

BUS TRANSACTION LOG GRAPHING TOOL FOR HARDWARE-SOFTWARE CO-DESIGN APPLICATIONS

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

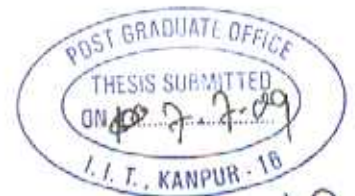
by
Bhave Nachiket Vishwas



to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2009

CERTIFICATE



It is certified that the work contained in the thesis entitled "*Bus Transaction Log Graphing Tool for Hardware-Software Co-Design Applications*" by *Bhavinachiket Vishwas* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

A handwritten signature in blue ink, which appears to be "Raj", written over a horizontal line.

Dr. Rajat Moona

Department of Computer Science and Engineering,
I.I.T. Kanpur.

July 2009

Contents

Acknowledgements	xi
Abstract	xiii
1 Introduction	1
1.1 Performance analysis of embedded systems	2
1.1.1 Performance analysis in Hardware-software co-design	3
1.2 Related Work	4
1.2.1 Sim-nML	4
1.2.2 FSim	5
1.2.3 Bus Logger	5
1.3 Overview of this work	6
1.4 Organization of the report	6
2 Profilers and Profile Viewers	7
2.1 Profilers	7
2.2 Profile Viewers	11
2.3 Role of the Log Viewer	12

3	The Log File	15
3.1	Log file design issues	15
3.2	Log file format	18
3.2.1	The header block	18
3.2.2	The data block	20
4	Library for Log Viewer	23
4.1	The Viewer Interface	23
4.1.1	Introduction	23
4.1.2	Interface definition	24
4.2	Viewer interface implementation details	29
4.2.1	initViewer implementation	29
4.2.2	readLog implementation	30
4.2.3	closeViewer implementation	30
4.2.4	seekByCycleCounter implementation	31
5	The Log Viewer Application	33
5.1	Organization of the application	33
5.1.1	The menu	35
5.1.2	The header section	35
5.1.3	The clock cycle input section	36
5.1.4	The timeline	36
5.1.5	The display panel	37
5.2	Types of visual data representations	38
5.2.1	The transaction type graph	38
5.2.2	The transaction type joint bar chart	39

5.2.3	The transaction type bar chart	40
5.2.4	The table view	41
5.3	Features of the log viewer application	41
6	Results and Conclusion	43
6.1	Experimental setup	44
6.1.1	The host machine	44
6.1.2	The target system	44
6.2	Results	45
6.3	Conclusion	53
6.4	Future Work	53





List of Figures

3.1	A simple log file format	17
3.2	The log file format	19
5.1	The log viewer application window	34
5.2	The menu	35
5.3	The header section	35
5.4	The clock cycle input section	36
5.5	The timeline	36
5.6	The display panel	37
5.7	The transaction type graph	38
5.8	The transaction type joint bar chart	39
5.9	The transaction type bar chart	40
5.10	The table view	41
6.1	The target system	44
6.2	Log viewer output for the bus connecting processor and L1 instruction cache during simulation of column-wise matrix copy	47
6.3	Log viewer output for the bus connecting processor and L1 instruction cache during simulation of row-wise matrix copy	48

6.4	Log viewer output for the bus connecting the L1 data cache and the L2 cache during simulation of column-wise wise matrix copy	49
6.5	Log viewer output for the bus connecting the L1 data cache and the L2 cache during simulation of row-wise wise matrix copy	50
6.6	Log viewer output for the bus connecting the L2 cache and the main memory during simulation of column-wise wise matrix copy	51
6.7	Log viewer output for the bus connecting the L2 cache and the main memory during simulation of row-wise wise matrix copy	52



Acknowledgements

There are many people who helped me in my journey from the commencement to the completion of this work. First and foremost I would like to thank my thesis supervisor Prof. Rajat Moona. His active participation and quest for perfection have always inspired me to give my best. Working with him has indeed been a learning experience.

