ECLIPSE Performance Analyzer

A Thesis
submitted in partial fulfillment
of the requirements for the degree
of
Master of Technology
in
VLSI Design Tools & Technology
by
Anup Gangwar

Under the guidance
of

Prof. Anshul Kumar **Prof. M. Balakrishnan**
Department of Computer Science and Engineering
Indian Institute of Technology Delhi, India
and
Dr. Ir. Jos van Eijndhoven
Information and Software Technology Sector
Philips Research Laboratories Eindhoven, The Netherlands

Department of Computer Science and Engineering
Indian Institute of Technology Delhi
December 2000

Abstract

ECLIPSE is a heterogeneous multiprocessor programmable architecture. In this report a method to measure the performance of such a system on a trace based approach is presented. A set of metrics is chosen and their derivation from ECLIPSE's accumulated hardware fields is explained. A set of performance views is described and a generic Performance Visualization tool which gives a graphical view of extracted metrics is presented. The separation of metric relationships from the visualizer is also explained.

Contents

List of Figures

Chapter 1

Introduction

1.1 Need for performance measurement

Qualitatively performance of a system [1] characterizes how well the system is able to perform a job. Quantitatively performance of a system is defined on a set of metrics whose values indicate the system performance. Performance measurement is necessary due to a number of reasons. Primarily it allows system evaluation which is also useful for system modeling. Performance measurement done during the simulator stage allows for early system analysis and proper design space exploration. It also allows for the proper selection of application and characterization of the workload. Finally performance measurement of a system allows benchmarking of a system against other known systems.

1.2 How to do performance measurement

Performance measurement of a system can be done at two levels.

1. The simulator level.

2. The *real system* level.

[1] Throughout this document system refers to a heterogeneous multiprocessor programmable system

Developing a methodology at system level has the advantage that it works both with the simulator as well as the *real system*. In both these levels the performance measurement may itself be performed in either,

- the application elements.
- or the architecture elements.

In case performance measurement is done in the application elements then, either the application software themselves have hooks to extract performance metrics or there is a separate thread of execution for performance measurement. This extraction of performance metrics certainly modifies the system behavior, so, now the performance metrics are being extracted for a system which is different and may lead erroneous results.

If measurement units are hardware elements, then application elements just have to *pull out* the measured values from the hardware elements. This does not modify the system behavior to a large extent as is the case when measurement is done in application elements. Secondly, if these same measurement units are present in the *real system*, then performance measurement works at both the levels. The biggest disadvantage of this latter technique is that application level metrics may not be appropriate. So, for example it is difficult to figure out with this technique exactly at what point in execution the current application is.

One last issue is whether the simulator itself has capability to represent metrics in some meaningful form or it generates a *trace* of the simulation which is in turn viewed using a viewer. Having a separate viewer is desirable as it separates the actual simulation process from the viewing process. Also it is possible to use different viewers on the same trace data.

ECLIPSE is a heterogeneous multi-processor media processing architecture of Philips. In this report the performance measurement system for this architecture is described. The measurements units are hardware elements and the measurement process has been implemented in the simulator. The visualization process is carried out using a separate tool.

1.3 Organization of this report

Chapter 2 deals with the ECLIPSE architecture and simulator. This chapter may be skipped by any one having some idea of the ECLIPSE architecture. Chapter 3 describes the metrics which have been extracted from the ECLIPSE simulator as well as discusses some general issues involved in performance evaluation of heterogeneous multiprocessor architectures. Most of the material in this chapter is taken from [2]. Chapter 4 discusses the general concepts in visualization as well as defines a set of views which have been used to study the ECLIPSE architecture. Chapter 5 describes implementation details of the complete performance evaluation system. Finally chapter 6 describes the simulation runs and results obtained for a MPEG decoding application, setup using the coprocessors available at the time of writing this report. A configuration file for the performance visualizer or *Telescope* is found in appendix along with instructions on running the visualizer.

Chapter 2

Architecture Application and Simulation

In this chapter a brief introduction of the ECLIPSE architecture is presented ECLIPSE hardware components contain a number of registers to hold task and some other information. This information is organized in the form of tables. Some fields of these tables are present for measuring performance of the particular hardware component. These fields which are described here are used in chapter 5 to derive a subset of high level metrics as explained in chapter 3. The set of performance metrics chosen is a subset of metrics as proposed in [2].

2.1 The Architecture

ECLIPSE is a Philips Research project aiming towards a heterogeneous media multi-processor, in which a TriMedia VLIW CPU cooperates with more dedicated coprocessors in a flexible way in order to keep the media data on-chip as much as possible. Coprocessors may be highly parameterized or even fully programmable. Use of coprocessors partially reduces the bandwidth requirements to external memory. Secondly, these coprocessors should off-load the CPU from media processing tasks that can be performed more efficiently in application domain specific coprocessors, while maintaining the flexibility of a programmable architecture. The ECLIPSE architecture is targeted for irregular media processing.

Each of the ECLIPSE hardware components contain certain performance measurement units. These performance measurement units are responsible for sampling some fields of the tables containing task and other information. These tables are stored in the registers contained in the component. Some fields of these tables are present to hold accumulated values for sampled fields from the tables. [1]

2.2 The Application

Application is modeled using YAPI (refer [6]), where the granularity of YAPI nodes exactly matches the granularity of the coprocessors. YAPI is an application programmer's interface which can be used to model signal processing applications as process networks. The communication between applications is based on the Kahn model with blocking reads on theoretically unbounded FIFOs. In practice, the FIFO's are bounded such that writes can also suspend the process execution. The purpose of YAPI is to provide a means to construct models of signal processing applications with a platform independent programmer's interface. As such, YAPI enables the reuse of application code across different projects. Furthermore, the application code may be used as input for performance analysis, hardware-software codesign, and mapping onto various platforms.

Typical signal processing architectures are parallel architectures because they have to process enormous amounts of data at high rates, especially in the video domain. These throughput requirements significantly constrain the design space of video systems. In order to explore this design space more effectively, we explicitly describe the potential parallelism in the functional specification by modeling video applications as networks of parallel processes. YAPI is a C++ library with a set of rules which can be used to model signal processing applications as process network. An example process network is

[1] It is not possible to provide more information about the architecture and performance measurement units as that piece of information is Philips Company Restricted.

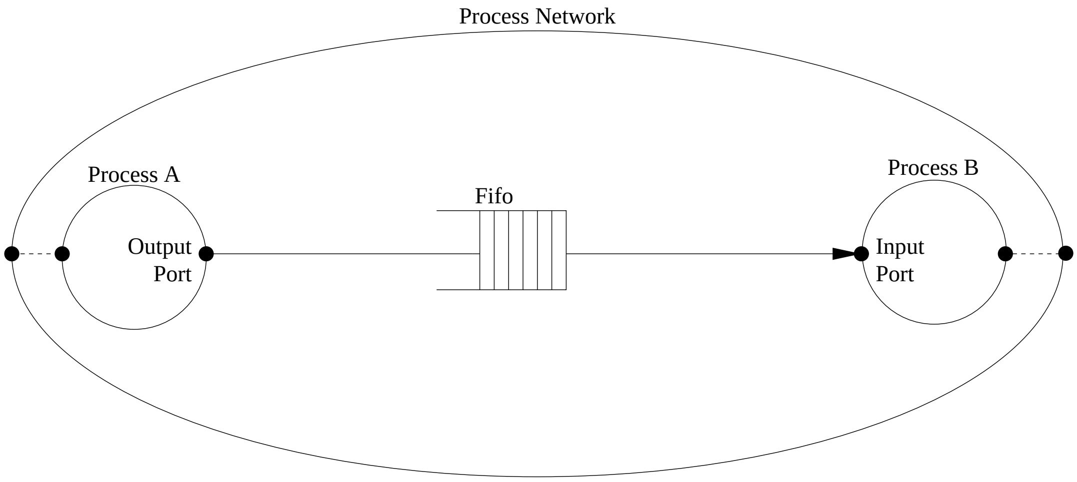


Figure 2.1: A Process Network

shown in figure 2.1.

