QuickStep:

A System for Performance Monitoring and Debugging Parallel Applications on the Alewife Multiprocessor

by

Sramana Mitra

A.B., Computer Science and Economics
Smith College, 1993

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
November 28, 1994

Certified by: ______________________________
Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ______________________________
Frederic. R. Morgenthaler
Chairman, EECS Committee on Graduate Students

QuickStep:
A System for Performance Monitoring and Debugging Parallel Applications on the Alewife Multiprocessor

by

Sramana Mitra

Submitted to the Department of Electrical Engineering and Computer Science
on November 30, 1994 in partial fulfillment of the
requirements for the Degree of
Master of Science
in Electrical Engineering and Computer Science

ABSTRACT

In Alewife, a large-scale multiprocessor with distributed shared memory, many sophisticated features have been incorporated to enhance performance. However for most parallel programs, the initial implementation usually produces sub-optimal performance. Alewife hardware offers features to monitor events that provide important information about program behavior. **QuickStep** is a tool that offers a software interface for monitoring such eventsand a graphical interface for viewing the results. The actual monitoring of the data takes place in hardware. This thesis will describe QuickStep's features and implementation details, evaluate the overhead due to the inclusion of the performance monitoring probesand look at case studies of parallel application optimization using QuickStep.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

MIT would have been a very rough, unidimensional experience had it not been for ballroom dancing. I acknowledge first, therefore, dancing. I have named my system **QuickStep** after a dance that is fast and fun!

Of course, none of my work would have gained form if not for the constant guidance and influx of ideas from John Kubiatowicz. My deepest appreciation to Kubi, although I think I have successfully resisted his very convincing attempts to have me stay for a Phd. Thanks Kubi — indeed, it is flattering!

Thanks to my advisor — Professor Anant Agarwal — for giving me the high level direction. Thanks also for being extremely receptive and supportive of my long-term career goals and ambitions.

Ken and Don — my officemates have kept me in constant good humor. I could not have asked for better officemates. Thank you for making MIT warm for me.

I appreciate Ricardo Bianchini's help with the application chapter, as well as with the thesis in general.

Thanks to David Kranz, David Chaiken, Beng Hong Lim, Kirk Johnson and Rajeev Barua for the useful feedback and Silvina Hannono for the support.

Nate — my house-mate — has been a source of enormous comfort and friendship and Ellen has been there for me from day one (actually day negative something). My sincere appreciation to both.

Finally, thanks to all my friends from dancing and elsewhere for keeping me going.

To my parents: See you soon!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Even though the peak performance rating of multiprocessor systems has improved substantially over the past several years, the initial implementation of parallel applications almost never harnesses the full processing power. Performance bottlenecks abound and it is difficult for the programmer to keep track of all aspects of performance optimization. Consequently, there is the need for tools to assist in performance debugging.

The Alewife machine is a large-scale multiprocessor with distributed shared memory built at the MIT Laboratory for Computer Science [1]. Alewife consists of a group of processing nodes connected by a two-dimensional mesh interconnection network. Each processing node consists of SPARCLE - a 33MHz processor, a floating point unit, 64Kbytes of static direct-mapped cache, 4 Mbytes of global shared memory, a network routing chip and a cache controller chip which enforces cache coherence between caches from different processing nodes, and provides a shared memory abstract view of distributed main memory (see Figure 1.1). Currently, the first batch of the Alewife *Communications and Memory Management Unit (CMMU)* chip is being tested by the members of the Alewife team and various software efforts in compiler and performance evaluation technology are in progress. **QuickStep** is one such project which explores the issue of performance debugging of parallel applications on the Alewife machine.

The Alewife CMMU has features to support performance monitoring in hardware. QuickStep utilizes these features to provide a performance monitoring and debugging platform.

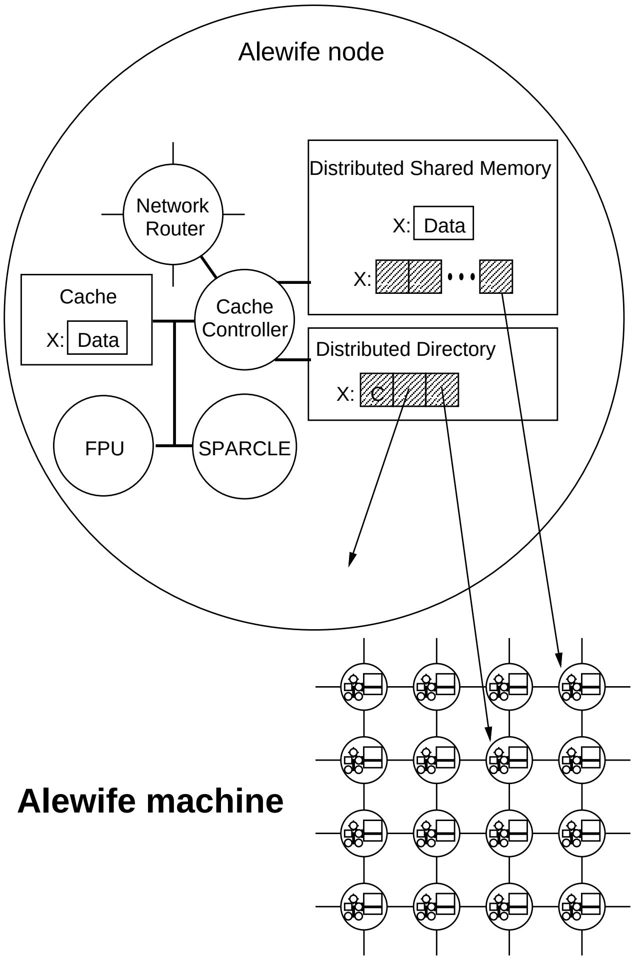

Figure 1.1: An Alewife processor node.

1.1 Performance Monitoring and Debugging Methods: Background

Several efforts have been directed at identifying performance bottlenecks in parallel programs. The popular techniques are *Static Analysis, Simulation, Emulation, Hardware Instrumentation and Software Instrumentation* [10].

1.1.1 Static Analysis

Static analysis although fast, has limited applicability. The most extensive research in static analysis was done at the University of Illinois as a part of the Cedar multiprocessor project [16]. Static analysis involves predicting the performance of loops, counts of local and global memory references, estimates of MFLOPS, etc. based on simple models of instruction latencies and memory hierarchies. The Illinois project later went on to use more sophisticated techniques like exploiting compiler dependency analysis in the predictive models. However, the static analysis techniques are in general inaccurate and hence, are inadequate means of providing performance debugging solutions.

1.1.2 Simulation

Simulation is a slow but precise method. In execution driven simulation, a program is instrumented so that each operation causes a call to a routine which simulates the effects of that operation. While reasonably accurate, simulation is a very slow process and it is not even realistic to simulate the behavior of an entire large program. Therefore, simulation is hardly an effective tool for performance debugging. It is used more for detailed analysis of architectural tradeoffs and is important because it allows evaluation without real hardware. Simulation has been used extensively in the Stanford DASH [17] project, as well as in Alewife during the architectural design phase.

1.1.3 Emulation

Emulation is a method of hardware system debugging that is becoming increasingly popular. Field-programmable gate arrays have made possible an implementation technology that is ideal for full system prototyping, yet does not require the construction of actual silicon chips [23]. Emulation, also called Computer Aided Prototyping, combines CAE translation and synthesis software with FPGA technology to automatically produce hardware prototypes of chip designs from netlists. It enables concurrent debugging and verification of all aspects of a system including hardware, software and external interfaces, leading to a faster design cycle. Using emulation for performance debugging of applications, however, is not very common.

1.1.4 Software Instrumentation

Software instrumentation is fast and flexible. Manually done, it involves instrumenting a program with write statements to print out special purpose information. More sophisticated tools involve automatic instrumentation. The common types of software instrumentation are accumulating an aggregate value (for example, time spent in a procedure) and tracing an event (a new trace event, usually time-stamped, is output each time it is executed). Software instrumentation, however, introduces inaccuracies due to their intrusive nature.

One of the earliest attempts at performance debugging in the sequential domain was *gprof* - an execution profiler that outputs data concerning execution timings in different routines [12]. Gprof monitors the number of times each profiled routine is called (*Call Count*) and the time spent in each profiled routine. The arcs of a dynamic call graph traversed by an execution of the program are also monitored and the call graph is built by post processing this data. The execution times are propagated along the edges of this graph to attribute times for routines to the routines that invoke them.

In the parallel world, a debugger called *Parasight* was developed at Encore Computer Corporation [2]. Parasight implements high-level debugging facilities as separate programs that are linked dynamically to a target program. Parasight was implemented on Multimax, a shared memory multiprocessor.

IPS is a performance measurement system for parallel and distributed programs that uses knowledge about the semantics of a program's structure to provide a large amount of easily accessible performance data and analysis techniques that guide programmers to performance bottlenecks [19]. IPS is based on the software instrumentation technique.

Quartz is another tool for tuning parallel program performance on shared memory multiprocessors. The principal metric in Quartz is *normalized processor time*: the total processor time spent in each section of the code divided by the number of other processors that are concurrently busy when that section of code is being executed.

Other related works can be found in [6], [7], [8] and [20]. A tool called *Memspy* is described in [18] that offers the additional feature of extremely detailed information to identify and fix memory bottlenecks. Memspy isolates causes of cache misses like cold start misses, interference misses, etc. which is very useful.

Mtool is a software tool for analyzing performance loss by isolating memory and synchronization overheads [11]. Mtool provides a platform for scanning where a parallel program spends its execution time. The taxonomy includes four categories: Compute Time, Synchronization Overhead, Memory Hierarchy Losses, and Extra Work in Parallel Program (versus Sequential). Mtool is a fairly general implementation that runs on MIPS-chip based systems like DEC workstations, SGI multiprocessors and the Stanford DASH machine. Mtool's approach is distinct in that where most performance debugging tools lump the compute time and memory overhead together as "work", Mtool offers important information about the behavior of the memory system. Studies have shown that this is critical to optimizing the performance of parallel applications. Mtool is typically estimated to slow down programs by less than 10%.

1.1.5 Hardware Instrumentation

Hardware instrumentation involves using dedicated counters and registers to monitor events. Monitoring of events occurs in hardware and hence is virtually unintrusive. The biggest advantages of hardware instrumentation are its accuracy and speed.

The drawback of hardware instrumentation is that it is not widely available and it may not be as flexible as simulation. In our case, availability is not an issue since Alewife hardware was designed to support instrumentation counters. However, it is only possible to provide a finite amount of instrumentation support in hardware, so it is not as flexible as software. In Alewife, for example, we have 4 statistics counters that monitor a subset of all events. Therefore, only a finite set of events can be monitored during a single run. However, since runs can happen fast, multiple runs allow monitoring of larger sets of statistics. Furthermore, the event monitoring hardware was carefully architected so that most key events could be captured.

QuickStep takes a hybrid of hardware and software approaches and provides a friendly interface for viewing the data collected by the kernel. As is true for most hardware instrumentation based performance monitors, it is not trivial to directly port QuickStep to some other hardware platform. However, the concepts are general and portable. The features of QuickStep will include Gprof like execution profiling facilities, as well as means of monitoring memory system behavior and network traffic patterns.

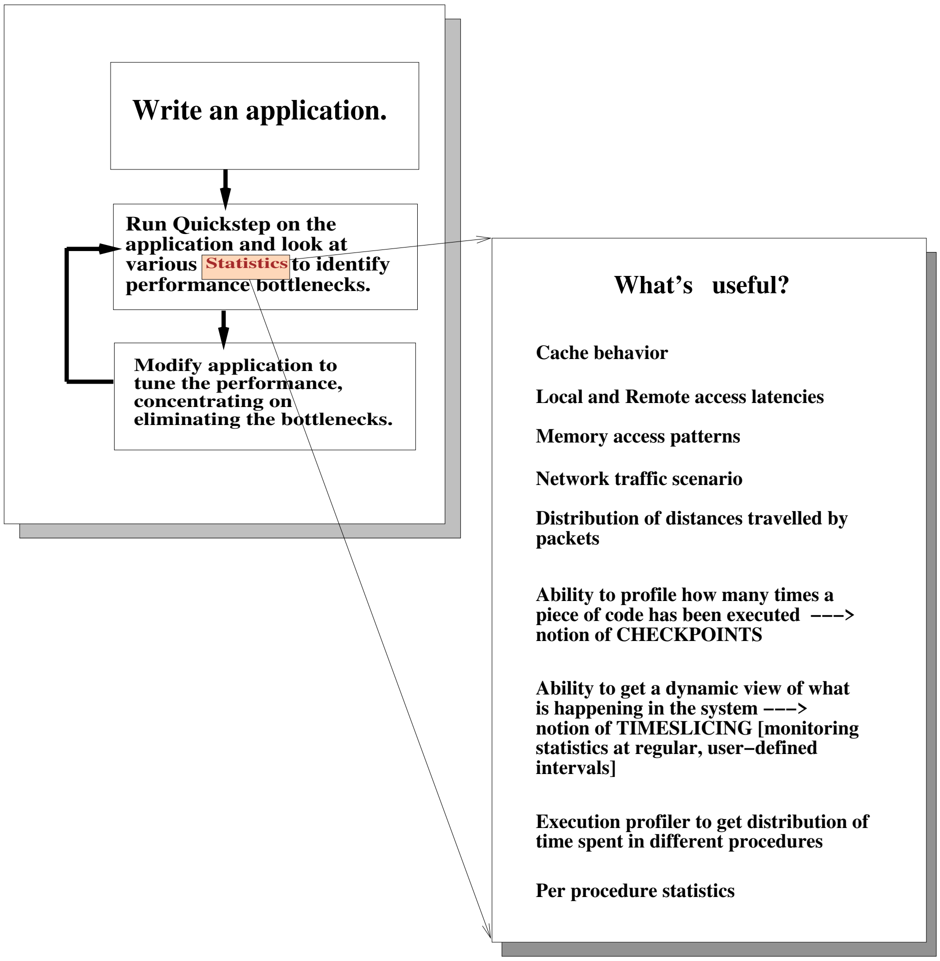

Figure 1.2: Flow chart of tuning the performance of an application using QuickStep.

1.2 Goals of the Thesis

The main goal of this work is to develop the QuickStep system with adequate features to provide a vehicle of further research on the Alewife machine. QuickStep provides a platform for monitoring cache and memory statistics, network statistics, various latencies and message frequencies, etc. for applications that are run on Alewife. It thus enables users to analyze the performance characteristics, bottlenecks and enhancement potentials and to accordingly fine-tune applications. Figure 1.2 shows a flow chart of tuning the performance of an application using QuickStep. It also shows what kind of statistics are useful for performance debugging. In principle, QuickStep is capable of providing all those features, although some of the profiling features have not been implemented yet.

The Alewife CMMU provides basic hardware support for monitoring events. 11% of the total chip area of the CMMU is dedicated to performance monitoring hardware. However, it is not possible to utilize this feature without a well-developed software interface which can handle the bit manipulations and produce comprehensible information. QuickStep provides this interface, as well as a graphical display environment for the information gathered.

1.3 Overview

The rest of this thesis proceeds as follows: Chapter 2 describes the features of QuickStep and illustrates the features that have been implemented so far with examples. This chapter also outlines other features that will be implemented in the next version of the system without too much modification of the existing model. Chapter 3 describes the principles followed in implementing QuickStep. Chapter 4 discusses the suite of programs used to test the validity of the system. Chapter 5 demonstrates the effectiveness of QuickStep by using it to analyze and optimize a few large parallel applications from the SPLASH suite. Finally, Chapter 6 summarizes the thesis.

Chapter 2

Features of QuickStep

In QuickStep, the menu-driven interface through which different statistics are requested is a friendly environment [Figure 2.1]. This chapter describes the different classes of statistics that can be monitored and gives examples of sample outputs. Besides the major categories described here, the interface offers the additional facility of specifying pre-collated groups of statistics that are commonly used. The user can select one or more of these groups without having to look through the detailed menus. The groups are named in a self-explanatory way, for example, *Data and Instruction Cache Ratios, Distribution on Local and Remote Accesses, Read and Write Latencies, Header Frequency through Network Queues,* etc.

2.1 Timesliced Statistics

Statistics can either be recorded at the end of the run, or at regular intervals during the run. QuickStep provides both these options. allowing the user to get some amount of profiling information. Chapter 6 discusses the more elaborate profiling capabilities that will be provided in the next version of the system.

