VISUALIZING THE MEMORY PERFORMANCE OF PARALLEL PROGRAMS WITH CHIRON

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

Dieter Polzin

February 1996

Supervised by

H.A. Goosen

Abstract

Writing efficient shared-memory parallel applications is a difficult task because problems such as false sharing, inefficient synchronization and load imbalance can reduce the application's performance. Until parallel compilers generate more efficient code, parallel programmers need tools which can help them understand and improve the performance behaviour of their parallel programs.

This thesis describes Chiron, a visualization system which helps programmers detect memory system bottlenecks in their shared-memory parallel applications. Chiron is different from most other performance debugging tools in that it uses three-dimensional graphics techniques to display vast amounts of memory-performance data. Both code- and data-oriented information can be presented in several views. These views have been designed to help the user detect problems which cause coherence interference or replacement interference. Chiron's interactive user-interface enables the user to manipulate the views and home in on features which indicate memory system bottlenecks. The visualized data can be augmented with more detailed numerical data and correlations between the separate views can be displayed. The effectiveness of Chiron is illustrated in this thesis by means of three case studies.

Acknowledgements

I would like to thank my supervisor, Henk Goosen, for introducing me to the field of visualization and for writing the very first version of Chiron,

The Foundation for Research and Development for their financial support,

Peter Hinz for his help with some of the coding and with generating trace files,

My friends and colleagues for their help and the interesting discussions,

And Joanne Moorcroft, whose support helped me through the last few agonizing months.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multiprocessors are gaining popularity as uniprocessors are unable to meet the increasing demand for high performance computers. However, parallel programs are more difficult to develop than sequential programs [4].

Shared-memory[1] parallel computers simplify the programming task by moving data automatically between processors and by presenting one large, shared address space to the application. Several vendors (such as Encore, Sun Microsystems, Cray Research Inc. and Silicon Graphics) now offer shared-memory machines with cost-effective performance compared to traditional super-computers. For example, an 8-CPU Silicon Graphics 4D/380 achieves half the performance of a Cray-2 at 2.5% of the cost, executing a particle simulator called MP3D [22].

Writing *efficient* shared-memory parallel applications is difficult because many problems such as memory contention, inefficient synchronization and load imbalance can reduce the application's performance. Memory contention, in particular, can have a large impact on a shared-memory parallel program's performance [1, 11].

A number of techniques exist for optimizing the memory performance of shared-memory multiprocessors, such as loop partitioning [20], data pre-fetching [13] and wavefront blocking [5]. However, studies have shown that the success of parallelizing compilers is limited [2, 5]. While a particular technique may speed up certain code patterns, it may cause performance degradation for the rest of the program.

This means that parallel programmers have to perform some of the optimization themselves, but to do this they need tools which can help them relate the memory performance

[1]The term shared-memory refers to the memory model and not the actual implementation. Shared-memory computers can therefore be implemented on top of a distributed memory.

to the parallel program. Several tools exist, such as MTOOL [10], MemSpy [14] and ParaView [36], but these tools cannot show the detailed behaviour (at object or source level) of large applications.

This thesis describes Chiron [12], a visualization system that uses interactive three-dimensional (3D) graphics techniques to display large amounts of memory performance data. The user interface allows the user to display the data in several ways and allows the user to home in on interesting features which are an indication of performance bottlenecks. The user can derive information from the spatial, colour and texture information of the visualization, which can also be augmented by more detailed textual displays.

Chiron [12] was designed to help programmers detect memory system bottlenecks in shared-memory applications. This thesis describes the design of Chiron and discusses the issues involved in implementing Chiron. This thesis also shows how Chiron was used to improve the memory performance, and thus the overall performance of three applications. Two of the three applications are real-world applications which generate large amounts of performance data.

Before Chiron is described in more detail, the following section outlines the memory performance problems addressed by Chiron. An overview of the thesis is presented in the final section of this chapter.

1.1 The memory performance bottleneck

Microprocessor speeds are increasing at a higher rate than that of main memory speeds [17]. To reduce memory access times, modern computer systems use memory systems with multiple levels of cache memory.

The cache memory is smaller and faster than main memory and keeps a copy of recently used data. If a program accesses data which is in the cache, the data is retrieved from the cache and not from main memory. The resulting memory access time is lower because the cache is made of faster memory.

If the data is not in the cache, a cache miss occurs and the data has to be retrieved from the next level of cache or main memory. Typically multiple words, called a block, are brought into the cache at a time. The block is likely to be referenced a number of times before it is replaced, because programs exhibit spatial and temporal locality of reference. For a more detailed discussion of caches see [35].

As the difference between memory speeds and processor speeds increases, so the impact of memory performance upon the program's overall performance increases as cache misses

become more and more expensive in terms of processor time. Programs will have to exhibit a memory reference pattern that exploits caches well, before they can attain good performance.

There are four possible causes of cache misses in a multiprocessor system [16]:

- compulsory misses
- capacity misses
- collision misses
- coherence misses

1.1.1 Compulsory misses

Compulsory misses happen when an object is brought into the cache because the program has referenced it for the first time. When this happens, the whole block containing the object is brought into the cache, which means that all the other objects in the block are pre-fetched. However, if the block leaves the cache before any of these objects are referenced the pre-fetch will not be effective.

By changing the layout of the data in memory so that data which is referenced in a similar time period by the same processor is located in the same block, the number of compulsory misses can be reduced.

1.1.2 Capacity misses

Capacity misses occur because the size of a cache is not large enough to accommodate the working set of the program. Seeing that the cache has a fixed size, cache blocks will have to be evicted from the cache to make space for new blocks.

1.1.3 Collision misses

Caches are often organized on a set-associative basis, so that lookups in the cache can be performed faster. This means that a cache block will be mapped onto a limited number of positions in the cache. Even if the cache is large enough to accommodate the working set, capacity misses will still occur if objects are not mapped evenly onto the different sets. These misses are referred to as collision misses.

If there is an imbalance in the number of collision misses between sets, it is possible to reduce these by changing the location of objects in memory. This can be achieved using blocking algorithms [21].

Chiron does not distinguish between capacity misses and collision misses, and instead refers to them as replacement misses.

1.1.4 Coherence misses

While forced misses and capacity misses apply to uniprocessors and multiprocessors, coherence misses apply only to multiprocessors. These misses occur because the memory system has to ensure that the data in the caches of different processors remains consistent. Many coherence protocols exist but the case studies for this thesis were performed on a memory system simulator with cache invalidate protocol. It is, however, relatively simple to change the simulator to support other protocols.

With the cache invalidate protocol a processor has to have exclusive access to a block before it can perform a write operation to that block. Before a processor can get exclusive access to a block, all other copies of the block have to be invalidated. If different processors access the same object, the cache block of that object has to be transferred between the different processor caches. This is known as true sharing.

If different processors access different objects in the same cache block, this cache block also has to be transferred between the different processor caches and false sharing occurs. An experimental study [3] has shown that this can have a large impact on a program's performance.

False sharing can be reduced by changing the layout of the data so that objects which are accessed by different processors fall into different cache blocks.

Chiron uses visual patterns to help the user differentiate between memory bottlenecks caused by true sharing and memory bottlenecks caused by false sharing.

1.2 Thesis outline

The next chapter examines existing performance debugging tools and compares these to Chiron. The majority of these tools use a combination of animation, two-dimensional (2D) graphs and summary statistics to present the performance data to the user. Chiron, however, tries to present the performance data more effectively using 3D graphics techniques employed in scientific visualization. This chapter also mentions some of the available visualization tools and explains why Chiron uses the SGI Inventor library.

Chapter 3 explains how Chiron's 3D views display various aspects of a parallel program's memory performance. These views can be placed into three categories: the global views which display a summary of the memory reference cost for all the data objects and source lines in a program; the temporal views which display the state of the memory system as time progresses; and the correlation views which display relationships between subsets of the data.

Chapter 4 describes some of the implementation issues and explains how manual level-of-detail suppression can improve the graphics performance of Chiron, especially when large amounts of data are displayed.

Chapter 5 describes various methods of monitoring the execution of a parallel program and describes how the trace files were generated for the case studies of the next chapter. The memory reference behaviour of these programs was simulated for the Paradigm [6] shared-memory architecture, which is also described in this chapter.

Chapter 6 describes how Chiron was used to improve the performance of three parallel applications. The process of performance debugging with Chiron is illustrated with a small program which performs vector addition. The effectiveness of debugging real-world applications with Chiron is shown with two applications called MP3D and Barnes-Hut. While the overall performance of Barnes-Hut was not improved significantly, the performance of MP3D was improved by 30%.

In the final chapter conclusions are drawn and ideas for future work are mentioned.

Chapter 2

Related Work

In the past few years a number of parallel program debugging tools have been designed. Most of these tools use a combination of two-dimensional (2D) graphs, animation, and summary statistics to display the program's behaviour.

2.1 Algorithm animation

In algorithm animation the program data and program operations are abstracted to create animated graphical views which will help the programmer understand the functional behaviour of the program. Tango [37] is a tool which uses algorithm animation.

Using Tango is a three step process. First the programmer must annotate the program with algorithm operations. Then the programmer must design the animation scenes, after which the mapping from the algorithm operation to the animation action can be specified. Only then can the programmer execute the parallel program and see how it behaves by looking at the animations.

This process involves a large amount of work for the programmer, although recent animation systems such as Obliq-3D [24] do simplify the task of constructing complex 3D animations.

While tools such as Tango are useful for examining the functional behaviour of small programs, it is not clear how effectively they can be used for large parallel applications. Algorithm animation is also not suitable for performance debugging.

2.2 Performance debuggers for message-passing systems

Performance debuggers for message-passing multiprocessors focus on processor utilization and the interprocessor communication overhead.

One attempt at presenting computation and interprocessor communication graphically, was achieved by mapping the nodes in a hypercube onto a two-dimensional grid using a gray code mapping function [30]. A metric of interest can then be mapped onto the resulting surface by setting the colour of each node. Animation is used to show the performance of the system over time. This approach has the advantage that hundreds or thousands of processors can be displayed at once and that the flow of granules of computation and communication throughout the system, over both space and time, can be shown. While this system helps detect performance bottlenecks, it does not show where in the program these bottlenecks occur.

The Prism [34] programming environment is a graphical environment for developing parallel programs for the Connection Machine. The user can debug parallel programs in a graphical setting and the data of multidimensional arrays can be visualized in a number of different ways. One visualizer builds a 3D surface from a series of data slices so that the user can view the behaviour of the underlying variable or expression. In addition, the user can query the value of any element with a point-and-click operation. Chiron supports similar point-and-click operations in all its views so that the user can supplement the visual data with more detailed textual data.

IPS-2 [23] uses multiple two-dimensional (2D) views to show the performance of shared-memory and distributed-memory parallel programs at different levels of abstraction. At the lowest level the user can see a detailed breakdown of processor activities while at the higher levels the user can see the procedure call tree and the critical path at a process level and a procedure level. Chiron also uses multiple views to display the performance data at different levels of detail but unlike Chiron, IPS-2 does not display the memory performance of programs.

ParaGraph [15] is another tool which uses two-dimensional graphs and animation to help the performance debugger analyze the performance and behaviour of parallel programs. ParaGraph uses more than twenty graphical displays to give the user as many visual perspectives of the processor utilization and communication overhead as possible. However, ParaGraph has been designed for message-passing processors and not for shared-memory multiprocessors.

2.3 Performance debuggers for shared-memory systems

Performance debuggers for shared-memory multiprocessors focus on memory performance in addition to processor utilization.

ParaView [36] is a performance debugging tool for shared-memory systems. It is very similar to ParaGraph in that it uses 2D graphs, animation and perspective views of 3D graphs to display the performance data. ParaView is an effective tool for detecting bottlenecks caused by poor cache performance, inefficient synchronization and load imbalance in small shared-memory applications but the use of 2D graphics as opposed to 3D graphics limits the amount of fine-grained data that can be displayed for larger applications. The information displayed by ParaView is of a data-oriented nature, while Chiron displays code-oriented and data-oriented information.

MAP [7] uses 2D graphics to display the memory access patterns of Schedule [7] parallel programs. Matrix elements are mapped onto a flat surface and the nodes are given different colours for a cache miss and a cache hit. Animation is used to show the behaviour of the system over time. Chiron displays a similar view for data objects in memory, but the view is static and shows the cost (in terms of lost CPU cycles) of data objects over the whole execution of the program. The third dimension (height) is used to display the cost of the data objects, which can be completely arbitrary objects instead of matrices.

MTOOL [10] also helps the user analyze performance losses in shared-memory programs. This system uses 2D histograms and summary statistics to display the cost of the program in terms of synchronization overhead, parallel overhead and memory system overhead. The code-oriented information can be displayed at different levels, from the program level right down to individual source lines. While Chiron also displays code-oriented memory performance information at a source line level, it displays data-oriented information at a data object level. Chiron also supplies more detailed information on the cause of the memory system behaviour (such as coherence interference) than MTOOL.

MemSpy [14] is similar to Chiron because it has been designed to provide information such as cache miss rates and causes of cache misses for both source level code objects (such as procedures) and data objects. MemSpy makes use of a matrix presentation to allow the programmer to view both code and data oriented statistics at the same time. Chiron provides similar information to MemSpy but in a graphical and interactive environment.

2.4 Scientific visualization

While the above tools have been shown to be effective, Chiron tries to improve on them by showing more performance data more effectively using 3D graphs and interaction. These techniques have been employed in scientific visualization and have already contributed to other sciences such as medical imaging, meteorology and biochemistry [9]. However, the data being examined in these sciences is inherently three-dimensional and maps more easily onto objects which can be visualized than the abstract data generated during performance evaluation.

Some research has been conducted in the area of abstract data visualization and some methods for presenting abstract data can be found in [38, 39]. However, these methods rely on 2D graphics and colour.

A survey of parallel debuggers has shown that the use of graphical techniques can improve the useability of a debugger but that it is necessary to develop more effective ways to portray the complexity of parallel programs [25]. This study also mentions that a problem with animation is that there is no explicit record of events, so the user has to keep a mental track of what has happened during several frames to recognize patterns. Instead of using animation, Chiron uses the third dimension in many of its 3D views to show the evolution of the memory system over time. By zooming in or out of the 3D view the user can determine the length of time over which the performance of the system is observed.

One system which does use interactive 3D graphics is N-vision [8], which uses coordinate systems within coordinate systems — also known as worlds within worlds — to display multivariate data. Program visualization systems such as Pavane [29] and Plum [27], on the other hand, use 3D graphics to display the behaviour of programs.

2.5 Visualization tools

A number of visualization tools exist which help scientists display their data. These tools include AVS (Application Visualization System), Iris Explorer, Vis5D [18] and Glyphmaker [28]. Our first attempt at displaying folded graphs used the Explorer visualization package but because Explorer did not allow extensive interaction with the underlying data, Chiron was written with the more flexible SGI Inventor 3D toolkit, which does provide support for interaction.

Glyphmaker is similar to Chiron because it allows the user to view correlations in the data (also known as correlative linking) by emphasizing features in multiple views. Chiron's lens is also very similar to Glyphmaker's conditional box which enables the user to focus on

certain areas of the data. However, because Glyphmaker uses Iris Explorer the interaction with the data is limited.

Chapter 3

The Design of Chiron

The aim of performance debugging tools such as Chiron is to help the programmer find and eliminate performance bottlenecks to speed up the program. One approach to finding bottlenecks is to use statistical methods to summarize performance information. Chiron's approach is to present the programmer with a visual representation of detailed, fine-grained performance data.

This data is displayed in several views which are placed in 3D space. Each view is a 3D representation of the memory performance data, which can be manipulated independently. For example, the user can rotate, scale or move each view. These views can be switched on or off independently so that the user can control the amount of information being displayed. Several views have further controls which allow the user to control the amount of detail in that particular view.

By supplying the user with as much information as possible, Chiron gives the user a general overview of a program's memory performance. From this overview, the user can home in on possible memory bottlenecks by zooming into particular features on any of the views.

Chiron makes extensive use of interactive 3D graphics to allow the user to view and manipulate the data. It is difficult to convey the interaction in photographs and diagrams so the interaction is described in a fair amount of detail in the text.

The views of Chiron fall into three categories:

- **Global Views:** These views present a summary of the memory reference cost of all data objects or source lines in the parallel program. Even though the cost for the objects is averaged over the entire run-time of the program, the cost is shown at a fine granularity. Thus the cost can be shown for all C/C++ objects, memory words or source lines in the program.

- **Temporal Views:** While the global views can often show where in a program bottlenecks are occurring, they do not show the cause of these bottlenecks. Here the temporal views can help the programmer determine the cause because they show the state of the system at any instant in time.
- **Correlation Views:** These views allow the user to map an additional metric onto the above views or allow the user to relate information from one view to another.

3.1 Global views

The information displayed by Chiron's global views is essentially two-dimensional (2D) in nature (object versus cost) but displaying this data in a 2D graph has a number of problems:

- The aspect ratio of a 2D graph does not match the aspect ratio of the screen. This means that fine detail is lost if a graph with a large amount of information is viewed in its entirety.
- It is difficult to select individual nodes in the graph if the entire graph is viewed because the detail is lost.
- If a more detailed view of a large graph is required, only a section of the graph can be displayed and the overview is lost.
- It is difficult to show correlations in the information by painting nodes in a 2D graph.

As a solution to these problems, Chiron maps the data onto a square 3D surface, called a folded graph [12].

3.1.1 The folded graph

The folded graph is created by first sorting the data according to some criteria and then mapping it along consecutive diagonals of a square grid which has a side length of $\sqrt{N}$, where N is the number of data objects.

The result of this mapping is shown in Figure 1. The diagonals onto which the objects are mapped are shown as solid lines, while the sequence of the diagonals is indicated by a dotted line. The information at each grid node (normally the cost of that node) is then used to assign a height to that node, thus creating a 3D surface. A folded graph is thus a standard surface plot with the nodes arranged in a certain order.

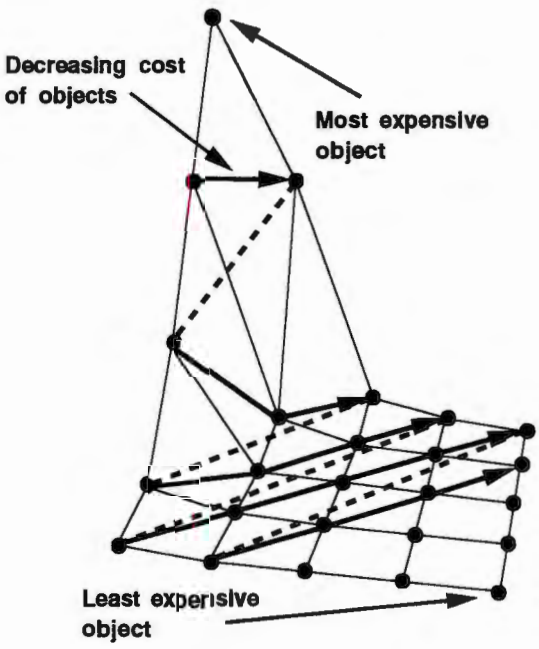

Figure 1: Creating the folded graph.