Process networks consist of sequential processes that are interconnected by directed first-in-first-out channels which are called FIFOs. Processes have a private state space which is not accessible by other processes. Furthermore, they have input and output ports through which they communicate with their environment. Ports are the formal arguments of processes. FIFOs connect output ports to input ports. They are the actual arguments of processes. Each port is connected to precisely one FIFO. Processes can read from their input ports and they can write to their output ports. The read method blocks if the corresponding FIFO is empty. The write method blocks if the corresponding FIFO is full. This avoids data loss during communication. The ports and FIFOs are typed which means that they can handle data of one type only. The structure of process networks, i.e., the binding of FIFOs to ports, is static.

2.3 The Simulation

The simulator is able to simulate a number of ECLIPSE like architectures. Which particular architecture to instantiate and simulate is definable using a setup file. The parameters which may be varied for various architectures include the number of coprocessors, the size of registers etc. Application is mapped onto this architecture using an application setup file. The application setup file specifies the tasks along with the task ports which are mapped onto the various coprocessors. This file also contains some parameters for performance measurement, like the trace file name, number of measurements to perform etc.

Figure 2.2 shows the top level break up of simulator classes. This tree shows the organization of units in simulator. In this figure *transport* refers to the data bus and *CPU* is the TriMedia CPU. Different coporcessors may be required in different system simulations. *Clients* refers to the coporcessors which are currently taking part in simulation along with their shells.

In order to completely abstract the functionality of each of the individual units from communication a set of five public access functions is provided. For accessing any unit the help of TokenPair is taken. A TokenPair is a class which maintains a key/value pair. The key in this case contains the id of a particular unit. A few examples of id's are Input.Shell.Ss.StreamTable.TAS_ID, Input.Shell.MeasureFreq etc. More information on this can be found in [8].

These five functions and their functionality is as described below:

- *Component()*–This is the constructor for the Component class.
- *Setup()*–This function sets up the tables and other parameters in an architecture element. The parameter values come from the arch setup file.
- *MicroscopeRead()*–This function fully abstracts a components implementation from it's interface. It is possible to access the full hardware of a component using this function. So, for example it is possible to access the stream synchronizer stream table of Input coprocessor by constructing a TokenPair with the appropriate ids.
- *MicroscopeWrite()*–This function is similar to the previous one, except that it provides a write functionality.
- *Run()*–This function is called by the Task Scheduler for executing the functionality of a component.

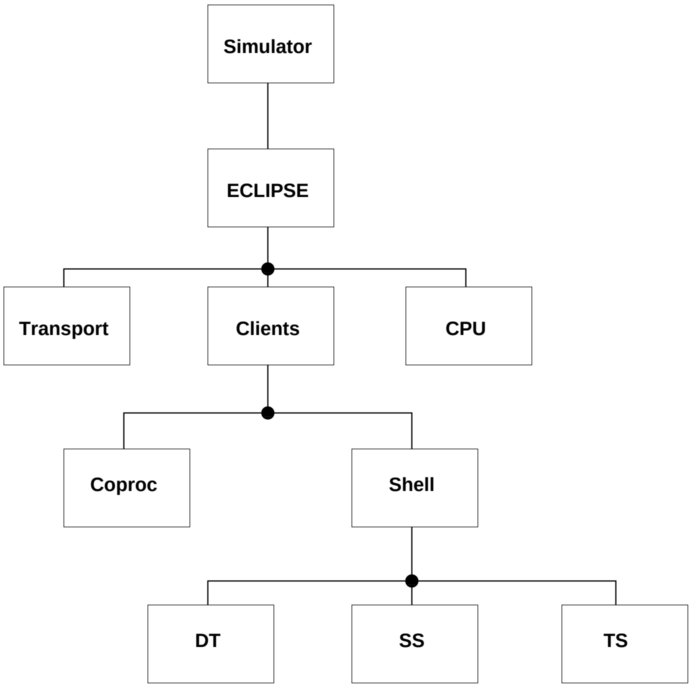


Figure 2.2: ECLIPSE Simulator

An example arch setup file is shown in figure 2.3. The first line tells the simulator to create an instance of Input coprocessor. The other information present in this file is, maximum number of tasks on this coprocessor is equal to 2, maximum number of task ports is 2 and the DTR has only one port.

An example application setup file is shown in figure 2.4. This description fills up the stream table of Input shell. It also tells where the initial buffer space is. A detailed discussion of these fields is present in [8].

```
Create Input

Input{
        NTasks: 2
        Shell{
                NStreams: 2
                 Dtr.NPorts: 1
        }
}
```

Figure 2.3: Example of Arch Setup file

```
Input.Shell{

        Ss.StreamTable{
                TASK_ID: 1
                BUF_SPACE: 0x100
        }
}
```

Figure 2.4: Example of Appl Setup file

Chapter 3

Performance Metrics and Collection

In this chapter a set of high level metrics are described to measure the performance of a multiprocessor architecture. A trace file format is suggested to collect metric data generated from the simulator. The metrics and trace file format are taken from reference [2]. This trace file is picked up by the visualizer described in chapter 4 to generate a set of views which are easily perceptible. The metric relations described here are utilized both by the simulator hooks as well as the visualizer. However, in order to separate the trace data from these metric relations, they are not put in the trace file.

3.1 Metrics and Definitions

3.1.1 Processor Metrics

- **Execute :** Fraction of the samples taken for which the processor is executing a task, i.e., the processor is performing computation.

- **Read :** Fraction of the samples taken for which there is an active input port, i.e., the processor is busy performing input.

- **Write :** Fraction of the samples taken for which there is an active output port, i.e., the processor is busy performing output.

- **Blocked :** Fraction of the samples taken for which all input and output ports are blocked, i.e., all tasks that run on the processor are blocked on I/O.

- **Idle :** Fraction of the samples taken for which there is no work to be done for this processor. This can only occur at run-out: all tasks are ready.

3.1.2 Task Metrics

- **Execute :** Fraction of the samples taken for which the process is executing on the processor.

- **Read :** Fraction of the samples taken for which the process is reading from an input port. The input port is active.

- **Write :** Fraction of the samples taken for which the process is writing to an output port. The output port is active.

- **Blocked :** Fraction of the samples taken for which the process is blocked on an input or output port.

- **Suspend :** Fraction of the samples taken for which the process is suspended, i.e., another process is running on the processor.

3.1.3 Processor Port Metrics

- **Active :** Fraction of the samples taken for which there is a pending read or write command for the port, data or room is available, the command is scheduled, but not all data has been transferred yet.

- **Idle :** Fraction of the samples taken for which there are no pending read or write commands for the port, i.e., none of the tasks that are running on the processor has issued a command for the port.

3.1.4 Task Port Metrics

- **Active :** Fraction of the samples taken for which there is a pending read or write command for the port, data or room is available, the command is scheduled, but not all data has been transferred yet.
- **Blocked :** Fraction of the samples taken for which there is a pending read or write command for the port, but no data is available yet in case of an input port, or no room is available in case of an output port.
- **Idle :** Fraction of the samples taken for which there are no pending read or write commands for the port, i.e., the task to which this task port belongs has not issued a command for this port.

3.1.5 ECLIPSE Metric Relations

Based on the metric definitions of previous sections a set of metrics has been chosen, specifying mathematical relations between and abstractions of the performance numbers presented in the earlier paragraphs. These metrics are then partially extracted from ECLIPSE's accumulated hardware fields (refer section 2.1). [1]

3.2 Performance Data File Format

In this section we describe a file format for representing performance data for multi-tasking, heterogeneous multi-processor systems. This file format serves as the definition of an unambiguous interface for transferring performance data from the simulator to the performance evaluator/viewer.

The performance data can be reported with different architecture and application elements, at different level of detail, and with different metrics. The file provides information on which elements, metrics, and level of detail are supported. This allows for simulator platforms with different capabilities. It is up to the viewer to extract and visualize the desired performance for files

[1] It is not possible to provide more information about the metric relations as that piece of information is Philips Company Restricted.

```
FORMAT
    #FORMAT Section
END

DATA
    #DATA Section
END
```

Figure 3.1: The FORMAT and DATA Sections

coming from different platforms. Finally, the performance data is reported in accumulated form. This allows for partial reports and it is up to the viewer to derive intermediate metric values.