Figures 2.2 and 2.3 are examples of graphs obtained from the cache-statistics menu. The ratio figures on the graphs are rounded up to integers, however, if the user wants to look at more precise values, an easy-to-read raw datafile is available which provides figures upto 14 decimal places. Both the graphs have been obtained by running *Water* from the SPLASH suite on a 16-node Alewife machine, with timeslices of 10,000,000 cycles each. Since the data cache hit ratios are more or less uniform over time, the timeslice mode does not provide much extra information. Data cache hit ratios are uniformly 98-99% for all processors and timeslices, with the exception of the second timeslice of processor 0. However, in Figure 2.3 we see clearly how the access pattern changes over time. For

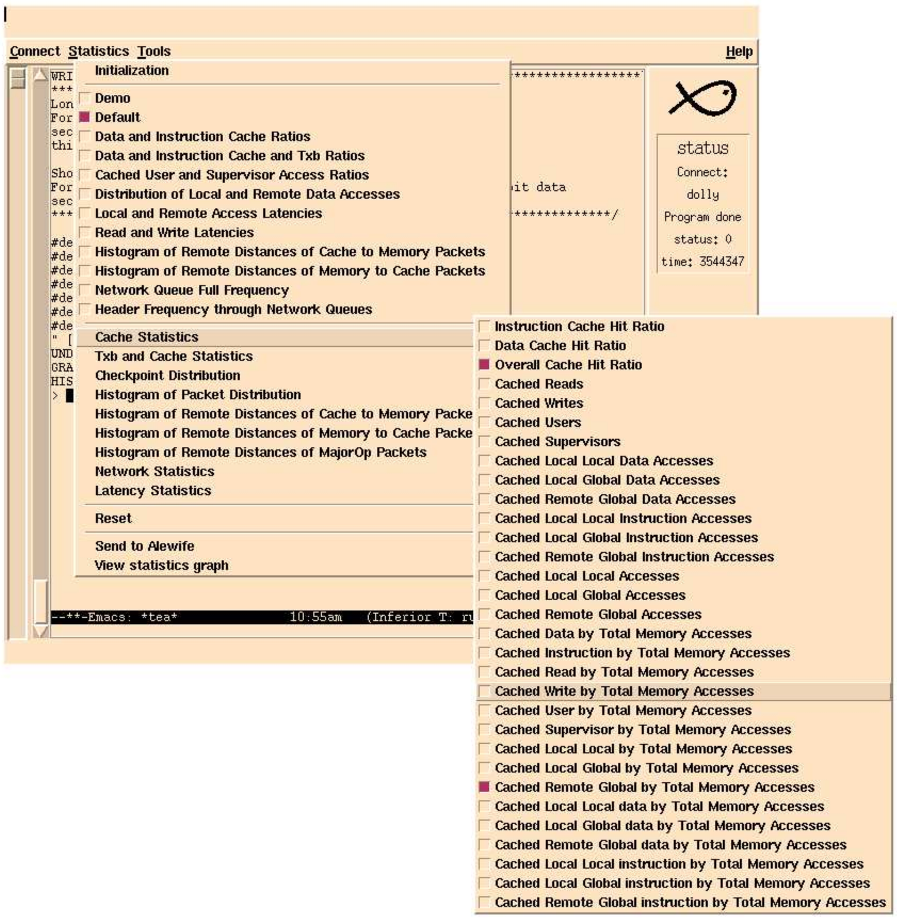

Figure 2.1: The Alewife user interface with pull-down menus for selecting the different statistics to be monitored.

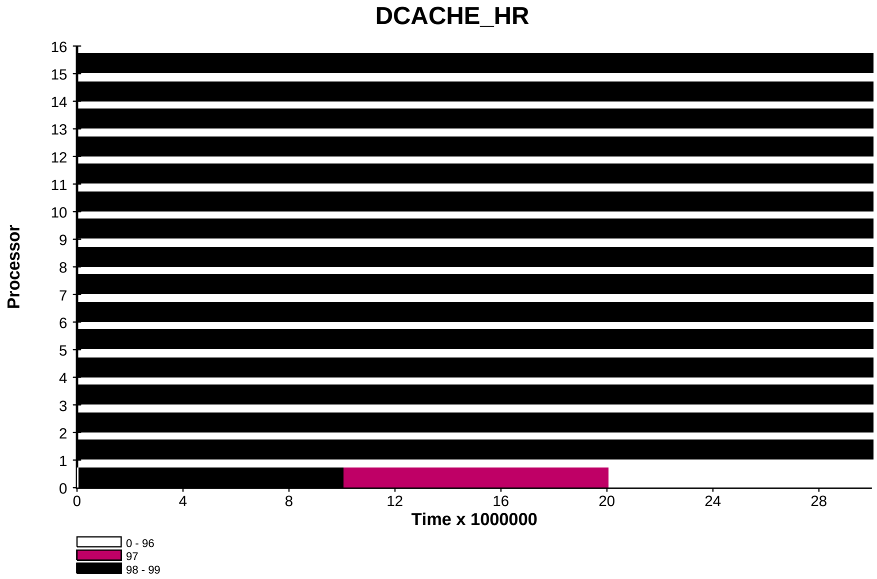

Figure 2.2: Water on 16 processors: Per processor data cache hit ratio.

instance, during the first timeslice, all processors other than processor 0 are waiting for the initialization to be completed. Hence, the number of remote data accesses are low for all processors except processor 0. In the subsequent timeslices, all the processors are doing real computation, and the access pattern reflects activity.

When statistics are collected in the timesliced mode, program behavior is perturbed to some degree. Chapter 3 describes how the timesliced statistics are implemented and discusses perturbation due to instrumentation. When statistics are to be collected at regular intervals, the program has to stop running during the collection phases. This makes timesliced instrumentation intrusive and hence comparatively inaccurate. On the other hand, the timesliced mode does provide important information about program behavior over time.

2.2 Overall Statistics

Often, however, the user simply wants an overall summary statistic for the run. In such cases, the timesliced mode is turned off and the counter values are recorded at the end of the run. Figure 2.4 shows an example of a statistic gathered at the end of the run, ie. in the non-timesliced mode. It is possible to post-process data gathered in time-sliced mode

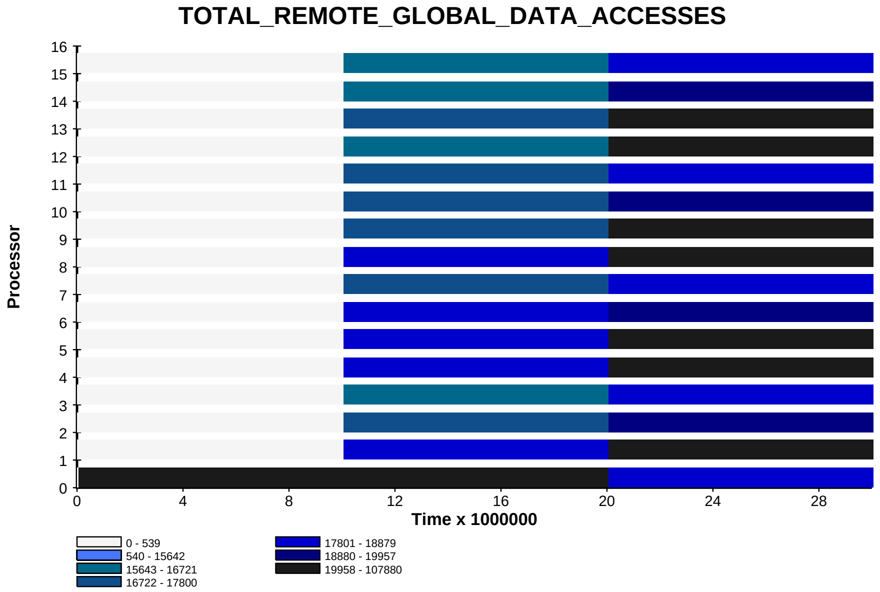

Figure 2.3: Water on 16 processors: Per processor distribution of remote shared data accesses [Table B.1].

to obtain the overall statistics. However, as discussed in the previous section, timesliced statistics perturb program behavior, while non-timesliced statistics do not. Furthermore, non-timesliced mode is naturally faster than the timesliced mode. As always, there is a trade-off between accuracy of statistics, perturbation of program and speed. Hence, we provide both timesliced and non-timesliced modes of collecting statistics.

2.3 Network Usage Statistics

Two types of hardware statistics yield histograms of values: *counts of packets* and *distribution of distances that packets have travelled*. The *counts of packets* mode watches network packets and whenever a packet appears in a designated direction (input or output), it is checked to see if it matches the class of packets that is being tracked. If so, then the corresponding histogram counter is incremented [Chapter 3]. This class of statistics is useful for tracking different classes of packets, especially from a low-level kernel programmer's point of view. Being able to track protocol packets, synchronization packets, boot packets, etc. can help a runtime system designer.

The *distribution of distances* mode increments histogram counters based on the number

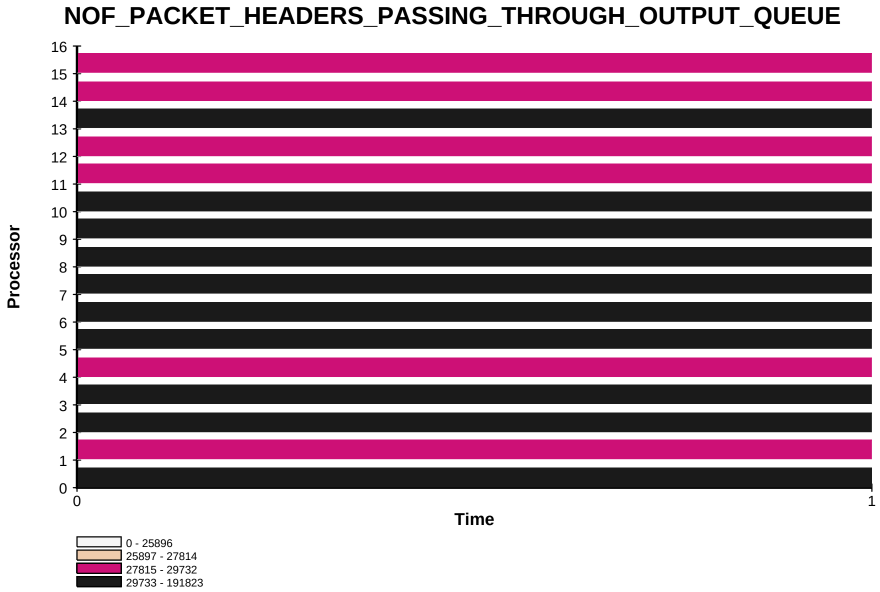

Figure 2.4: Water on 16 processors: Counts of packet headers passing through output queues [Table B.2].

of hops a packet of the specified class has travelled. Figure 2.5 gives an example of such a statistic. The y-axis has the processor number and the x-axis has the number of hops. The colors represent the numbers of packets in each category. This is very useful for debugging application performance because it provides a way of judging whether the application is showing good communication locality or not. Ideally, most of the communication ought to be nearest neighbor and if the remote distance histogram reflects that this is not the case, then the programmer can debug the application. It is easy to see the effect of debugging some aspect of a program by simply comparing histograms obtained from running different versions.

2.4 Checkpoints

Checkpoints are a set of useful debugging features offered by QuickStep. They are single cycle instructions that can be included at different points in a program and useful profiling information can be obtained by looking at the counts of the checkpoints. For instance, information can be obtained about how many times a procedure is called, or how often a particular section of code is executed.

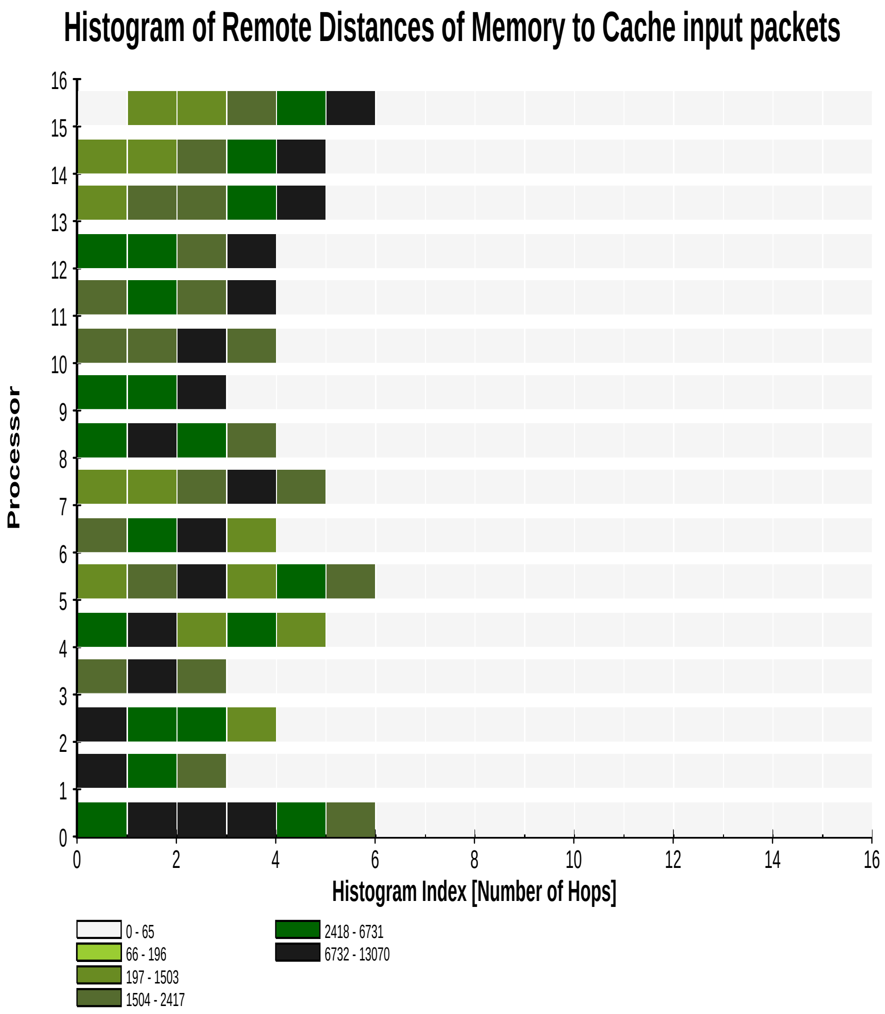

Figure 2.5: Water on 16 processors: Histogram of distances of memory-to-cache input packets [Table B.3].

To use the checkpoint facility of QuickStep the program being run needs to be annotated with the checkpoint instruction. This section gives an example of using checkpoints. The checkpoint instruction takes one argument, with two parts separated by a colon. The first part is the checkpoint group name and the second part is the checkpoint name. Checkpoint group is an abstraction which is used to group several checkpoints together. [A maximum of 16 checkpoints are allowed per group.] The argument translates to a 12-bit checkpoint address.

The three different checkpoint modes operate as follows:

- **Classwise**: In this mode, only the checkpoint group name to be tracked needs to be specified through the user interface. The address translation works such that the corresponding counter is incremented when the first 8 bits of the translated address matches a checkpoint.

- **Single**: In this mode, both the checkpoint group name and the checkpoint name to be tracked need to be specified through the user interface. The address translation works such that the corresponding counter is incremented when all 12 bits of the translated address matches a checkpoint.

- **Histogram**: In this mode, only the checkpoint group name to be tracked needs to be specified through the user interface. The histogram mode of checkpoint monitoring gives a distribution of accesses for checkpoints of a certain class.

Figures 2.6, 2.7 and 2.8 use the checkpoint features of QuickStep to monitor the procedure *Proc1* listed in Appendix A. *Proc1* is annotated with some checkpoints and when those checkpoints are tracked using QuickStep, the expected frequencies are obtained.

The procedure *Proc1* is started up on all 16 processors with arguments 10 and 200. Checkgr2:Check2 is monitored in Figure 2.6 and rightly comes out to be $(10 * ProcessorId)$ on each processor. [The graph shows the ranges of frequencies and table 2.1 shows the exact numbers.]

Checkgr2 is monitored in Figure 2.7 and rightly comes out to be $(10 * ProcessorId + 2)$ on each processor. [The graph shows the ranges of frequencies, while table 2.1 shows the exact numbers.]

Checkgr3 is monitored in Figure 2.8 and rightly comes out to be 1 each for Check1 (Histogram Id = 0) and Check3 (Histogram Id = 2) and $(200 - (10 * ProcessorId))$ for Check2 (Histogram Id = 1) on each processor. [The graph shows the ranges of frequencies and table 2.2 shows the exact numbers. Data for processors 0 through 6 only are represented in the table, but the rest of the data from the raw file have been verified to be consistent.]

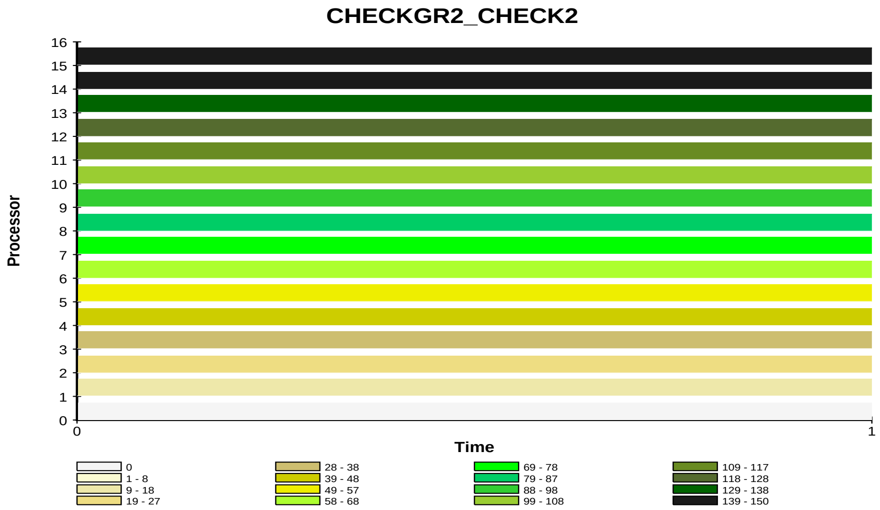

Figure 2.6: Result of monitoring Checkgr2:Check2.