The nodes in a folded graph are closer together than in a 2D graph, resulting in a more compact graph. The aspect ratio of a folded graph is similar to that of the screen so screen space is better utilized. This means that the user can see more objects at once without having to scale down the graph until individual nodes are indistinguishable from each other. Selecting individual nodes is therefore easier on a folded graph.

The surface of the folded graph can also be painted with different colours. Chiron implements a number of correlation views by mapping colours onto specific nodes.

3.1.2 Normalizing the folded graph

The height of the folded graph is normalized with respect to the side length of the surface to keep the aspect ratio of the graph fixed. The height of the other nodes is thus proportional to the height of the most expensive node.

If there is a big difference in cost between the most expensive node and the other nodes, the folded graph will drop off very sharply from the most expensive node, as shown in Figure 2(a). If the cost is distributed more evenly amongst the nodes the graph will drop off more slowly, as shown in Figure 2(b).

If the difference in height between the nodes is found to be insufficient to represent the difference in cost between the objects, the user can increase the height of the graph. The height of the other nodes will be increased proportionally and the user will be able to detect differences in cost more easily.

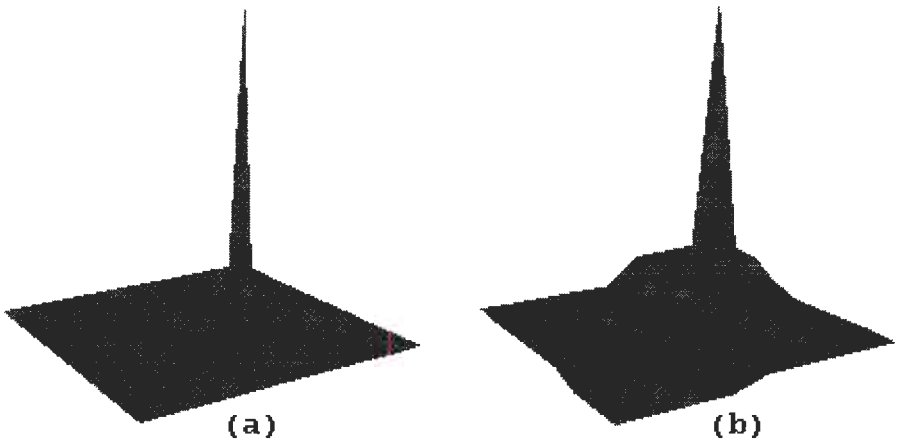

Figure 2: Comparison of normalized folded graphs with different cost.

3.1.3 The SourceView

One of the performance analyst's first questions is: "Which lines in my code are causing the performance bottlenecks?" To help the analyst answer this question, Chiron displays a global view, called the SourceView, which shows the memory reference cost incurred by each line of source code over the entire execution of the program. Showing the cost at a source line level instead of a procedure level helps the user determine the source of the bottleneck more accurately.

The SourceView is implemented as a folded graph, where each line of code is represented by a node. The nodes are sorted on cost before they are mapped onto the folded graph, with the height of each node representing the cost of the source line.

In this case the cost for a particular node is the total amount of CPU cycles, over the entire execution of the program, that processors spent waiting for cache blocks to be moved in on the particular source line. This metric is more accurate than the total number of cache misses that were caused by a line of code because it takes the different memory latencies of a miss into account. Seeing that one source line can perform a large number of memory references, the difference in memory latencies can have an effect on the source line's overall performance impact.

3.1.4 Selecting a source node

The user can select each node in the SourceView by clicking on it with the left mouse button. The selected node is then highlighted in red and textual information about it is printed in the SourceView's text window (see Figure 27 in Appendix B). At the top of this window, a copy of the source line and its cost in terms of CPU cycles are printed. In

addition, the file name and the line number of the source line are displayed. Below that the block of code surrounding the selected line is displayed.

3.1.5 The ObjectView

While some performance bottlenecks can be found through code-oriented statistics, other bottlenecks are easier to find through data-oriented statistics. For example, consider a program which accesses one data object in a number of source lines. This data object could be responsible for many cache misses but the cost might be distributed across all the lines of code which access it so that none of the lines can be identified as a bottleneck.

Chiron displays a global view of the cost of all the C/C++ data objects in the application. Once again, the view is implemented as a folded graph, which is created by sorting the C/C++ objects according to decreasing cost and then mapping them onto the square surface. Each node on the surface thus represents one object and the height is proportional to the cost in CPU cycles which the program spent waiting to access the object.

3.1.6 Sorting objects on memory address

Alternatively, the objects can be sorted according to decreasing memory address before they are mapped onto the surface. The user can then relate the cost of an object with its location in memory. This in turn, can give the performance analyst important clues to the cause of a performance problem. For example, if expensive objects are close together, as in Figure 3(a), it is possible that these objects occupy the same cache block and that false sharing between them is causing the bottleneck.

The roughness of the surface in this view also gives an indication of the difference in cost between the various objects. In Figure 3(a) the difference in cost between the most expensive objects and the least expensive objects is large. The surface is very smooth because the difference in cost between most of the objects is much smaller than the difference in cost between the most expensive and the least expensive object.

Figure 3(b) shows an ObjectView of the same objects after the program has been optimized. Now the surface is much rougher because the difference in cost between the most expensive object and the least expensive object is much smaller than in the previous example. Small differences in cost between objects will be more visible because the height differences are greater.

The sorting method can be changed at run-time so that the user can toggle between the ObjectView which is sorted according to cost or sorted according to memory address.

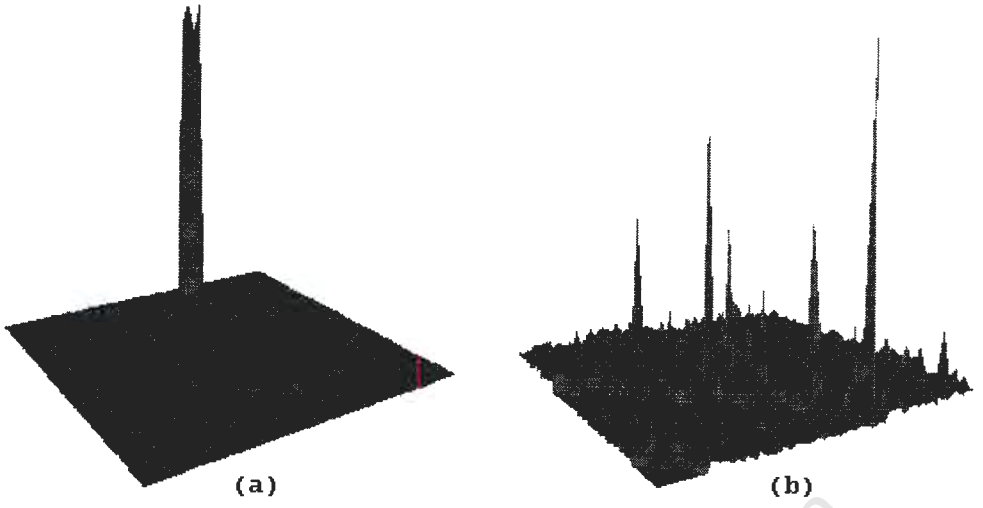

Figure 3: Comparison of ObjectViews with different cost, sorted on address.

3.1.7 Selecting an object node

As in the SourceView, the user can select any node in the ObjectView with the mouse. The selected node is then highlighted in yellow and the details about the node are printed in Chiron's object information window (see Figure 27 in Appendix B). The highlight colour is different to that of the SourceView so that it is easy to distinguish between the two. In addition, the different colours are useful in the correlation views, as will be described later on.

The details which are displayed for the selected node include the following information:

- The rank of the node in the folded graph
- The memory address of the node
- The number of cycles lost by accesses to the object
- The data object's name

Displaying the rank of the the object in the folded graph is especially useful when the folded graph contains a large number of objects so that it is difficult to determine the rank visually.

The address of the object can help the user find references to the object in the other views while the number of cycles tells the user exactly how expensive the object is. The name of the selected object is only given if the location of data objects is captured in the trace information. This is described in more detail in the following chapter.

3.2 Temporal views

In a shared-memory multiprocessor the memory performance plays an important role in the program's overall performance. While the global views help the user detect which objects or source lines are responsible for most of the memory overhead, the user has to be able to relate this cost to the underlying memory behaviour of the system. This helps the user determine whether the memory performance is indeed a bottleneck and what course of action needs to be taken to improve the performance.

Chiron's temporal views show the detailed memory behaviour of a program by visualizing the movement of cache blocks as time progresses, for individual processors and individual cache blocks. Each cache level is displayed in a separate view which can be independently manipulated.

To see the memory behaviour of the entire program the user has to zoom out of the view using Chiron's interactive controls. From the overall view the user can visually identify hotspots of activity and can then zoom into the region of activity to study the behaviour in more detail. The user can also interactively display either the cache occupancy view [12] or the cache temperature view [12] of the memory system. These views will also be referred to as BlockViews.

3.2.1 The cache occupancy view

The cache occupancy view shows the time periods during which a particular cache block resides in the cache of a CPU. This view is created by drawing an elongated cube at location x_1, y_1, z_1 if processor x_1 moves cache block y_1 into its cache at time z_1. The cube extends to position x_1, y_1, z_2, where z_2 is the time that processor x_1 moves block y_1 out of its cache.

Before the cache occupancy view is created, the cost of each cache block[1] is calculated as the total cost over the whole execution of the program, in terms of CPU cycles spent by all processors waiting for that block to be moved into the cache. The blocks are then sorted in ascending order according to this cost. The most expensive block is then drawn at the top of the graph, while the least expensive cache block is drawn at the bottom of the graph.

An example of this visualization is shown in Figure 4. In this figure, cube Y-X represents the cache block with virtual address Y in CPU X's cache. Cube 3-1, therefore, shows how

[1] A cache block is a contiguous block in global memory which is moved into the cache as a unit. It is also referred to as a cache line.

cache block 3 is moved into the cache of CPU 1 at time t_2 and is moved out of the cache at time t_3. Cube 3-2 on the other hand shows when cache block 3 occupies CPU 2's cache. Cube 2-1 shows how block 2 is moved into the cache of CPU 1 at time t_1. In addition, cache block 2 is more expensive than cache block 3, because it is drawn higher up on the graph.

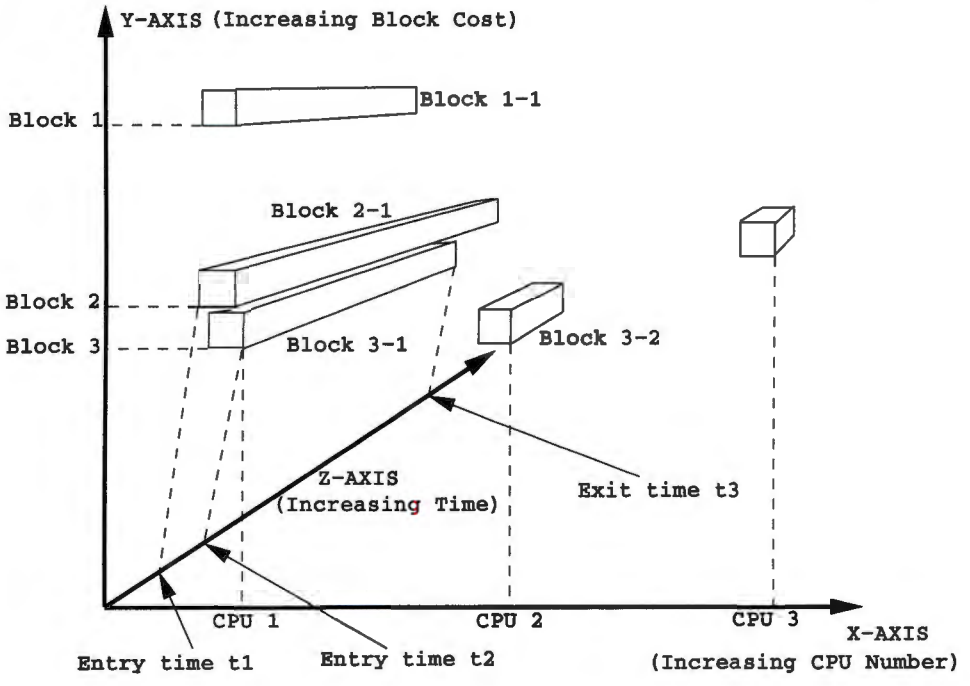

Figure 4: The cache-occupancy view.

Seeing that cubes with the same address are placed at the same height on the graph, the user can easily see how a cache block migrates between CPU caches, by looking at the graph along the x-axis. Figure 5 is an example of such a view, with the cache blocks of lower numbered CPU's drawn on deeper levels into the page. In this figure one can see how a block migrates from CPU 3's cache into CPU 1's cache, then migrates into CPU 2's cache after which it migrates back into CPU 3's cache.

If the viewer zooms out far enough, the cache behaviour for the entire execution of the program can be seen. From this viewpoint time flows from left to right, so the program starts at the left-hand side of the screen and ends at the right hand side. Figure 28 (Appendix B) is an example of such a view.

If the program starts executing at time $t = 0$, the leftmost point of the BlockView will represent time $t = x$, where x is the time when the first block being traced is accessed. As mentioned in chapter 5 the user can enable tracing after the application reaches a given line in the source code. This feature is useful when the user is not interested in the performance of certain sections of the code.

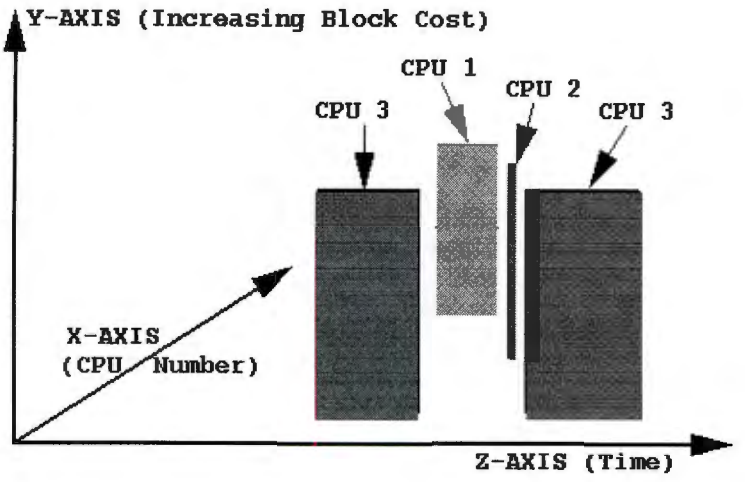

Figure 5: A cache block moving between CPU caches.

In such a case the cache behaviour will only be displayed in the BlockView once the given line is reached. An example would be the initialization phase of a scientific application where the data structures in shared memory are initialized. Seeing that the application only does the initialization once and spends most of its time in a computational loop, the user may not be interested in the performance of the initialization phase and will not want to fill the display with unnecessary data.

3.2.2 Sorting cache blocks on memory location

The cubes in the BlockView can be sorted according to memory location instead of cost. In some cases, sorting the blocks using this method helps the programmer relate the cache blocks back to the code. Figure 28 (Appendix B) shows a program which accesses successive elements inside an array. Successive cache blocks are moved into the cache which causes the diagonal step effect in the figure.

3.2.3 Selecting cache blocks

Each cache block in the cache occupancy view can be selected with the mouse and the following information for the block is printed in Chiron's object information window:

1. The virtual address of the block. This helps the user identify cache blocks and determine which cache blocks are located adjacent to each other in memory.

2. The memory location which was accessed when the block was brought into cache. This information helps the user determine whether false sharing is occurring in the cache.

3. The total cost of the block, which helps the user determine how expensive cache blocks are in relation to each other,

4. The C++ objects inside the block. This helps the user relate the cache block back to the objects in the source code.

3.2.4 Coherence and replacement misses

To help the user narrow down the cause of a bottleneck, the BlockView distinguishes between two types of blocks:

- Blocks which were evicted due to coherence misses
- Blocks which were replaced due to the replacement misses

As a default the BlockView displays all the blocks but a menu option allows the user to display only the replacement blocks or only the coherence blocks. This option affects the display of the blocks only and does not cause a re-sorting of the blocks. The user can then compare the different views to determine whether coherence misses or replacement misses are causing the high block traffic in a certain region of the program.

3.2.5 Coherence misses

With the cache invalidate protocol, a coherence miss results in the eviction of a cache block from one or more processor caches before that block can be moved into the cache which caused the miss. In the cache occupancy view a coherence miss will therefore cause the movement of a cache block on the same x-z plane because blocks in this plane have the same address. Figure 5 shows this movement, as viewed along the x-axis.

A common cause of coherence interference is false sharing. To determine whether false sharing is causing the movement of a particular cache block, the user has to select a few cubes which are at the same height on the graph. The user has to then look at the memory location which caused the selected block to be moved into the processor's cache to see whether this address differs between different instances of the block. If it is the same address, then the same object is being accessed by the different processors and true sharing is occurring. On the other hand, if it is not the same address then false sharing is probably occurring.[2]

[2]The definition of false sharing is not precise so there are grey areas between true sharing and false sharing.

Alternatively, the user can use the correlation view between the ObjectView and the BlockView to determine whether false sharing is occurring. This will be described in more detail in the following section.

3.2.6 Replacement misses

Seeing that a replacement miss occurs between two blocks with differing addresses, the block movement occurs from one y-level (representing the address of the evicted block) to the y-level which represents the address of the block responsible for the miss.

If the BlockView is sorted on memory location, as in Figure 28 (Appendix B), the user can see the effects of replacement interference more clearly. In this figure, the program is running through an array and adjacent blocks in the bottom right hand corner are replaced, causing the step effect. In the top right hand corner adjacent blocks are moved into the cache over the same time period (indicated by similar z values) as the blocks which were moved out of the cache. Each one of the blocks which was evicted is being replaced by the block which is moved into the cache just after the former block has left the cache. The user can verify this fact by selecting individual blocks and looking at the textual information.

When the user selects a block which was evicted due to a replacement miss, the address of the block which caused the replacement, the memory location which was referenced inside that block and the objects which reside in that block are displayed in addition to the standard information for a block. This tells the user which objects are colliding with each other.

3.2.7 The cache temperature view

The BlockView can also show the cache temperature view [12]. This view is similar to the cache occupancy view, but instead of showing the time period during which a cache block resides in a cache, this view indicates at what times during the execution of the program a block moves into or out of a CPU's cache.

The view is created in the same manner as the cache occupancy view, but instead of drawing an elongated cube between the time that the cache block enters the cache and the time that it leaves the cache, a point is drawn when the block enters the cache and a point is drawn when that block leaves the cache.