The statistics file consists of a format section and a data section. These sections are delimited by the keywords, respectively, FORMAT, DATA and END as shown in figure 3.1. In essence the FORMAT section defines how the performance data is organized, while the data section contains the actual performance data measurements. Each of these sections is described in details below.

3.2.1 The FORMAT Section

The FORMAT section contains information on how the performance data is organized. It describes which architecture and application elements can appear in the file, and how they are related. It also describes which metrics are used for measuring the performance data, and in which sequence they appear in the file. This is done by means of a number of architecture and application element definitions. Such definitions take up one line in the FORMAT section. An example of an architecture element definition is shown below.

```
ARCH Processor proc_id {} { Exec Read Write Blocked Idle }
```

The definition starts with the reserved keywords ARCH or APPL, followed by the keyword used for the type of this architecture or application

```
FORMAT
    ARCH Processor proc_id {} {Exec Read Write Blocked Idle}
    APPL Task      task_id {} {Exec Read Write Blocked Suspend}
    ARCH ProcPort  pprt_id {proc_id} {Active Idle}
    APPL TaskPort  tprt_id {task_id} {Active Blocked Idle}
END
```

Figure 3.2: Example of the FORMAT section

element, followed by three more fields which can be seen as formal parameters to specify:

1. The identifier (name) of the architecture or application element.
2. The parent name of this element.
3. A list of metrics for this element for performance data.

This particular FORMAT section (figure 3.2), is applicable for the type of systems which we deal with, and defines the architecture elements Processor and ProcPort. The application elements defined are Task and TaskPort. It also shows that a task is mapped on a Processor and a TaskPort on a ProcPort. The metric identifiers which appear, such as Execute, Read, Write etc. are in essence formal parameters., but their exact names give meaning to what is being measured.

3.2.2 The DATA Section

The DATA section contains the actual performance data itself, which is segmented over one or more REPORT sections. A REPORT section corresponds to the state of collected performance data in the system at a certain moment in time, triggered by an event.[2]

[2]It is not possible to provide more information about the Data section as that piece of information is Philips Company Restricted.

```
DATA
    REPORT ''PDC(1)'' 15000 x1
     #Report generated at 15000 cycles.
     #Performance Data.
    REPORT ''PDC(2)'' 30000 x1
     #Report generated at 30000 cycles.
     #Performance Data.
END
```

Figure 3.3: Example of the DATA section

Chapter 4

Performance Data Visualization

In this chapter a brief overview of the visualization process as well as the research done in this area is made. Next a set of views for visualizing the performance data are described in detail. The views explained, while being easily perceptible, allow to probe extensively into the system behavior. Some of the views described need metric relations. To this end a metric definition file or view setup file is described.

4.1 Visualization Concepts

In the next few sections the various concepts associated with data visualization are explained. Some of these concepts have arisen from the unique challenges of interpreting performance data while others have proved useful in the design of effective performance views and data. A detailed discussion is presented in references [3] and [4].

4.1.1 Context

To present performance information clearly, it is necessary to establish some context to which users can relate that information:

- *Perspective*–The point of view from which information is presented. Typical perspectives for performance information include the hardware, the application program etc.

- *Semantic Context*–The relationship between performance information and the constructs and abstractions (such as data structures and control structures) in application program. Another example is selecting a graphical image from an element and highlighting the corresponding portion of the application program.

- *Subview Mapping*–A mapping between a subset of a graphical view (for instance, a rectangular subregion) and the corresponding subset of the data being rendered. This mapping implies that the data can be reconstructed from the image, which would not be the case if the data were reduced so that details were lost.

4.1.2 Scaling

Scaling graphical views as data sets become very large is a major challenge. This is especially true of performance visualizations as the number of processors or the duration of execution substantially increases. Several techniques have been used to handle this scaling problem:

- *Multidimensional and multivariate representation*–A representation of data with many attributes per data point. Like most scientific data, performance data are typically multi-dimensional, with both space-like (processor, memory etc.) and time-like dimensions and other parameters (such as problem size) that may vary as well. Such a multivariate representation is conceptually compact, but the technical challenge for visualization is to represent as many dimensions as possible on a flat video surface.

- *Macroscopic and microscopic views*–The level of detail represented by a given view. A macroscopic view conveys the big picture, while a microscopic view depicts the fine details.

- *Display manipulation*–The interactive modification of a display, through techniques such as scrolling or zooming, to handle a large amount of data of varying detail.

- *Composite view*–A Synthesis of one or more views into a single view that is intended to enhance visual relationships among the views and present more global information. Examples include combining lower dimensional displays into a single higher dimensional display or taking time along a third axis to construct a three-dimensional display.

4.1.3 User Perception and Interaction

Successful visual performance tuning depends on a synergistic feedback loop between the user and the visualization tool. The tool produces images that the user interprets, while the user selects views and options to guide the tool in detecting and isolating performance bottlenecks. Important concepts in this category include:

- *Perception and Cognition*–The sensory development of an impression, awareness, or understanding of a phenomenon. Human visual perception can grasp patterns, distinguish variations, classify objects etc.

- *Observing Patterns*–The observation of spatial, temporal, or logical patterns of behavior, which often indicate important interrelationship between models and data. For example, a repetitive pattern over time is often related to a program's iterative loops.

- *User interaction*–Selections that users make regarding alternate views, levels of detail, and display parameters. Such interaction lets users customize the visualization for a given situation to enhance understanding. For example in a processor oriented display, users can select a particular ordering of processors to make patterns more evident.

4.1.4 Comparison

Comparisons and cross-correlations between related views or representation can provide much insight into behavioral characteristics and their causes. Several graphical techniques can be used for visual comparison.

- *Multiple views*–The visual representation of data using multiple displays from different perspectives. Any single visualization or perspective can only display a portion of the relevant behavior. Viewing the same underlying phenomenon from diverse perspectives gives a more well-rounded impression.
- *Small multiples*–A series of images showing the same combination of variables indexed by changes in another variable, much like successive frames of a movie. Animation is one example of this technique (indexing over time).
- *Cross-execution views*–The visual comparison of information from program executions that may differ in various ways, such as in problem size or machine size.

4.1.5 Extraction of Information

Several techniques enable visual extraction of useful information from a morass of data:

- *Reduction and filtering*–Representing raw data by statistical summaries, such as maxima and minima, means standard deviations, frequency distributions and so on. This notion extends to graphical reductions, where a visual display conveys the general trends rather than detailed behavior.
- *Clustering*–Multivariate statistical analysis and presentation techniques for grouping or categorizing related data points. The intent is to classify points or identify outliers in a multidimensional data space. Classical examples include scatterplots and frequency histograms.
- *Encoding and abstracting*–Using graphical attributes such as, color, shape, size, and spatial orientation or arrangement to convey information (for instance, additional dimensions).
- *Separating information*–Visual differentiation among layers of information through color highlighting, foreground/background, and so forth.

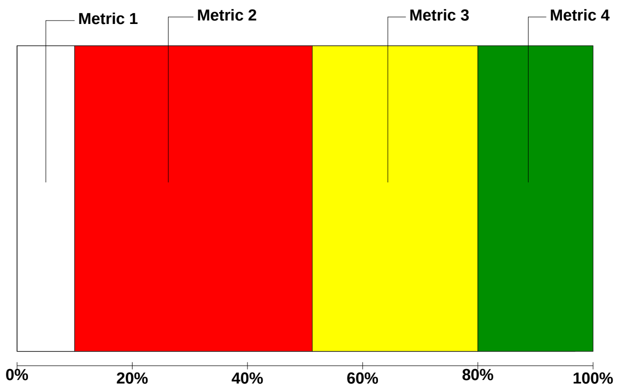


Figure 4.1: Unexpanded Hierarchic View - Processor

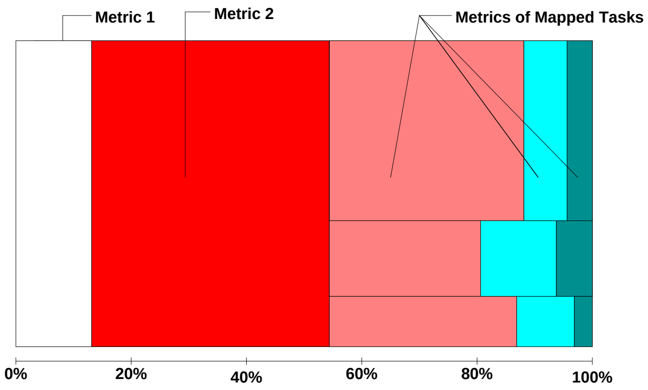
4.2 Performance Views

In this section a set of performance views are described. Following is a list of requirements on the visualization process.