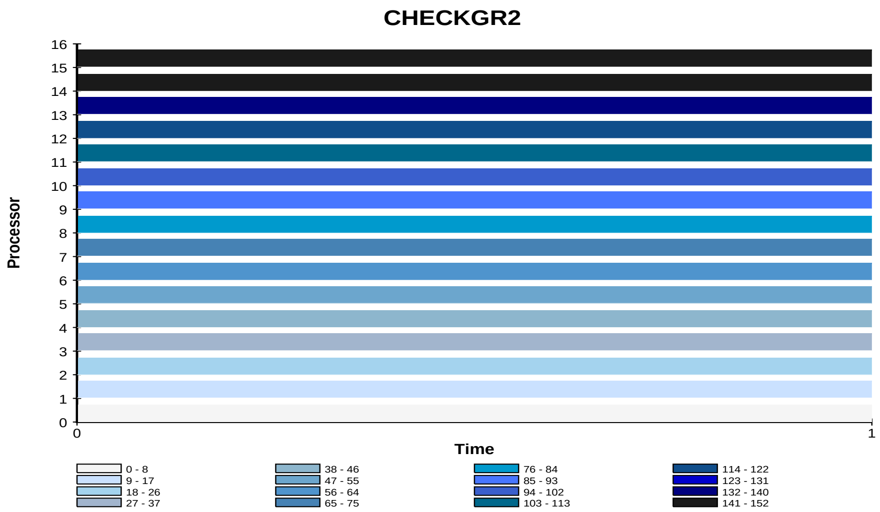

Figure 2.7: Result of monitoring Checkgr2.

Processor Id	Checkgr2	Checkgr2:Check2
0	2	0
1	12	10
2	22	20
3	32	30
4	42	40
5	52	50
6	62	60
7	72	70
8	82	80
9	92	90
10	102	100
11	112	110
12	122	120
13	132	130
14	142	140
15	152	150

Table 2.1: Data obtained from the raw data file for the classwise and single checkpoint graphs.

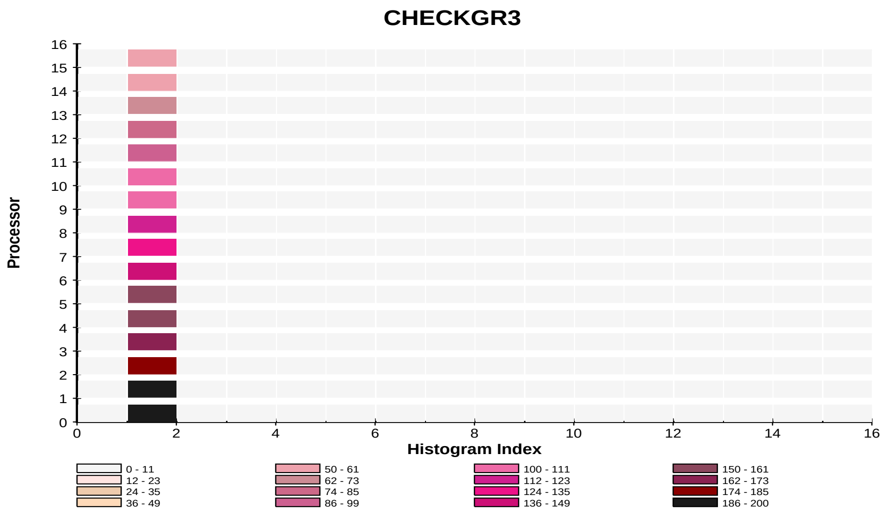

Figure 2.8: Result of monitoring Checkgr3.

Processor Id	Checkgr3	Histogram Id
0	1	0
0	200	1
0	1	2
1	1	0
1	190	1
1	1	2
2	1	0
2	180	1
2	1	2
3	1	0
3	170	1
3	1	2
4	1	0
4	160	1
4	1	2
5	1	0
5	150	1
5	1	2
6	1	0
6	140	1
6	1	2

Table 2.2: Data obtained from the raw data file for the checkpoint histogram.

2.5 Additional Features

The first major addition that is planned for the next version of QuickStep is a profiling feature. Currently, statistics gathering cannot be turned on or off in the middle of a run. However, this is a feature that would be of enormous usefulness. For instance, users of the QuickStep system have commented that it would be useful if a certain set of statistics could be computed on a per procedure basis. The statistics could be of various types: the amount of time spent in the procedure, the cache behavior and the network statistics for the procedure, etc.

This feature can be incorporated easily, by encoding the turning on and off of statistics counters into a function. Ideally, the user should be able to specify the name of the procedure and the statistics to be monitored. The compiler/linker would then incorporate the function in that procedure automatically, the process being transparent to the user.

Furthermore, there are several classes of statistics that the CMMU supports which have not been implemented in this version. These include *synchronous trap statistics*, *hitmiss statistics*, *remote transaction statistics*, *memory controller statistics* and *transaction buffer statistics*.

From a presentation point of view, we are currently using the Proteus *Stats* program as the display environment. Most of the data we are presenting would be easier to read in a 3-dimensional graphical display environment, which stats does not support. There is room for improvement in the way the statistics are represented through graphical displays.

Chapter 3

Implementation

In this chapter, we discuss the implementation details of the QuickStep performance monitoring system. Figure 3.1 shows a flow chart of the QuickStep system. This chapter is organized according to the flow chart as follows: We first describe the user interface in Section 3.1. Next, the Alewife architechtural support for performance monitoring and the resource allocation procedure are described in Section 3.2. Finally, the data collection and reporting is described in Section 3.3, and the graphical display is discussed in Section 3.4.

3.1 User Interface

The high-level interface for the QuickStep system is an integrated part of the general Alewife interface developed by Patrick Chan. Figure 2.1 shows a snapshot of the interface. It consists of menu items for:

- Connecting to the Alewife machine or to the NWO simulator (NWO is a simulator that has been developed as a part of the Alewife design process by David Chaiken)
- For selecting the statistics to monitor and display the graphical output of QuickStep
- For running *Parastat*— a graphical monitor of the status of the individual nodes of the Alewife machine (also developed by David Chaiken)

The code for the interface is written in Tcl, an X-windows programming environment.

The user requests the statistics that he or she wants to monitor by clicking on the relevant menu items. The major classes of statistics that are currently offered are the following:

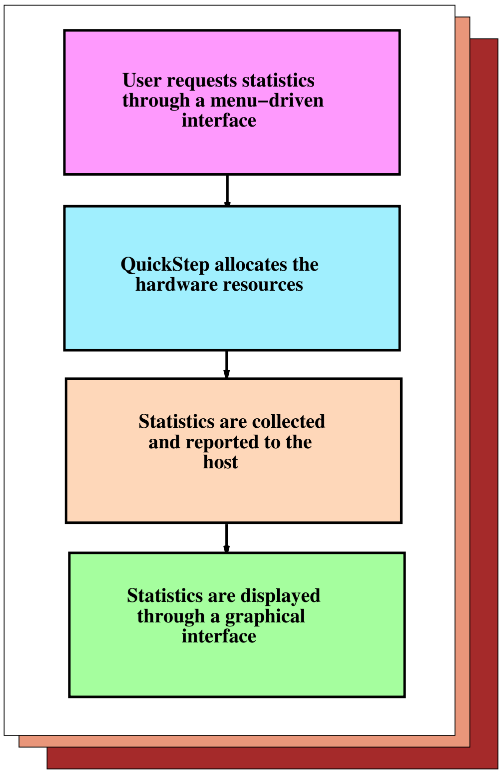

Figure 3.1: Flow chart of the QuickStep system.

- Cache Statistics
- Checkpoints for profiling
- Histograms of Remote Distances of cache to memory and memory to cache packets
- Network Statistics
- Latency Statistics for different types of memory accesses

This information is then passed on to the host[Figure 3.2] by selecting *Send to Alewife* from the *Statistics* menu. Internally, this transfer of information happens through the exchange of a message which is decoded by the host running QuickStep.

3.2 Resource Allocation

In this section, we first describe the hardware support that Alewife provides for performance monitoring. Then, we discuss the resource allocation problem, and how it is solved in QuickStep. We also describe the configuration file in which resource requirement information for different statistics are stored.

3.2.1 Alewife's Performance Monitoring Architecture

Alewife, being a vehicle for research in parallel computation, has several built-in features that assist in monitoring events like *data and instruction cache hit ratios, read accesses, write accesses, distances travelled by packets,* etc. In particular, the CMMU has 4 dedicated 32-bit statistics counters, and 16 20-bit histogram registers. The histogram registers are also counters that are incremented when certain events occur. The histogram registers monitor events like histograms of checkpoints, packet distributions and packet distances. The histogram control field of the statistics control register [Figure 3.3] is used to configure the histogram registers as a unit to count different events. Each statistics counter is independently configured with a 32-bit event mask.

When an overflow of the statistics or histogram counters occurs, the hardware takes a trap. A 32-bit overflow counter for each statistics and histogram counter is implemented in software, which are then incremented. 64-bit resolution is thus achieved by extension into software.

The user interacts with the machine through the host (see Figure 3.2). The host is attached to processing node 15 of the mesh, and supports an interactive user interface.

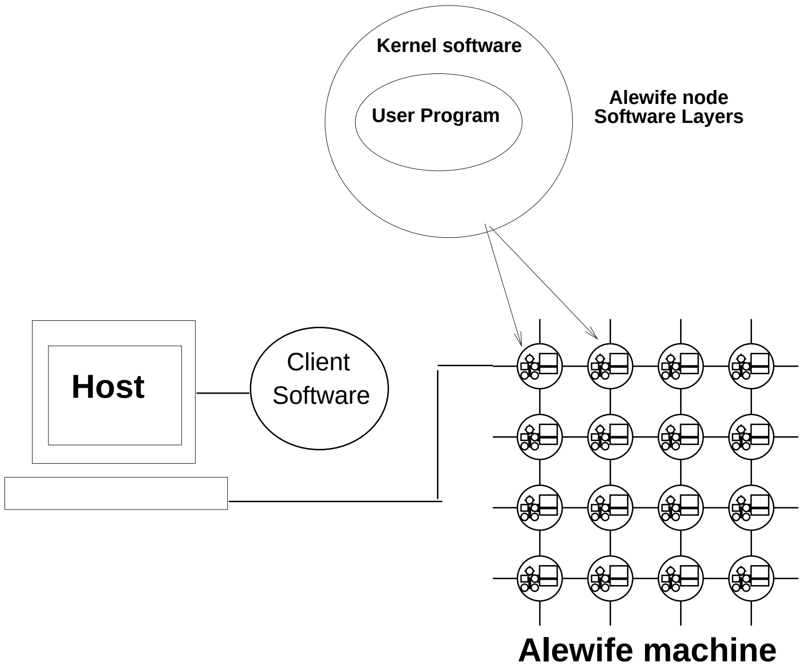

Figure 3.2: Software layers in Alewife.

The user can boot the machine, load and run programs via this interface. The code for instrumenting the statistics gathering facility is included as a part of the Alewife kernel and the statistics monitoring mode is activated by adding features to the host interface. The Alewife kernel supports a message-passing interface [15] which is used to communicate between the host and the machine.

Alewife also supports a timer interrupt facility which is used to interrupt processors at specified times to collect statistics for a certain interval. This feature of the Alewife architecture is utilized in QuickStep to provide snapshot views of the behavior of a program over time, as described in Chapter 2.

3.2.2 Hardware Mask Configuration

As mentioned before, the CMMU has registers dedicated to monitor statistics. These registers are divided into two sets: the statistics counters and the histogram registers. Each set is controlled by one or more control registers. The statistics and histogram registers can work independently, or work together (to compute latency statistics).

The histogram registers are controlled as a single unit by the StatCR (statistics control) register. The registers work together as bins (except when computing latencies) to keep a histogram of different events. Chapter 2 provides examples of this mode. The StatCR also controls other aspects of the statistics registers such as traps.

The counter registers work independently of each other and each has an associated 32 bit control register called its *mask*. These masks can be used to make the counters count a general set of events, or a subset of events. For instance, a counter can be set up to count cache hit ratios, or just data cache hit ratios.

Figure 3.3 shows the fields of the StatCR register and of a typical statistics counter mask. The *histogram control field* of the StatCR register holds the histogram mask, the *StatCounter0 Control, StatCounter1 Control, StatCounter2 Control and StatCounter3 Control* fields are responsible for enabling and disabling statistics counters 0 through 3.

The statistics counter masks have a 4-bit major function specifier and a 28-bit minor function specifier each. The major function specifier bits determine the class of statistics to be monitored (eg. checkpoints, network statistics, etc.) The minor function specifier fields determine the specifics within a class of statistics.

Let us look at a counter mask from the configuration file in Figure 3.6. 201EFF20 is the hexadecimal mask for counting number of cached data accesses. The major function specifier is 2, which represents the cache statistics. Bits 17 through 21 represent the type of processor request. Bit 21, for instance, denotes an instruction match. Since we are counting data accesses specifically, bit 21 is turned off. Bits 17 through 20 are read and write requests and are hence turned on. Bit 5 represents cached accesses and hence needs to be on. The rest of the bits are configured accordingly.

3.2.3 The Resource Allocator

As mentioned before, the Alewife CMMU has only 4 statistics counters and 1 set of 16 histogram registers. Consequently, only a small number of events can be monitored during a single run. Hence, when the user requests a large number of statistics, several runs are needed to satisfy such requests. In such cases, allocation of counters need to take place across runs.

QuickStep has a resource allocator to take care of this task. Say, the user has requested 3 statistics: Data Cache Hit Ratios, Cached Unshared Data Accesses, Cached Local Shared Data Accesses and Cached Remote Shared Data Accesses. For Data Cache Hit Ratios we need 2 counters to count *number of cached data accesses* and *total number of data accesses*. For Cached Unshared Data Accesses we need 2 counters to count *number of cached unshared data accesses* and *total number of unshared data accesses*. For Cached Local Shared Data accesses we need 2 counters to count *number of cached local shared*

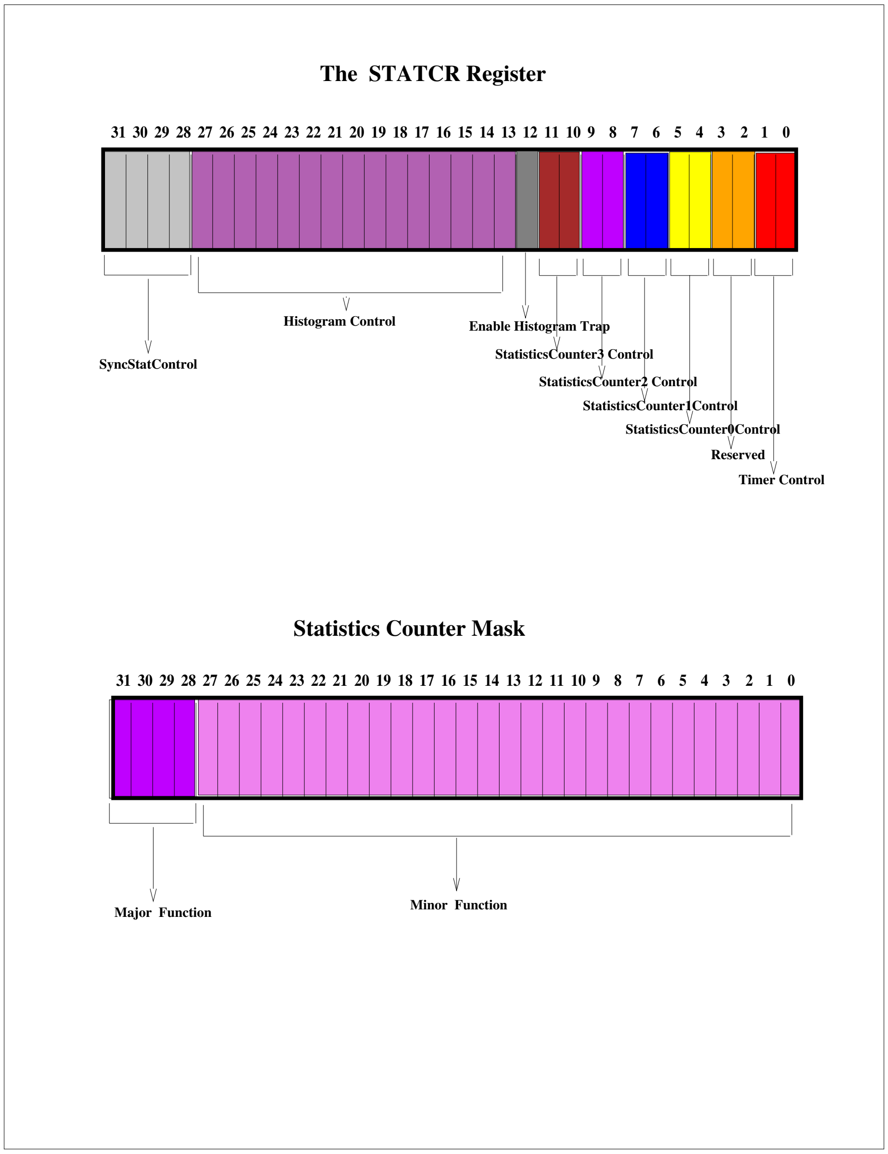

Figure 3.3: Statistics counter mask fields.

data accesses and *total number of local shared data accesses*. For Cached Remote Shared Data accesses we need 2 counters to count *number of cached remote shared data accesses* and *total number of remote shared data accesses*. That is a total of 8 events and 8 statistics counters are needed to compute them.