I would like to thank Amit Kulkarni who has been a helpful colleague and more importantly a wonderful friend during the course of this work. I would also like to thank my senior Hemant Shinde whose words of advice have been like a guiding light in the darkest of times. I am grateful to the entire batch of MTech CSE who made my stay at IIT Kanpur enjoyable and impossible to forget.

Last but not the least I would like to thank my family members for their unconditional support and confidence in me. Making them proud is one of the greatest joys of my life.



Abstarct

Current embedded systems are becoming increasingly complex even though they are faced with strict time-to-market constraints as never before. In such a scenario there is a need of using new paradigms in embedded system design. Hardware-software co-design is one such paradigm in which the design of hardware and software proceeds concurrently. This leads to a flexible design process in which the costs of changes in the design are low. The hardware-software co-design process involves the use of automated tools for designing, prototyping and analyzing the system. These automated tool generators require the description of the target processor in some architecture description language in order to generate processor centric tools. At IIT Kanpur, an architecture description language called Sim-nML[1] has been developed. Various automated tool generators are also developed that use Sim-nML description of a processor in order to generate tools like simulators[2], debuggers[3] etc. Along with tools for simulation of the target system, one requires tools to record performance related parameters of system and later analyze them to improve the performance of the system. These tools help the designers to ensure that the system meets its performance specifications.

In this work, we have developed a tool called log viewer which helps in performance analysis of the system. This tool provides graphical representation of bus

transaction log records which are generated during the simulation of execution of an application on the system. The bus transaction log records collected during the simulation contain information relevant to the performance of the system such as the time required by the bus transactions, the amount of data transferred during the bus transaction etc. A bus transaction log viewing library has also been developed in order to facilitate the reading of the log files. The library takes advantage of specially designed log file format in order to perform fast search in the file. The log viewer provides a variety of representations of the bus transaction log records in order to facilitate analysis.



Chapter 1

Introduction

An embedded system is a computer system that is used to perform few specialized tasks. Generally it is a part of a larger machine or system. Designing an embedded system includes designing the hardware as well as the software for the system. The traditional way of doing this is to design the hardware first. Once the hardware has been designed, the software which is to be run on it is developed. Modern day embedded systems are becoming increasingly complex and hence designing them is also becoming an increasingly complex task. The issues faced with the traditional design approach can be summarized as follows.

- The cost to correct hardware design errors increases as the design progresses.
- There should be a separation of functionality to be implemented in hardware and software at the beginning of the design process. This leads to a loss of flexibility leaving no scope for revision. This may lead to suboptimal designs.
- The design of such systems can have many additional constraints, including performance, cost, time-to-market, power, space and reliability requirements.

These issues led embedded systems designers to use new design paradigms. These design paradigms include concurrent development of hardware and software. This gives the designers a choice between multiple hardware-software designs in the early phases of design. They can analyze the trade-offs and choose an optimal design.

1.1 Performance analysis of embedded systems

Performance analysis of a program is the process of collecting performance related data about the program during its execution and using this data to study the issues regarding the performance of the program. In the design of an embedded system, it is important that the system meets its performance specifications along with its functional specifications. In some systems there are strict performance requirements and hence performance of such systems has to be optimized. In other systems, performance optimizations help the system exceed specifications and hence outperform competition. In order to optimize the performance of a system, performance analysis is performed during embedded system design.

The process of collecting data about a program execution is called as profiling. The data may include frequency and duration of function call, cache utilization etc. The profilers can collect the data about a program execution using techniques such as hardware interrupts, code instrumentation, operating system hooks or the use of instruction set simulators. The examples of such profilers are gprof[4], valgrind[5] etc. The size of the data collected during the process of profiling is generally very large. Therefore specialized tools are required that provide suitable representation of the data so that the performance issues can be studied. The development of

such a tool has been the main motivation of this work.

1.1.1 Performance analysis in Hardware-software co-design

Hardware-software co-design[2] is a design approach in which the design of hardware and software proceeds in parallel. Due to this the hardware-software design can take place co-operatively. This design approach can be described as follows.