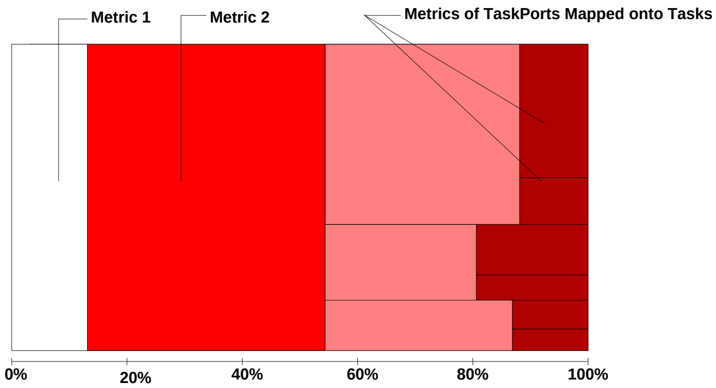
- In all cases, the visualization provides a view of how the resources are spent. Color coding is used to provide a quick overview of in what way these resources are used.
- Performance data is visualized per architecture element.
- A selection of which instances to view can be made.
- A selection of the level of detail to be visualized can be made.
- A selection of which time frame (between elements) to view can be made.
- The performance data can be visualized either as *static*, being the average over the given time segment, or as *dynamic*.

4.2.1 The Hierarchic View

The first view which is described is termed as the Hierarchic view. As shown in figure 4.1 and 4.2, the resource usage, in this case cycles, are shown according to metric definitions, (refer section 3.1). The detail level is at that of



(a) First Level Expansion - Tasks of Processor



(b) Second Level Expansion - TasksPorts of Tasks

Figure 4.2: Hierarchic Views

a processor (for this example) and they clearly show the metric fractions for processor. In the next view, figure 4.2, the *accountable metrics* of processor (refer section 3.1.5) i.e. exec, read and write are replaced by the exec, read and write states of the mapped tasks. The height of each of the tasks is in direct proportion to the resource utilization and all the tasks are arranged in descending order of resource utilization. In the next view this procedure is taken one step further and the resource utilization of individual tasks is broken up into taskports.

4.2.2 The Dynamic View

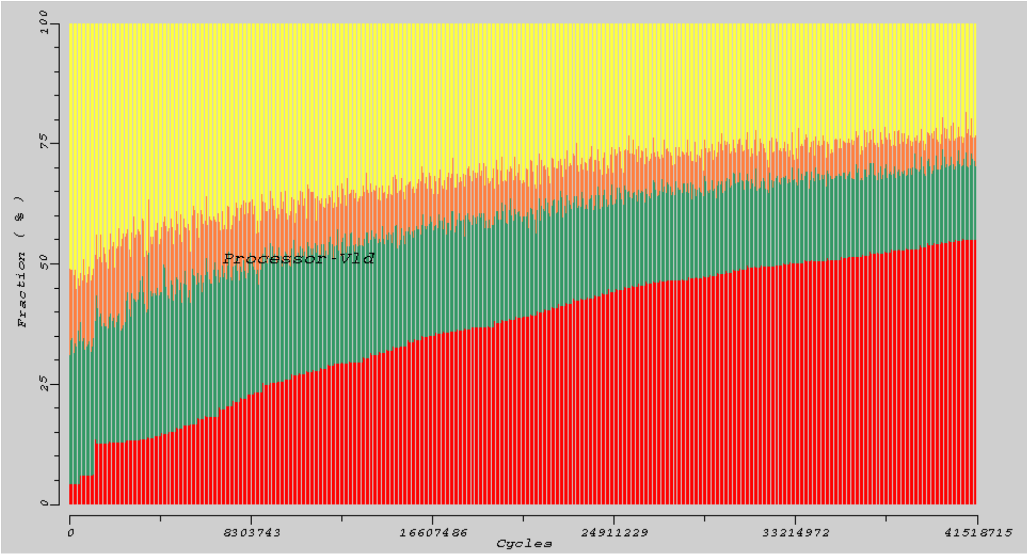


Figure 4.3: Dynamic View

The next view (refer figure 4.3) which is described is termed as the Dynamic view. In this view, the metric fractions are plotted as a function of time hence the name dynamic view. This sort of view does not have detail sub levels as this leads to complex pictures.

4.2.3 The Bar Graph View

In a Bar Graph view (refer figure 4.4), the metric value is plotted as a function of time. Some metric definition may not fall into the semantic meaning of a state. Examples of such metrics are the number of read and written bytes. For such metrics currently only a Bar graph view is defined. Though a Bar Graph view could as well be applied to other metrics.

4.3 The View Setup File

The view setup file, shown in figure 4.5 specifies how a particular set of metrics should be viewed. This file consists of three primary sections, view

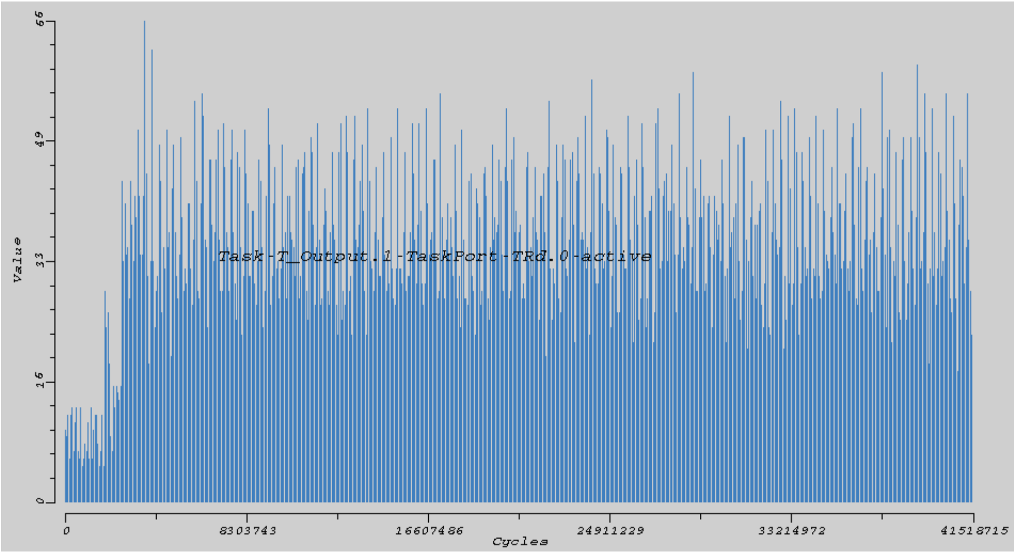


Figure 4.4: Bar Graph View

setup, metric setup and color setup. Each of these sections is described below:

- *The view setup section*–The view setup section starts with the reserved keyword VIEWSETUP and ends with the keyword END. It defines the refinement relations for the Hierarchic view. An entry in the view setup section looks as shown below,

```
TypeName Processor {exec} {exec read write} Task {exec read write}
```

  The definition starts with the reserved keyword TypeName followed by five fields which are described below:

  1. The identifier (type name) from trace file, to which this definition belongs.
  2. The list of states of identifier for which this definition holds.
  3. The list of *metrics* which are to be removed from the view, for redrawing the *new metrics*.
  4. The identifier (type name) of element which is accountable to these metrics.

```
VIEWSETUP
    #View Setup
END

METRICSETUP
    #Metric Setup
END

COLORSETUP
    #Color Setup
END
```

Figure 4.5: The View Setup File

5. The list of metrics of accountable element which are to be drawn in the new view.

- *The metric setup section*–The metric setup section starts with the reserved keyword METRICSETUP and ends with the keyword end. It defines which metrics of a particular type are not to be used for calculating metric fractions. For example the number of read or written bytes for memory. An entry in the metric setup section looks as shown below:

  ```
  TypeName TaskPort NonStates {acc_space}
  ```

  The definition starts with the reserved keyword TypeName which is followed by the identifier (type name) of the element to which this definition corresponds. This is followed by the reserved keyword NonStates and finally a list of states.