The resource allocator is intelligent enough to be able to figure out how many counters will be needed and how many runs will be required given the counter requirement. It can also eliminate duplicate events and optimize the number of runs required to satisfy the requested set of statistics. In this case, the resource allocator assigns *number of cached data accesses, total number of data accesses, number of cached unshared data accesses* and *total number of unshared data accesses* to the first run. The *number of cached local shared data accesses, total number of local shared data accesses, number of cached remote shared data accesses* and *total number of remote shared data accesses* are assigned to the second run.

The fact that all the requested statistics cannot be computed in one run due to limitations in the availability of hardware resources, implies, there is always a window of error. Hence, each statistic needs to be gathered several times and averaged over all data points to eliminate this effect. Since the hardware can only provide a finite set of resources, this effect is pretty much unavoidable.

3.2.4 The Configuration File

The information about what the mask values are for each event to be monitored is stored in a configuration file that is read by the resource allocator. The configuration file uses a configuration language described in Figure 3.4.

The operations that are to be performed on the counters to get the requested statistics are specified by the *CounterOperations* keyword. The specific operations that are allowed are described in Figure 3.5

Example of a Configfile

Figure 3.6 shows a sample configuration file with four records.

The first record provides the resource specifications for *Data Cache Hit Ratio* of statistics class *Cache Statistics*. The 2 counter masks provide mask-configurations for monitoring *the total number of cached data accesses* and *the total number of data accesses*. The record assumes that counter 0 will be monitoring the number of cached accesses and counter 1 will be monitoring the total number of accesses. The statistics that are reported if this record

The Configfile reserved words are the following:

Name : Name of the statistic, with dots separating each menu subclass for the main user interface.

CounterMasks : Masks necessary for the relevant events; represented as hexadecimal values (preceeded by #x)

CounterOperation : Operations to be performed on the counter values; the set of operations allowed are described below

HeaderStrings : Headers describing each statistic that is obtained by computing the result of each CounterOperation

HistogramMask : Histogram Mask necessary for the relevant events;

HistogramHeaderStrings : Headers describing the result of the Histogram Operation

HistogramOperation : Currently, "Histogram" is the only operation allowed which reports the value of the histogram counters.

TimeFlag : TimeFlag = 0 means timeslicing is not implemented, TimeFlag = 1 means it is.

Help : Describes the details of what are available as a part of the statistic

Accumulator : Accumulator = 1 implies latency statistics are being computed, and counter zero will need to be used as an accumulator

GroupNames : Name of statistics group

StatisticsNames : If a statistics group has been defined, then the statistics constituting that group are referenced here

EndRecord : An "EndRecord" is placed at the end of each statistics record

EndOfFile : Needed at the end of the file

Figure 3.4: The configuration language.

List of Operations allowed by the Configuration File Language

Value : Takes 1 argument;
Reports value of the counter which is passed as the argument.

Div : Takes 2 arguments;
Reports result of dividing the value of the counter that is passed as the first argument by the value of the counter that is passed as the second argument.

Sum : Takes multiple arguments;
Adds all the counter values that are passed as arguments.

Mul : Takes multiple arguments;
Reports the product of all the counter values that are passed as arguments.

Sub : Takes 2 arguments;
Reports the difference of the 2 counter values that are passed as arguments.

DivMul : Takes 3 arguments;
Reports the result of multiplying the first argument (a number) with the result of dividing the value of the counter that is passed as the second argument by the value of the counter that is passed as the third argument.

Note : The Counter arguments are passed as numbers: 0, 1, 2 and 3 – referring to Counter 0, Counter 1, Counter 2, and Counter 3.

Figure 3.5: The operation keywords.

is chosen by the user are specified by the header strings: *number of cached data accesses, total number of data accesses* and *data cache hit ratio*. The CounterOperation keyword gives the operations required to get those statistics. For example, the number of cached data accesses is the value of counter 0 and the total data accesses is the value of counter 1. The data cache hit ratio is obtained by dividing the value of counter 0 with that of counter 1 and multiplying the quotient by 100. TimeFlag = 1 implies that this statistic is available in the timesliced mode as well.

The other 3 records provide resource specifications for Cached Unshared Data Accesses, Cached Local Shared Data Accesses and Cached Remote Shared Data Accesses.

The configuration file is read by a parser which gets all the arguments related to the sets of statistics that have been requested. It then passes that information on to the resource-allocator, which determines the number of runs required and assigns the masks for each run.

Currently the configuration file is hand-generated, thereby leaving room for errors. In the next implementation of QuickStep, we would like to modify the configfile language somewhat, so as to allow for a more automated procedure for generating the configfile.

3.3 The Machine Side

Since all the resource allocation information is processed by the host, the task on the machine side is very simple. The host passes all the information for a particular run in a message to the machine. The machine (kernel) configures the statistics counter masks and the histogram control mask accordingly[Figure 3.3]. It also clears all the counters. If timesliced mode is requested, then the timer is programmed to go off at regular intervals. Finally, the counters are enabled at the beginning of the run. If timesliced mode is off, then statistics are gathered at the end of the run and the data is sent back in packets to the host.

3.3.1 Data Collection and Reporting

When timeslicing is not used, the counter values are simply collected at the end of the run and sent back to the host in packets.

However, as described in Chapter 2, QuickStep offers the option of monitoring timesliced statistics. This feature is implemented by using a timer interrupt facility supported by the Alewife hardware. In our first implementation, the timesliced mode would cause an interrupt to happen at regular intervals. The interrupt handler would then disable all the

```
Name "Cache_Statistics.Data_Cache_Hit_Ratio"
CounterMasks #x201EFF20 #x201EFF3F
CounterOperation Value 0 Value 1 DivMul 100 0 1
HeaderStrings "#ofCachedDataAcc" "#ofDataAcc" "DCache-HR"
TimeFlag 1
Help "Offers 3 figures: Number of Cached Data Accesses,
   Number of Total Data Accesses, and
   Data Cache Hit Ratio"
EndRecord


Name "Cache_Statistics.Cached_Local_Local_Data_Accesses"
CounterMasks #x201EF320 #x201EF33F
CounterOperation Value 0 Value 1 DivMul 100 0 1
HeaderStrings "Cached Local-Local-Data Accesses"
              "Total Local-Local-Data Accesses"
              "Cached-Local-Local-Data"
TimeFlag 1
Help "Offers 3 figures: Cached Local-Local-Data Accesses,
   Total Local-Local-Data Accesses, and
   Cached Local-Local-Data Ratio"
EndRecord


Name "Cache_Statistics.Cached_Local_Global_Data_Accesses"
CounterMasks #x201EF520 #x201EF53F
CounterOperation Value 0 Value 1 DivMul 100 0 1
HeaderStrings "Cached Local-Global-Data Accesses"
              "Total Local-Global-Data Accesses"
              "Cached-Local-Global-Data"
TimeFlag 1
Help "Offers 3 figures: Cached Local-Global-Data Accesses,
   Total Local-Global-Data Accesses, and
   Cached Local-Global-Data Ratio"
EndRecord


Name "Cache_Statistics.Cached_Remote_Global_Data_Accesses"
CounterMasks #x201EF920 #x201EF93F
CounterOperation Value 0 Value 1 DivMul 100 0 1
HeaderStrings "Cached Remote-Global-Data Accesses"
              "Total Remote-Global-Data Accesses"
              "Cached-Remote-Global-Data"
TimeFlag 1
Help "Offers 3 figures: Cached Remote-Global-Data Accesses,
   Total Remote-Global-Data Accesses, and
   Cached Remote-Global-Data Ratio"
EndRecord
```

Figure 3.6: A sample configuration file.

statistics and histogram counters and a packet reporting the counter/histogram information would be sent to the host. This protocol created a problem since whenever a large number of processors were involved, too many messages were going towards the host, thereby clogging up the network. We solved this problem by buffering the statistics counter values in arrays of structures (described in Figure 3.7). Note, the statistics counters are 32-bits in length and the histogram registers are 20-bits in length. However, additional overflow handling mechanism implemented in software provides 64-bit resolution for each register. Hence, when the counter values need to be stored, both the upper 32-bits and the lower 32-bits need to be recorded. The data-structures shown in Figure 3.7 demonstrate provisions for handling this task. We staggered the reporting of data to the host by ensuring that no two processors are reporting data during the same timeslice and thereby lightened the load on the network. The number of messages was reduced by sending large chunks of the array in a single message.

3.3.2 Instrumentation Overhead due to TimeSlicing

Interrupting a program every so often is expected to perturb a program in terms of memory behavior, execution times, etc. We have done an experiment with perturbation characteristics regarding execution times. Figure 3.8 shows the results of the experiment. We ran 3 applications (namely, *Water, SOR* and *Mergesort*) first without timeslicing, and then timeslicing with decreasing intervals (ie. increasing number of times the system is interrupted). We found that the slowdown factor (ratio of execution time *with* timeslicing to execution time *without* timeslicing) is nominal for upto about 180 interruptions. It was not possible to get measurements with higher numbers of interruptions because the current network overflow software for Alewife cannot handle a higher degree of network congestion.

In our implementation of the timesliced mode of statistics gathering, we have faced some problems. When data reporting messages are sent to the host too often, the network clogs up. However, if very large buffers are used for storing the data, then the memory requirement on each processor limits the memory available to the application, and hence causes capacity misses, thereby deteriorating its performance. There are a couple of solutions to this problem:

- Increasing the memory on each node
- Adding I/O nodes to decrease network congestion

We expect these features to be available in later versions of Alewife.

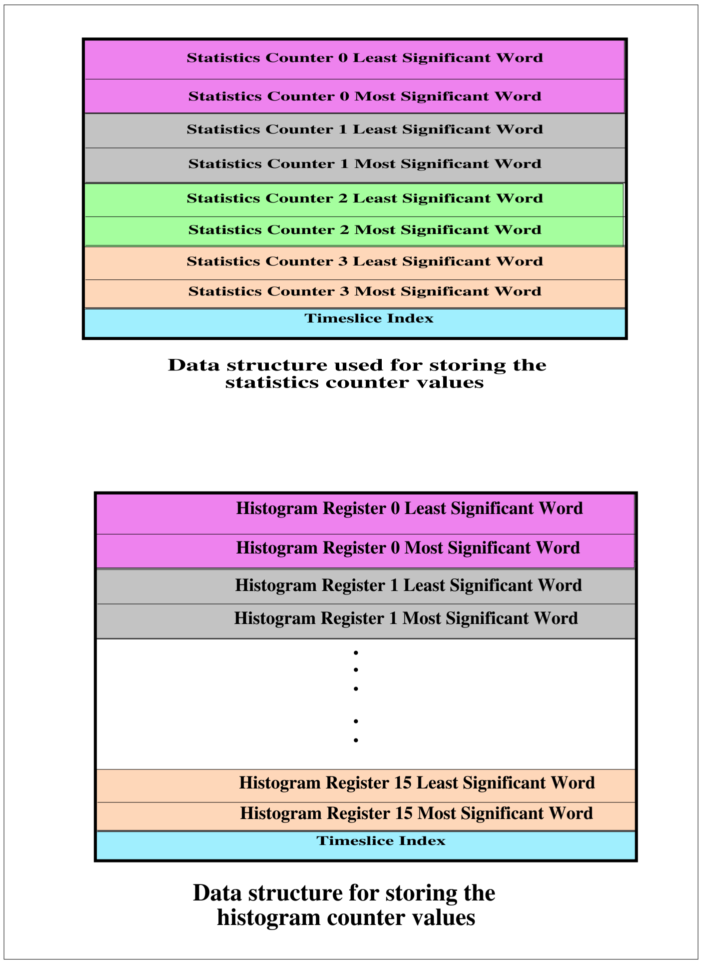

Figure 3.7: Data structure for storing counter values.

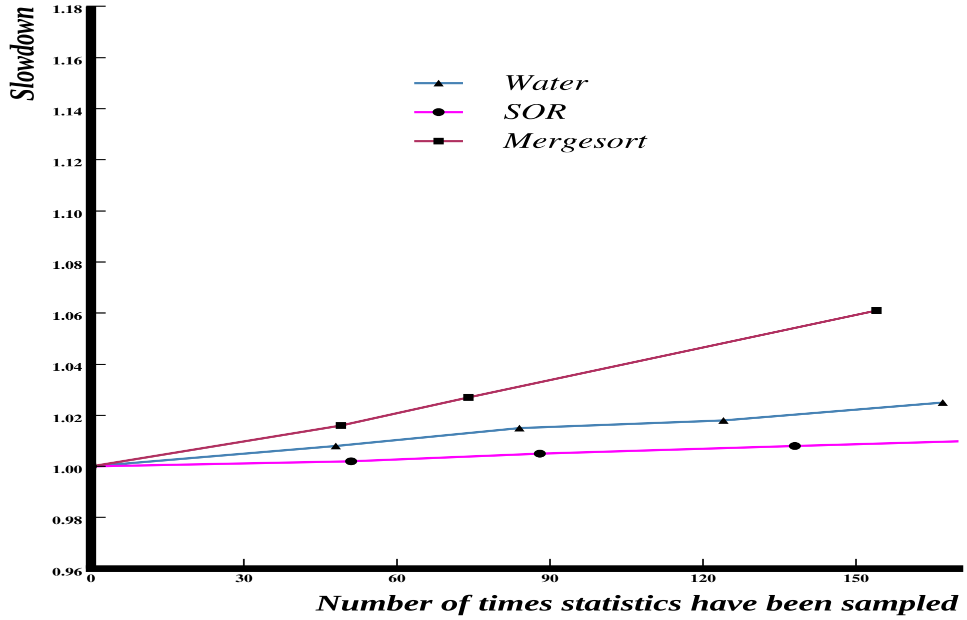

Figure 3.8: Instrumentation overhead due to timeslicing: Monitoring timesliced data and instruction cache hit ratios for 3 applications.

3.4 Post-Processing

The data packets are received by the host and stored in memory until the end of the run. At the end of the run, they are output into a raw data file in a simple column format. A sample data file is given in Figure 3.9. The raw file is then processed to generate a binary file that is in the Proteus [5] trace file format, that can be viewed with a graphical interface supported by the Proteus Stats program. Chapter 2 shows examples of graphs obtained as outputs of the QuickStep system. The column headings from the raw data file are used to generate headings and menus for the graphs. The graphs give approximate ranges that are helpful as an easy-to-grasp summary. However, the datafile values are available if a user would like to look at more precise statistics. The *Index* column represents the processor number and the *timestamp* field represents the timeslice id. [In Figure 3.9, a small program was run on a 16-node Alewife machine and only overall statistics were gathered.]

```
CntRecord
"#ofCachedDataAcc" "#ofDataAcc" "DCache-HR" "Cached Local-Local-Data Accesses"
"Total Local-Local-Data Accesses" "Cached-Local-Local-Data" Index  Timestamp
331    332    99.69879518072288 331    332    99.69879518072288 3  0
330    331    99.69788519637463 330    331    99.69788519637463 6  0
330    331    99.69788519637463 330    331    99.69788519637463 9  0
332    333    99.69969969969969 332    333    99.69969969969969 4  0
329    330    99.69696969696969 329    330    99.69696969696969 7  0
329    330    99.69696969696969 329    330    99.69696969696969 12 0
330    331    99.69788519637463 330    331    99.69788519637463 10 0
331    332    99.69879518072288 331    332    99.69879518072288 5  0
328    329    99.69604863221885 328    329    99.69604863221885 13 0
328    329    99.69604863221885 328    329    99.69604863221885 14 0
332    333    99.69969969969969 332    333    99.69969969969969 2  0
327    328    99.6951219512195  327    328    99.6951219512195  15 0
331    332    99.69879518072288 331    332    99.69879518072288 8  0
329    330    99.69696969696969 329    330    99.69696969696969 11 0
333    334    99.7005988023952  333    334    99.7005988023952  1  0
EndRecord
```

Figure 3.9: Excerpt from a sample data file.

3.5 Summary

User-friendliness is the main principle that has been followed in the design of QuickStep. We have also ensured that it is easy to add new records to the configuration file for monitoring new statistics. Another design principle that we have followed is to keep most of the task of resource allocation outside the main kernel. Consequently, the resource allocation is done on the host side and minimum amount of work is left for the kernel. Of course, the actual reading and storing of counter values is done in the kernel.

Chapter 4

Validation of the System

4.1 Overview of the Benchmark Suite

The **QuickStep** system can be used to obtain various statistics. However, the statistics can be utilized to analyze and fine-tune performance of applications only if the system has been validated and there is some guarantee that the data is authentic. For this purpose, a suite of small programs with predictable behavior has been put together. This set of synthetic benchmarks have been run on the Alewife machine and the statistics gathered have been found to tally with the expected figures. In the next section, we give examples of some of the benchmark programs and the output graphs that verify the correctness of the corresponding statistics class.

4.2 Examples

4.2.1 Example 1: Cached Reads

Bench1.c from Appendix A is an example of a synthetic program that is used to verify the statistic *cached reads*. It is run with arguments 40 as the probability of misses and 10000 as the loopbound. The expected hit ratio for read accesses is 60%. Table 4.1 shows the values obtained from the data file in which the result of monitoring cached reads are recorded [Results from two separate runs are presented]. As expected, the cached read ratio does turn out to be around 60%. The variation is due to the fact that there is a brief shutdown phase at the end of each program which causes a few extra accesses, thereby introducing a slight inaccuracy. Since the statistics counters are user programmable, it would be easy to turn them on and off right before and after the critical loop, thereby getting the hit ratios for the loop only. However, this would involve not using the capabilities offered by QuickStep.