The first step includes generation of system specification. This specification is analyzed based on various cost matrices like performance, design time, power and space. The system functionality is divided into modules. The next step is to partition the modules based on whether their functionality will be implemented in hardware or software. This partitioning is done in such a way that the cost criteria are satisfied in an optimal manner. After this step, the design of hardware and software proceeds in parallel. The hardware is designed typically using hardware description languages (HDLs). HDLs are used to describe the hardware at the lowest level. HDL tools are used for the simulation and synthesis of the required hardware. The software design involves use of high level language such as C, C++ and tools such as instruction set simulators, debuggers, profilers, cross-compilers etc. These tools aid in the development of the software for the target environment.

After the co-design the hardware and the software are simulated together. This is called as co-simulation. The results of these simulations are checked against the specifications. If the specifications are not met, the design process is repeated by re-partitioning. Once the specification is met, the design is translated into an actual system.

The performance of the system can be analyzed during the co-simulation. An

instruction set simulator equipped with suitable logging facility can act as profiling tool that can collect data about the execution of each instruction. This data can then be studied using a suitable tool in order to find performance related problems of the system. Once these problems are determined, they can be solved during the next cycle of the design process.

1.2 Related Work

1.2.1 Sim-nML

Every embedded system has its own unique requirements. These requirements may force the designers of the system to use new processor cores or modify existing processor cores. Whenever a new processor core is created, it requires a new set of software tools such as simulators, debuggers, disassemblers, compiler back-ends etc. So, whenever a new processor core is designed, there is an overhead of developing software tools for that processor. To avoid this overhead, there is a need for automated software tool generation. Automated software tool generators require a specification of the processor architecture.

At IIT Kanpur, a language called Sim-nML[1] has been developed for this purpose. Sim-nML is a language for describing arbitrary processor architectures. It can be used to describe the processor architecture at the level of the instruction set. Thus hardware specific low level implementation details are hidden from the designer. Sim-nML is flexible, easy to use and is based on attribute grammar.

Sim-nML description of a processor can be viewed as the programmer's model of the processor. The model consists of the following.

- Syntax and semantics of instructions
- Addressing modes
- Definition of registers and memory
- Resource usage model
- Methods for handling traps and other synchronized events

The description of processor architecture in Sim-nML can be used by processor centric tools such as instruction set simulators, debuggers, compiler back-ends and profiling tools in a retargetable manner. A Sim-nML description of processor architecture acts as a specification for retargetable processor centric tool generators.

1.2.2 FSim

A retargetable function simulator generator (*fsimg*[2]) was developed at IIT Kanpur. It uses the Sim-nML description of the target processor architecture in order to generate a functional simulator (*fsim*).

1.2.3 Bus Logger

A Bus Logger[6] has been developed which can interface with *fsim* and can be used to log the bus transactions that take place during the simulation of execution of a program using *fsim*. The log files generated by the Bus Logger can be used in performance analysis.

1.3 Overview of this work

In this work, we have developed a bus log viewing facility for reading log files generated by the bus logger in the retargetable functional simulator *fsim*. The log files contain the information about the bus transactions that take place during the simulation of a program execution by *fsim*. A library of API functions for reading log files as well as traversing a log file in an efficient manner has been designed and implemented.

A log viewer has been created using this library. The log viewer uses Qt 4.0 [7], which is a C++ toolkit for cross-platform GUI application development. In the log viewer the data contained in the log files can be viewed in different forms like graphs, bar charts, tables etc. It can be used as an effective tool for studying bus transactions during the simulation of execution of a program using *fsim*.

1.4 Organization of the report

The rest of the report is organized as follows. In chapter 2, we describe profilers and profile viewers. In chapter 3, we describe the log file which is viewed using the log viewer. In chapter 4, we describe the library which is used by the log viewer. It includes the viewer interface and its implementation. In chapter 5, we describe the design and features of the log viewer application. In chapter 6 we describe the results and conclude this work.

Chapter 2

Profilers and Profile Viewers

Profilers and Profile viewers play an important role in performance analysis of large and complex applications. The log viewer along with the bus logger interfaced with a suitable instruction set simulator can act as a performance analysis tool. The bus logger logs the bus transactions that take place during the simulation of a program execution using an instruction set simulator. The log viewer allows the users to view the data available in the log files in a convenient manner. Thus the bus logger acts as a profiler while the log viewer acts as a profile viewer. In this chapter, we describe profilers and profile viewers and establish the usefulness of the log viewer as a profile viewer.