The advantage of this view is that memory performance bottlenecks, caused by excessive cache block movement, are indicated by a high concentration of points. When viewed from

a distance, such a concentration is easily visible as a cluster of points, as is illustrated in Figure 6. The user can then zoom into the area and determine the cause of the bottleneck.

Figure 6: The cache temperature view.

3.3 Correlation views

While the individual view can convey a great deal of information it is often necessary to relate this information to information from another view. For example, the user can easily see where bottlenecks are occurring in the BlockView, but it is necessary to relate this information back to source lines or data objects to be able to correct the problem.

As a solution, Chiron can create a correlation view when the user selects an object in one view and wants to see how this object correlates with objects from another view. The objects that it correlates with are highlighted by drawing them in a different colour or by drawing them as different 3D objects. For example, if the user selects a node in the SourceView, all the objects which were accessed by the selected source line are highlighted with a particular colour in the ObjectView.

In some cases the user rather wants to see how a metric of interest correlates with the data in the current view. The user can then select an additional metric through one of the menus in Chiron and this metric is then mapped onto the graph. This is achieved by mapping an additional colour onto the surface of a folded graph or by changing the height of objects in the graph.

3.3.1 The ClassView

The ClassView is a correlation view of the latter type. It shows the user which C++ class an object in the ObjectView belongs to, by colouring nodes from the same class with the same colour. Chiron presents a list of all the classes in the program and the user can then select these classes on an individual basis.

Each class is assigned a different colour from a predefined palette so that the user can easily distinguish between nodes from different classes. If many classes were created during the

run of the program, it might be difficult to distinguish between the different colours; in this case the user can change the colour of a class through a menu option.

The advantage of this view is that the user can correlate cost and class. If the folded graph is sorted according to cost, the user can quickly see whether the most expensive objects predominantly fall inside one particular class. If the folded graph has been sorted according to memory location, as in Figure 30 (Appendix B), the user can relate the cost of objects to the memory layout of the objects. In this figure the colour of a node represents its class while the position of the nodes on the folded graph represents their ordering in memory.

3.3.2 SourceView to BlockView

This view shows the correlation between a node in the SourceView and cache blocks in the BlockView. After the user has selected a node in the SourceView, all the blocks which were touched by this source node are highlighted in the BlockView. This correlation information is useful as it enables the user to relate sections of the BlockView back to the source.

3.3.3 SourceView to ObjectView

Chiron also displays the correlation between the SourceView and the ObjectView. When the user selects a node in the SourceView, all the objects which are touched by the selected source line are highlighted with the same colour in the ObjectView.

The user can then select the objects in the ObjectView and see which object is being represented by the node from the information in Chiron's text window. An object which is selected by the user is highlighted in a different colour to the objects which are referenced by the SourceView so that the user can easily distinguish between the two. The objects which are referenced by the selected source line also remain highlighted until the user selects another node in the SourceView. This enables the user to rapidly get more information about the objects which seem of interest.

Figure 31 (Appendix B) is an example where a source node was selected in the SourceView and an object was then selected in the ObjectView. The surface with the single red node represents the SourceView and its selected node, while the surface with the multiple red nodes and the single yellow node represents the ObjectView. The multiple red nodes are the objects which are touched by the selected node in the SourceView.

3.3.4 ObjectView to SourceView

In a very similar manner, every time the user selects a node in the ObjectView all the source nodes which referenced that particular object are highlighted in the SourceView. The user can thus see which source lines access the selected object in the ObjectView.

The single yellow node in Figure 31 (Appendix B) is the object which the user has selected in the ObjectView while the multiple yellow nodes are the source lines which touch the selected object.

The correlation views between the SourceView and the ObjectView are important for helping the user decide which source lines and which objects have the greatest impact on the program's performance.

The user will be looking for a source line which is reasonably expensive and which touches a number of objects which are also reasonably expensive.

The idea is to reduce the cost for more than one object at once. The volume below the surface represents the total memory access cost of the program, so reducing the cost for one object will not have that much effect on the total volume. However, reducing the cost on a number of objects at once will result in a greater reduction of the volume below the surface, which means a larger reduction in the total memory access cost of the program.

When the objects in the ObjectView are large in the sense that the structure being represented by the node contains many fields, it is difficult for the user to determine whether a field inside this structure is being excessively accessed, or whether all the fields in the structure are being accessed with equal distribution.

However, the correlation view between the ObjectView and the SourceView enables the user to see which source lines are touching the selected object. By then selecting each source node and looking at the actual source line it represents, the user can determine which field in the selected object is being accessed by the source line. For example the source line might be:

```
Global->num_collisions = cell->num_coll * coll_prob;
```

If the user selected the Global object, then it is quite clear that this line is accessing the `num_collision` field inside the Global structure.

If the majority of source lines access the same field in the object, then that field is probably accessed more often than any other field in the object and therefore contributes the most to the cost of the object.

3.3.5 ObjectView to BlockView

When a node in the ObjectView is selected, the blocks which were moved into the cache due to references to the selected object can be highlighted. The standard functions of the BlockView can then be used to determine what is affecting the performance of these objects.

Highlighting only those cubes representing cache loads caused by references to the selected object helps the user detect false sharing in the object's cache block. If all cubes are highlighted, then false sharing is definitely not occurring because the same object is referenced all the time. If some cubes are highlighted while some are not highlighted it is possible that false sharing is occurring, because different processors could be accessing different objects in the same block.

Figure 7 is an example of how the cubes would be highlighted if false sharing is occurring. In this figure, the user is looking down the Y-axis (which is pointing out of the page) at one particular cache block which is moving between CPU 1 and CPU 2's cache. The cubes representing cache loads caused by references to a particular object are drawn as solid blocks. Some of the cubes in CPU 2's cache are highlighted, showing that references to a particular object caused the cache block to be moved into CPU 2's cache. Because CPU 1's cubes are not highlighted, the user can see that references to another object are repeatedly causing the cache block to be moved out of CPU 2's cache into CPU 1's cache. This indicates that false sharing is occurring between the two caches.

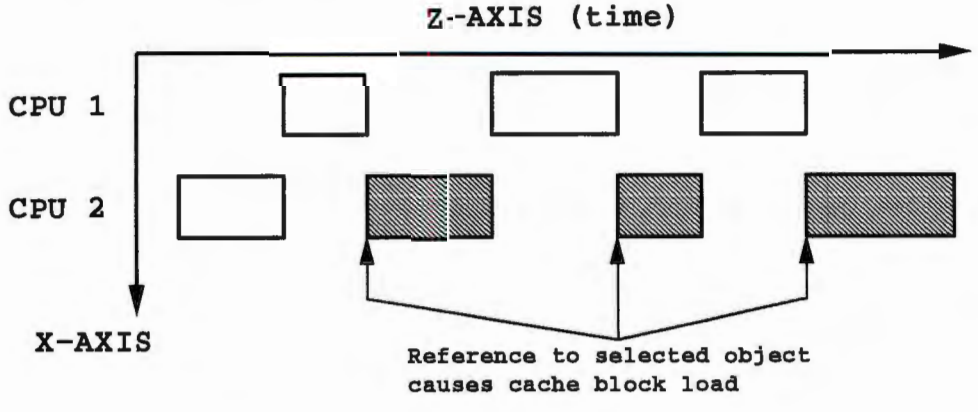

Figure 7: Highlighting cubes in the BlockView to show false-sharing.

Chapter 4

Implementation of Chiron

Chiron was written in C++ and uses the Inventor 3D toolkit from Silicon Graphics to display and manipulate 3D objects. The Inventor toolkit consists of a library of C++ classes, which allow the programmer to create and manipulate 3D objects in a 3D scene [32].

Some of the objects which the user can create are cubes, meshes and polygons. In addition, each object can be given a position, a rotation, a colour and a lighting model. The objects and attributes are stored in a database called the scene graph. Inventor includes methods for creating and manipulating the scene graph so that the visualization program can change the scene at run-time.

Inventor also allows the user to attach callback functions to external events. When the event occurs, the callback function for that event is called. In the case of a selection — with the mouse, say — Inventor also tells the callback function which object in the scene graph was selected. Chiron uses these callback functions to enable the user to select nodes in the global views and to select cubes in the temporal views.

4.1 Reading the trace file

Before Chiron can visualize any data, the data has to be read in from the trace file and stored in Chiron's data structures. The trace data is stored in an ASCII file with the information for each event stored on an individual line. This means that the trace file can be viewed and edited with a standard editor if the user wants to change the event information.

While Chiron reads sequentially through the trace file, four linked lists are built:

- A linked list of structures for each source line that causes at least one miss. Each structure contains the line number of the source line and the total number of CPU

cycles that all the processors spent waiting for a cache block to be moved into their cache, on this source line.

- A linked list of structures for each object in the folded graph. Each structure contains the address of the object which it represents and the total number of cycles which were lost when the processors accessed this particular object. The structure also contains the name of the C++ class to which this particular object belongs. The class of a structure is determined when that structure is created, by performing a lookup into a database. Every time a create or delete message is encountered in the trace file, the database is updated, so that subsequent lookups on memory locations will return the correct class.

- A linked list of structures for each cache block in the L1 BlockView. Each structure stores the address of the cache block which it represents and stores the total cost over all CPU's associated with this block. Each structure also keeps a linked list for each processor, which stores the times when that particular cache block was moved in and out of the particular processor's cache.

- A linked list of structures for each cache block in the L2 BlockView, similar to the above linked list for L1 blocks.

While Chiron reads sequentially through the trace file the linked lists are updated, depending on the event that is read in from the trace file. After scanning through the trace file, the four linked lists are used to create four Inventor scene graphs. These four graphs are then joined together to form the scene graph which is rendered by Inventor.

Each one of the resulting four subgraphs can be included or excluded in the rendered scene graph, at runtime, thus allowing the user to switch each of the four graphs on and off. Each graph can also be manipulated independently, so that the user can change the scale, position and rotation of a graph without this affecting any of the other graphs.

Chiron can easily be extended by adding more data structures to store any additional information and then creating Inventor subgraphs to display that information.

4.2 The user interface

Chiron is built around the SoSceneViewer, the source code of which is supplied with Inventor. The SoSceneViewer consists of a menu-driven user interface which allows the user to manipulate the 3D scene. For example, the user can attach manipulators to 3D objects, in order to scale, rotate or move the object. The SoSceneViewer also allows the

user to edit the colour of an object. This can be very useful, when the colours which Chiron chooses automatically are not to the user's liking.

The main device for interacting with the rendered objects is the mouse. As the mouse was designed for moving around in 2D space, it is difficult to navigate in 3D space with it. To help the user navigate in 3D space, the SoSceneViewer allows the user to select 1 out of 4 different viewers for moving around in the scene.

All four viewers have been included in Chiron, but the most useful viewers are the PlaneViewer and the ExaminerViewer. The PlaneViewer allows the user to move around rapidly in the x-z, x-y or y-z plane while the ExaminerViewer allows the user to rotate the scene in any direction around a user-defined point.

Three menus have been added to Chiron, in addition to the menus from the SoSceneViewer. The three menus are:

- The Views menu
- The LineView menu
- The MeshView menu

The Views menu allows the user to switch the four graphs on and off individually. The LineView menu contains options for the temporal views and the MeshView menu contains options for the global views.

Figure 27 (Appendix B) shows the Chiron interface after all four views have been switched on. This figure also shows Chiron's class list window, source line information window, and object information window.

4.3 Setting the level of detail

One problem that has to be addressed during visualization, is how much detail needs to be displayed. If too much detail is displayed the screen becomes cluttered and it becomes more difficult to interpret the data. If not enough detail is displayed, the potential of the graphics display is not fully exploited.

Displaying great amounts of detail implies a greater performance overhead on the workstation. With an interactive system like Chiron, it is important that this overhead is kept within the capabilities of the workstation otherwise the interaction will not take place in real-time.

As a solution to these problems, Chiron presents the user with various options to set the amount of detail that will be displayed. Manual level-of-detail suppression allows the user to decide how much detail is necessary, given the speed and resolution of the graphics workstation.

4.4 Global views

The surfaces for the global views are created by means of Quadmeshes. The Quadmesh is created by storing the 3D coordinates of each gridnode on the surface in an array, and then passing this array to Inventor.

Each node in the surface is then given a colour; in the initial view all the nodes in the SourceView and the ObjectView are coloured green. When the user selects a node, Inventor notifies Chiron which node was selected and Chiron then changes the colour of that node. In the correlation views, the colour of multiple nodes is changed.

4.4.1 Performance

Seeing that the Extreme graphics board of our Indigo2 graphics workstation is optimized to display triangles, large Quadmeshes can be displayed and rotated efficiently. The largest folded graph from our performance debugging, consisted of just over 20000 nodes and could still be rotated in real time.

The more serious performance degradation in the global views occurs when the user selects a node. In a folded graph of 20000 nodes or more, it can take up to 1 second for Inventor to determine which node was selected.

However, in the performance tests which were conducted, the SourceView never had more than 200 nodes. Both real applications used a loop to simulate the behaviour of a dynamic system as time progressed and a limited number of lines inside these loops accessed the data.

The ObjectView consisted of more than 20000 nodes only when memory words versus cost were displayed. As more nodes are displayed, the screen area that each node occupies is reduced. This makes it more difficult to select a particular node unless the user zooms into an area, in which case the overview of the whole graph is lost.

As a result, the user has to navigate around the graph to access all the nodes and the whole process of viewing the graph becomes cumbersome and less efficient. In such a case it is more useful to look at objects versus cost.

4.4.2 Level of detail in the ObjectView

By default, the ObjectView displays a folded graph of C/C++ objects versus cost. While this view is useful when a program creates many objects, it is less useful if the program only creates a few objects or the size of the objects differs by a large amount. For example, one object might take up 16 Kbytes of memory while another object might take up 4 bytes of memory. It is to be expected that the larger object is responsible for more cycles seeing that it will probably be referenced more often, however, each object will still be represented by a single node on the graph.

In this case the ObjectView can be misleading. An object which appears to be very expensive on the graph, might not be that expensive in terms of cycles per byte, if that object occupies a great deal more memory than the less expensive objects.

As a solution to this problem Chiron's ObjectView can display a folded graph of memory words instead of objects. To get this view, the user has to run Chiron with the '-c' option from the command line.

When run with this option each memory location which is accessed, is considered to be one object and for each of these objects a counter keeps track of how many cycles were caused by this object. Now each object is the same size and each object will occupy one node on the folded graph.

When run with this option, Chiron still keeps track of the class that a particular memory location belongs to. This enables Chiron to colour memory words from the same class in the same colour.

The difference is best explained by an example. A program creates an object called **my_array** which is an instance of the class **Array**, with the following statement:

```
Array* my_array = new Array;
```

where **Array** is defined as:

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

If Chiron is run without the '-c' command line option, **my_array** will be considered to be one object and all cycles which were lost due to accesses to the elements of **my_array**, will

be added to the counter of `my_array`. If all 100 elements of the array were accessed and each access caused lost a CPU cycle, then the cost of `my_array` will be 100 cycles and `my_array` will belong to the `Array` class. If another object of class `Array` is created, this object will appear as a separate node on the graph and it will also belong to the `Array` class.

If Chiron is run with the '-c' command line option, then each element inside `my_array` will be considered to be one object. In the above example, where all 100 elements of the array were accessed, we would instead have 100 objects, each with a cost of 1 cycles. These objects all belong to the class `Array`, so they will be given the same colour if the class colour is mapped onto the surface. This view is very helpful, as the user can see the object at a fine-grained level, but can still relate each object fairly easily back to the source.

The advantage of the latter method, is that if one particular element inside an object is responsible for a number of cycles, the user will be able to pick this up. With the former method this will not be possible, but the advantage is that fewer objects have to be displayed and therefore the response of Chiron will be much better.

If Chiron is used to analyze a program which creates many objects which are roughly equal in size, then the first method can quickly show the user which objects need to be looked at. This was demonstrated very well when the performance of MP3D was examined.

If the application being analyzed does not create many objects, it is more useful to run Chiron in the second mode. An example of such a case occurred when the performance of Barnes-Hut was analyzed and Chiron showed that only nine objects had been created.

4.5 Temporal views

Seeing that the global views give a static representation of the data, the duration of execution of an application does not affect the performance of Chiron a great deal, as a fixed number of lines or objects are accessed, irrespective of the duration of the execution.

However, seeing that the BlockView is a temporal view, the longer the execution of the program takes, the more data will have to be displayed. During the execution of a parallel program thousands of cache blocks may be accessed. This has a negative impact on the graphics performance, so Chiron displays only the cache occupancy for the 100 most expensive blocks.

However, the number of blocks which have to be displayed is still very large. For example, on a three timestep run of MP3D with 8 processors, up to 10000 cache blocks have to be displayed.

Displaying such a large amount of blocks as cubes would not only be impossible from a performance point of view, it would also be impossible for the user to understand a view of the whole graph as most of the screen will be filled with data and the CPU planes will obscure each other. For this reason Chiron offers various options for setting the level of detail.

4.5.1 Coherence misses and capacity misses

Two menu options in Chiron allow the user to toggle the display of cache blocks which were evicted due to coherence or capacity misses. These options allow the user to reduce the amount of blocks which are displayed, as the user can display only coherence blocks or only capacity blocks.

The display objects for cache blocks of coherence misses and cache blocks of capacity misses are stored in separate scene graphs so that Inventor only traverses the graph of those blocks which are displayed, thus improving the performance of Chiron.

4.5.2 Switching processor planes on or off

Because the cache blocks from different processors might obscure one another, especially if they are displayed as cubes, the user can switch cache blocks belonging to a particular processor on and off. The cache blocks of a particular processor are displayed in the same y-z plane and are therefore called processor planes.

4.5.3 Drawing cache blocks as points or lines

The points and line option in the BlockView menu allows the user to toggle the drawing mode of the blocks between points and lines. By default, each cache block is represented by two points: one point to represent the time that the block was brought into the cache and one point to represent the time that the block was moved out of the cache.

Displaying the cache blocks in this manner has a number of advantages:

- Points can be drawn and rotated efficiently.
- This view corresponds to the cache temperature view mentioned in the previous chapter.
- Points use up less screen space than cubes. Because the cache occupancy for each processor is shown in a different plane and these planes are stacked next to each

other, some cubes will be obscured by other cubes. Thus it will not be possible to see all the data at once.

Once the user has zoomed into the graph so that the density of the points is too low too relay any information, the blocks can be drawn as lines by choosing the points/lines option. Each block is then represented by a line, which starts at the point of time when the block is moved into the cache and ends at the point of time when the block is moved out of the cache.

Seeing that lines can be drawn more efficiently than cubes, all blocks in the BlockView are drawn as lines when the user selects this option from the menu. The user can then select any line to get detailed information on the cache block which it represents.

4.5.4 The lens

Drawing the cache blocks as points or lines is useful when viewing the graph down the x-axis. However, if the user wants to view the graph from a different angle, lines do not represent the 3D nature of the BlockView adequately and the user will want to see cache blocks as cubes.