4.2.2 Example 2: Remote Accesses

Bench5.c from Appendix A is another synthetic benchmark which flushes the cache of the local processor before each read access [$j = tmp- > d1$]. The actual data resides in processor 0 and hence every access in the first loop is a remote access. For instance, if the program is run with arguments 40 and 10000, 4000 of those accesses ought to be remote accesses on all the processors except on processor 0. Furthermore, the number of cached remote accesses ought to be 0 on every single node. We do see this behavior in the graphs obtained by running QuickStep on the program (graphs are not included).

We also use this program to validate histograms of remote distances. For instance, figure 4.2 shows that each of the other processors have sent out 4000 Read Request (RREQ) packets to processor 0 and are represented in the graph according to the number of hops they each have travelled. Figure 4.1 shows the numbering scheme of processors on the mesh, from which we see that processors 1 and 2 are 1 hop away from node 0, processors 3, 4 and 8 are 2 hops away, processors 5, 6, 9 and 10 are 3 hops away, processors 7, 11 and 12 are 4 hops away, processors 13 and 14 are 5 hops away and processor 15 is 6 hops away from processor 0. Figure 4.2 reflects this information by showing, for example, processors 5, 6, 9 and 10 have sent out 4000 packets each that have travelled 4 hops.

Figure 4.3 shows the reverse of figure 4.2 in that it shows that 8000 packets have travelled 1 hop, 12000 packets have travelled 2 hops, 16000 packets have travelled 3 hops, 12000 packets have travelled 4 hops, 8000 packets have travelled 5 hops and 4000 packets have travelled 1 hop from processor 0, to go out to the caches of the other processors carrying the data requested by each of them. Each node had sent out 4000 read requests and 2 of these are 5 hops away from processor 0. The rest of the data represented in the two graphs is also consistent.

4.2.3 Example 3: Timesliced Statistics

Bench11.c from Appendix A is a modified version of bench5.c in which the first and third loops of reads do not require any remote access, however, all accesses in the second and fourth loops are to data stored in the memory of processor 0. Consequently, when the program is run, the output reflects this information in Figure 4.4. The program was run with arguments 1 and 1000. Hence, initially, for all processors except node 0, 900 accesses are

Run Id	Cached Read Ratio	Processor Number	Mean
1	60.43912570467808	3	60.5046871656633
	60.41563582384958	6	
	60.60665747488675	4	
	60.41563582384958	9	
	60.396039603960396	7	
	60.39211803148826	12	
	60.57948162018331	5	
	60.396039603960396	11	
	60.36857227781631	13	
	60.36857227781631	14	
	60.68796068796068	2	
	60.344998512937444	15	
	60.664765463664075	8	
	60.64154285152023	10	
	60.62604587065656	1	
	60.73180302138513	0	
2	60.43912570467808	3	60.50713018408938
	60.41563582384958	6	
	60.60665747488675	4	
	60.41955274094597	9	
	60.396039603960396	7	
	60.39996039996039	12	
	60.57948162018331	5	
	60.396039603960396	11	
	60.3724985139687	13	
	60.3724985139687	14	
	60.68796068796068	2	
	60.35285955000496	15	
	60.66863323500492	8	
	60.645415190869734	10	
	60.629921259842526	1	
	60.73180302138513	0	

Table 4.1: Results of running bench1.c on a 16-node Alewife machine.

2A	2B	2E	2F	3A	3B	3E	3F
28	29	2C	2D	38	39	3C	3D
22	23	26	27	32	33	36	37
20	21	24	25	30	31	34	35
0A	0B	0E	0F	1A	1B	1E	1F
08	09	0C	0D	18	19	1C	1D
02	03	06	07	12	13	16	17
00	01	04	05	10	11	14	15

Figure 4.1: Numbering scheme for the mesh of Alewife nodes.

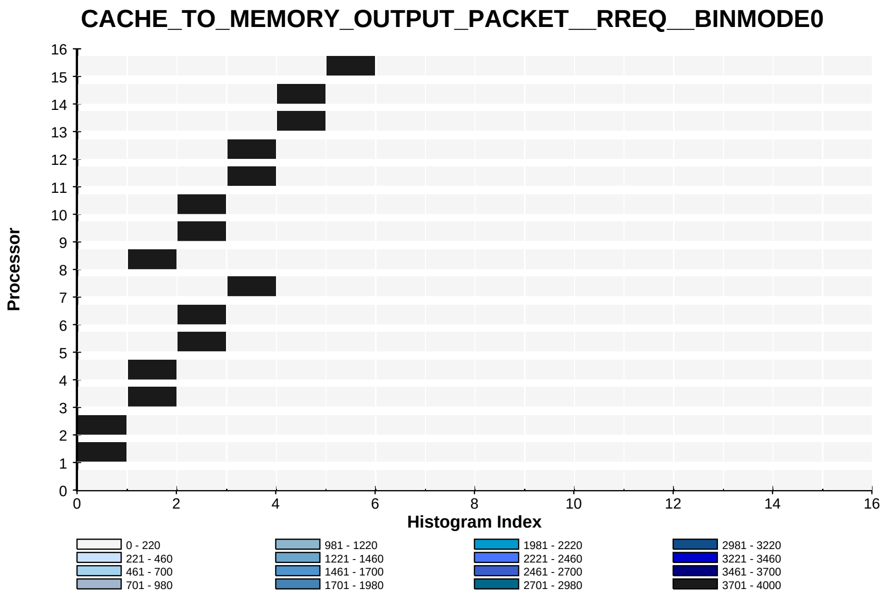

Figure 4.2: Bench5.c on 16 processors: Per processor distribution of distances travelled by RREQ packets going from caches of each processor to the memory of processor 0.

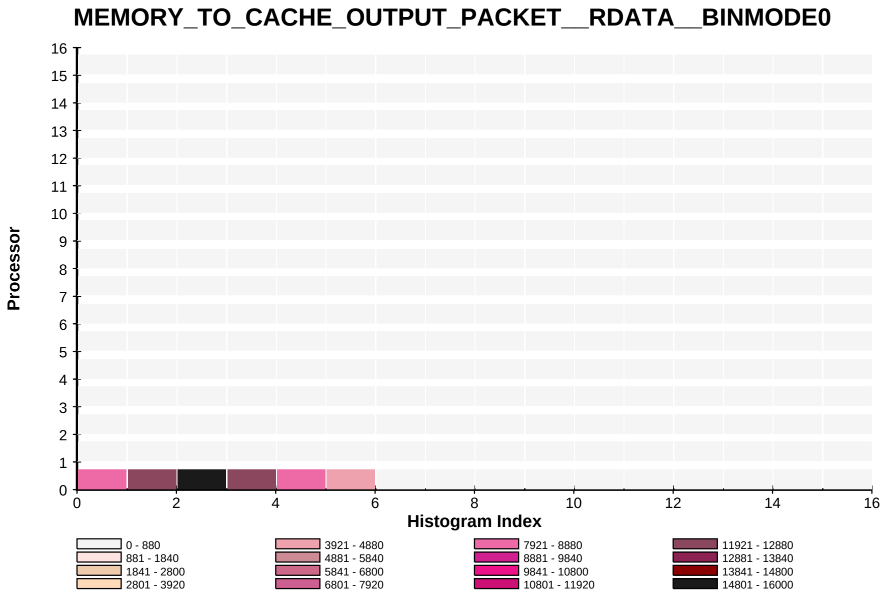

Figure 4.3: Bench5.c on 16 processors: Per processor distribution of distances travelled by RDATA packets going from the memory of processor 0 to the caches of each processor.

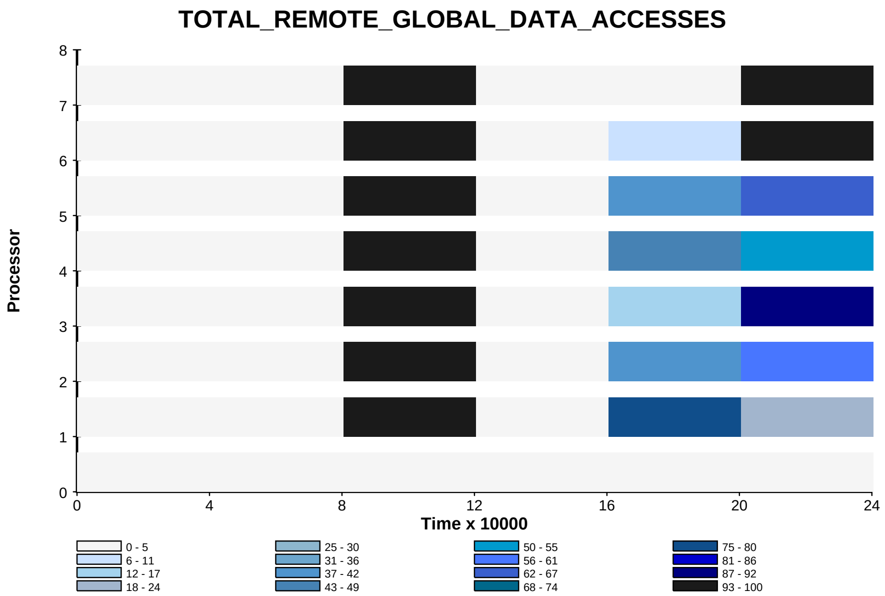

Figure 4.4: Bench11.c on 8 processors: Per processor distribution of remote accesses over time.

local and 100 accesses are remote. Then, another 900 local accesses take place, followed by a second set of 100 remote accesses. In the graph, we see the remote accesses showing up in the third, fifth and sixth timeslices. The first timeslice covers the initialization phase, the second timeslice covers the first loop, the fourth timeslice (in case of processor 7, the fourth and fifth timeslices) cover the third loop.

4.3 Summary

The different modules of QuickStep have been tested individually. Figure 4.5 gives a summary of the status of testing. The hardware (module 1) has been tested during the simulation and testing phase of the CMMU. Modules 2 through 8 have been tested by printing out the output of each module on a phase by phase basis and comparing them with results obtained by doing the same task by hand. Finally, the validation suite has been used to test the authenticity of the actual statistics.

The synthetic benchmark suite that is used to validate the different statistics supported by QuickStep, however, is by no means complete, since the number of statistics available is huge. Only a small subset of these have been validated. The classes of statistics that have been tested include cache statistics, histograms of remote distances and latency statistics, although not all subclasses have been validated under these categories. The validation suite covers a sample of statistics from each of these classes. We have done adequate validation and testing to think that the hardware is counting the right events and that the full vertical slice of the software is processing the information correctly. Of course, bugs are likely to be present in the system. We expect to get more feedback and bug-reports from the users of the system, which will make the system increasingly solid.

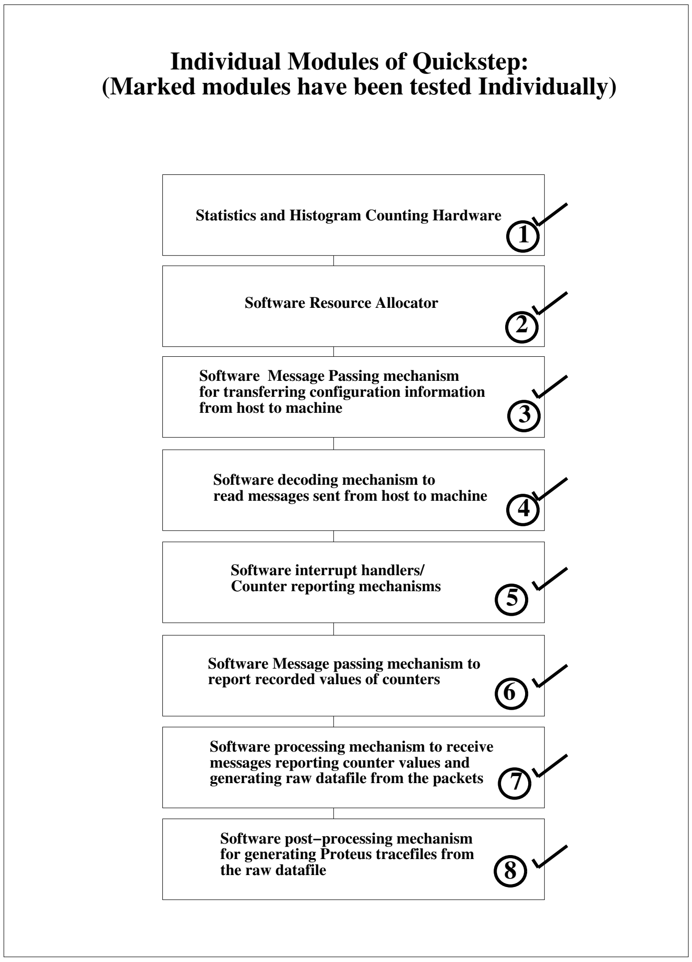

Figure 4.5: Status of validation and testing of QuickStep.

Chapter 5

Case Studies Using QuickStep

QuickStep has been developed as a vehicle for analyzing parallel program performance on the Alewife multiprocessor system. The main goal of the system is to aid in identifying performance bottlenecks and tuning programs to reduce the effect of such bottlenecks. Furthermore, it can also be used to analyze effects of optimization of application code. We use **MP3D** and **SOR**— two large parallel applications [3] to demonstrate appilcations of Quickstep.

5.1 Case Study 1: MP3D

5.1.1 Description

In this chapter, we use MP3D— an application from the SPLASH suite to demonstrate how QuickStep provides useful insight into program behavior as a result of optimization. Mp3d simulates the interactions between particles flowing through a rectangular wind tunnel and a solid object placed inside the tunnel. The tunnel is represented as a 3D space array of unit-sized *cells*. Particles move through the space array and can only collide with particles occupying the same cell in the same time step. A complete description of this program can be found in [21].

Ricardo Bianchini has done a study on the performance of large parallel applications on Alewife. Ricardo's study includes experimentation with multiple implementations of Mp3d. In this chapter, we have used three different implementations of Mp3d and run each on a 16-node Alewife machine, with 18000 particles for 6 iterations. A 15MHz clock has been used for each set of runs.

The 3 versions of Mp3d that will be compared are described below:

1. *Orig Mp3d*: This is the original SPLASH implementation.
2. *Mp3d*: This is a modified version of the original program in which some useless code (variables updated but never read) has been eliminated.
3. *MMp3d*: This is another modified version in which the partitioning of the data has been altered. This version reduces sharing by allocating particles to processors in such a way that a certain processor's particles rarely move through cells used by other processors.

5.1.2 Analysis Using QuickStep

This section compares the 3 versions of Mp3d based on some specific statistics: cache hit ratios, invalidations, remote accesses, etc. We will show how the modifications affect the original program using QuickStep.

Data Cache Hit Ratios

Program	Data Cache Hit Ratio Range
Orig Mp3d	93%
Mp3d	95-96%
MMp3d	96-97%

Table 5.1: Average data cache hit ratios for running the 3 versions of Mp3d on a 16-node Alewife machine.

Table 5.1 shows the data cache hit ratios across processors for *Orig Mp3d*, *Mp3d* and *MMp3d* respectively. Orig Mp3d has an average data cache hit ratio of 93%, while Mp3d and MMp3d have hit ratios of 95-96% and 96-97% respectively. Although the improvement is marginal, the modifications do enhance cache performance.

Read Invalidations

Figures 5.1, 5.2 and 5.3 show the distances travelled by memory-to-cache read invalidation packets for *Orig Mp3d*, *Mp3d* and *MMp3d* respectively. The y-axis has the processor number and the x-axis has the number of hops travelled by packets before coming into a processor. The colors represent the number of packets that have travelled that many hops.

The overall conclusion that can be drawn from these graphs is that the number of invalidation packets that are floating around in the system is much greater in the case of

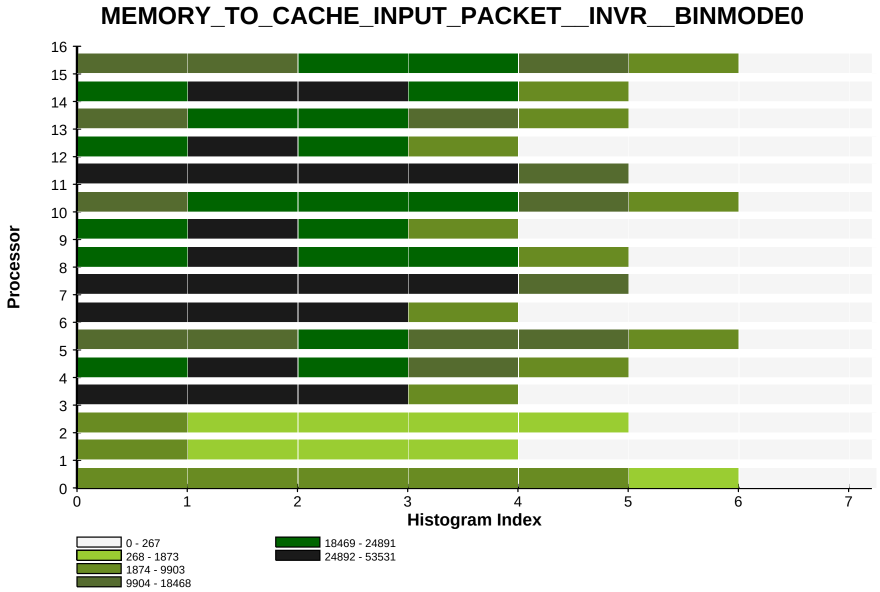

Figure 5.1: Orig Mp3d: Per processor distribution of remote distances travelled by read invalidation packets [Table B.4].