2.1 Profilers

Profilers are tools that collect information about the behavior of programs during their execution. These tools collect different kinds of information like the memory access patterns, time spent in execution of different functions etc. As opposed to

static code analysis, in profiling tools the program is executed in order to perform analysis. Thus the program analysis represents the actual execution behaviour rather than the static approximation.

Profilers use different techniques such as code instrumentation, hardware interrupts, operating system hooks, instruction set simulation etc. In code instrumentation, additional code is inserted in the target program to collect profiling data. Hardware interrupts are used to transfer the control to the profiler whenever certain hardware related event takes place. Similarly operating system hooks are used to transfer control to profiler whenever certain operating system calls take place. Using instruction set simulators, execution of the program can be simulated and profiling data can be collected during simulation.

Data collected by profiling tools can be used to identify the hotspots in the code where large fraction of time is spent during execution. When hotspots are identified, they can be analyzed and attempts can be made to optimize the code such that the time spent in these hotspots is reduced.

The profiling data can also be used to find the time spent by a program to access data. The memory access patterns that may result in cache misses, which in turn may result in increased execution time, can be found out. Changes can be made to these memory access patterns such that the cache misses are reduced and thus the execution time of the program is reduced. Using profiling data we can identify parts of the memory allocated to the program that are never accessed. Using this information one can find unused variables in the program thus uncovering the bugs in the program.

There are several profiling tools available. Some of these are described below.

- **gprof :** gprof[4] is an open source text based profiling tool developed by the GNU. It gathers statistical information about the target programs execution. It uses a technique called code instrumentation for this purpose. In code instrumentation the target code is modified in order to collect information about the program execution. For example GNU compiler inserts code at the head and tail of every function in order to collect timing information about the functions. This is enabled through a command line switch -p to gcc[8]. The information generated can be used to perform call graph analysis. A call graph is a representation of the subroutine calls in the program in a graphical manner.
- **Intel VTune Analyzer :** Intel VTune Analyzer[9] is a performance analyzer for programs executing on systems with Intel processors. It is a commercial tool developed by Intel. Along with the profiler, it also contains an integrated profile viewer. The profile viewer is in the form of a GUI to view the information collected during program execution. The results are presented in details to the extent that time taken by an individual instruction can also be measured. The features of VTune Analyzer include the following.
 - **Call Graph :** A call graph view is provided which can be used to find out critical functions in the target program and to get a high level algorithmic view of the target program.
 - **Time Based and Event based Sampling:** Sampling can be done with respect to time as well as the events that occur during program analysis. This provides the programmer the flexibility required to accurately analyze the performance of the program.

- **Counter Monitor :** Keeps track of resource consumption during program execution.
 - **Tuning Assistant :** Provides automated advice for performance tuning based on extensive knowledge.
 - Support for tuning multi-core processors.
- **Valgrind:** Valgrind[5] is an open-source tool licensed under GPL used primarily for memory profiling. Valgrind can be used for detecting memory leaks, memory debugging and for detecting errors due to synchronization in multiple threads. It works with programs independent of a programming language and is available for most popular platforms. It does not execute a program directly but converts it into a machine independent Intermediate Representation (IR). It then uses just-in-time compilation techniques to execute the program. The IR makes code instrumentation easy as compared to the instrumentation of the original code.

Many tools have been developed using the Valgrind framework. The Memcheck tool[10] is built over Valgrind framework and is used to detect various memory related errors such as memory leaks, invalid memory accesses, use of uninitialized variables etc. The Cachegrind tool[11] is another similar tool that is used to get detailed information about cache usage during program execution.

2.2 Profile Viewers

Profile viewers are tools that are used to view the data collected during process of profiling in a suitable format. These tools include graphing tools for representing the data. Graphing tools are used to represent the available data in a graphical form. Generally raw data is difficult for human beings to interpret. This is because the raw data is generally available in large quantities and it is very difficult to interpret anything from it directly. In graphing tools, the data may be represented in different forms like simple graphs, bar-charts, joint bar-charts, pie charts, tree-views, tables etc. The aim of a graphing tool is to provide the user a view of the data which is useful to him and easy for him to interpret.

