-turnaround 3d animation system. *IEEE Transactions on Visualization and Computer Graphics*, pages p175–193, June 1995.

[37] Fernando Nasser and Michael Stumm. Extending gdb for functional and performance debugging of parallel programs on shared-memory multiprocessors. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 214–216, San Diego, California, May 1993. [Extended abstract].

[38] V. Natarajan, D. Chiou, and B. S. Ang. Performance visualization on monsoon. *Journal of Parallel and Distributed Computing*, 18(2):169–180, June 1993.

[39] P.Machanick. Spacelib: A library for shared-memory parallel applications. In *The 8th National Conference for MSc and PhD Students in Computer Science*, pages 215–220. UNISA, South Africa, June 1993.

[40] R. Rajamony and A. L. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. In *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, pages 250–256, 1996.

[41] Lotufo Rasure, Jordan. Teaching image processing with khoros. In *1994 IEEE International Conference on Image Processing*, November 1994.

[42] William Ribarsky, Eric Ayers, John Eble, and Sougata Mukherjea. Glyphmaker: Creating customized visualisations of complex data. *IEEE Computer*, pages p57–64, July 1994.

[43] S. R. Sarukkai and D. Gannon. SIEVE: A performance debugging environment for parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):147–168, June 1993.

[44] SGI. *pixie - add profiling code to a program.* Release 3.10.1 manual page entry.

[45] SGI. *Prof - Analyze profile data.* Release 3.10.1 manual page entry.

[46] SGI. Iris explorer users guide. Silicon Graphics Computer Systems, 1992.

[47] Sanjay Sharma, Allen Malony, Michael Berry, and Priyamvada Sinvhal-Sharma. Runtime monitoring and performance visualization of concurrent programs for shared memory multiprocessors. Technical Report CSRD Rpt. No. 987, University of Illionis, Center for Supercomputing Research and Development, March 1990.

[48] J.P Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. Technical report, Computer Systems Laboratory, Stanford, 1993.

[49] Evan Speight and John K. Bennett. Paraview: Performance debugging of shared-memory parallel programs. Technical Report ELEC TR 9303, Rice University, March 1994.

[50] Advances Visual Systems. Avs ... the future of visual computing, 1994. URL: http://www.avs.com/products/avs.html.

[51] Unknown. The geometry toolbox. Khoral Research Inc, 1995. URL: http://www.khoros.unm.edu/khoros/khoros2/toolboxes/geometry.html.

[52] J.E. Veenstra. Mint tutorial and user manual. Technical Report 452, Computer Science Department,University of Rochester, June 1993.

[53] A. Waheed and D. T. Rover. Performance visualization of parallel programs. In G. M. Nielson and D. Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 174–182. IEEE Computer Society Press, October 1993.

[54] Jeremy Walton. Now you see it - interactive visualisation of large datasets. In *Aplli-cations of Supercomputer in Engineering III.* Computational Mechanics Publications, 1993.

[55] M. Young, D. Argiro, and S. Kubica. Cantata: Visual programming environment for the khoros system. *Computer Graphics*, 29(2):pp 22–24, May 1995.

[56] M. Young, D. Argiro, and S. Kubica. An object oriented visual programming language toolkit. *Computer Graphics*, 29(2):pp 25–28, May 1995.