It is extremely expensive from a performance point of view to draw every block in the graph as a cube so Chiron only draws the blocks within a certain region as cubes. This region is defined by means of a 'lens' which is represented by a yellow wire rectangle (see Figure 27 in Appendix B).

The user can select the lens with the mouse and can slide it along the time axis of the BlockView to any position. When the lens is placed in the new position, the cache blocks inside the lens are only redrawn when the user selects the redraw button in the top left-hand corner of Chiron's interface. Drawing the blocks inside the lens as cubes or planes is a lengthy operation which can take a few seconds so the user has to be certain that the lens is in the correct position before selecting the redraw option.

The L1 BlockView and the L2 BlockView each have their own lens, however, if the lens in one view is moved to a new position, the lens in the other view moves along with it and will be placed in the same position on the z-axis. The advantage of this is that the user can position the lens on a particular event in the one view, and can use the lens in the other view to see what happened at the exact same instant in time in this view.

For example, the user might want to see what happened in the L2 cache at the time that a particular block was moved out of a L1 cache. To do this the user would move the lens so that one of its edges is aligned with the block moving out of the cache. Then the same

edge of the lens in the L2 view will be positioned at the correct time and the user will be able to see what happened in the L2 cache at that particular instant in time.

Each lens can still be selected separately, so that the user can use the "View Selection" option to rapidly zoom into the area covered by the lens. This menu option instantly brings the lens into the centre of the screen, with the lens filling up most of the screen. This means that the region of interest is immediately accessible to the user.

4.5.5 Highlighting cache blocks for correlation views

If the user selects the redraw button after selecting a node in the SourceView or the ObjectView, the cache blocks which relate to the selected node are highlighted. This is achieved by drawing the highlighted blocks as cubes or planes (depending on the selected menu option) while the other cache blocks are drawn as lines or points.

Figure 32 (Appendix B) shows how cache blocks are highlighted as planes in the L1 BlockView after an object was selected in the ObjectView. The blocks which are not highlighted are drawn as lines. Section 3.3.5 of the previous chapter describes how this highlighting of cache blocks in the BlockView helps detect false sharing in a program.

4.5.6 Face sets

Cache blocks inside the lens, or highlighted cache blocks can be drawn either as cubes or as faces by selecting a toggle option from Chiron's menu. Drawing the blocks as faces is more efficient from a performance point of view because each block will be represented by one single rectangle as opposed to 6 rectangles per block if the cube option is selected.

Instead of drawing the rectangles parallel to the y-z plane, Chiron draws the rectangles which represent coherence blocks, at 45° to this plane and rectangles which represent replacement blocks, at $-45°$ to this plane.

When the graph is being viewed down the x-axis, with the y-z plane parallel to the plane of the screen, the user will be able to see the rectangles from coherence and replacement blocks equally well and will thus not be able to distinguish between the two.

Figure 8 is a 2D schematic of the above scenario with the z axis going into the page, so that the rectangles are seen edge on. The top rectangle represents a replacement block while the lower rectangle represents a coherence block. From this figure it should be clear that the user will be able to see both rectangles equally well, although both rectangles will not be very bright, seeing that their surface normals point at 45° to the normal of the users headlight.

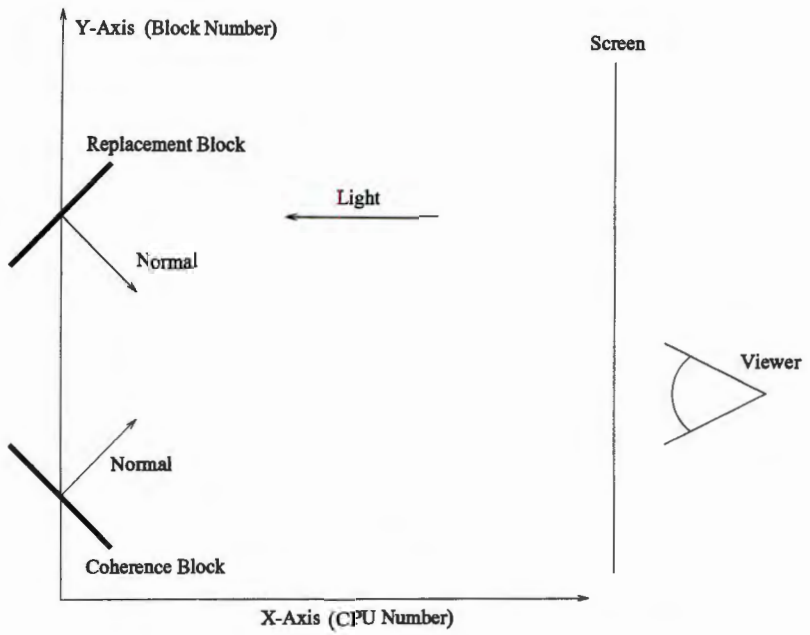

Figure 8: Viewing face sets along the X-axis.

If the user rotates the whole graph by 45° counterclockwise along the z-axis, the rectangles of the replacement blocks will be viewed edge on and will thus be invisible. However, the coherence blocks are now at 90° to the user's line of vision and thus achieve maximum visibility. This is illustrated in Figure 9.

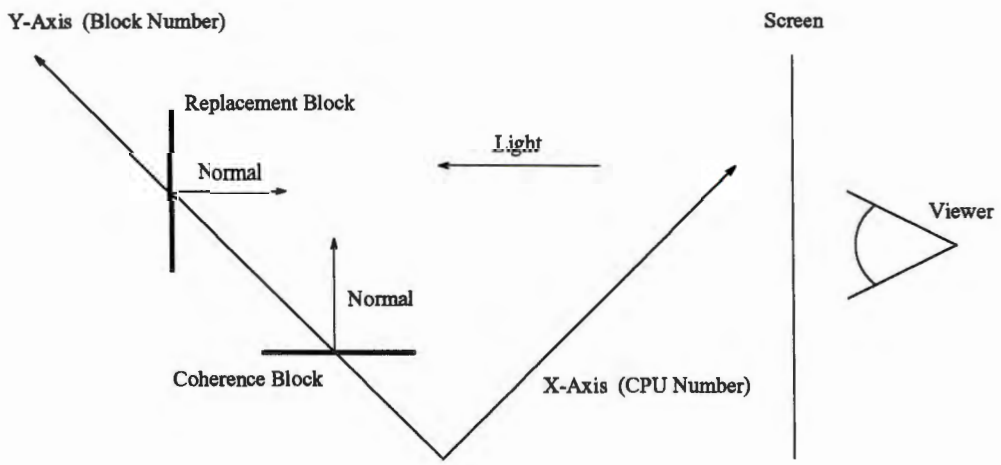

Figure 9: Viewing face sets at 45° to the X-axis.

The advantage of displaying the rectangles in this manner, is that the user can rapidly distinguish between replacement and coherence blocks just by rotating the graph through an angle of 90°, using the the mouse. Because the two kinds of rectangles are at 90° to each other, when the one has achieved maximum visibility, the other will be virtually invisible, seeing that it will be viewed edge on. Thus the rectangle from the opposite type

will not obstruct the view of the rectangle which is being viewed face-on.

The user can therefore distinguish between the two types of cache misses by manipulating the image itself. This results in a tight visual coupling between the different modes.

Alternatively, the user can display only coherence or replacement blocks by choosing the appropriate option in the BlockView menu. However, this can be time consuming for big datasets because the operation involves updating the Inventor scene graph and thus the tight visual coupling is lost.

4.5.7 Cubes

The face set view is designed for viewing along the x-axis and loses its meaning when viewed from any other angle. Chiron can, however, draw cache blocks in the cache occupancy view as elongated cubes.

Drawing the blocks as cubes is the most expensive option but it is also the most useful when the user has zoomed into the view and wants to see all the detail from any angle. Seeing that cubes are 3D objects it does not matter from which direction in space they are viewed, it is easily possible to recognize individual cubes and to determine their orientation in space.

In addition, cubes can be easily selected from any position, so if the user wants to select multiple cache blocks for more information, cubes are the most effective option.

The cubes in both BlockViews are drawn by means of Inventor cube objects and are shaded to emphasize their orientation.

Colour

All cubes belonging to the same CPU are given the same colour. In Figure 4 it is difficult to distinguish between blocks from different CPU's and between blocks which are deeper in the page because all blocks have the same colour. Colour is therefore a vital clue for the performance analyst.

Perspective

Because cubes are 3D objects, the amount of perspective used to render the cubes affects the user's interpretation of the scene. Chiron therefore allows the user to change the amount of perspective by means of a zoom slider.

With the lowest zoom angle an orthogonal projection of the scene is achieved. Distances can be measured and compared in an orthogonal projection, which is useful if the user wants to compare the start times and end times of cubes.

With a large zoom angle, a perspective projection of the scene is achieved and the cubes are distorted due to the effects of perspective. This distortion is useful for conveying 3D information — for example cubes which are further away from the user's viewpoint will appear smaller. This helps the user see the scene in three dimensions even though the scene is being displayed on a two dimensional device.

In addition to the the zoom slider the user can move in and out of the graph, using Chiron's controls. Used in conjunction with the zoom slider, the user can get the required amount of detail and perspective.

Chapter 5

Generating the data

Visualizing the performance of a program with Chiron is a two-step process. First, the performance of the program is monitored and specific events are captured in a trace file. Then Chiron analyzes that trace file and visualizes the data.

5.1 Monitoring a program's execution

It is important that the trace data accurately represents the behaviour of a program, if real-world applications are to be debugged successfully. However, the monitoring process can be intrusive and can alter the behaviour of the program.

There are three methods of monitoring parallel programs:

- **Hardware monitoring:** This the most expensive method as it requires external hardware to monitor the system, but it is non-intrusive and it enables real-time performance debugging. However, it is difficult to relate the events which are generated to the program, which in turn makes it difficult for the programmer to gain insight into the functional behaviour of the program [31].
- **Software monitoring:** Software monitoring, on the other hand, does relate the events to the actual program but the timing is not as accurate and it incurs an extra overhead on the execution of the program, which affects the order of events [31].
- **Simulation:** Event-driven simulation avoids the problems associated with the above methods of monitoring because routines can be inserted into the simulator to write the required information to the trace file. Even though this can degrade the performance of the simulator, the order in which the memory references are generated

does not change because the parallel program is executed together with the simulator. This is vital to ensure that the memory reference pattern between the monitored and unmonitored program does not change. However, the performance overhead of simulation prevents the simulation of applications with large data sets.

Chiron was developed to help study the performance of shared-memory multiprocessors which are currently under development. Simulation was therefore used to generate the trace data for the following case studies. Chiron can also visualize the trace data which is generated by other methods, provided the required events can be captured in the trace file. This means that hybrid monitoring [31] — which is a combination of software monitoring and hardware monitoring — would have to be used to capture timing information as well as functional information.

The simulation is performed in two steps: a trace collection process and a trace analysis process.

The trace collection process is performed by Mint [40], which interprets an executable and passes all references to shared memory to the memory analyzer. The memory analyzer assigns each reference to the appropriate processor and calculates the time that it will take for the reference to be completed. This time delay depends on the communication time between processors and the memory latencies of the caches in the cache hierarchy.

If particular events occur during the simulation of a reference, the analyzer writes the details of this event to the trace file. The information in the trace file is then visualized by Chiron during the visualization phase. Should it be necessary to write out more information about a program's performance, the simulator can be modified to write out the new information.

Once the analyzer has simulated a particular reference, control is passed back to Mint, which then continues interpreting the executable. Mint uses the time taken to execute instructions and the duration of memory references, to determine the order of these references. It is important that this order is correct if we want to study the performance of real applications on real multiprocessors.

Writing the events to a file instead of passing them directly on to Chiron is fast and enables the user to analyze the data any number of times after the simulation, without having to run the simulation every time. This is useful as parallel programs are not necessarily deterministic and separate executions can therefore generate different results.

One disadvantage of writing the events to a trace file, is that the trace of even a small application can grow extremely large. Fortunately, modern hard disks are fast enough and big enough to be able to store the vast amount of data generated during a program trace.

For example, the case studies for this thesis generated trace files less than 40 Mbytes in size. Chiron can read a file of this size in less than 50 seconds and files of this size can easily be accommodated on modern multi-gigabyte disks. Seeing that the events are stored in the trace file in ASCII format it is possible to compress the trace files using standard compression tools to reduce the file size by 50%.

5.2 The Paradigm simulator

This research was conducted in the context of studying the architecture and behaviour of an experimental multiprocessor, called Paradigm [6]. The memory analyzer used for the following case studies therefore simulates the Paradigm memory system. At the bottom level (the first level) each CPU has its own cache, which is usually on-chip. A number of the first level (L1) caches are then connected via a bus to one cache on the next level (L2). More levels can be added to this, where each new level is created by connecting a number of caches from the previous level to one cache on the new level.

At the topmost level the caches are connected via a bus to the global memory which contains the shared address space. In general, the cache size increases as one goes up the memory hierarchy and the memory access time increases as well. This means that the access time for a particular block of data depends on which level of memory the data is found in.

Paradigm is a cache coherent multiprocessor with invalidate protocol. If there are multiple copies of a block in different caches, these copies all have to contain the same information. If a processor wants to write to one of these blocks all the other copies have to be invalidated before it can perform the write operation.

Even though Chiron was designed specifically to visualize the memory performance of the Paradigm architecture, it can also visualize the memory performance of any shared memory system which has a similar memory hierarchy. For example, the Silicon Graphics (SGI) 4D/380 shared-memory multiprocessor also has multiple levels of cache, but the processors do not share any caches as they do in the Paradigm architecture. Chiron can still visualize the performance of a program running on the SGI 4D/380, as long as the performance data can be captured in the right format in a trace file.

Chiron can visualize the memory performance of a multiprocessor with an unlimited number of processors, but the visualization is limited to two cache levels. However, Chiron can easily be extended to show the performance of extra cache levels seeing that the performance of each cache level is shown in a separate view.

Chiron currently does not visualize the performance of the TLB (Translation Lookaside Buffer) even though this can have a large impact on a program's performance [22]. If the performance of the TLB can be captured in a trace file, an additional view can easily be added to Chiron to display its performance.

Figure 10 shows a configuration of the Paradigm architecture which will be used in the case studies later on. In this example there are four processors, each with an on-chip cache. A group of two processors is connected via a bus to one L2 cache and the L2 caches are then connected to the global memory via another bus.

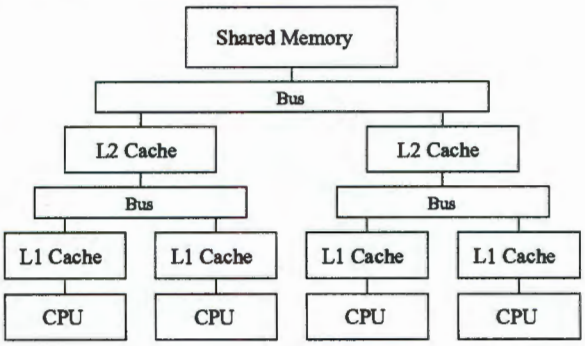

Figure 10: The Paradigm architecture.

5.3 Generating a trace for Chiron

The process of generating the trace data from a C++ application is as follows. First the programmer writes a C++ application using ANL Macros to split the computation among the processors. The application is then preprocessed and the ANL macros are replaced with calls to the Mint simulator. When the application is compiled, these calls are replaced by stubs and the executable is generated.

The executable is then interpreted by Mint and each memory reference by the program is passed to the Paradigm memory reference simulator which determines how long each reference will take, by taking factors such as cache coherency cost and memory latencies into account. Mint also passes certain parameters to the memory reference simulator, such as the time at which the memory reference was made, the CPU which issued the reference and the source line where the reference occurred.

Mint also intercepts malloc calls and on every such call sends a message to the Elf memory tracking system, specifying the memory location and size of area being allocated by the malloc. Elf then writes this information to the trace file. The name of the pointer to the allocated memory is also written to the trace file. Chiron uses this information to map memory references onto objects instead of memory words.

At the start of the simulation the memory simulator writes a header to the beginning of the trace file, describing the configuration of the current simulation. This header includes how many L1 and L2 caches there are, and how big the L1 and L2 cache blocks are. This header is used by Chiron to interpret the trace data.

The block sizes and cache sizes for the Paradigm simulator are defined in a configuration file. By changing the parameters in this file the user can see how the new memory architecture will affect the performance of the program without having to recompile the simulator.

While running the program through the simulator a file is generated which contains a list of all the source files which the program accesses. This file is used by Chiron to map line numbers back to the correct source file.

The process of generating a trace file is shown schematically in Figure 11.

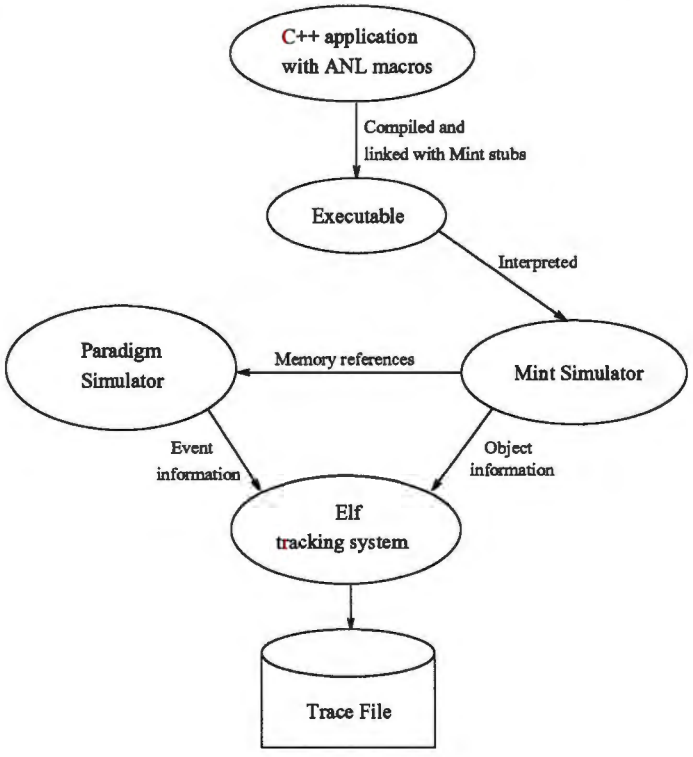

Figure 11: The process of generating a trace file.

5.4 Switching tracing on and off

Very often the user will only be interested in the memory performance of a particular section of code. For example, the application called MP3D which is used as a case study

in chapter 6 consists of an initialization section and a computational loop. The computational loop can be executed any number of times whereas the initialization section is only executed once.

In such a case it is unlikely that the performance debugger will be interested in the performance of the initialization section and will rather concentrate on the performance of the computational loop because the application spends most of its time there.