- *The color setup section*–The color setup section starts with the reserved keyword COLORSETUP and ends with the keyword end. This section defines the colors for the various metrics which are to be used in drawing the views. An entry in this section looks as shown below:

  ```
  TypeName Processor {exec 255 251 48 read 255 158 48}
  ```

  The definition starts with the reserved keyword TypeName which is followed by two fields as explained below:

1. The identifier (type name) to which this definition corresponds.
2. A list of metric and color values in the form

   ```
   <metric> <r> <g> <b>.
   ```

   where metric refers to the name of the metric and each of r, g and b correspond to the integer value of color R, G and B component respectively.

Chapter 5

Performance Measurement System Implementation

In this chapter implementation of the complete performance measurement system is described. This includes the simulator and simulator hooks as well as the performance visualizer or *Telescope*. The simulator is written in C++ and the visualizer in Java.

5.1 Performance Measurement System

The performance analysis system, as shown in figure 5.1 consists on the top level of three entities. The simulator along with it's hooks which is described in detail in the sections 2.3 and 5.2. The trace file which is explained in detail in section 3.2 and the performance visualizer, which is described in section 5.3. The simulator along with the hooks, writes the data in a trace file. This file is next picked up by the performance data visualizer along with a view setup file. Since the simulation is pretty slow keeping the performance data in a file is justified. This also allows for the possibility of comparing simulation results for various runs.

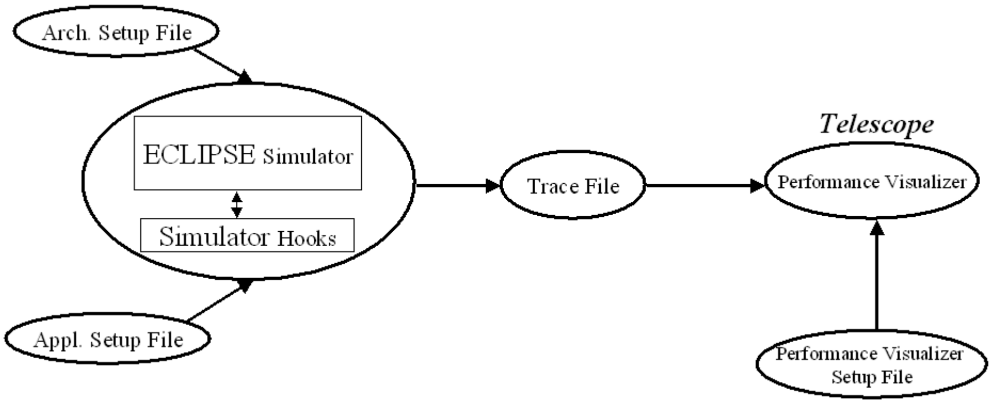


Figure 5.1: The Performance Measurement System

5.2 Simulator Hooks

The hooks are organized as a C++ class, here after referred as the performance measurement class, which is called at the simulator level in the ECLIPSE simulation environment (refer section 2.3). This gives the performance measurement class access to the complete eclipse hardware. Access to internal ECLIPSE hardware is made using the MicroscopeRead and MicroscopeWrite functions. This provides a common interface for accessing hardware elements while abstracting the implementation details. The functionality of hooks is implemented using two classes, namely, *PerfMeasurement* class and *PerfData* class. Each of these classes is described in detail in next section.

5.2.1 PerfData class

The PerfData class represents the performance data associated with each coprocessor. There are four vectors present in each instance of this class. They correspond to the processor itself, the tasks mapped onto this processor, the processor ports of this processor and the task ports of each task. For each of these elements the data is stored in a class which has fields for determining the parent of each element, so, it is possible to resolve to which task a task port belongs.

5.2.2 PerfMeasurement class

The PerfMeasurement class represents the actual measurement functionality. The functionality of this class can be divided into two parts. The initialization part and the computation part. The initialization part is responsible for reading setup information from appl.setup file and then configuring the appropriate hardware elements. This configuration involves,

- setting up the value of *NMeasurements*, *MeasureFreq* and enabling the *measure_flag*.
- Echoing the FORMAT section in trace file.
- Initializing the data structure.

The NMeasurements values is run down (to zero) as measurements progress and the measure_flag is set to false when the measurements complete. Thus these two fields need to be reset after generation of each REPORT section. This setting up thus is performed by both, the initialization part as well as the computation part.

The computation part performs the following functions,

- Collects the performance data from each hardware element.
- Echoes the performance data in trace file.
- Reconfigures the measurement hardware.

For collecting the performance data, the computation part simply walks the task table and Stream Synchronizer stream table for each of the coprocessors. It then manipulates the data as per metric relationships and stores them in the appropriate part of data structure. Once this process is complete is calls the writeToFile functionality in each of the coprocessor perf data objects. This writes the performance data to file in proper order. Finally it reconfigures the hardware and passes control back to the simulator.

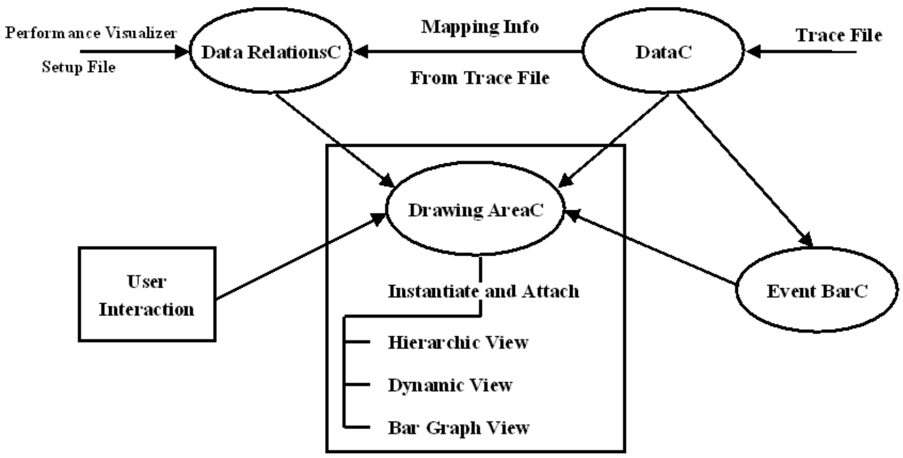


Figure 5.2: Telescope Classes

5.3 The Performance Visualizer - *Telescope*

Telescope is implemented using Java. Java has a number of advantages over other languages for front end development.

- Is portable across various platforms.
- Is an object oriented language.
- Has built in capability for rendering complex graphical shapes, using the Java2D and Java3D API.
- Has a large widget set for developing GUI.
- Has advanced event handling capabilities.

Figure 5.2 shows the Telescope classes. In this figure an oval represents a class and a square represents a collection of classes. The two inputs to Telescope are a setup file (Section 4.3) and a trace file (Section 3.2). The DataC class is responsible for parsing the trace file and extracting information from it. It has a number of public methods using which it is possible to get reported data for any particular cycle. The mapping information present in the trace file, is picked up by the DataRelationsC. This particular class also parses the view setup file and extracts view setup information.

The EventBarC class is responsible for getting the reported cycles information and maintaining the last selected interval information. The information present in all these classes along with the user interaction is taken up by the DrawingAreaC class. This class is responsible for setting up the data for each of the view types, instantiating an object for that particular view and attaching the object to itself, so that the view object is visible [1]. To get rid of the problem of keeping all the element names unique, each sub element is specified by prefixing the parent name with it as shown,

```
Element     : Input
Description : Top Level Processor
Identifier  : Input

Element     : Inp.out
Description : Input  Output ProcPort
Identifier  : Input-ProcPort-Inp.out
```

So, using this naming convention it is possible to have two elements with same name under different coprocessors. One example where this is useful is in naming the read and write ports of each coprocessor.