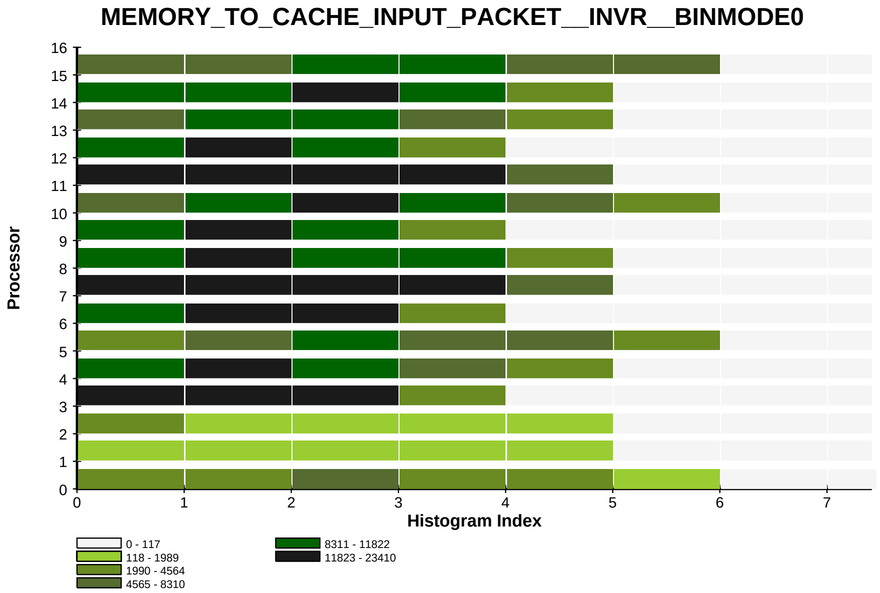

Figure 5.2: Mp3d: Per processor distribution of remote distances travelled by read invalidation packets [Table B.5].

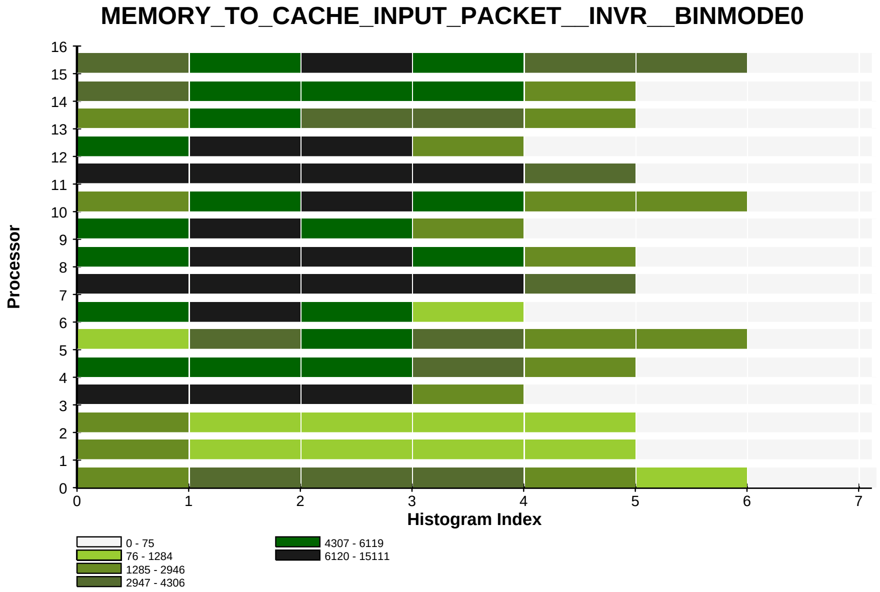

Figure 5.3: MMp3d: Per processor distribution of remote distances travelled by read invalidation packets [Table B.6].

Orig Mp3d than *Mp3d* and *MMp3d* performs the best in this regard. This is expected since the modification in *MMp3d* is designed to reduce sharing of data. From a locality point of view, in all 3 cases, the majority of the packets travel either 2 or 3 hops. However, in absolute terms, the number of packets in the system is lower in case of *MMp3d* than *Mp3d*, which in turn is lower than *Orig Mp3d*.

Remote Access Latencies

Figures 5.4, 5.5 and 5.6 show the average of remote access latencies for *Orig Mp3d*, *Mp3d* and *MMp3d* respectively. For *Orig Mp3d*, latencies are in the range 53-66 cycles. For *Mp3d*, average latencies vary between 54-72 cycles. For *MMp3d*, latencies range between 34-56 cycles. Hence, it can be concluded that taking out the useless code sections does not affect latencies, but the modification in the data partitioning does have a significant effect.

Data Distribution

Figure 5.7 represents the data distribution graphs that have been used to obtain results presented in Table 5.2. Figure 5.7 shows the percentage of data accesses that are to remote

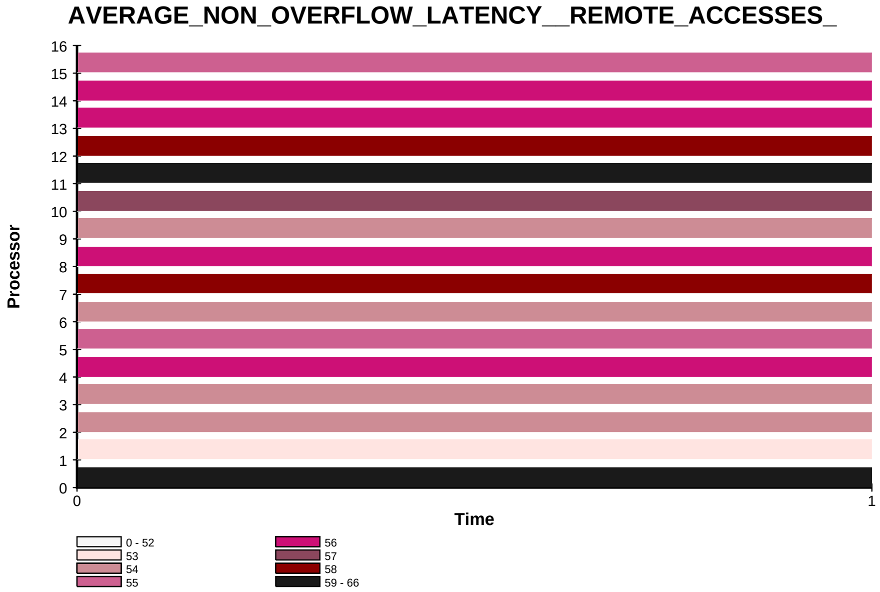

Figure 5.4: Orig Mp3d: Average remote access latencies [Table B.7].

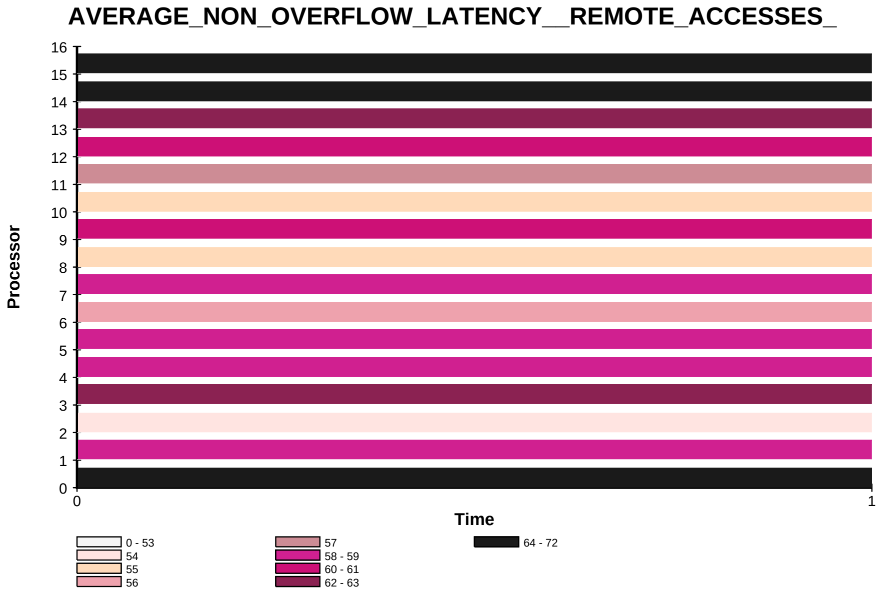

Figure 5.5: Mp3d: Average remote access latencies [Table B.8].

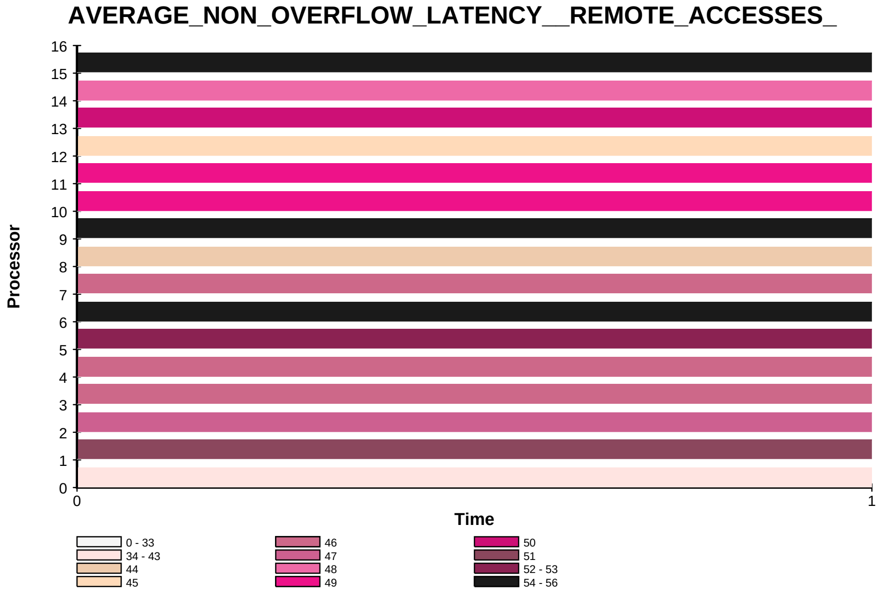

Figure 5.6: MMp3d: Average remote access latencies [Table B.9].

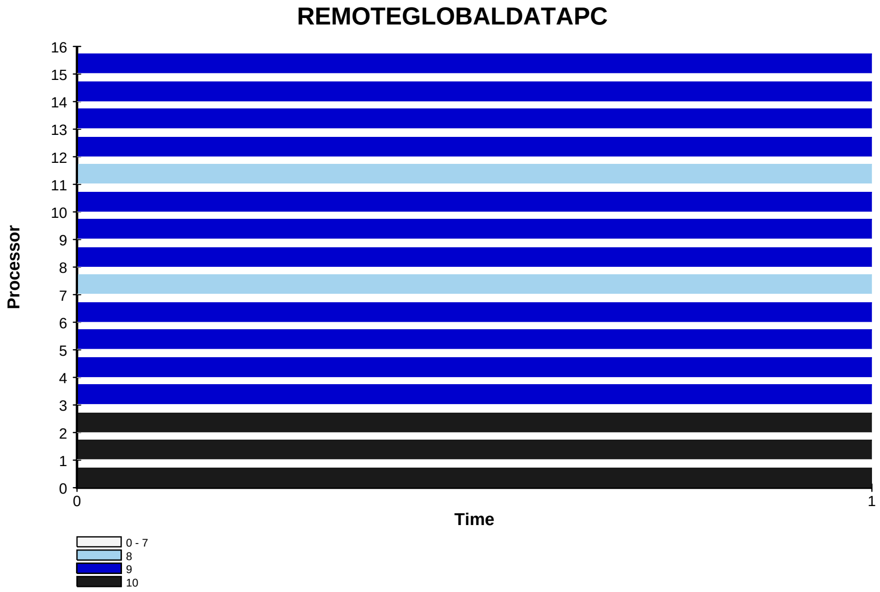

Figure 5.7: Orig Mp3d: Percentage of remote global accesses.

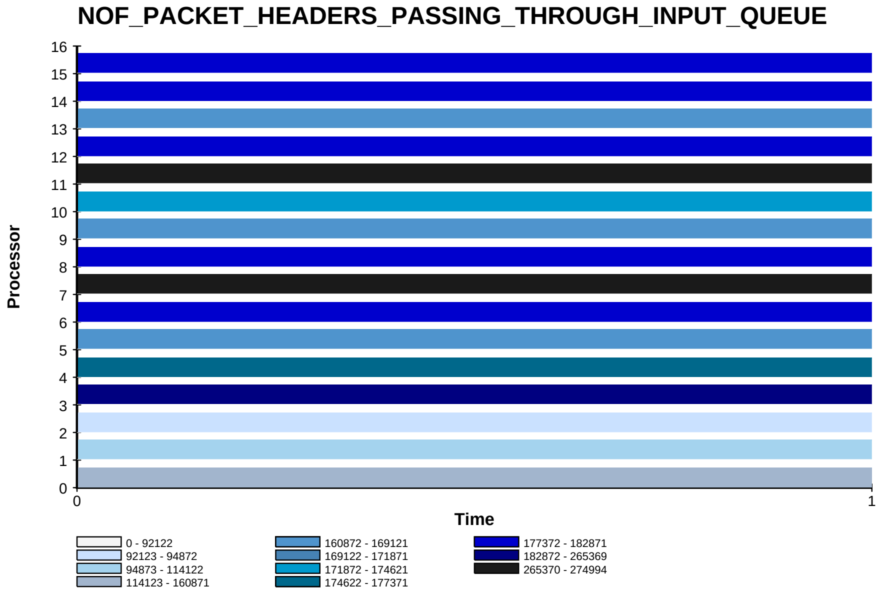

Figure 5.8: Orig Mp3d: Packet headers passing through output queue [Table B.10].

shared memory for each processor. Table 5.2 summarizes the data collected from several graphs of the same type. The table shows that a higher percentage of data accesses are to remote shared memory in case of *Orig Mp3d*. Data locality is much better exploited in the modified versions of Mp3d than in the original version.

Program	% of Remote Shared Accesses	% of Local Shared Accesses	% of Unshared Accesses
Orig Mp3d	8-10%	36-38%	51-53%
Mp3d	4-5%	33-35%	58-60%
MMp3d	4-5%	32-35%	58-61%

Table 5.2: Data distribution for running the 3 versions of Mp3d on a 16-node Alewife machine.

Network Traffic

Figures 5.8, 5.9 and 5.10 show the network traffic passing through output queues for *Orig Mp3d*, *Mp3d* and *MMp3d* respectively, as a function of processor number. For *Orig Mp3d*, traffic is much higher (92,000-280,000 packets). For *Mp3d* and *MMp3d*, traffic is a lot lower, ranging between 46,000-124,000 packets and 27,000-63,000 packets respectively. Hence, network performance is significantly affected by the modifications.

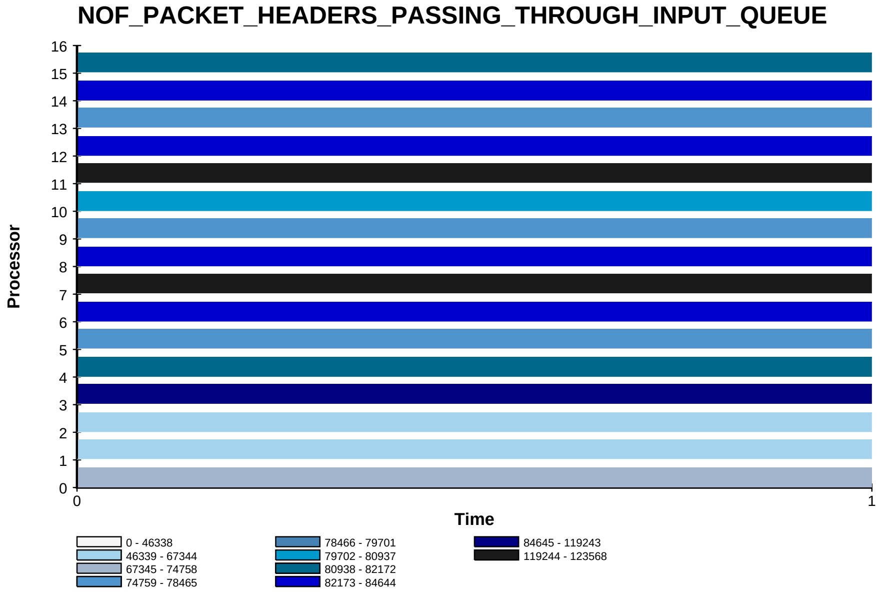

Figure 5.9: Mp3d: Packet headers passing through output queue [Table B.11].