We have implemented two functions TRACE_ON and TRACE_OFF which enable the user to switch tracing on and off anywhere in the program. Mint recognizes these two functions as special functions and sends messages to Elf which enable or disable the writing of messages to the trace file.

The significance of this is that the memory performance will be simulated whether tracing is enabled or not and only the writing of the messages to the trace file will be affected by the two functions. The advantage of this is that the memory reference stream will not be affected by the TRACE_ON and TRACE_OFF functions, but the disadvantage is that the performance of the program will suffer even if tracing is disabled, seeing that each memory reference will still be sent to the memory system simulator.

Most important, though, the user can decide how much trace information needs to be stored, thus reducing the size of the trace file and reducing the amount of information which will have to be displayed during the visualization phase.

5.5 Events written to the trace file

The following lines (with one event per line) are from an example trace file:

```
i 32
i 128
i 8
i 2
c Global 10000000000 6060 1
m 10000000650 0 1 1000315
m 10000000650 0 2 1000315
h 10000000650 0 2 1000315 7164
h 10000000650 0 1 1000315 7165
W 10000000650 0 1000315 57
m 10000000620 0 1 1000370
h 10000000620 0 1 1000370 7366
```

```
W 10000000620 0 1000370 20
m 10000000700 0 1 1000644
h 10000000700 0 1 1000644 10270
W 10000000700 0 1000644 20
m 10000000544 0 1 1001203
m 10000000544 0 2 1001203
h 10000000544 0 2 1001203 10663
h 10000000544 0 1 1001203 10664
W 10000000544 0 1001203 57
c bodytab 10000007120 40000 1
```

The first four lines are the header lines — indicated by the 'i' at the start of the line. Lines starting with a 'c' are the object creation events which Elf received from Mint. These events specify the name of the object, its location in memory, the size of the object and the processor which created it. When the memory for an object is de-allocated by the parallel program, Mint sends the Elf system a delete message for that object. This message is also captured in the trace file and is indicated by a 'd'.

When the trace file is scanned by Chiron, the information from the object creation and deletion messages is entered into the Elf database. For memory addresses referenced by certain events in the trace file, Chiron performs a lookup on the address in the Elf database. If the memory address falls inside an object, the Elf database supplies the name of the object.

This information is used by Chiron to visualize the cache performance of objects instead of memory words. In a C++ program these objects are equivalent to instances of C++ objects; in a C program the objects refer to chunks of memory allocated by a malloc call.

All other messages in the trace file are caused by events which were sent from the simulator to Elf. The format of these messages is generally:

```
MessageId MemoryPtr ProcessorId Source Time
```

MessageId is a alphabetic character identifying the message type. **MemoryPtr** is the address which was accessed when the event happened. **ProcessorId** is the identifier of the processor which generated the event; **Source** refers to a line number in the source file where the memory reference occurred. **Time** is the absolute time when the event occurred, in terms of simulator timesteps from the start of the program. Some messages contain more fields of information; a detailed description of each type of message is included in Appendix A.

Chapter 6

Debugging Program Performance with Chiron

This chapter presents the process of performance debugging with Chiron, and the results achieved for three parallel applications. The first application to be debugged is a small parallel program which performs vector addition, while the remaining two parallel programs are scientific applications from the SPLASH [33] benchmark suite. The concepts of Chiron are illustrated with the small program while the remaining two applications show how Chiron is used for realistic applications. The results from these case studies show that Chiron is particularly useful for detecting coherence interference.

All three programs are written in C or C++ and use the ANL [4] macros for the parallel constructs. The ANL macros use the concept of monitors to synchronize the manipulation and access to shared data. These macros can be ported to any machine which supports shared-memory, so parallel programs which use the ANL macros are highly portable. In addition, ANL macros exist to simulate multi-processing on a uniprocessor, using multiple processes. These macros were used with the Paradigm memory simulator to generate the memory reference traces for our applications.

6.1 Vector addition

The first application is a trivial program which adds two arrays in parallel and stores the result in a third array, as shown in Figure 12. The behaviour of such a small program is easy to understand, which makes it suitable as an initial example to explain Chiron's views.

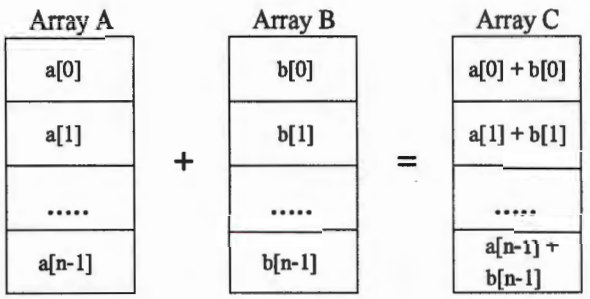

Figure 12: Adding two arrays.

The following pseudo-code shows how CPU 0 initializes the two arrays to be added, then creates the slave processes to perform the addition, before printing the results of the addition.

1. Allocate a chunk of shared memory, called `Global`, which contains `ArrayA`, `ArrayB`, `ArrayC` and the monitor `GS`.
2. Initialize the monitor `Global->GS`
3. Initialize `Global->ArrayA` with data
4. Initialize `Global->ArrayB` with data
5. Create slave processes for CPU 1 to CPU 3, which perform the addition
6. Wait for all slave processes to finish
7. Print the result of the addition, which is stored in `Global->ArrayC`

The following pseudo-code shows how the slave process on each of the remaining three processors adds one subscript of the vector at a time, until the whole vector has been added.

1. Get `index`
2. While `index` is greater or equal to 0
3. BEGIN
4. `Global->ArrayC[index] = Global->ArrayA[index] + Global->ArrayB[index]`
5. Get `index`
6. END

Seeing that `ArrayA`, `ArrayB` and `ArrayC` are in shared memory, all processors can access the data in these arrays to perform the addition. The difficult part is assigning each processor a different index so that different processors do not access the same data.

To achieve this the ANL GETSUB macro is used to get an index:

```
GETSUB(Global->GS, index, ArrayLength-1, 3)
```

The first parameter to the GETSUB macro is the name of the monitor which controls the access to shared memory and which was initialized by CPU 0. The next parameter is a variable through which the result of GETSUB is passed. The third parameter to GETSUB specifies the maximum value of the subscript while the last parameter specifies how many processors will be competing for a subscript.

If there are no more subscripts, GETSUB returns -1 in index, otherwise GETSUB returns a subscript in the range 0 to `ArrayLength-1`. The execution of the GETSUB macro is an indivisible operation (because it is a monitor operation), which ensures that no two processors will receive the same subscript.

However, a processor requires exclusive access to a lock inside the monitor to ensure the indivisibility of the monitor operation. The following case study will show that contention can occur on such a lock because processors frequently require exclusive access to the same memory address.

6.1.1 Adding one element at a time

The first version of the vector addition program used the above example of the GETSUB macro to assign one element at a time to the processors performing the addition.

The execution of this program was simulated with four L1 caches of size 1 Kbyte, each with a block size of 16 bytes, and with two L2 caches of size 8 Kbytes with a block size of 128 bytes. The three arrays contained 96 elements each, where each element was 4 bytes big. The arrays were kept small to simplify the explanation of Chiron's views and all three arrays almost fit into the 1 Kbyte L1 cache, thus minimizing replacement interference.

The ObjectView of this version of the program displays an almost flat plane with a very high peak in the corner, as can be seen in Figure 33(a) (Appendix B). This shows that the most expensive object is much more expensive than the other objects and therefore it is likely that reducing the cost of accessing this object will improve the performance of the program.

Selecting the most expensive object reveals that it is the GS variable. The high access cost of this variable is to be expected, seeing that it is accessed during every GETSUB call. Each processor has to acquire exclusive access to a lock inside the GS variable to ensure the indivisibility of the GETSUB operation. Seeing that the GETSUB routine is called inside a loop one can expect that some contention will occur for the lock variable.

However, the source to object correlation view shows that the most expensive source node touches almost all the objects in the program with the exception of the GS variable — see Figure 33(b). Chiron's text window shows that this source node represents the line which performs the addition. Seeing that this line runs through all the elements in all three arrays, it accesses many objects. The cumulative effect of the cost of each individual access can explain the high cost for this source line.

The ObjectView and the SourceView have thus indicated two possible performance bottlenecks: the access to the GS variable and the addition of the arrays. This shows how important it is to display both code-and data-oriented performance information. To see whether these bottlenecks can be eliminated, the user needs more detailed information — which can be found in the BlockViews.

Figure 28 (Appendix B) shows the BlockView of the L1 cache. The cache blocks have been sorted according to memory location to help identify them. As indicated in the figure, the cubes in the lower third of the screen represent cache blocks occupied by **ArrayA**, while the cubes in the middle of the screen represent cache blocks occupied by **ArrayB**. The cubes in the top third of the screen represent the cache blocks occupied by **ArrayC** with the exception of the topmost two lines of cubes which represent cache blocks which are occupied by the GS variable.

The purple cubes — which are on the plane which is parallel to the page and furthest away from the viewer — represent cache blocks accessed by processor 0. The white cubes, which are closest to the viewer represent cache blocks accessed by processor 3.

Time flows from left to right in this drawing so the steep slope of the purple plane from the bottom left corner, upwards, shows how processor 0 moves successive cache blocks into its memory as it initializes **ArrayA** and then **ArrayB**.

After that the other three processors access **ArrayA** and **ArrayB**, which is shown by the three 'trapezoids' which are stacked on top of each other like pancakes, with the white 'trapezoid' being on top. The two planes below it are not clearly visible.

The 'staircase' in the top third of the screen represents the cache blocks which are accessed as the three processors write to successive elements in **ArrayC**. The purple triangle further to the right represents processor 0 reading elements from **ArrayC** before printing out the

results.

The short length of the cubes in the above 'staircase' means that these cache blocks are staying in the cache for a short time only. There is once again the possibility of a bottleneck.

Figure 13 shows a close-up of a group of short cubes. All four cubes represent the same cache block in the cache of three different processors. The reference which brings the same cache block into the cache of a processor is to a different location for each cube. False sharing is probably occurring because if CPU 1 and CPU 2 had not required exclusive access to the block, CPU 3 could have accessed the two locations (represented by its two cubes) without having to reload the cache block.

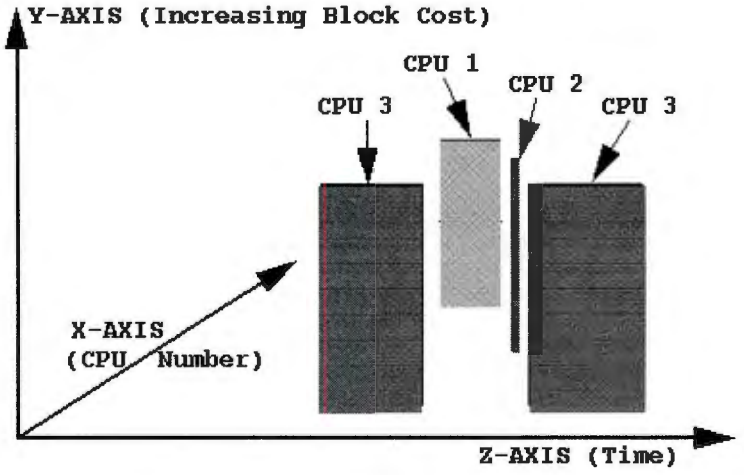

Figure 13: False sharing causes a block reload for CPU 3.

Because one cache block is 16 bytes big, four elements from an array fit into one block. Figure 14 shows how the first four elements of `ArrayC` fit into one L1 cache block and how the three processors access the elements inside that block. When processor 1 writes to `ArrayC[0]` the L1 cache block is moved into processor 1's cache. When processor 2 writes to `ArrayC[1]` the cache block has to be moved out of processor 1's cache and into processors 2's cache. Then processor 3 writes to `ArrayC[2]` and the cache block has to be moved across into its cache. Finally, when processor 1 writes to `ArrayC[3]` the original cache block has to be moved back into its cache. This means that processor 1 had to perform an unnecessary cache load to access `ArrayC[3]`.

The two lines of short cubes at the extreme top of Figure 28 (Appendix B) also indicate a potential performance bottleneck. If the BlockView is sorted according to cost these two lines of cubes are drawn at the top of the BlockView. This means that the two cache blocks represented by the cubes are the most expensive blocks. The high cost of these

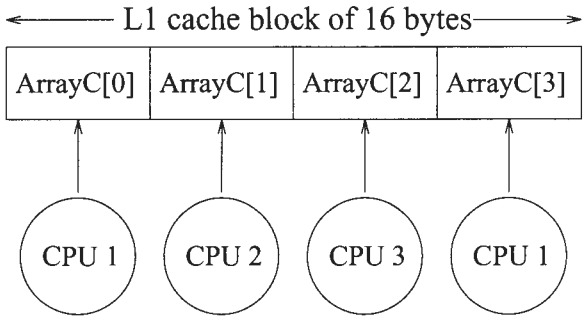

Figure 14: Four consecutive array elements fall into one L1 cache block.

two cache lines is explained by the numerous short cubes, which indicate that these cache blocks are moving rapidly between processors' caches.

Selecting any of these cubes shows that they represent cache blocks which are occupied by the `GS` variable. This information supports the information from the ObjectView, which shows that the `GS` variable is the most expensive variable.

The ObjectView to BlockView correlation view for the `GS` variable (see Figure 15) highlights all the cache blocks containing `GS`. This means that accesses to this variable cause the block movement between the caches and true sharing is occurring.

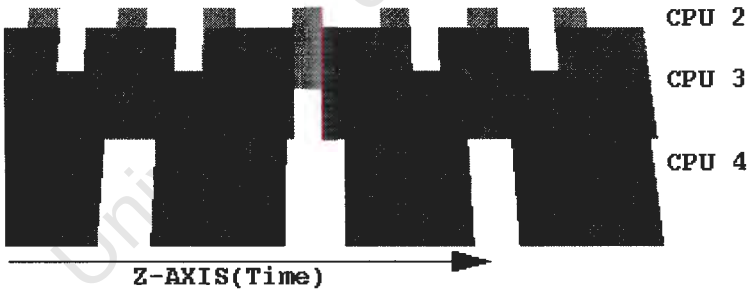

Figure 15: Highlighting of cache blocks across CPU planes indicates true sharing.

The problem is that each processor involved in the addition has to have exclusive access to the `GS` variable before it can get an index. The only way to reduce the cost of this cache block is to reduce the number of times that a processor calls the GETSUB routine. This means changing the work allocation scheme so that each processor adds a number of elements after a GETSUB call instead of adding only one element.

6.1.2 Adding 4 elements at a time

To prevent the false sharing in the L1 cache, caused by writing to `ArrayC`, each processor must be given 4 consecutive elements to add. The first element of these 4 elements has

to be block aligned so that the 4 elements fit exactly into one L1 cache block. Also, by giving each processor 4 elements at a time instead of one element at a time, the monitor `Global->GS` has to be accessed four times fewer than previously.

The views of the new program's performance are very similar to the previous views. The `GS` variable is still the most expensive variable, but its cost has been reduced from 12666 cycles to 3917 cycles, which is a 3 fold improvement. The addition line is still the most expensive source line but its cost has been reduced from 7505 cycles to 5662 cycles, which is a 25% improvement.

In the BlockView there is noticeably less movement of cache blocks between processors for the `GS` variable. There is no more false sharing in the array which stores the result, but processor 1's blocks are moved out of the cache after every write to an element. The BlockView of the L2 cache level reveals that false sharing is occurring at the L2 level. Because more than one L1 block maps onto the same L2 block, different processors can write to different L1 blocks which map onto the same L2 block.

Processor 2 and processor 3 share the same L2 cache so they cannot cause false sharing on the L2 level. However, processor 1 is connected to a different L2 cache so false sharing will occur on this level if processor 1 requires exclusive access to the same L2 cache block as the cache block requested by processor 2 and processor 3.

Figure 16 shows how CPU 1 accesses a L1 cache block which maps to a L2 cache block in two different L2 caches. False sharing can occur between the two L2 caches, because this L2 block also contains L1 cache blocks which are exclusively accessed from CPU 2 and CPU 3.

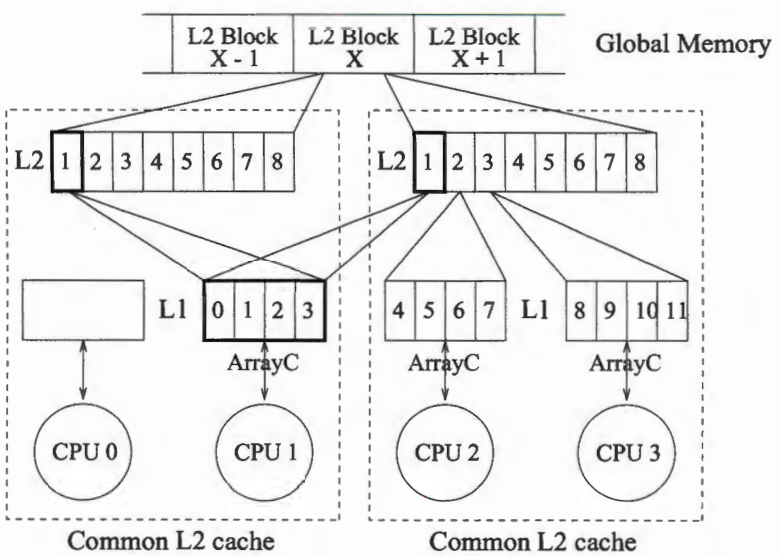

Figure 16: False sharing in the L2 cache.

6.1.3 Adding 32 elements at a time

To prevent false sharing between different L2 caches, the vector addition program has to be modified so that each processor is assigned 32 elements at a time. As the L2 cache size is 128 bytes, 32 elements will fit into one cache block if the first element is aligned to the L2 block boundary.

The ObjectView of the new version shows that the GS variable is still the most expensive variable but it is only responsible for 962 cycles, which is a 13 fold improvement over the initial 12666 cycles. The source line which performs the addition is still the most expensive with a cost of 1539, but this is almost a 5 fold improvement over the initial cost of 7505.

Figure 29 (Appendix B) shows the L1 BlockView for this version of the program. All false sharing has been eliminated and the yellow, red and white trapezoids do not overlap anymore because different processors do not access the same L1 cache block anymore.

The following table shows how the execution time in terms of simulator cycles has been reduced by improving the program's memory performance. The table also shows how the number of cycles have been reduced for the three arrays and the GS variable. The last row in this table shows that the cost for all three arrays is roughly equal seeing that no more false sharing occurs for **ArrayC**.

Program	Time	ArrayA Cycles	ArrayB Cycles	ArrayC Cycles	GS Cycles
No Opt.	86108	1765	1798	5382	12666
L1 Opt.	81649	996	1036	5072	3917
L2 Opt.	78758	986	1007	1056	962

Table 1: Execution times and memory cost for the array addition.