5.3.1 DataC class

The DataC class consists of two separate private classes. These two classes are TraceFormatC and TraceDataC. These are responsible for maintaining the information associated with the format and data sections respectively. The mapping information is picked up by the associated object of TraceData. This class works through the following public access functions:

- *getStates*–This function returns the states associated with any element. This data is present in the format section of the trace file.
- *getParamters*–This function returns the metric values reported in data section for a particular element and a particular cycle.

[1] An object of DrawingAreaC class is what is visible to the user

- *getCyclesList*–This function returns the list of cycles for which metric values have been reported. So, after combining this function with the *getParameters* function it is possible to get the reported metrics for all the cycles.

- *getReportsList*–This function returns a list of report titles for all the report sections. This is utilized by the EventBar class, to present the user with a context to relate the cycle information with.

- *getExplicitMapTree*–This function returns the tree containing explicit mapping information which is used by the DataRelationsC class and the TreeAreaC class.

- *getImplicitMapTree*–This function returns the tree containing implicit mapping information which is only used by the DataRelationsC class.

5.3.2 DataRelationsC class

The DataRelationsC class is responsible for maintaining the relation ships between elements reported in the trace file as well as the viewing information which it gets from the view setup file. This class essentially works through four public access functions:

- *getNonStates*–This function return names of metrics which have been specified as non states in the view setup file. This is utilized by the DrawingAreaC class for setting up a particular view.

- *getStatesToRemove*–Once a hit [2] has been reported on a particular element. The object which has been hit calls this particular method to get the names of states which have to be removed from the view.

- *getChildStatesToDisplay*–This method returns the states of mapped elements (child) which have to be drawn in the view.

[2] Mouse click

- *getMappedElements*–This function utilizes both the implicit as well as the explicit map tree information to return a space delimited list of identifiers which are mapped onto this element.

5.3.3 EventBarC class

The EventBarC class is responsible for maintaining the last selected interval information, displaying this and delivering the selected interval information on request. This class works through the following public access functions:

- *getBeginCycle*–This returns the *actual* begin cycle associated with the users last selected interval. This function always returns the begin cycle by snapping to events for which the trace data contains reports.

- *getEndCycle*–Similar to the previous function this function returns the end cycle for users last selected interval after snapping to events.

5.3.4 User interaction classes

This is a collection of classes which are responsible for managing the user interaction. This involves the displaying of trees, responding to events etc. The classes which are responsible for this user interaction are:

- *LogAreaC*–This class is a static and is used for logging any important information. For instance the fraction values associated with any hierarchic view.

- *TreeAreaC*–An object of this class is what is visible to the user in form of a tree. This class calls the public access functions in DrawingAreaC class to generate any particular view.

5.3.5 DrawingAreaC and associated classes

This class is responsible for setting up the data for a particular view, instantiating the object and attaching the object to itself. After the particular view has been instantiated it responds to any event itself. So, for example the view object can report the cycles and fractions information, once hit. To extend the set of views currently defined, a public access function needs to be defined in this class, and a corresponding menu item in the TreeAreaC class to call this member function along with a class rendering this view. This class only provides access functions for instantiating a particular view along with the member functions for printing a view, for all other functionality it calls objects of other classes.

Chapter 6

Simulation Setup and Results

In this chapter the simulation setup is described in detail. This includes the architecture and application setup with the available coprocessors as well as the setting up of performance measurement units in hardware. A description of the available coprocessors and their behavior is also made. Finally an analysis of the results obtained from this setup is presented.

6.1 Simulator and Telescope Setup

At the time of writing this report the simulator factory contained four coprocessors. The input coprocessor, the output coprocessor, the MPEGHdr coprocessor and the VLD coprocessor. The application setup configured a MPEG decoding application on this particular architecture. The decoded file was a MPEG stream with 7 frames.

To start the measurement process, some setup parameters need to be defined in the application setup and in the architecture setup. These are presented next:

6.1.1 Architecture Setup

The parameters which need to be defined are,

- *NWidthMeasureFreq*–This sets the width of measure_freq field implemented in the hardware. In all the simulations this value was set to

16.

- *NWidthClockDivider*–This sets the width of clock divider field implemented in hardware. In all the simulations this values was set to 22.

6.1.2 Application Setup

The parameters which need to be defined are,

- *Measure_Freq*–This field selects a particular bit of the clock divider at the rising edge of which the measure flag is set so that measurements take place. In simulations this value was set to 14.
- *NMeasurements*–This field selects the number of measurements to perform before which reconfiguration is again needed. In all the simulations this value was set to 255.
- *PerfMeasurement.FileName*–This particular variable defines the name of the trace file to which the performance data is written.

6.1.3 Telescope Setup

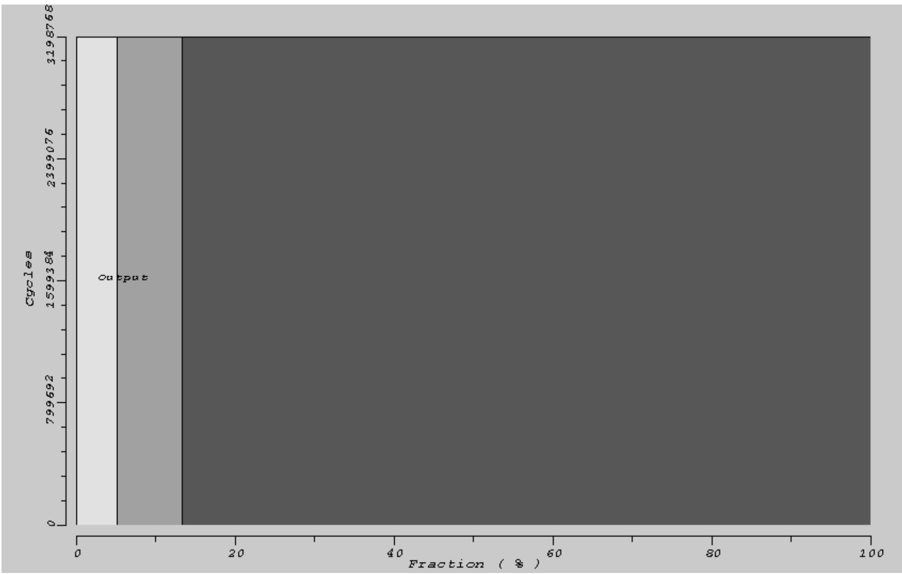
The full telescope setup file is presented in appendix A. The view setup section defines the view setup for ECLIPSE metric relations. So, the *exec* state of a processor becomes expandable by the *exec read and write* states of a task. And the *read and write* states of processor are expandable by the *active* state of mapped processor ports. The acc_space field which is extracted on a per task port basis represents the relative FIFO filling rate. So, this has been mentioned as a non state in the metric setup section. This telescope setup file is listed in A also this file is present in the telescope directory in ECLIPSE source tree.