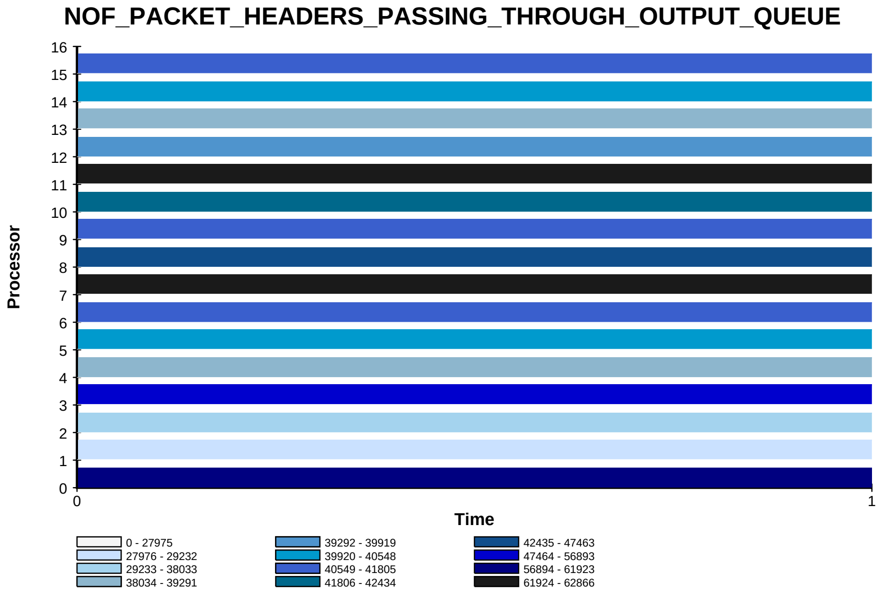

Figure 5.10: MMp3d: Packet headers passing through output queue [Table B.12].

5.1.3 Summary

We conclude that the performance of Mp3d is enhanced by the modifications in all the categories that we consider. *Mp3d* and *MMp3d* do not perform much differently as far as data distribution is concerned, but *MMp3d* performs significantly better than Mp3d as far as network traffic, remote access latencies and read invalidations are concerned. Data cache hit ratios show only marginal variation.

5.2 Case Study 2: SOR

5.2.1 Description

SOR (Straightforward 2D successive over-relaxation) can be used to compute the successive over-relaxation of the temperature of a metal sheet. In each step of the algorithm, the temperature of a certain point in the sheet is calculated based on the temperature of its neighboring points. This application is implemented using two matrices (called "odd" and "even") for representing the metal sheet. During even-numbered steps, processors read from the odd matrix and write to the even matrix. During odd-numbered steps, processors read from the even matrix and write to the odd matrix.

In the next section, we will analyze the behavior of SOR based on cache miss ratios and show how we achieved improved performance using QuickStep. All experiments correspond to a 256 x 256 SOR running for 10 iterations.

5.2.2 Analysis Using QuickStep

In the first version of SOR that we have considered (henceforth referred to as *zgrid*), processors take turns in getting a row index from a central pool. The index represents the next row the processor must update. Figure 5.11 shows the data cache hit ratios for this implementation. We see that *zgrid* achieves good load balancing but lousy locality [data cache hit ratios are in the 69-86% range], as processors are likely to update a different set of rows in each phase of the computation.

We identified the source of bad performance and changed the way row indices are assigned in the next version of SOR (*mgrid*) to a round-robin scheme. As a result, locality improved significantly [data cache hit ratios shot up to the 92-97% range], as is demonstrated in Figure 5.12. However, we found that there is still room for further improvement.

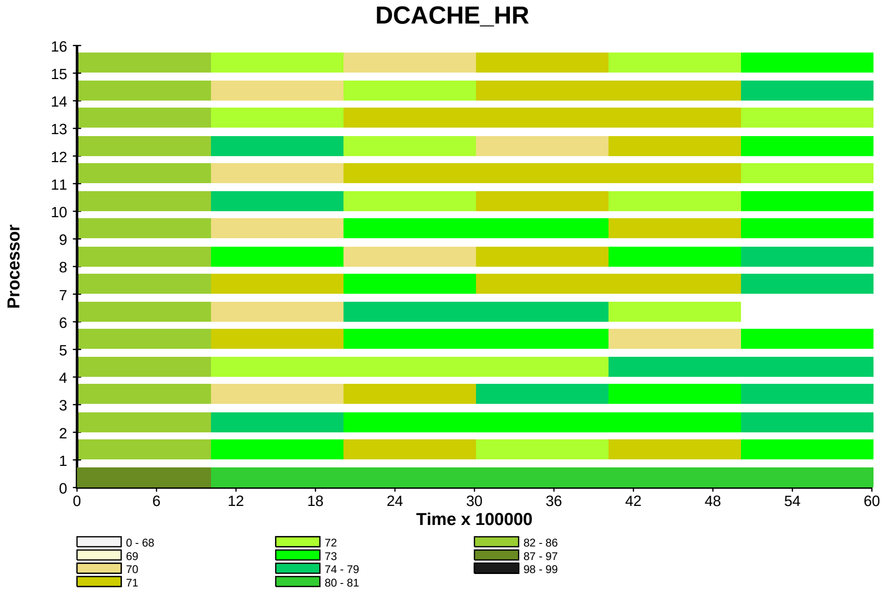

Figure 5.11: ZGRID: Data cache hit ratios [Table B.13].

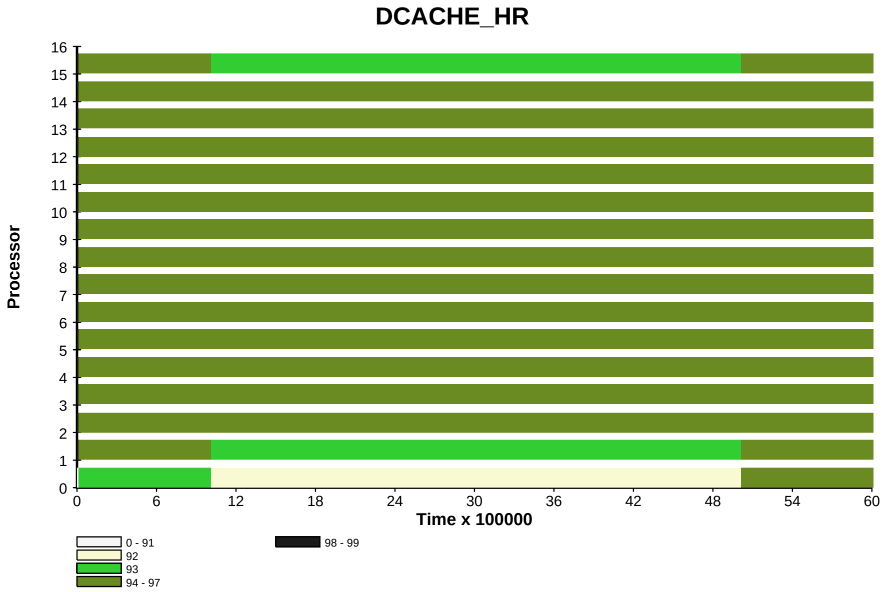

Figure 5.12: MGRID: Data cache hit ratios [Table B.14].

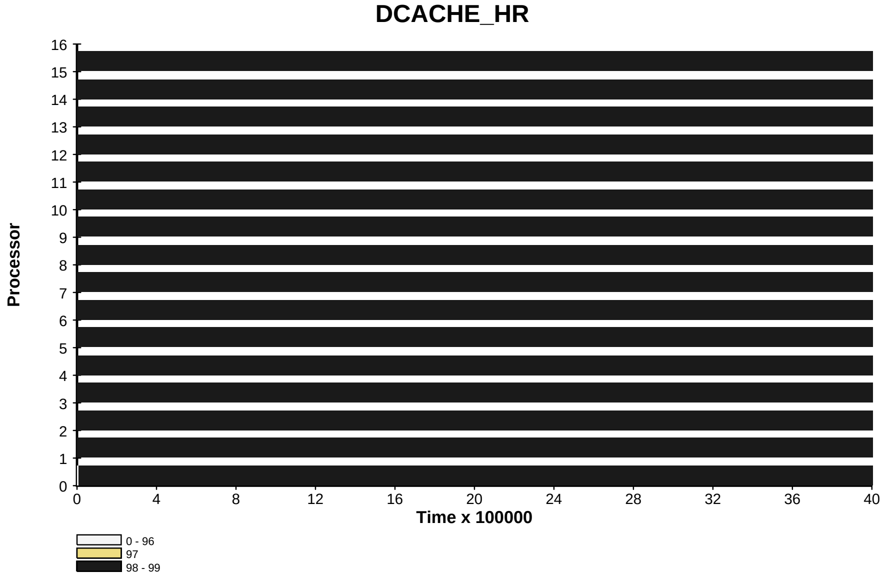

Figure 5.13: CGRID: Data cache hit ratios [Table B.15].

We did a coarse-grained implementation of SOR (*cgrid*), assigning a group of consecutive rows of each of the matrices to each processor. Since this row assignment persists throughout the whole computation, this implementation exhibits excellent locality of reference [data cache hit ratios are in the 98-99% range (Figure 5.13)]. Only the boundary rows must be communicated between neighboring processors once the cold start misses are over.

5.2.3 Summary

Program	Execution Times
Zgrid	5542931 cycles
Mgrid	5200530 cycles
Cgrid	3565709 cycles

Table 5.3: Execution times for the three versions of SOR on a 16-node Alewife machine.

In this section, we have used QuickStep to identify performance problems in SOR and subsequently modified the program to achieve better performance. Table 5.3 summarizes the improvement in overall performance by showing the execution times for each version of SOR.

Chapter 6

Conclusions

6.1 Summary

The **QuickStep** performance monitoring system has been developed to aid performance tuning and debugging of parallel applications on the Alewife multiprocessor. With the CMMU providing hardware instrumentation support, general instrumentation overhead is low, although not absent. The main limitation of QuickStep comes from the fact that there are only 4 statistics counters and if more than 4 events are to be monitored, then the events have to be split over several runs. The strength of the system, however, is that it is smart enough to handle the resource allocation process efficiently and without any help from the user.

The members of the Alewife team are presently using QuickStep in various areas like parallel application studies, compiler research, etc. The system will also aid in analyzing/justifying the design decisions made during the Alewife design process and evaluating architectural design trade-offs for our next machine. The general feedback is that people have found it easy to use, and the information gathering capabilities have been found to be useful. Suggestions for additional features have been outlined in the next section. With those features, QuickStep will be a more powerful vehicle for performance monitoring and debugging for the Alewife family of machines.

As long as a system has some means of gathering the statistics that we have discussed in this thesis through either software or hardware instrumentation, the higher level design principles of the QuickStep system are quite general. For instance, the resource allocator, the configuration file and parser and the message-passing platform are all general concepts that can be used in any system of a similar nature. If statistics are gathered by using software instrumentation, obviously, they would be less accurate. Hence, if making decisions about the hardware design is an option and if the designers know in advance that providing performance optimization support is a goal of the system, then having dedicated hardware for monitoring statistics is a very good idea. No other method would be as accurate and non-intrusive. Thus, the accuracy of the reported statistics is the most impressive feature of the QuickStep system.

6.2 Future Work

As mentioned in Chapter 2, there are several features that we would like to add to the system. In this section, we discuss those features, along with some details about the support that the current version of QuickStep provides to enable a reasonably easy implementation of those features.

- **Execution Profiler**: User feedback has indicated the need to be able to profile programs on a per procedure basis. For instance, it would be convenient to be able to monitor statistics for a particular procedure. It would also be useful to get Gprof like graphs describing how much time has been spent in each procedure. The checkpoint feature already provides an equivalent of the call count feature of Gprof.

  QuickStep already has provisions for allocating resources and configuring statistics counter masks. The statistics counters can be enabled and disabled by the user. Therefore, in principle, all that is required is a couple of procedures encapsulating the different operations which can be included at the beginning and end of the piece of code to be profiled. Ideally, we would like to have the compiler/linker do this automatically and the user would only need to specify the name of the procedure he/she would like to profile. We have the string translation and message decoding mechanisms that would be required to support this feature. The compiler/linker support can also be provided.

- **Additional Statistics Classes**: The major function specifier field of the statistics counter masks represent the following classification of statistics (This classification is how the hardware views the statistics classes and is therefore somewhat low-level):

  1. Checkpoints
  2. Processor Bus Traffic
  3. Synchronous Statistics

4. Hit-Miss Scheduling
5. Remote-TXN Statistics
6. Memory-side Statistics/Scheduling
7. Multiphase Transactions
8. Transaction Buffer Statistics
9. Network Statistics

Out of these, major functions 1, 2 and 9 have been implemented. Major function 7, together with the histogram registers constitute the latency statistics, which have also been implemented. Major functions 3, 4, 5, 6 and 8 have not been implemented as yet, although all that is required is to set up the appropriate resource records in the configuration file. The resource allocation, mask configuration, data collection, etc. are exactly similar to the ones that have been implemented.

- **Lifeline Graph**: A lifeline graph representing what the processor is spending its time on at a given point in time will be a nice addition to QuickStep's repertoire. However, this is a complicated addition and does not fit as readily into the model that has been developed so far.

- **Display Environment**: At present, the display environment for QuickStep is the Proteus Stats program. Stats does offer quite a flashy and sophisticated graphical display in color. But, a lot of the data generated by QuickStep would be easier to read with a full-fledged 3-dimensional display. In the next version, it would be useful to spend some time on this issue.

  Since a raw datafile is generated anyway and the Proteus trace file is generated from the datafile, there is every reason to believe that writing a different filter to post-process the datafile will be easy. Therefore, whatever format the new display environment may require the data to be in can be easily incorporated.

- **Load Balancing Aid and Synchronization Time Monitor**: These two features have been requested by users, although the hardware is not capable of supporting them. Future modifications to the hardware may be able to incorporate these features.

Appendix A

Code Listings

A.1 Proc1: A Procedure Annotated with Checkpoints

```
void proc1(int miss_prob, int loopbound)
{
  int i, j;
  int k, l;

  checkpoint("Checkgr2:Check1");
  k = miss_prob*my_pid();
  l = (loopbound-k);
  for (i = 0; i < k; i++){
    checkpoint("Checkgr2:Check2");
    softflush(&num);
    num = i;
  }
  checkpoint("Checkgr2:Check3");

  checkpoint("Checkgr3:Check1");
  for (i = 0; i < l; i++){
    checkpoint("Checkgr3:Check2");
    num = i;
  }
  checkpoint("Checkgr3:Check3");
}
```

A.2 Bench1.c: A Program for Validating Hit Ratios

```
/*
   Sramana Mitra
   September 26, 1994
   bench1.c: A synthetic benchmark program to
   test cache hit-miss statistics; verifiers for
   cached reads and overall data cache hit-ratio
   statistics.
*/

#include <primops.h>
#include <stdio.h>
#include <parallel.h>

int num = 9;
void proc1();
void clear_counters();

main(int argc, char *argv[])
{
  int missprob, loopbound;

  missprob = atoi(argv[1]);
  loopbound = atoi(argv[2]);

  do_in_parallel(proc1, missprob, loopbound);
}

void proc1(int miss_prob, int loopbound)
{
  int i, j;
  int k, l;

  k = (int)((float)miss_prob*0.01*(float)loopbound+0.5);
  l = (loopbound-k);
  /*
  Counters are cleared so that the data can be obtained
  for this section of the code only.
  */
  clear_counters();
```

```
  for (i = 0; i < k; i++){
    softflush(&num);
    j = num;
  }
  for (i = 0; i < l; i++)
    j = num;
}

void clear_counters()
{
  int i;
  unsigned OldCCR = CReg->ContCR;

  trap(SUPERVISOR_MODE_TRAP);
  disint(STATISTICS_MASK);
  clrstat(CLEAR_STAT_CNT_ALL_MASK);
  for (i = 0; i < 4; i++) {
    CReg->StatArray[i].Count = 0;
  }
  slink->StatCnt0_MSW = 0;
  slink->StatCnt1_MSW = 0;
  slink->StatCnt2_MSW = 0;
  slink->StatCnt3_MSW = 0;
  CReg->ContCR = OldCCR;
}
```

A.3 Bench5.c: A Program for Validating Remote Access Patterns

```
/*
   Sramana Mitra
   September 27, 1994
   bench5.c: A synthetic benchmark program to test
             uncached remote accesses. The cache hit
             ratios on remote accesses ought to be 0
             due to flushing of the cache before each
             access on every node except 0.
*/
```

```
#include <primops.h>
#include <stdio.h>
#include <parallel.h>

int num = 9;
void proc1();
/* struct for dynamic allocation */
typedef struct dummy {
    unsigned  d1;
    unsigned  d2;
    unsigned  d3;
    unsigned  d4;
    unsigned  d5;
} DUMMY;
typedef DUMMY *DUMMY_PTR;


main(int argc, char *argv[])
{
  int missprob, loopbound;
  DUMMY_PTR  temp;

  missprob = atoi(argv[1]);
  loopbound = atoi(argv[2]);

  temp = (DUMMY *)shmalloc(sizeof(DUMMY));
  temp->d1 = 1;
  temp->d2 = 2;
  temp->d3 = 3;
  temp->d4 = 4;
  temp->d5 = 5;
  do_in_parallel(proc1, temp, missprob, loopbound);
}

void proc1(DUMMY_PTR tmp, int missprob, int loopbound)
{
  int i, j;
  int k, l;


  k = (int)((float)missprob*0.01*(float)loopbound+0.5);
  l = (loopbound-k);
```

```
  for (i = 0; i < k; i++){
    softflush(tmp);
    j = tmp->d1;
  }
  for (i = 0; i < l; i++)
    num = i;
}
```

A.4 Bench11.c: A Program for Validating the Timesliced Mode

```
/*
   Sramana Mitra
   September 27, 1994
   bench11.c: A synthetic benchmark program
              to test uncached remote accesses.
*/

#include <primops.h>
#include <stdio.h>
#include <parallel.h>

int num = 9;
void proc1();
/* struct for dynamic allocation */
typedef struct dummy {
    unsigned  d1;
    unsigned  d2;
    unsigned  d3;
    unsigned  d4;
    unsigned  d5;
} DUMMY;
typedef DUMMY *DUMMY_PTR;


main(int argc, char *argv[])
{
  int missprob, loopbound;
  DUMMY_PTR  temp;
```

```
  missprob = atoi(argv[1]);
  loopbound = atoi(argv[2]);

  temp = (DUMMY *)shmalloc(sizeof(DUMMY));
  temp->d1 = 1;
  temp->d2 = 2;
  temp->d3 = 3;
  temp->d4 = 4;
  temp->d5 = 5;
  do_in_parallel(proc1, temp, missprob, loopbound);
}

void proc1(DUMMY_PTR tmp, int missprob, int loopbound)
{
  int i, j;
  int k, l;


  k = (int)((float)missprob*0.01*(float)loopbound+0.5);
  l = (loopbound-k);
  for (i = 0; i < l; i++)
    num = i;
  for (i = 0; i < k; i++){
    softflush(tmp);
    j = tmp->d1;
  }
  for (i = 0; i < l; i++)
    num = i;
  for (i = 0; i < k; i++){
    softflush(tmp);
    j = tmp->d1;
  }
}
```

Appendix B

Tables for Graphs

This chapter provides tables for graphs in chapters 2, 4 and 5 to aid readers who have a black and white copy of the thesis.