The reader should note that the time given in the first column is the wall clock time (in CPU cycles) from the start to the end of the program. The figures given in the other columns are cumulative totals for all four processors. Thus the program's overall execution time is improved by only 7350 cycles (8.5%) even though 11704 CPU cycles were saved on accesses to the GS variable.

6.2 MP3D

MP3D is a particle-based wind tunnel simulator which is used by NASA to simulate space vehicles moving through the upper atmosphere at hypersonic speeds. In MP3D, particles flow through the wind tunnel and collide with other particles, the boundaries and the obstacle, before exiting the wind tunnel.

The space in the wind tunnel is divided into a number of cube-shaped cells. For every time step that MP3D simulates, each particle inside a cell is moved to determine whether it stays in its cell or enters another cell. After this calculation has been performed for all particles, the collisions inside each cell are simulated.

MP3D was originally written for a vector architecture before it was implemented on shared-memory multiprocessors. The version of MP3D used in this case study has been converted to C++ and optimized for shared memory multiprocessors by aligning and padding the C++ objects to fit cache block boundaries [22].

A 3 timestep execution of MP3D with 1000 molecules was simulated by the Paradigm simulator. The L1 cache for this run was set to a size of 16 Kbytes with a block size of 64 bytes. The L2 cache was set to a size of 128 Kbytes with a block size of 256 bytes. A total of 8 processors was used, with 4 processors sharing one L2 cache. Each processor had its own L1 cache.

The above cache sizes are small, compared to today's multiprocessors, but the smaller size means that replacement interference will already occur with a small number of particles. This means that we did not have to use a large number of particles to study the effects of replacement interference on MP3D.

Tracing was only enabled after the initialization phase of MP3D was completed by processor 0. All of the eight processors moved 1000 particles over 3 timesteps, which took approximately 2.5 million CPU cycles to complete.

The resulting trace file was 15 Mbytes big, which is small when one considers the size of hard disks today. On an Indigo2 workstation with 96 Mbytes of memory, Chiron requires about 45 seconds for the initial setup.

6.2.1 The global views of the original MP3D

Figure 17(a) shows the ObjectView which has been sorted on cost, of the initial version of MP3D. From this figure the user can deduce that the most expensive objects have a very high cost in comparison to the others, as the folded graph drops off very suddenly near the beginning and is very flat for the remaining square. This is also evident from the ObjectView which is sorted according to memory address, as shown in Figure 17(b).

By selecting the most expensive object, the user can see in Chiron's information window that this object is an instance of the Quadrant class and that it's total memory access cost is over 400 thousand CPU cycles.

Figure 17(b) shows the ObjectView after it has been sorted on memory location, with all instances of the Quadrant class highlighted. The highlighted objects form the tip of the

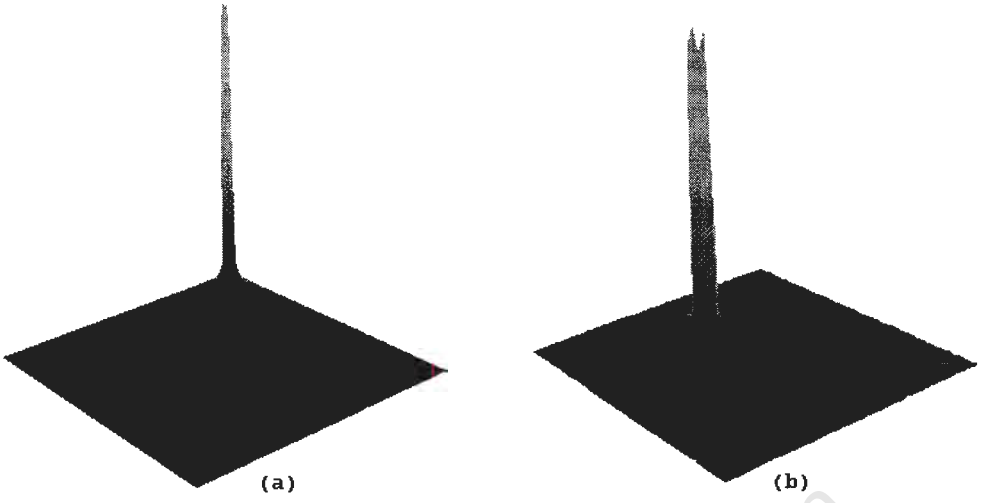

Figure 17: ObjectView of MP3D, sorted on cost and then on memory address.

'sword', which shows that these objects have a much higher memory cost than that of the other objects.

The picture also shows that the Quadrant objects are located close together in memory space so it is possible that false sharing is the cause of the high cost. To determine whether this is the case, it is necessary to look at Chiron's BlockViews.

Table 2 shows the total and average memory access cost for all of MP3D's classes. This statistical data is shown in Chiron's information window if the user selects the classes in Chiron's class window. This data shows that on average each instance of Quadrant costs more than 230 thousand CPU cycles while most other objects cost less than 5000 cycles. This highlights the fact that the Quadrant objects are by far the most expensive objects.

Object	Instances	Cycles	Cycles/Inst
BoundaryCellWall	1936	2755851	1423
BoundaryCellEntrance	336	453906	1350
BoundaryCellExit	336	435912	1297
ReservoirCell	1	21922	21922
Cell	752	1251271	1663
MasterQuadrant	1	4678	4678
Quadrant	7	1612582	230368
Particle	1000	919200	919

Table 2: Object cost statistics for MP3D.

In addition, the total cost for the Quadrant class is the second highest amongst all the classes even though there are only seven instances of it. Improving the performance of the Quadrant class should therefore improve the overall performance of the program.

6.2.2 The temporal views of the original MP3D

Figure 18 shows the initial view of the L1 and L2 BlockView, with the L2 BlockView on top. The most expensive cache blocks at the top of the two views are clearly visible as high concentrations of points which melt together to form solid lines in some areas.

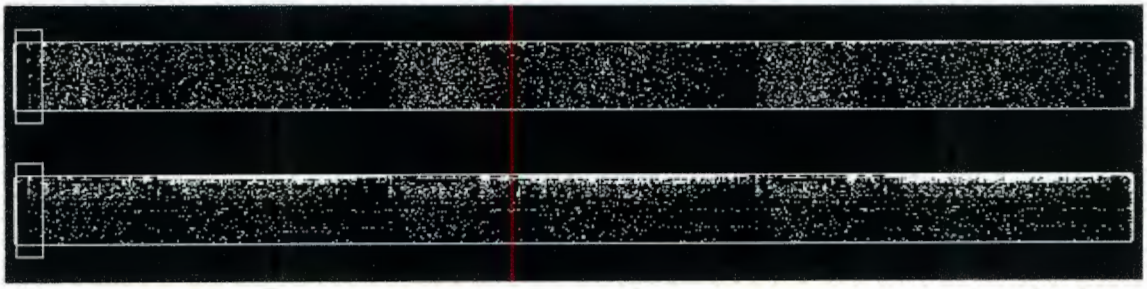

Figure 18: L1 and L2 BlockView of original MP3D.

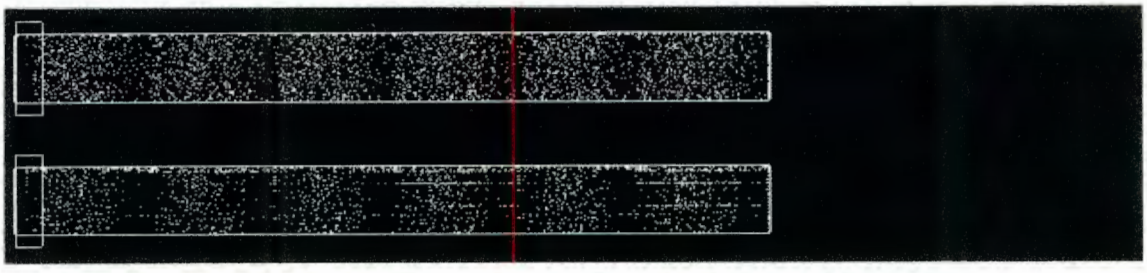

Figure 19: L1 and L2 BlockView of improved MP3D.

Six regions are visible as clusters of points in the L2 BlockView. These regions represent the two processes (move and then collide) of each of the three timesteps. These six regions are also visible in the L1 BlockView. The BlockView also shows that the three timesteps take almost equally long to execute, which tells the performance analyst that the load balance has not changed much over these timesteps.

By toggling between the display of coherence blocks and the display of replacement blocks, the user can see that the number of blocks which were evicted due to replacement misses is roughly equal to the number of blocks which were evicted due to coherence misses. However, coherence interference is causing definite hotspots of activity in the more expensive cache blocks, while replacement interference is causing a more uniform movement in all the cache blocks.

By zooming into an area of high activity on the L1 BlockView and selecting some of the most expensive cache blocks, the user can immediately see that the four most expensive cache blocks are occupied by the Quadrant objects.

The first assumption might be that false sharing is occurring in the L1 cache. However,

this is not the case, as each processor accesses a different address to any other processor inside the same L1 cache block. In addition, the Quadrant objects have been specifically aligned to the start of L1 cache blocks so that false sharing would not occur in the L1 cache.

A closer look at the L1 BlockView shows that the most expensive cache blocks do not move across CPU's — that is from the cache of one CPU to the cache of another CPU. Instead, just after a cache block is moved out of a processor's cache another cache block (with a different address) is moved into the cache of another processor. This is an indication of false sharing occurring in the L2 cache.

This can be confirmed by looking at the same region in the L2 BlockView with the lens. The most expensive cache block here contains the four most expensive cache blocks in the L1 BlockView. Seeing that the L1 blocksize is 64 bytes and the L2 blocksize is 256 bytes, four consecutive L1 blocks are mapped to one L2 block.

False sharing is occurring in the L2 cache because the same block is bouncing between the two L2 caches and different memory locations are being accessed by the two caches.

After selecting a few L2 blocks and seeing which objects are being accessed it becomes clear what is happening in MP3D. There are eight Quadrants, each of which is assigned to one processor. The first Quadrant, which is called MasterQuadrant, is assigned to processor 0. The remaining seven Quadrants are assigned to the remaining processors.

When the Quadrants are created each one is aligned to an L1 block boundary and by coincidence the Quadrant for processor 1 is also aligned to the L2 block boundary. Because each Quadrant fits inside one L1 block, the first four Quadrants fit in the same L2 block.

Figure 20 shows how processors 1 to 4 access the same L2 block every time they access their Quadrant. But processors 0 to 3 are connected to the first L2 cache and processors 4 to 7 are connected to the second L2 cache. Even though processor 4 is accessing a different L1 block it accesses the same L2 block as processors 1 to 3, but in a different cache. This causes the false sharing and the resulting coherence interference.

6.2.3 The global views of the improved version

The problem is easy to fix. The second Quadrant (which is assigned to processor 1) is aligned to an L2 block boundary and then shifted up by one cache block so that it is placed into the second L1 block of that L2 block. Then the Quadrants for processors 2 and 3 fall into the same L2 block but the Quadrants for processors 4 to 7 fall into the next L2 block. Now different L2 cache blocks are accessed in the separate L2 caches (as shown in Figure 21) and no more false sharing occurs.

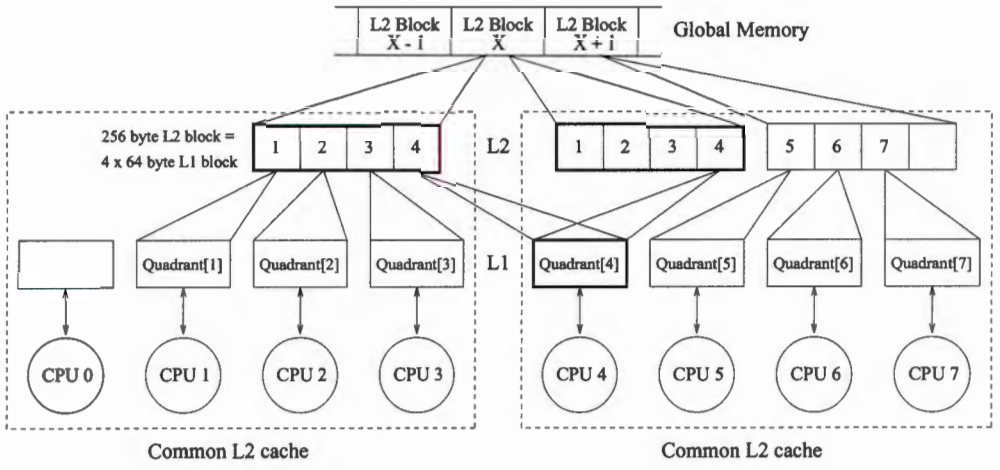

Figure 20: Bad allocation of Quadrants causes false sharing in the L2 cache.

After this change MP3D was executed with the same parameters and the same configuration for the simulator. This run took 1954965 cycles to complete, which is a 22% speed improvement over the initial version.

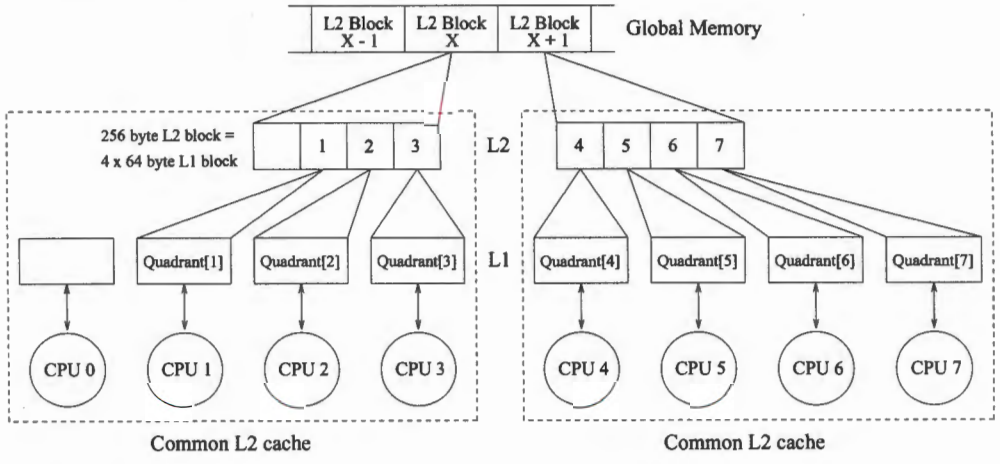

Figure 21: New allocation of Quadrants prevents false sharing in the L2 cache.

The statistics from the ObjectView of this execution are shown in table 3. Comparing this table with the previous table shows that the number of cycles per instance for all objects has remained almost unchanged. However, a very significant reduction from 230368 cycles per instance to 2977 cycles per instance has been achieved for the Quadrant particles. The most expensive object in the ObjectView is now an instance of BoundaryCellWall which has a memory cost of just over 31 thousand cycles.

The fact that the most expensive object has a much lower cost can also be seen from the texture of the ObjectView. Figure 22(a) shows the ObjectView from the first version of MP3D, after it was sorted on memory address. The surface is very smooth and hardly any perturbations on the surface are visible. The most striking feature is the 'sword' which

Object	Instances	Cycles	Cycles/Inst
BoundaryCellWall	1936	2306571	1191
BoundaryCellEntrance	336	360062	1071
BoundaryCellExit	336	367538	1093
ReservoirCell	1	14546	14546
Cell	752	106291	1413
MasterQuadrant	1	4286	4286
Quadrant	7	20845	2977
Particle	1000	771596	771

Table 3: Object cost statistics for improved MP3D.

is formed by the Quadrant objects. The smoothness of the graph can be ascribed to the large difference in cost between the most expensive object and the least expensive object.

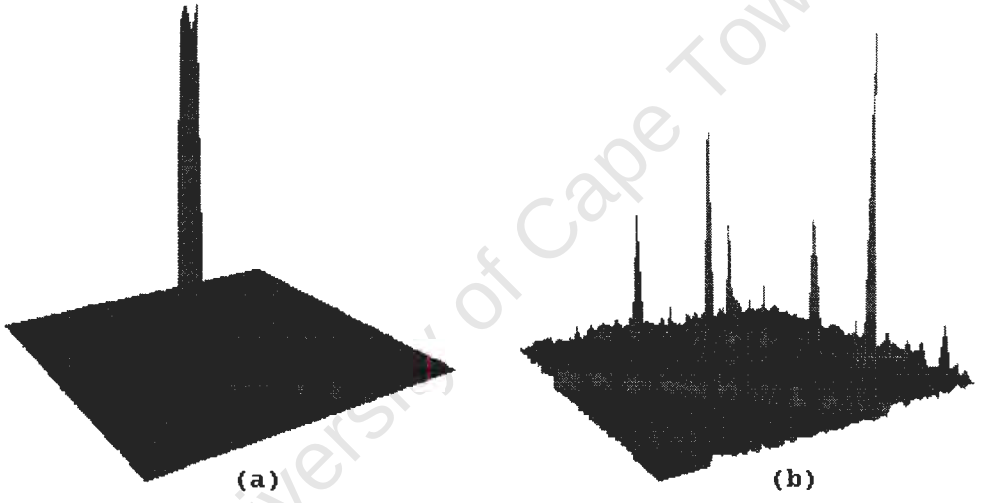

Figure 22: Comparison of ObjectViews from un-optimized MP3D and optimized MP3D.

The ObjectView, sorted on address, from the optimized version of MP3D is shown in Figure 22(b). The 'sword' has disappeared and the perturbations in the surface are more visible because the difference between the most expensive object and the least expensive object is not as large.

The Quadrant objects are not the most expensive objects anymore. Unfortunately, the most expensive object in this graph does not map onto any C++ object because during the tracing only the memory locations of C++ objects were written to the trace file.

Seeing that Chiron can only determine the address of this object, the user cannot relate this object back to the source. However, selecting this object causes all the nodes in the SourceView which access it, to be highlighted.

Figure 23 shows the ObjectView to SourceView correlation, after the most expensive object has been selected in the ObjectView. Two source nodes are highlighted in the SourceView,

which means that the source lines represented by these nodes access the most expensive object. By querying these two nodes the user can deduce from the source code information that the node in the ObjectView represents a global variable which is shared by all the processors and which keeps track of the total number of collisions in the system.

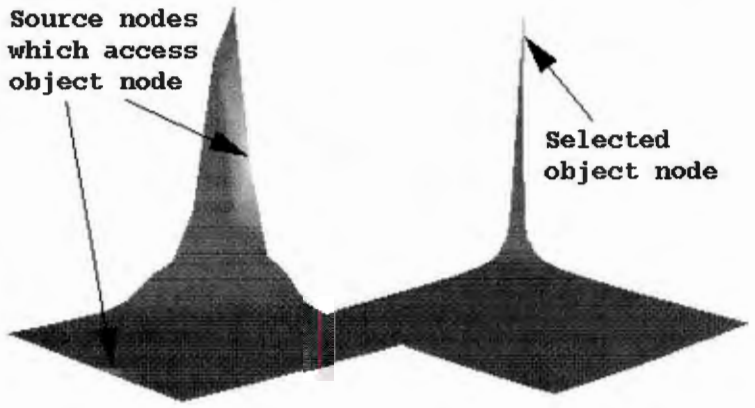

Figure 23: Correlation: ObjectView to SourceView.