6.2 Results and Analysis

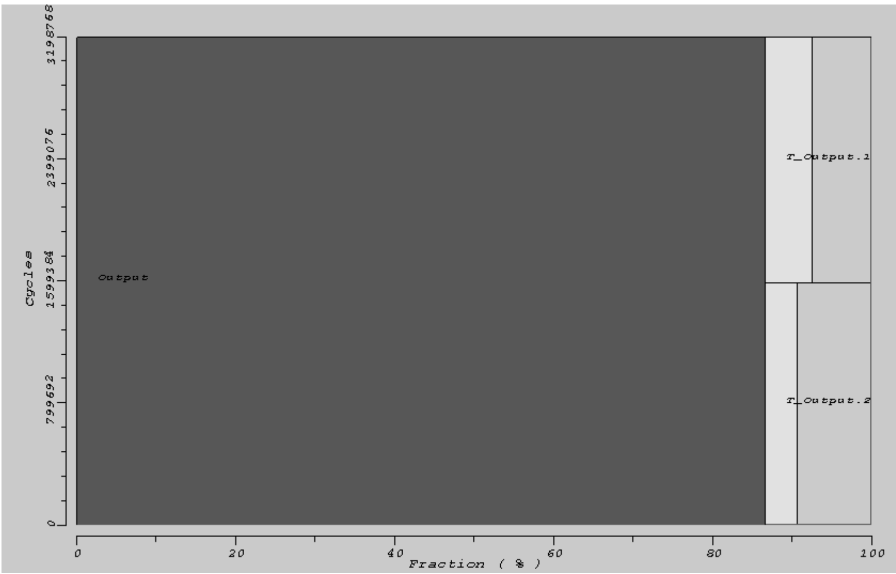
In figures 6.1 through 6.3 the performance views as obtained for the MPEG decoding application are shown. Here only the relevant set of views for output coprocessor are presented. Figure 6.1(a) shows resource utilization for the output coprocessor. The view in figure 6.1(b) shows the resources utilization for output which has been refined on a per task basis. This view clearly shows that the tasks 1 and 2 which have been mapped onto this coprocessor consume the maximum number of resources.

The next figure (6.2) shows the time (dynamic) views. From figure 6.2(a) it is clear that apart from a slight variation in the coprocessor performance in the initial stages the coprocessor performance is uniform.

The last set of figures (6.3) shows the resource utilization of the tasks mapped onto this coprocessor. This is shown only for the mapped task 1 and mapped task 2 since these are the only two tasks which show up in the expanded hierarchic view, figure 6.1(b).

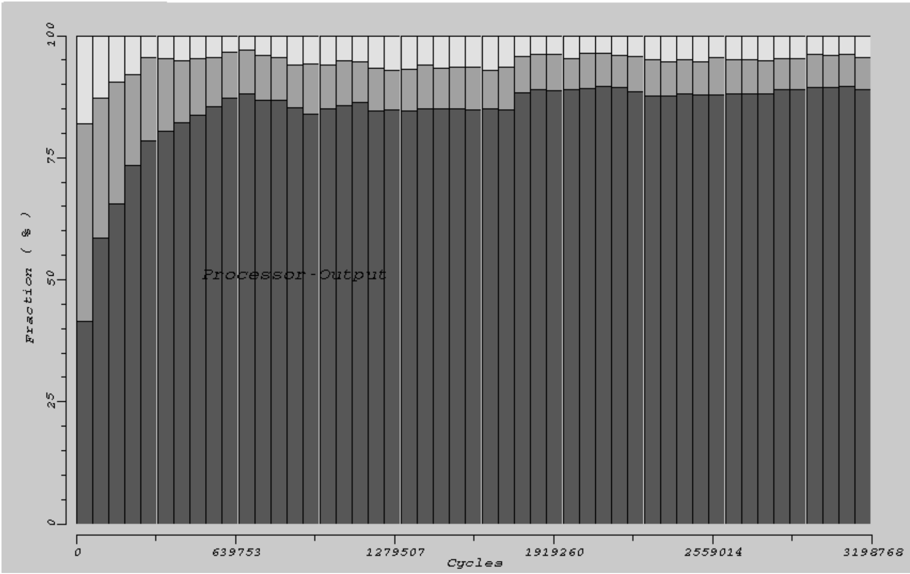


(a) Output Processor - Resource Utilization



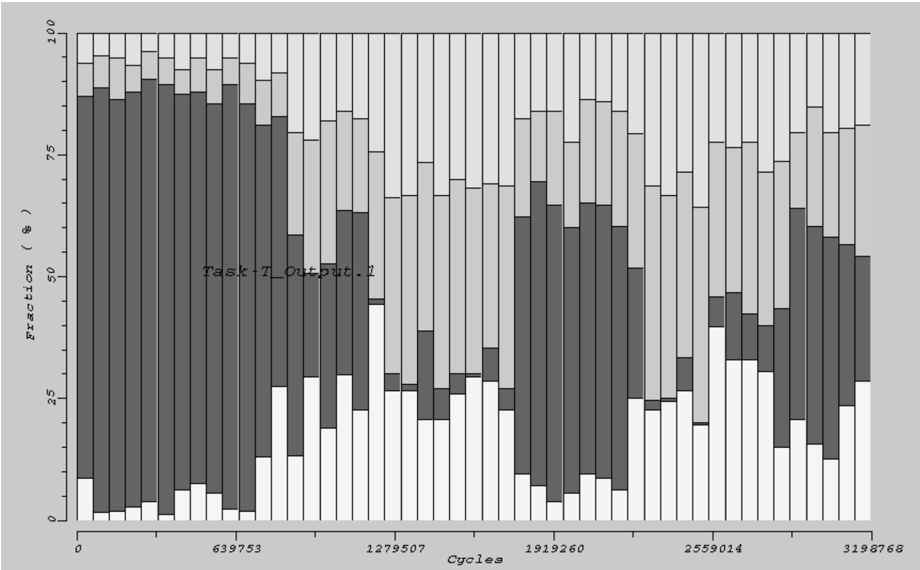
(b) Output Processor - Tasks

Figure 6.1: Output Processor Performance Views

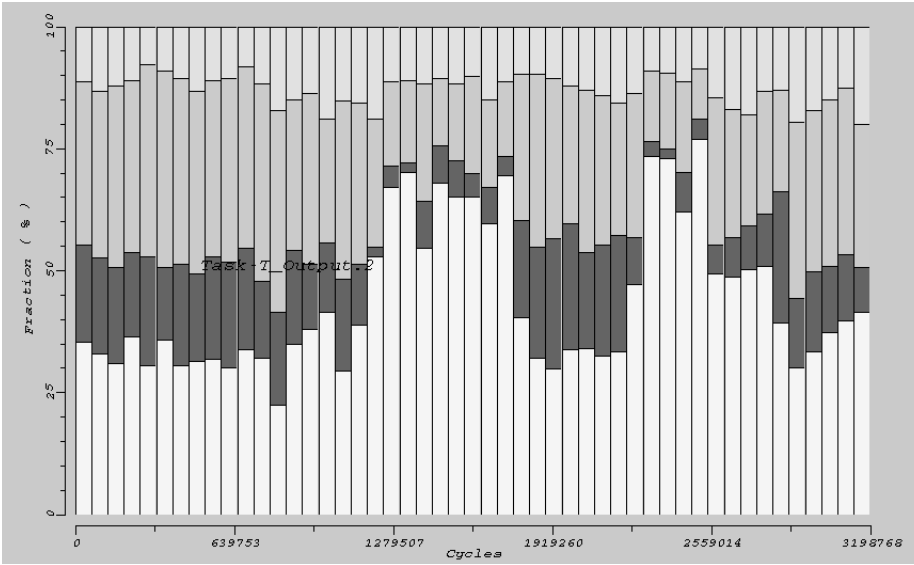


(a) Output Processor - Dynamic View

Figure 6.2: Output Processor Performance Views (contd)



(a) Output Processor Mapped Task 1 - Dynamic View



(b) Output Processor Mapped Task 2 - Dynamic View

Figure 6.3: Output Processor Performance Views (contd)

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The initial goal was to design a tool to study performance of ECLIPSE architecture. However during the course of project it was realized that a significant amount of work in the area of performance measurement for such systems had already been done in another group [2]. So, it was decided to take a part of their work and try to match it to the project requirements. This lead to the development of Telescope. Since Telescope is customizable it can be used for a variety of systems. The separation of metric relations from the rest of the visualization process was another major contribution of the work, which also forms the basis of the Metric Definition File being proposed. Telescope was informally tested in a different environment as mentioned in [14]. Development of Telescope in Java also enhanced the understanding of this language in the group, as well as established that it is a good alternative to TCL/Tk.

The performance views as obtained for ECLIPSE were as expected. However the metric extraction process requires certain enhancements. This is evident in the case of Mpeghdr coprocessor being shown as *executing* in most of the views. The problem is with the separation between *blocked and run* and *blocked and suspended* states. But, since the metrics were derived from the hardware fields present in ECLIPSE, which provided no segregation between these two states, it was not possible to separate these two.