The role of the profile viewer in the process of performance analysis is to help the users understand the data collected during profiling. The users should be able to locate hotspots in the program execution, understand the sequence of subroutine calls etc. The graphing tools may provide specialized view of the profiling data for each of the tasks the user wishes to perform during analysis.

Some profiling tools have profile viewers as their integral part. An example of this is the Intel VTune Analyzer. The VTune Analyzer shows different kinds of graph to help the users understand performance bottlenecks and fine-tune the application accordingly. On the other hand some profiling tools do not provide profile viewing facilities. An example of this is gprof. Such profiling tools generally write their output into files. These files can be read by third party profile viewers in order to produce graphical output. An example of such a profile viewer is Kprof[12]. Kprof takes the profiling data produced by tools such as gprof, Function Check[13] and Palm OS Emulator[14] as input. It produces easy to understand list views

and tree views as output.

2.3 Role of the Log Viewer

As described in section earlier a profiling tool collects information about program execution. In order to collect the information the profiling tools use different techniques such as code instrumentation. One such technique is the use of an instruction set simulator for simulating program execution equipped with a suitable logging facility. An instruction set simulator can gather all the information about a program execution from invocation to termination. This technique is used by the bus logger.

The retargetable simulator fsim[2] is an instruction set simulator. During the simulation of execution of a program using fsim, the bus logger logs every bus transaction that takes place on a bus into a log file. The log viewer provides representations of this data contained in the log file in different forms which are easy to understand and useful. Such representations of the log data can be used in performance analysis and performance tuning of the target application.

Since fsim can simulate programs written for arbitrary processor architecture, the bus logger and the log viewer can also be used as a performance analysis tool for arbitrary processor architectures. During the process of embedded system design using the Hardware-Software co-design paradigm described in section 1.1.1, the log viewer can be used after the co-simulation step in order to do performance analysis and check the results against system specification. Using the information provided by the Log Viewer suitable changes can be made to the software as well as the hardware so that the system specifications are met. Since the logs are

recorded for the bus of the systems, the log viewer can help in identifying the hardware errors by indicating the time taken by the hardware to provide data on the bus. Therefore the Log Viewer can play an important part in the design of an embedded system.





Chapter 3

The Log File

In this chapter we discuss the log file which is generated by the bus logger. This file acts as an input to the log viewer.

3.1 Log file design issues

The size of the log file generated by the bus logger is expected to be very large. Consider the example of a matrix multiplication program. Let the size of input matrices be 100x100. For arm architecture[15], the number of machine instructions that are executed are of the order of 10^7 . For every machine instruction that is executed, there is at least one bus transaction generated between the processor core and the instruction cache. If the logs are recorded for this interface, at least one log record is written to the log file for every instruction executed. The size of one log record is typically 10 to 15 bytes. Therefore the size of the log file generated is expected to be of the order of a gigabyte. In order to effectively use the data available in the log file, it is essential that operations such as traversing

the log file or searching the log file should be efficient. Keeping these issues in mind we designed the log file structure for a faster access.

The log file consists of a header which contains information like bus name, number of log records etc. The header is followed by the log records which contain information about individual bus transactions. The size of a log record is not fixed and may vary depending on the bus transaction that is being logged. This size depends upon parameters such as address size on the bus, size of data etc.

A simple structure of the log file would be the as follows. The header information is stored at the beginning of the file. After the header, the log records are stored in the file sequentially in the order of the bus transactions.

Using this structure, reading log records for a sequence of bus transactions would be easy. Even though the log records are kept in sorted order, searching a log record for a particular bus transaction based on the clock cycle at which it occurred would be time consuming. This is because the log records are not of a fixed size. Due to this, it is not possible to randomly access log records in any order. Thus binary search is not possible. The only possible search operation is sequential search. As the size of log files are typically very large, the sequential search operation is time consuming.

As a solution to the above problem, we could maintain an index of clock cycle numbers and offset of corresponding log records. However if the index is a dense index, the size of the log file would increase considerably and if the index is a sparse index, the search operation will become slower.