Once again it is necessary to look at the BlockViews too see what is the cause of the most expensive object's cost.

6.2.4 The temporal views of the improved version

The overall view of the two BlockViews, in Figure 19, shows that the most expensive cache blocks are not as active as they were in Figure 18. It is important to note that the BlockViews in the latter figure are shorter than the BlockViews in the former figure. As the length of the BlockView indicates the execution time of the program, the version of MP3D shown in the latter figure, took less time to execute.

It is also important to note that the distribution of points is much more even in the latter figure. This implies that less processor time is being wasted by waiting for the most heavily utilized cache blocks to be brought into the cache.

Figure 24 shows the cache block in the L1 BlockView which contains the most expensive object. All instances of the cache block which were loaded due to references to the most expensive object are drawn as planes, while the other instances and other cache blocks are drawn simply as dots.

The user can deduce from this figure that false sharing in the L1 cache is not the cause of the block movement because the same address is accessed every time the particular cache block has to be loaded. A look at the L2 BlockView shows that false sharing is not occurring in the L2 cache either.

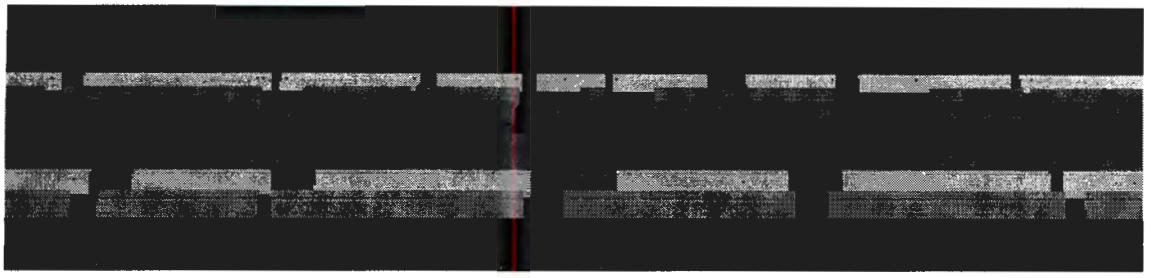

Figure 24: Loading of a cache block due to references to the most expensive object.

The high cost of the cache block is therefore caused by true sharing. This is to be expected because all eight processors need to write to the global counter. To improve the memory performance of the counter would therefore require changing MP3D so that the synchronization variable is accessed less often.

The ObjectView to L1 BlockView correlation view also shows that the synchronization variable occupies the 2nd most expensive cache and not, as expected, the most expensive cache block.

Examination of the most expensive cache block and the correlation view between the ObjectView and the BlockView shows that many cache loads for the most expensive cache block are caused by accesses to a lock variable.

The correlation view between the SourceView and the ObjectView reveals that this lock variable is accessed before a processor can print out which plane it is currently processing. To ensure that a processor is not interrupted during the printing process, a processor has to gain exclusive access to this variable.

The only means of reducing the movement of this block is to reduce the number of accesses to it. Fortunately, the information which is printed is only necessary for debugging purposes and for this reason the printing can be disabled via a compiler directive.

6.2.5 The final version of MP3D

After recompiling MP3D with the printing disabled, the above simulation was repeated and MP3D's execution time was reduced from 1954965 cycles to 1718087 cycles — an improvement of 12%. However, this improvement in performance is more likely due to a reduction in the overhead associated with the "printf" system call rather than the improvement in memory performance.

The most expensive object in the ObjectView is still the global collision counter from the previous simulation but now this variable also occupies the most expensive cache block in the L1 BlockView. There are no obvious memory system bottlenecks anymore and

any improvement in the program's performance will require more extensive changes to MP3D's algorithm. Seeing that it is difficult to verify the correctness of MP3D, further modifications of the program are beyond the scope of this thesis. MP3D's execution time was thus reduced from 2.5 million cycles to 1718087 cycles — an improvement of 30%.

6.3 Barnes-Hut

The final application which was examined with Chiron, is a hierarchical N-Body simulation from the SPLASH benchmarks [33], called Barnes-Hut. This program simulates the movement of bodies in 3-D space, under the effects of gravity. The program runs over a number of time-steps by computing the forces experienced by each body for every time-step. After computing the forces, the positions and velocities for each body are updated.

The program is written in C, and uses an octree to represent the space in which the objects lie. The root of the octree represents the cubical space that contains all the bodies. A leaf in the tree represents the cubical space containing one body, while the interior nodes which are called cells represent the cubical space containing the bodies found below this node. The depth of the tree below a node will thus depend on the density of points inside the region represented by that node.

Each time-step consists of a number of operations:

- Load bodies into the tree
- Find the center of mass of every cell
- Partition bodies among processors
- Compute the forces on all bodies
- Calculate the new position and velocity of each body
- Compute the dimensions of the root cell

The force calculation is the most expensive operation because the tree has to be traversed once for every body to calculate the total force acting on that body.

The bodies found below a node are represented as a point mass in this node so if the center of mass of a cell is far enough away from the body being examined, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell. The force that this center of mass exerts on the body is then computed.

The parallelism for each time-step is achieved by assigning a number of particles to a processor. Load balancing is obtained by keeping track of the amount of work done for each particle. This information is then used when partitioning the bodies among the processors in the following time-step.

6.3.1 The first run of Barnes Hut

The first execution of Barnes Hut was simulated on the Paradigm architecture with an L1 cache size of 64 Kbytes and a block size of 64 bytes. This cache size is similar to that of the Silicon Graphics 4D/340 multiprocessor. The L2 cache was set to a size of 512 Kbytes with a block size of 256 bytes. A total of 8 processors was used, with 4 processors sharing one L2 cache. Each processor had its own L1 cache.

In this simulation 128 bodies were moved over four timesteps. This was a very small simulation which took 2619319 simulated cycles to complete and the resulting trace file had a size of 3 Mbytes.

During the simulation Mint picked up the creation of only 6 objects. Because some of these objects are rather large (4 Kbytes) it is more sensible to run Chiron with the '-c' option so that each node on the graph represents a memory word instead of a C/C++ object. Now the objects can be viewed at a much finer level of detail, but with the class colour mapping function of Chiron the user can still easily see which C object each node belongs to.

For this particular trace, the class colouring shows that the most expensive objects are fields inside the Global structure. This structure is used to store information which is shared between the processors. The statistics in Chiron's information window also report that the Global structure is the most expensive object with a cost of 4622 cycles/instance. Because Chiron was run with the '-c' option, each instance is equivalent to one word, or 4 bytes. All other classes have a cost of less than 1000 cycles/word.

The high cost per word of the Global object makes it the prime candidate for performance improvements. By selecting the most expensive object and then looking at the nodes in the SourceView which touch this object, the user can see that the most expensive field represents a vector inside the Global structure.

To determine the cause of this field's high miss rate, the user has to look at the two BlockViews.

The four time-steps are clearly visible in the L1 cache temperature view shown in Figure 25. A large amount of cache activity can be seen at the start of each time-step while the largest portion of each time-step shows significantly less cache activity.

Figure 25: L1 and L2 BlockView of original Barnes-Hut, sorted on memory location.

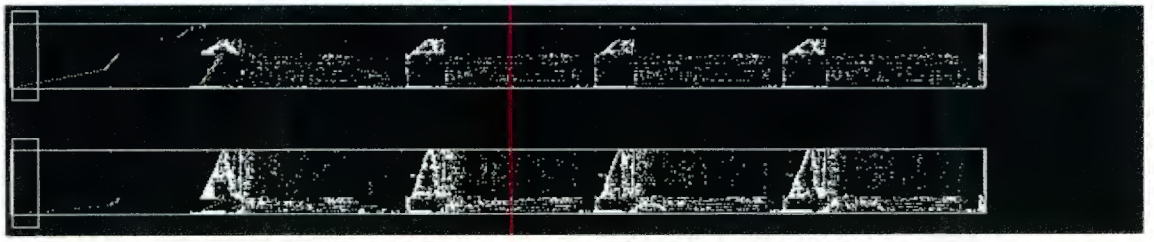

Figure 26: L1 and L2 BlockView of improved Barnes-Hut, sorted on memory location.

The correlation view with the SourceView helps the user deduce that the areas of high activity correspond to the phase which loads the bodies into the tree and partitions them amongst the processors. The areas of low activity, on the other hand, correspond to the force calculation phase. This is to be expected, seeing that the force calculation phase is the computationally intensive phase in which fewer memory references will occur.

If the display of coherence interference blocks is disabled, hardly any points are displayed in both BlockViews. This shows that coherence interference is responsible for most of the cache loads.

Because the BlockView in Figure 25 is sorted on memory address, the user can determine which cache blocks are occupied by bodies, which are occupied by cells and which are occupied by Global because the objects are located sequentially in memory.

From this the user can see that the cache blocks which contain bodies or cells are very active during the work partitioning phase, but are inactive during the force calculation phase. The blocks which contain Global, however, are more active during the force calculation phase.

The correlation view with the most expensive object shows that this object falls inside the most expensive cache block. This view also shows that false sharing in the L1 cache and in the L2 cache are causing the performance bottleneck.

As a fix, the most expensive object was padded to an L1 block boundary and the simulation was performed again. The performance improvement was minimal (less than 2%) and the cost per word of Global did not decrease significantly.

It would be possible to pad every field inside the Global structure to an L1 block boundary, but this would not necessarily prevent false sharing in the L2 cache. Padding every field inside the Global structure to an L2 cache boundary would prevent false sharing in the L2 cache but the under-utilization of cache memory would result in more forced misses and more replacement interference.

6.3.2 Reducing false sharing inside Global

Some of the slightly less expensive cache blocks which also contained Global, were then examined because these blocks showed the most activity during the force calculation phase. The correlation view between the BlockView and the SourceView showed that a block of code which computes information about load balancing is responsible for the cache activity in these blocks.

This block of code is contained inside an ifdef compiler directive because it only calculates information which is displayed to the user and is not needed for any other calculations. A quick solution to the problem was to recompile the application without the flag enabled.

After recompiling, Barnes-Hut was run with the same parameters. This time the four time-steps were completed after 2262810 simulated CPU cycles. This is a 13% improvement over the initial run.

The ObjectView of this trace did not change much from the initial version of Barnes-Hut. However, a comparison between Figure 25 from the initial version of Barnes-Hut and Figure 26 from the improved version of Barnes-Hut, shows that there is less cache activity in the L2 cache during the latter's force calculation phase. There is also less cache activity in the L1 cache during the force calculation phase.

6.3.3 The cache performance of cells and bodies

Instead of concentrating on the most expensive cache blocks other blocks were also investigated. The L1 BlockView and L2 BlockView show that false sharing is occurring in the caches during the work partitioning phase. The cache blocks which show the most activity during this period contain cells and bodies. The activity in the cache is happening because different processors work on cells which fall within the same L2 cache block.

One solution would be to pad the cells so that each cell occupies one L2 cache block. However, this would waste a large amount of memory (especially for a large number of bodies) and would increase the number of replacement interference.

Another solution would be to change the work partitioning phase so that processors only look at bodies which fall inside the same L2 cache block. However, this would degrade the load balancing and the improvement in memory performance could easily be lost to a degradation in the work allocation.

Because the program spends most of its time in the force calculation phase it is important to keep this phase as optimised as possible, even if this means a slightly higher memory performance overhead during tree building and body partitioning phase.

Chapter 7

Conclusion

Writing efficient shared-memory parallel applications is a difficult task because many problems such as false sharing, inefficient synchronization and load imbalance can reduce the application's performance. Until parallel compilers generate more efficient code, parallel programmers need tools which can help them understand and improve the performance behaviour of their parallel programs.

As the difference in speed between processors and memory increases, so the memory performance of a program becomes more and more important. Chiron is specifically designed to help programmers identify bottlenecks in the memory performance of their shared-memory parallel applications. This is achieved by displaying large amounts of code-oriented and data-oriented memory performance data, using interaction and 3D graphics techniques.

Chiron presents this data to the user in a number of views. The user interface allows the user to interact with this data and narrow in on features which are an indication of performance bottlenecks. Once the user has homed in on such a feature, the information conveyed by the 3D visualization can be augmented by numerical data which is more detailed.

The post-mortem analysis of trace files enables Chiron to visualize the performance of any shared-memory system which can produce a trace stream in the correct format. Chiron can therefore be used to visualize the performance of existing multiprocessors such as the Silicon Graphics 4D/380, provided that the necessary information can be captured in a trace file.

The size of the trace file which Chiron can process, depends on the workstation's amount of memory. The memory requirements of Chiron depend the data access pattern of the

trace file, but in general Chiron requires 0.3 Mbytes of RAM for every 1 Mbyte of trace data.

It takes Chiron roughly 40 seconds to process and display the data of a 16 Mbyte trace file on an Indigo2 workstation with an Extreme graphics card. Chiron's level-of-detail suppression enables the user to view even large trace files interactively on this type of workstation. Level-of-detail suppression allows the user to display the right amount of detail, taking the capabilities of the workstation and the ability of the user to absorb information into account.

One of Chiron's most useful views is the cache temperature view. Memory bottlenecks caused by coherence interference or replacement interference can be seen very clearly in this view as a cluster of points. Because points can be drawn efficiently on a fast workstation, the user can view the memory performance of the entire program and can then home into the area of high cache activity, using Chiron's interactive controls.

Finding the cause of the problem is simplified by the correlation views and the detailed data which can be obtained for any object in Chiron's views. For example, the ObjectView to the BlockView correlation can quickly show whether accesses to the same object are causing the cache movement of a particular block, or whether false sharing is the cause.

The global views are useful because they give an overall view of code-oriented and data-oriented information. The array addition program has shown that it is important to display both types of information, because it is possible for a source line or an object to be the cause of a performance bottleneck.

7.1 The case studies

The case studies have shown that Chiron can help even an inexperienced performance debugger detect memory performance bottlenecks quickly. These case studies, which ranged from a small application to large scientific applications, also show that Chiron is particularly effective for detecting bottlenecks caused by coherence interference.

The performance of the vector addition program was improved by 8.5% by changing the work allocation. This reduced the coherence interference in the L1 cache and the L2 cache.

The L1 and L2 coherence interference of MP3D was reduced by changing the location of objects in memory. An overall performance improvement of 30% was achieved.

The memory performance of Barnes-Hut could not be improved. Chiron did show that a memory performance bottleneck occurred at the start of each time-step but fixing this bottleneck will affect the performance of the next phase of the time-step. Another bottleneck

was found, which was caused by the collection of load balancing data for presentation to the user. This code was disabled by a compiler directive and the performance of Barnes-Hut improved by 13%.

The performance debugging of MP3D has shown that Chiron is also useful for detecting subtle cache problems such as false sharing across shared L2 caches. Shared caching can reduce false sharing in the L2 cache [11], particularly, if objects which fit into the same L2 cache block are only accessed exclusively by processors which share that cache. After MP3D was restructured to prevent the L2 false sharing which was detected with Chiron, coherence interference in the L2 cache was reduced by 75%.

7.2 Future work

While Chiron has proved to be a useful tool, it is still in the prototype phase and there is room for many improvements and additions which will speed up the process of performance debugging.

The case studies have shown that the BlockViews are not very effective for determining the cause of replacement interference. They are effective for determining the cause of coherence interference because coherence interference occurs in the same cache line. However, replacement interference occurs between different cache lines and because the BlockViews only shows a fixed number of these, it is possible that the cache blocks which are causing the replacement interference are not shown. It is therefore worthwhile to look at new views which are more effective at determining the cause of replacement interference. For example, one view could show which objects are colliding with a particular object. This view could also show which object is colliding most often with the selected object.

The graphics performance of Chiron is another problem. While the manual level of detail options provide a means of keeping the graphics performance level of Chiron within acceptable limits, future versions should support automatic level-of-detail suppression. Depending on the distance between the viewer's camera and an object, the latter could be drawn as points, then lines and finally as cubes. Small blocks which are close together could be represented by one single block until the camera has moved in close enough for the individual blocks to become visible. A mechanism for implementing this is described in [26].

Another option would be to display a dynamic number of the most expensive cache lines instead of the currently fixed number of 100. If there is a great deal of movement in the more expensive cache lines only the 50 most expensive cache blocks would be drawn. If

there is less movement in the cache blocks the most expensive 200 blocks, say, could be drawn.

The cache occupancy view currently does not show when the status of a cache block changes after it has been moved into a processor's cache. Very often the user would like to see when the cache is upgraded or downgraded as this will help detect false sharing. Chiron could show the state of a cache block by changing the colour of its cube when the state changes. The user should then also be able to see which memory location was accessed when the state of the cache block changed by selecting the correct part of the cube.

Chiron can be extended to show the performance problems caused by load imbalance and inefficient synchronization. Seeing that the BlockView has proven to be be very effective, Chiron has been extended to visualize load balancing and synchronization of a program in two separate views which are similar to the L1 and L2 BlockView. This forms part of the work of another thesis [19].

Appendix A

Trace Messages

This appendix lists the messages which Elf writes to the trace file after it receives a similar message from the Paradigm or the Mint simulator system. The format of a message is generally:

```
MessageId MemoryPtr ProcessorId Source Time
```

MessageId is an alphabetic character identifying the message type. **MemoryPtr** is the address which was accessed when the event happened. **ProcessorId** is the identifier of the processor which generated the event; **Source** refers to a line number in the source file where the memory reference occurred. **Time** is the absolute time when the event occurred, in terms of simulator timesteps from the start of the program. Some messages contain more fields of information, as can be seen in the following list of messages. The character in brackets shows which character in the trace file identifies the given message.

- **Create Object (c)**: Every time shared-memory is allocated for a variable, Elf writes a message to the trace file specifying the name of the variable, its address, the amount of space that has been allocated and the id of the processor. This message is identified by a 'c' in the trace file.
- **Delete Object (d)**: Whenever an object is deleted, Elf writes a message to the trace file specifying which memory area was read and which processor did the delete. Chiron uses this message and the above message to keep track of the location of objects in memory at a particular instant in time so that memory locations can be mapped onto objects. This message is identified by a 'd'.
- **Load Shared Block (g)**: This event occurs when a block is moved into a cache and the processor which requested the block has shared access to the block. The message

for this event specifies the address of the block which was moved into the cache, the cache which generated the event, the cache level, the source line and the time when the event occurred. This information is used by the BlockView to determine when a particular block was moved into a cache.

- **Load Private Block (h)**: This event generates the same message as the above message. In this case the block which is moved into the cache will be owned exclusively by the processor which requested the block. Chiron does not distinguish between this message and the above message.