7.2 Future Work

A number of extentions are possible to the current work. These include:

1. Implementation of hooks for extraction of more metrics from the simulator. Currently the metrics being extracted only provide information as present in the hardware fields. This can be extended to include metrics which are extracted from the simulation process only and not confirm to any hardware.

2. Implementation of a callback mechanism in the simulator hooks. Currently, the performance data collector only collects the data after some predefined time interval. This happens when the performance measurement process has stopped in hardware. The drawback with this process is that critical application specific events (e.g. completion of decoding of a slice in MPEG decoder application) are lost. The call back mechanism would ensure that a report is generated as and when this occurs and would give a better overall picture to the user.

3. Currently Telescope implements only a small set of views. This can be extended to include some 3D views like Kiviat tubes, which can provide more information from a different perspective.

Appendix A

Telescope Setup File

```
VIEWSETUP

TypeName Processor {exec} {exec read write} Task {exec read write}

TypeName Processor {read write} {read write} ProcPort {active}

TypeName Task {read write} {read write} TaskPort {active}

TypeName ProcPort {active} {active} TaskPort {active}

END

METRICSETUP

TypeName TaskPort NonStates { acc_space }

END

COLORSETUP

TypeName Processor {exec  255 251 48 read 255 158 48  write 25 132 103
                    blocked 255 0 0 idle 255 255 255 }
```

```
TypeName  Task {exec 227  255 48  read 255  173 143  write 20  131 103
blocked 255 8 77 suspended 255 255 192 }

TypeName TaskPort {active 255 159 221 blocked 150 0 0 idle 255 224 192
                    }

TypeName ProcPort {active 255 182 174 idle 255 8 255 }


END
```

Appendix B

Telescope Users Manual

B.1 Introduction and Requirements

Telescope is a general purpose customizable tool for performance evaluation of ECLIPSE like heterogeneous multiprocessor programmable architectures. It expects a trace file and setup file as input. It can currently display the performance data in three types of views, with direct postscript rendering capability. It supports selection of time scale to analyze, multiple element selection, multiple concurrent windows with shared data, multiple concurrent open trace files along with other user interaction capabilities. However Telescope is not a general purpose visualization tool. It does not claim to have the rendering capabilities of visualization tools like Polka [5] etc. However it is possible to effectively use Telescope in performance analysis problems where resource utilization is is to be analyzed. This is based to a large extent on the metrics which are defined and extracted.

Telescope requires Java2 version 1.2 or greater. The postscript rendering capabilities of version 1.2 are very poor. Every graphics shape in version 1.2 is rendered as a rasterized image. So, it is possible to get postscript file sizes of the order of hundreds of megabytes. More over the Swing implementation of version 1.2 contains a lot of bugs. So Java2 version 1.3 is recommended which has better postscript rendering capabilities along with fixes for a number of bugs reported in version 1.2.

B.2 Launching and Running Telescope

Telescope is launched by typing the command,

```
java -jar <path to jar>/Telescope.jar [trace_file setup_file] ..
```

If a set of trace and setup files is specified on the command line, Telescope will open a new window for each set. A sample trace file and setup file is present in the Telescope directory in the ECLIPSE CVS tree this may be used for initial experiments. The opening window contains four prominent areas:

1. *The Tree Area*–This is the area where the architecture and application tree are drawn, initially unexpanded. All the view drawing options are launched in this area. Selections are made using the left mouse button and the view menu is launched using the right mouse button. It is possible to make multiple selections by pressing *control* key along with a left mouse click. A large number of selections can be made by pressing *shift* along with left click at the final element.

2. *The Drawing Area*–The drawing area is initially shown as a blank area towards the right of the tree area. This area spawns a new tab for any new view which is created. The name of the tab is the complete identifier (refer 5.3) associated with any element. A left click on any area of the graph will report the information associated with that shape in the log area. If this particular view is a hierarchic view, then pressing the shift and control keys along with a left click has different meaning. Pressing a shift along with a left click will expand this view if the view is expandable while pressing a control key will unexpand this particular view, provided it is unexpandable. The fact whether a particular *state* of a view is expandable or not is reported when that particular *state* is clicked. Some information is automatically reported when a view is created, and/or modified. For a hierarchic view this is the total fractions associated with all the metrics.

3. *The Event Bar Area*–The event bar area shows the last selected interval. This is automatically snapped to reported events as and when a left mouse click occurs in this area. An event selection is made by either clicking the mouse at the initial position and dragging it till the final position or clicking

it once at the initial position and pressing a shift along with a left click at the final position. The current position of mouse pointer is reported along with the report title towards the bottom of this area.

4. *The Legend Area*–This area contains the legend for this particular window. These colors are fixed in the setup file and it is not possible to change these from Telescope.

B.3 Points to keep in mind

Mentioned below is a set of points which are important to remember when working with Telescope.

- When telescope is launched a lot of warning about missing fonts may be reported, this is not a bug. This is the problem of X server not being able to locate all the fonts required by Java.
- When Telescope is printing then the whole tool seems to be suspended (windows are not refreshed etc.). This has been deliberately done to keep the views from being modified when Telescope is printing.
- In the trace file, if any element is being specified as a parent of another element, then the element identifier, section 5.3, and not name is expected. This resolves the problem of having all the element names being globally unique.

B.4 Information and Reporting Bugs

Telescope was developed by Anup Gangwar as part of M.Tech theses at IIT Delhi and Sector IST, PRLE. His supervisors were Dr. Ir. Jos van Eijndhoven at PRLE and Prof. Anshul Kumar and Prof. M. Balakrishnan at IIT Delhi. Telescope is based on the Multi-processor Performance Data Collection and Visualization document by Gerben Essink and Gerben Hekstra with feedback from Pieter Van Der Wolf. All the bugs may be reported to anup@cse.iitd.ernet.in.

Bibliography

[1] Gills Kahn, "The Semantics of a Simple Language for Parallel Programming," *in Proc. of the IFIP Congress 74,* 1974, North Holland Publishing Co.

[2] Gerben & Gerben, "Multi-Processor Performance Data Collection and Visualization", *Nat. Lab. Technical Note (Philips Company Restricted)*, August 2000.

[3] Michael T.Heath, Allen D.Malony and Diane T.Rover, "Parallel Performance Visualization: From Practise to Theory", *IEEE Parallel & Distributed Techonlogy: Systems & Applications*, Vol. 3, No. 4, Winter 1995.

[4] Michael T.Heath, Allen D.Malony and Diane T.Rover, "The Visual Display of Parallel Performance Data", *IEEE Computer*, Vol. 28, No. 11, November 1995.

[5] Polka Animation System.
http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html

[6] E.A. de Kock and G.Essink, "Y-chart Application Programmer's Interfact, The YAPI Programmer's and Reference Guide", *Nat. Lab. Technical Note*, November 2000.

[7] J.T.J van Eijndhoven et al., "ECLIPSE Generic Communication Architecture", *Nat. Lab. Technical Note (Philips Company Restricted)*, March 1999.

[8] J.T.J van Eijndhoven et al., "ECLIPSE Detailed Generic Architecture Specification", *Nat. Lab. Technical Note 2000/051 (Philips Company Restricted).*

[9] J.T.J van Eijndhoven et al., "ECLIPSE CPU-Coprocessor Architectures", *Nat. Lab. Technical Note 405/99 (Philips Company Restricted).*

[10] Java 1.2 Application Programmers Interface.
http://java.sun.com/products/jdk/1.2/docs/api/index.html

[11] Developer's Guide to Swing.
http://java.sun.com/products/jdk/1.2/docs/guide/swing/index.html

[12] Java 2D Application Programmers Interface.
http://java.sun.com/products/java-media/2D/forDevelopers/2Dapi/index.html

[13] Online Training at java.sun.com.
http://developer.java.sun.com/developer/onlineTraining

[14] Basant Kr. Dwivedi, "Modeling & Mapping of Multimedia Applications onto Multiprocessor Architectures", *M.Tech Thesis, Department of Computer Science & Engineering, IIT Delhi*, December 2000.