Proc#	Timeslice0	Timeslice1	Timeslice2
0	19958-107880	19958-107880	17801-18879
1	0-539	17801-18879	19958-107880
2	0-539	16722-17800	18880-19957
3	0-539	15643-16721	17801-18879
4	0-539	17801-18879	19958-107880
5	0-539	17801-18879	19958-107880
6	0-539	17801-18879	18880-19957
7	0-539	16722-17800	17801-18879
8	0-539	17801-18879	19958-107880
9	0-539	16722-17800	19958-107880
10	0-539	16722-17800	18880-19957
11	0-539	16722-17800	17801-18879
12	0-539	15643-16721	19958-107880
13	0-539	16722-17800	19958-107880
14	0-539	15643-16721	18880-19957
15	0-539	15643-16721	17801-18879

Table B.1: Water on 16 processors: Per processor distribution of remote shared data accesses.

Proc#	# of packet headers
0	29733-191823
1	27815-29732
2	29733-191823
3	29733-191823
4	27815-29732
5	29733-191823
6	29733-191823
7	29733-191823
8	29733-191823
9	29733-191823
10	29733-191823
11	27815-29732
12	27815-29732
13	29733-191823
14	27815-29732
15	27815-29732

Table B.2: Water on 16 processors: Counts of packet headers passing through output queues.

Proc#	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
0	2418-6731	6732-13070	6732-13070	6732-13070	2418-6731	1504-2417
1	6732-13070	2418-6731	1504-2417	0-65	0-65	0-65
2	6732-13070	2418-6731	2418-6731	197-1503	0-65	0-65
3	1504-2417	6732-13070	1504-2417	0-65	0-65	0-65
4	2418-6731	6732-13070	197-1503	2418-6731	197-1503	0-65
5	197-1503	1504-2417	6732-13070	197-1503	2418-6731	1504-2417
6	1504-2417	2418-6731	6732-13070	197-1503	0-65	0-65
7	197-1503	197-1503	1504-2417	6732-13070	1504-2417	0-65
8	2418-6731	6732-13070	2418-6731	1504-2417	0-65	0-65
9	2418-6731	2418-6731	6732-13070	0-65	0-65	0-65
10	1504-2417	1504-2417	6732-13070	1504-2417	0-65	0-65
11	1504-2417	2418-6731	1504-2417	6732-13070	0-65	0-65
12	2418-6731	2418-6731	1504-2417	6732-13070	0-65	0-65
13	197-1503	197-1503	1504-2417	2418-6731	6732-13070	0-65
14	197-1503	1504-2417	1504-2417	2418-6731	6732-13070	0-65
15	0-65	197-1503	1504-2417	1504-2417	2418-6731	6732-13070

Table B.3: Water on 16 processors: Histogram of distances of memory-to-cache input packets.

Proc#	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
0	1874-9903	1874-9903	1874-9903	1874-9903	1874-9903	268-1873
1	1874-9903	268-1873	268-1873	268-1873	0-267	0-267
2	1874-9903	268-1873	268-1873	268-1873	268-1873	0-267
3	24892-53531	24892-53531	24892-53531	1874-9903	0-267	0-267
4	18469-24891	24892-53531	18469-24891	9904-18468	1874-9903	0-267
5	9904-18468	9904-18468	18469-24891	9904-18468	9904-18468	1874-9903
6	24892-53531	24892-53531	24892-53531	1874-9903	0-267	0-267
7	24892-53531	24892-53531	24892-53531	24892-53531	9904-18468	0-267
8	18469-24891	24892-53531	18469-24891	18469-24891	1874-9903	0-267
9	18469-24891	24892-53531	18469-24891	1874-9903	0-267	0-267
10	9904-18468	18469-24891	18469-24891	18469-24891	9904-18468	1874-9903
11	24892-53531	24892-53531	24892-53531	24892-53531	9904-18468	0-267
12	18469-24891	24892-53531	18469-24891	1874-9903	0-267	0-267
13	9904-18468	18469-24891	18469-24891	9904-18468	1874-9903	0-267
14	18469-24891	24892-53531	24892-53531	18469-24891	1874-9903	0-267
15	9904-18468	9904-18468	18469-24891	18469-24891	9904-18468	1874-9903

Table B.4: Orig Mp3d: Per processor distribution of remote distances travelled by read invalidation packets.

Proc#	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
0	1990-4564	1990-4564	4565-8310	1990-4564	1990-4564	118-1989
1	118-1989	118-1989	118-1989	118-1989	118-1989	0-117
2	1990-4564	118-1989	118-1989	118-1989	118-1989	0-117
3	11823-23410	11823-23410	11823-23410	1990-4564	0-117	0-117
4	8311-11822	11823-23410	8311-11822	4565-8310	1990-4564	0-117
5	1990-4564	4565-8310	8311-11822	4565-8310	4565-8310	1990-4564
6	8311-11822	11823-23410	11823-23410	1990-4564	0-117	0-117
7	11823-23410	11823-23410	11823-23410	11823-23410	4565-8310	0-117
8	8311-11822	11823-23410	8311-11822	8311-11822	1990-4564	0-117
9	8311-11822	11823-23410	8311-11822	1990-4564	0-117	0-117
10	4565-8310	8311-11822	11823-23410	8311-11822	4565-8310	1990-4564
11	11823-23410	11823-23410	11823-23410	11823-23410	4565-8310	0-117
12	8311-11822	11823-23410	8311-11822	1990-4564	0-117	0-117
13	4565-8310	8311-11822	8311-11822	4565-8310	1990-4564	0-117
14	8311-11822	8311-11822	11823-23410	8311-11822	1990-4564	0-117
15	4565-8310	4565-8310	8311-11822	8311-11822	4565-8310	4565-8310

Table B.5: Mp3d: Per processor distribution of remote distances travelled by read invalidation packets.

Proc#	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
0	1285-2946	2947-4306	2947-4306	2947-4306	1285-2946	76-1284
1	76-1284	76-1284	76-1284	76-1284	76-1284	0-75
2	76-1284	76-1284	76-1284	76-1284	76-1284	0-75
3	6120-15111	6120-15111	6120-15111	1285-2946	0-75	0-75
4	4307-6119	4307-6119	4307-6119	2947-4306	1285-2946	0-75
5	76-1284	2947-4306	4307-6119	2947-4306	1285-2946	1285-2946
6	4307-6119	6120-15111	4307-6119	76-1284	0-75	0-75
7	6120-15111	6120-15111	6120-15111	6120-15111	2947-4306	0-75
8	4307-6119	6120-15111	6120-15111	4307-6119	1285-2946	0-75
9	4307-6119	6120-15111	4307-6119	1285-2946	0-75	0-75
10	1285-2946	4307-6119	6120-15111	4307-6119	1285-2946	1285-2946
11	6120-15111	6120-15111	6120-15111	6120-15111	2947-4306	0-75
12	4307-6119	6120-15111	6120-15111	1285-2946	0-75	0-75
13	1285-2946	4307-6119	2947-4306	2947-4306	1285-2946	0-75
14	2947-4306	4307-6119	4307-6119	4307-6119	1285-2946	0-75
15	2947-4306	4307-6119	6120-15111	4307-6119	2947-4306	2947-4306

Table B.6: MMp3d: Per processor distribution of remote distances travelled by read invalidation packets.

Proc#	Average remote access latencies (in cycles)
0	59-66
1	53
2	54
3	54
4	56
5	55
6	54
7	58
8	56
9	54
10	57
11	59-66
12	58
13	56
14	56
15	55

Table B.7: Orig Mp3d: Average remote access latencies.

Proc#	Average remote access latencies (in cycles)
0	64-72
1	58-59
2	54
3	62-63
4	58-59
5	58-59
6	56
7	58-59
8	55
9	60-61
10	55
11	57
12	60-61
13	62-63
14	64-72
15	64-72

Table B.8: Mp3d: Average remote access latencies.

Proc#	Average remote access latencies (in cycles)
0	34-43
1	51
2	47
3	46
4	46
5	52-53
6	54-56
7	46
8	44
9	54-56
10	49
11	49
12	45
13	50
14	48
15	54-56

Table B.9: MMp3d: Average remote access latencies.

Proc#	# of packet headers (in cycles)
0	114123-160871
1	94873-114122
2	92123-94872
3	182872-265369
4	174622-177371
5	160872-169121
6	177372-182871
7	265370-274994
8	177372-182871
9	160872-169121
10	171872-174621
11	265370-274994
12	177372-182871
13	160872-169121
14	177372-182871
15	177372-182871

Table B.10: Orig Mp3d: Packet headers passing through output queue.

Proc#	# of packet headers (in cycles)
0	67345-74758
1	46339-67344
2	46339-67344
3	84645-119243
4	80938-82172
5	74759-78465
6	82173-84644
7	119244-123568
8	82173-84644
9	74759-78465
10	79702-80937
11	119244-123568
12	82173-84644
13	74759-78465
14	82173-84644
15	80938-82172

Table B.11: Mp3d: Packet headers passing through output queue.

Proc#	# of packet headers (in cycles)
0	56894-61923
1	27976-29232
2	29233-38033
3	47464-56893
4	38034-39291
5	39920-40548
6	40549-41805
7	61924-62866
8	42435-47463
9	40549-41805
10	41806-42434
11	61924-62866
12	39292-39919
13	38034-39291
14	39920-40548
15	40549-41805

Table B.12: MMp3d: Packet headers passing through output queue.

Proc#	Timeslice0	Timeslice1	Timeslice2	Timeslice3	Timeslice4	Timeslice6
0	87-97	80-81	80-81	80-81	80-81	80-81
1	82-86	73	71	72	71	73
2	82-86	74-79	73	73	73	74-79
3	82-86	70	71	74-79	73	74-79
4	82-86	72	72	72	74-79	74-79
5	82-86	71	73	73	70	73
6	82-86	70	74-79	74-79	72	-
7	82-86	71	73	71	71	74-79
8	82-86	73	70	71	73	74-79
9	82-86	70	73	73	71	73
10	82-86	74-79	72	71	72	73
11	82-86	70	71	71	71	72
12	82-86	74-79	72	70	71	73
13	82-86	72	71	71	71	72
14	82-86	70	72	71	71	74-79
15	82-86	72	70	71	72	73

Table B.13: ZGRID: Data cache hit ratios.

Proc#	Timeslice0	Timeslice1	Timeslice2	Timeslice3	Timeslice4	Timeslice6
0	93	92	92	92	92	94-97
1	94-97	93	93	93	93	94-97
2	94-97	94-97	94-97	94-97	94-97	94-97
3	94-97	94-97	94-97	94-97	94-97	94-97
4	94-97	94-97	94-97	94-97	94-97	94-97
5	94-97	94-97	94-97	94-97	94-97	94-97
6	94-97	94-97	94-97	94-97	94-97	94-97
7	94-97	94-97	94-97	94-97	94-97	94-97
8	94-97	94-97	94-97	94-97	94-97	94-97
9	94-97	94-97	94-97	94-97	94-97	94-97
10	94-97	94-97	94-97	94-97	94-97	94-97
11	94-97	94-97	94-97	94-97	94-97	94-97
12	94-97	94-97	94-97	94-97	94-97	94-97
13	94-97	94-97	94-97	94-97	94-97	94-97
14	94-97	94-97	94-97	94-97	94-97	94-97
15	94-97	93	93	93	93	94-97

Table B.14: MGRID: Data cache hit ratios.

Proc#	Timeslice0	Timeslice1	Timeslice2	Timeslice3	Timeslice4	Timeslice6
0	98-99	98-99	98-99	98-99	98-99	98-99
1	98-99	98-99	98-99	98-99	98-99	98-99
2	98-99	98-99	98-99	98-99	98-99	98-99
3	98-99	98-99	98-99	98-99	98-99	98-99
4	98-99	98-99	98-99	98-99	98-99	98-99
5	98-99	98-99	98-99	98-99	98-99	98-99
6	98-99	98-99	98-99	98-99	98-99	98-99
7	98-99	98-99	98-99	98-99	98-99	98-99
8	98-99	98-99	98-99	98-99	98-99	98-99
9	98-99	98-99	98-99	98-99	98-99	98-99
10	98-99	98-99	98-99	98-99	98-99	98-99
11	98-99	98-99	98-99	98-99	98-99	98-99
12	98-99	98-99	98-99	98-99	98-99	98-99
13	98-99	98-99	98-99	98-99	98-99	98-99
14	98-99	98-99	98-99	98-99	98-99	98-99
15	98-99	98-99	98-99	98-99	98-99	98-99

Table B.15: CGRID: Data cache hit ratios.

Bibliography

[1] Anant Agarwal *et. al*. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprcoessor. Proceedings of Workshop on Scalable Shared Memory Multiprocessors, 1991.

[2] Ziya Aral and Ilya Gertner. High-Level Debugging in Parasight. In *Parallel and distributed debugging. Proceedings of the ACM SIGPLAN/SIGACT workshop*, May 5-6, 1988, Madison, Wisconsin, January 1989.

[3] Ricardo Bianchini. Alewife Systems Memo #43. *Application Performance on the Alewife Multiprocessor*, September 1994.

[4] B. Bliss, M. C. Brunet and E. Gallopoulos. Automatic Parallel Program Instrumentation with Applications in Performance and Error Analysis. *University of Illinois Technical Report No. CSRD-1025*, June 1990.

[5] Eric A. Brewer and Chrysanthos N. Dellarocas. *Proteus User Documentation*.

[6] Helmar Burkhart and Roland Millen. Performance-Measurement Tools in a Multiprocessor Environment. In *IEEE Transactions on Computers*, Vol. 38, No. 5, May 1989.

[7] Jack Dongarra, Orlie Brewer, James Arthur Kohl and Samuel Fineberg. Tools to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors. In *Journal for Parallel and Distributed Computing*, 9, pp 185-202, 1990.

[8] Raymond R. Glenn and Daniel V. Pryor. Instrumentation for a Massively Parallel MIMD Application. In *Journal of Parallel and Distributed Computing*, 12, pp 223-236, 1991.

[9] Aaron J. Goldberg. Reducing Overhead in Counter-Based Execution Profiling. *Stanford Technical Report No. CSL-TR-91-495*, October 1991.

[10] Aaron J. Goldberg. Multiprocessor Peformance Debugging and Memory Bottlenecks. *Stanford Technical Report No. CSL-TR-92-542*, August 1992.

[11] Aaron J. Goldberg and John Hennessy. Mtool: An Integrated System for Peformance Debugging Shared Memory Multiprocessor Applications. In *IEEE Transactions on Parallel and Distributed Systems*, 4:1, pp 28-40, January 1993.

[12] S. Graham, P. Kessler and M. McKusick. Gprof: A call graph execution profiler. In *Proceedings of the ACM/SIGPLAN Symposium of Compiler Construction*, pp 120-126, 1982.

[13] John Kubiatowicz, David Chaiken and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pp 274-284, October 1992.

[14] John Kubiatowicz. Alewife Systems Memo #19. *Users Manual for the Alewife 1000 Controller*, November 1991.

[15] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference (ISC) 1993*, Tokyo, Japan, July 1993.

[16] A. Kwok and W. Abu-Sufah. Tcedar: A performance evaluation tool for cedar. *Technical Report CSRD 84, University of Illinois at Urbana-Champaign, 1984*.

[17] Daniel Lenoski *et. al*. The Stanford Dash Multiprocessor. In *IEEE Computer*, pp 63-79, March 1992.

[18] Margaret Martonosi and Anoop Gupta. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Performance Evaluation Review*, 20:1, pp 1-12, June 1992.

[19] Barton P. Miller *et. al.*. IPS-2: The Second Generation of a Parallel Program Measurement System. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990.

[20] Zary Segall and Larry Rudolph. PIE: A Programming and Intrumentation Environment for Parallel Processing. In *IEEE Software*, pp 22-37, November 1985.

[21] J. P. Singh, W. D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[22] SPARC Architecture Manual. 1988. SUN Microsystems, Mountain View, California.

[23] Stephen Walters. Computer-Aided Prototyping for ASIC-Based Systems. IEEE Design and Test of Computers, 1991.

[24] Benson Wen. Foghorn: Parallel Performance Statistics on Alewife. Bachelor's Thesis, Massachusetts Institute of Technology, June 1993.