- **Clean Replace (b)**: If a block has to be moved into the cache but the set onto which that block maps is full, another block will have to be evicted before the required block can be moved into the cache. When this event occurs a message is written to the trace file specifying the address which was accessed, the processor id, the cache level, the source line, the time at which the event occurred and the address which was replaced. This information is used by the BlockView to show the user the memory location for the reference which caused a particular block to be replaced.

- **Dirty Replace (a)**: This message is identical to the above message except that it indicates the replacement of a block which was dirty. This means that the block which is replaced will have to be flushed before the new block can be brought in. This message contains the same information as the above message and Chiron currently does not distinguish between the two types of messages.

- **Clean Invalidate (f)**: This event occurs when a cache block is invalidated because another processor has requested an exclusive copy of the same block. The message in the trace file specifies the address of the block which is moved out, the processor which it is moved out by, the cache level, the source line and the time at which the event occurred. This information is used by the BlockView to determine when a particular cache block is moved out of a processors cache.

- **Dirty Invalidate (e)**: This event occurs when a block which is dirty is invalidated. This message is the same as the above message and Chiron currently does not distinguish between the two messages.

- **Cache Protect (p)**: If a processor has a copy of a block in its cache but its wants to write to that block, it first has to have write ownership of the block. If this is not the case the processor has to request ownership of the block. This entitles notifying the other processors of the upgrade and waiting until they have flushed their copies of the block. This message is not used by Chiron but it could be used to indicate a change of status of a cache block in the BlockViews.

- **Complete Read (R)**: This message is sent to Elf on the completion of a read operation. The information in the message includes the memory location which was read, the processor id, the source line and the time in CPU cycles that the processor had to wait for the read operation to complete. This information is used by the SourceView to calculate the total cost in CPU cycles for each source line and by the ObjectView to calculate the total cost of each object. The BlockView also uses this information to calculate the total cost of each cache block that was accessed.

- **Complete Write (W)**: This message is sent to Elf on the completion of a write operation and its information is identical to the above message. The information is also used by the BlockView, ObjectView and the SourceView.

- **Read Cacheblock (r)**: When a read operation is initiated, the address of the variable being read and the processor id are written to the trace file. This message is not used by Chiron, but it was useful for debugging errors in the tracing mechanism, because this message is generated by Mint and not by the memory simulator. This message is identified by a 'r'.

- **Write Cacheblock (w)**: This message is identified by a 'w' and is identical to the above message except that it signifies the start of a write operation by a particular processor to a location in memory. This message is also not used by Chiron.

- **Cache Miss (m)**: Every time a read or write operation results in a miss in the cache, this message is written to the trace file. The message is identified by a 'm' in the trace file and specifies the memory location which was referenced, which processor issued the reference, on which cache level the miss occurred and on which source line the reference happened. This information is not used by Chiron but it can be used by a system which uses the number of cache misses as a metric.

Appendix B

Colour Prints

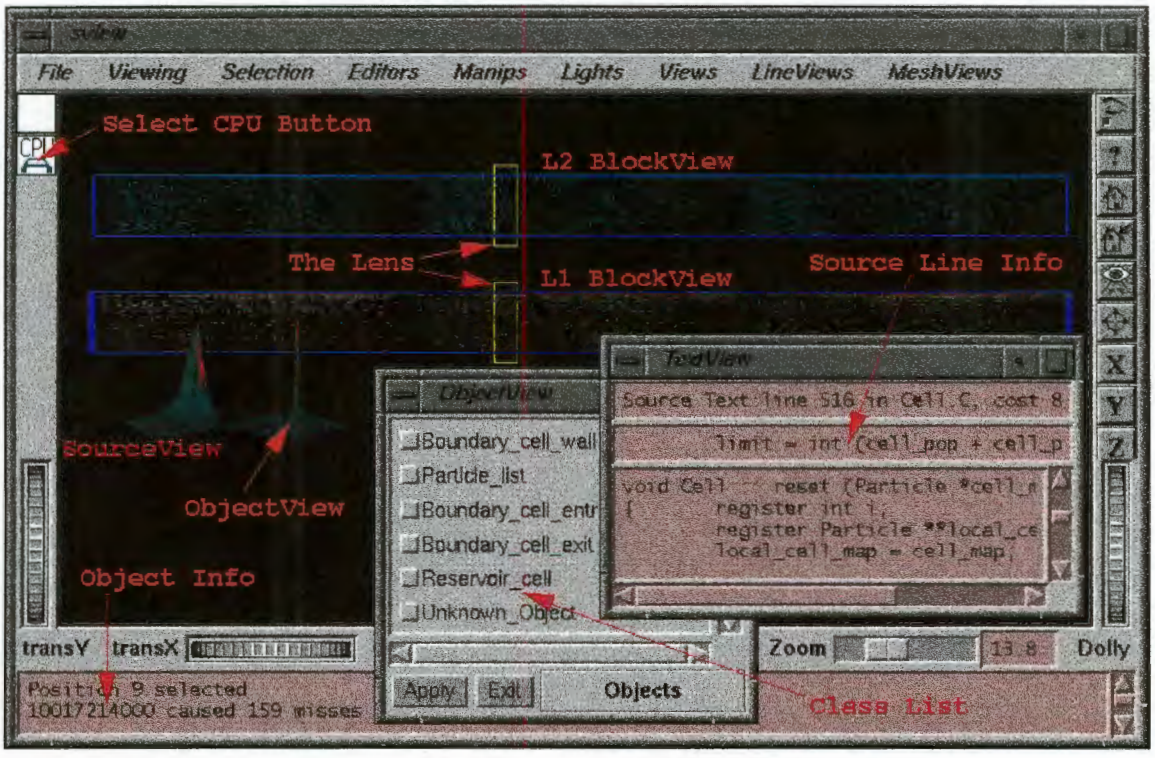

Figure 27: Chiron's user interface.

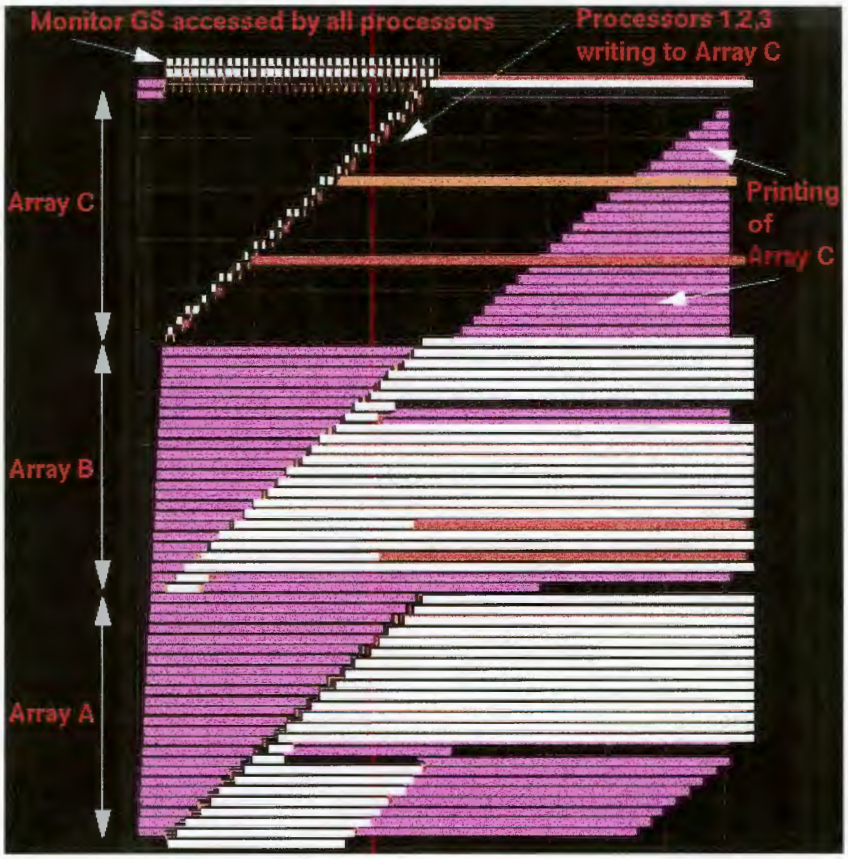

Figure 28: L1 BlockView of un-optimized array addition.

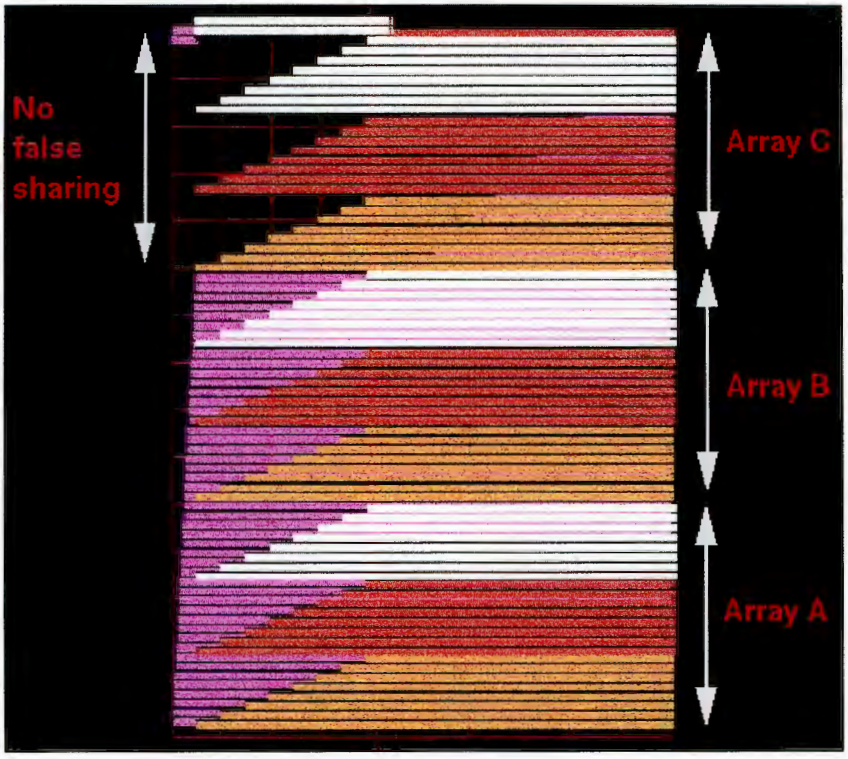

Figure 29: L1 BlockView of optimized array addition.

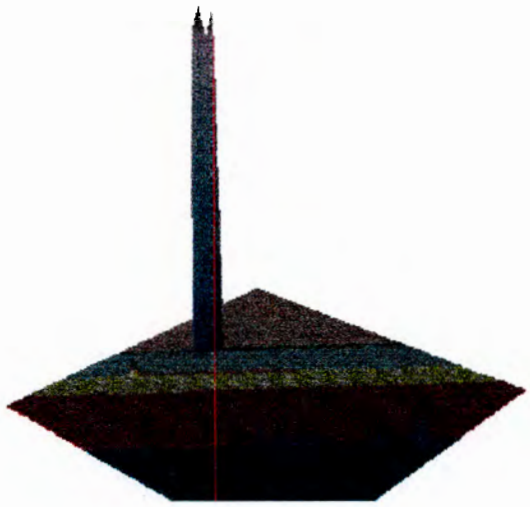

Figure 30: Class information mapped onto the ObjectView.

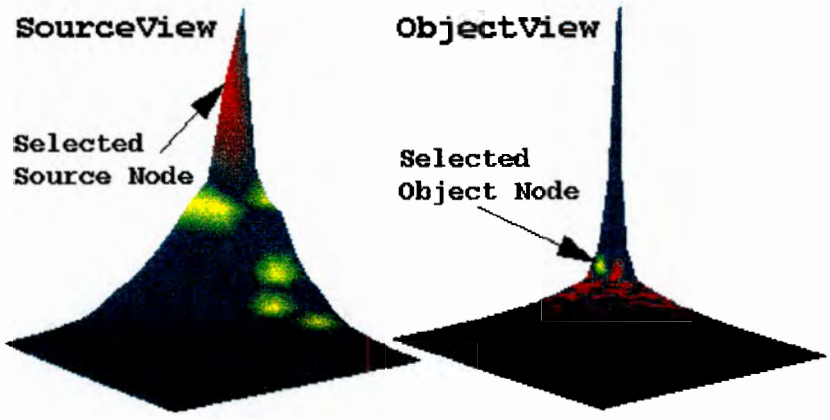

Figure 31: Correlation between the SourceView and the ObjectView.

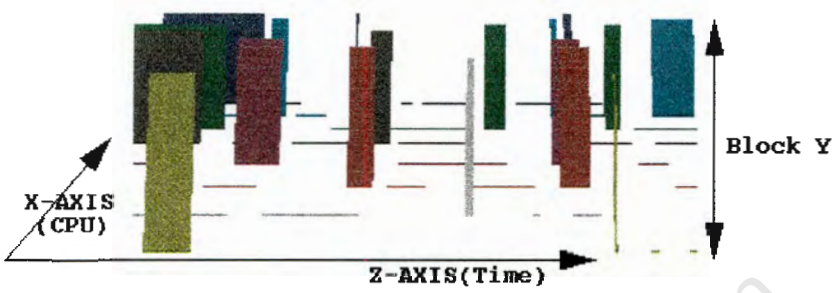

Figure 32: Cache blocks which correlate with an object are drawn as faces.

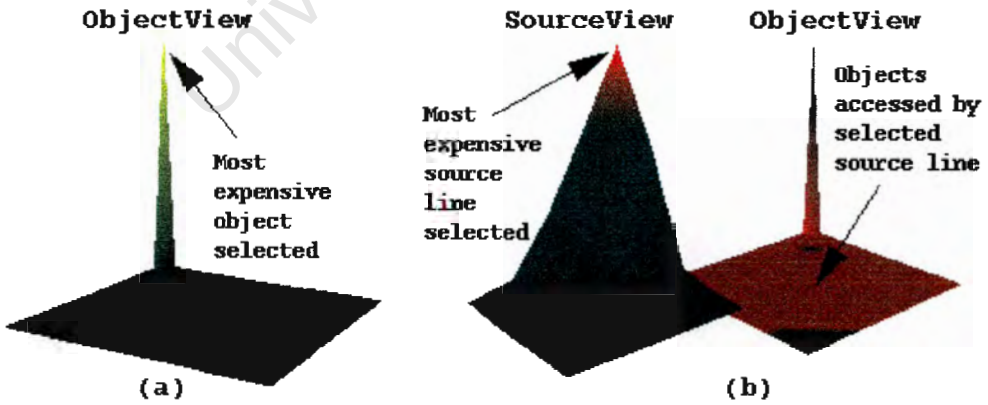

Figure 33: Sequence of selecting most expensive object, source node.

Bibliography

[1] R. Bianchini, M.E. Crovella, L. Kontothanassis, and T.J. LeBlanc. Memory contention in scalable cache-coherent multiprocessors. Technical Report 448, Computer Science Department, University of Rochester, April 1993.

[2] W. Blume and R. Eigenbaum. Performance analysis of parallelizing compilers on the perfect benchmarks program. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[3] W.J. Bolosky and M.J. Scott. False sharing and its effect on shared memory performance. Technical report, Computer Science Department, University of Rochester.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Analyzing and visualizing performance of memory hierarchies. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, chapter 1, pages 1–25. Addison Wesley, May 1989.

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

[7] J. Dongarra, D. Sorensen, and O. Brewer. Tools to aid in the design, implementation, and understanding of algorithms for parallel processors. In R.H. Perrott, editor, *Software for Parallel Computers*, pages 195–219. Chapman and Hall, 1992.

[8] S. Feiner and C. Beshers. Worlds within worlds. In *INTERCHI '93*, pages 76–83. ACM, Amsterdam, The Netherlands, April 1993.

[9] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice.* Addison Wesley Publishing Company, Inc., 1990.

[10] A.J. Goldberg and J.L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. Technical report, Computer Systems Laboratory, Stanford University, December 1991.

[11] H.A. Goosen and D.R. Cheriton. The performance of shared multilevel caches. Technical report, Computer Science Department, Stanford University, May 1993.

[12] H.A. Goosen, A.R. Karlin, D.R. Cheriton, and D.W. Polzin. Chiron parallel program performance visualization system. *Computer-Aided Design*, 26(12), December 1994.

[13] E.H. Gornish, E.D. Granston, and A.V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368. ACM SIGARCH, June 1990.

[14] A. Gupta, M. Martonosi, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of Sigmetrics '92*, pages 1–12. ACM, June 1992.

[15] M.T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[17] J.L. Hennessy and N.P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[18] W.L. Hibbard, B.E. Paul, D.A. Santek, C.R. Dyer, A.L. Battaiola, and M.V. Martinez. Interactive visualization of earth and space science computations. *IEEE Computer*, pages 65–72, July 1994.

[19] P. Hinz. Visualizing the performance of parallel programs. Master's thesis, University of Cape Town, 1996.

[20] D.E. Hudak and S.G. Abraham. Compile-time optimization of near neighbour communication for scalable shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 15(4), August 1992.

[21] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM, April 1991.

[22] P. Machanik. 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.

[23] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, pages 206–217, April 1990.

[24] M. A. Najork and M. H. Brown. Obliq-3d: A high-level, fast-turnaround 3d animation system. *IEEE Transactions on Visualization and Computer Graphics*, pages 175–193, June 1995.

[25] C.M. Pancake and S. Utter. Debugger visualizations for shared-memory multiprocessors. In M. Durand and F.E. Dabaghi, editors, *High Performance Computing II*, pages 145–158. Elsevier Science Publishers, 1991.

[26] W. Paverd. Information visualization. Master's thesis, University of Cape Town, 1996.

[27] S. P. Reiss. 3d visualization of program information. *SIGCHI '94 Software Visualization Workshop Proceedings*, April 1994.

[28] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. Glyphmaker: Creating customized visualizations of complex data. *IEEE Computer*, pages 57–64, July 1994.

[29] G. Roman and K.C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, pages 11–24, December 1993.

[30] D.T. Rover, G.M. Prabhu, and C.T. Wright. Visualization of program performance on concurrent computers. In *Computing in the 90's, The first Great Lakes Computer Science Conference*, pages 154–160. Kalamazoo, MI, USA, October 1989.

[31] 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.

[32] Silicon Graphics. *IRIS Inventor Programming Guide Volume I: Using the Toolkit*, 1992.

[33] J.P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[34] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. Data visualization and performance analysis in the prism programming environment. In N. Topham, R. Ibett, and T. Bemmerl, editors, *Programming Environments for Parallel Computing*, pages 37–52. Elsevier Science Publishers, April 1992.

[35] A. J. Smith. Cache memories. *Comput. Surv.*, pages 473–530, September 1982.

[36] E. Speight and J.K. Bennett. Paraview: Performance debugging of shared-memory parallel programs. Technical Report ELEC TR 9403, Rice University, March 1994.

[37] J.T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, pages 27–39, September 1990.

[38] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 1983.

[39] E. R. Tufte. *Envisioning Information.* Graphics Press, 1990.

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