To overcome the problems mentioned above, we introduce a log file structure in which the log file is divided into fixed sized blocks. Each block has associated clock cycle number of the bus transaction corresponding to the first log record stored in

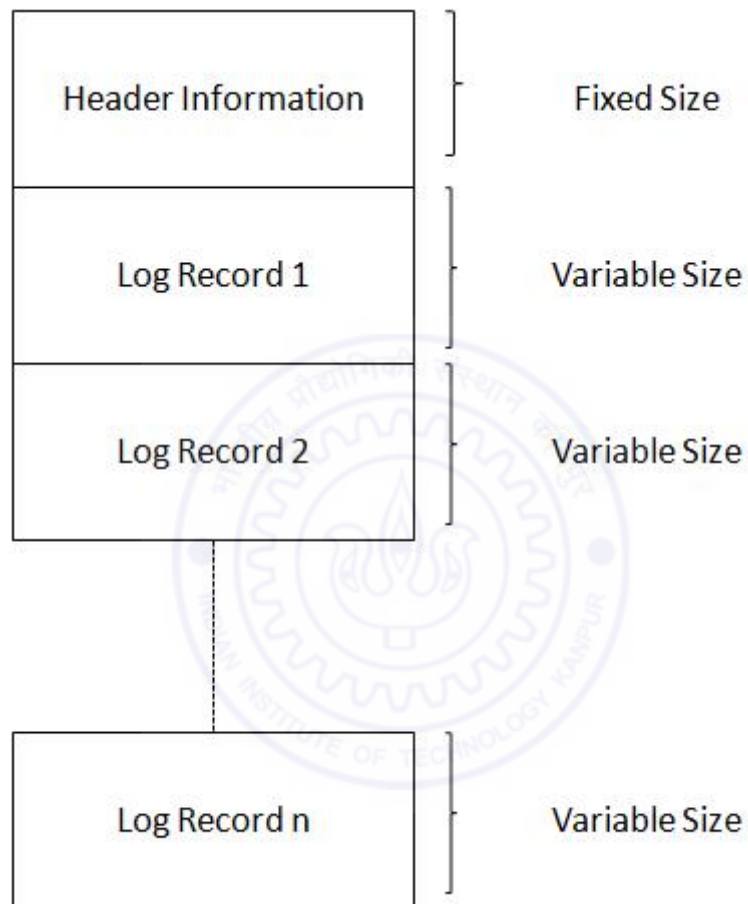


Figure 3.1: A simple log file format

the block. Thus binary search on blocks is possible. The actual record can then be searched sequentially within the block. In addition, there is no need for an index. The detailed explanation of the log file structure used by our approach is given in the following section.

3.2 Log file format

The log file contains several fixed sized blocks. The size of each block is 16 KB. For a time efficient execution, the log file is read from and written to the disk one block at a time. There are two different types of blocks, the header block and the data block.

3.2.1 The header block

The header block is the first block of the log file and is used to store the following information.

1. Bus Name: This field is a sequence of 32 bytes which is used to store the name of the bus. First 31 bytes in this sequence are used to store a human readable bus name. The last character is set to the null character ('\0'). If the length of the name of the bus is less than 31, then unused bytes are also set to the null character.
2. Timestamp: It is an 8 byte unsigned integer used to store the timestamp for the time of creation of the log file. A timestamp is stored as an integer providing the number of seconds since midnight, January 1, 1970 GMT. The endianness of this field is the same as the endianness of the target machine

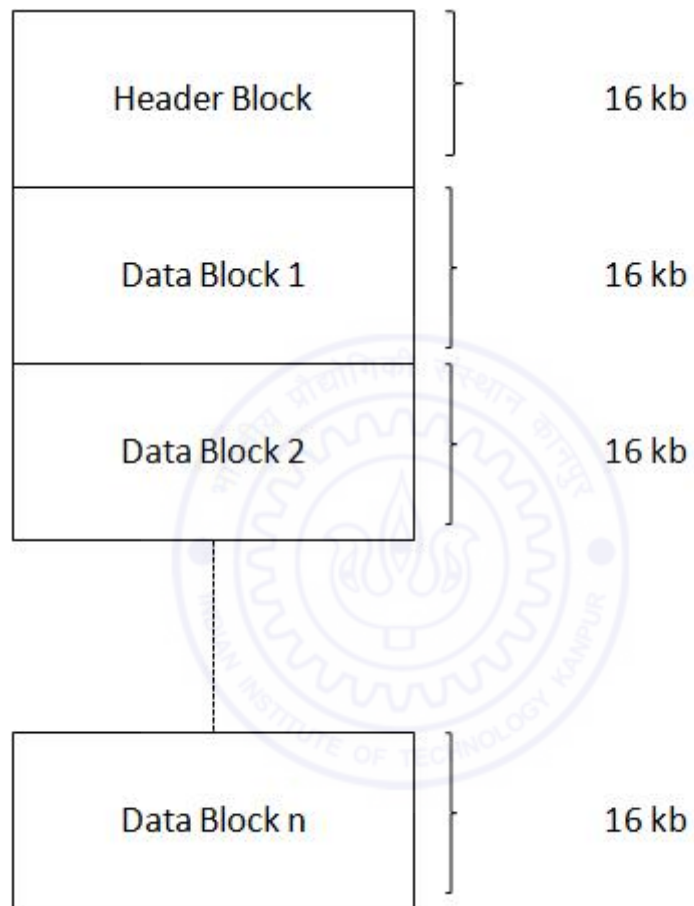


Figure 3.2: The log file format

being simulated.

3. Size of the Address: The size of the address on the bus in number of bits are usually specific to the bus and differ from one bus to another. For a given bus, the size of the address is however always a constant. This value is stored as one byte unsigned integer in the header block.
4. Number of Records: It is an 8 byte integer used to store the number of records present in the log file. The endianness of this field is the same as the endianness of the target machine being simulated.
5. Endianness: It is a one-byte integer used to store the endianness of the machine being simulated. The value 0 is used to indicate the little endianness and the value 1 is used to indicate the big endian machine.

The remaining space in the header block is reserved for future use.

3.2.2 The data block

The data blocks are used to store the log records. The first 8 bytes in a data block contain the cycle counter (i.e. the timestamp) associated with the first log record in the block. Endianness of this field is the same as the endianness of the target machine. Cycle counter is a counter which is initialized to zero at the beginning of the simulation and represents a clock cycle number of the CPU at the start of the bus cycle. The cycle counter is followed by sequentially stored variable length log records. Each log record contains the following data items in the order of storage in the log records.

1. Transaction type: This field is used to indicate type of the bus transaction corresponding to the log record. The transaction type is a single byte unsigned integer and its interpretation is specific to the logger.
2. Offset: The offset field in a record is used to store the difference between the cycle counter associated with the record and the cycle counter associated with the first record in the same data block. The offset of the first record in the a data block is therefore always zero. The offset is stored as a two byte unsigned integer. The endianness of this field is same as the endianness of the processor being simulated.
3. Transaction duration: This field indicates the number of CPU clock cycles taken by the transaction to complete. It is stored as a single byte unsigned integer.
4. Address: This field stores a sequence of bytes representing the address on the bus corresponding to the log record. The endianness of this field is the same as the endianness of the processor being simulated.
5. Size of data: This field indicates the size of the data being transferred in number of bytes in the current bus transaction. It is stored as a two byte unsigned integer. The endianness of this field is the same as the endianness of the processor being simulated.
6. Data: This field provides the sequence of bytes representing the data associated with the bus transaction. The endianness of this field is the same as the endianness of the processor being simulated.

A log record is never split across data blocks. Therefore no partial records are stored in any data blocks. Unused bytes at the end of the data block therefore are set to zero. Since a transaction type cannot be 0, a record starting with 0 indicates the end of the records in that particular data block.



Chapter 4

Library for Log Viewer

In this chapter we describe the library which is used to read the bus log records from the log file. The library is developed in C. First we describe the library interface which includes some data structures and a set of API functions and then we discuss the implementation of the library.

4.1 The Viewer Interface

4.1.1 Introduction

The viewer interface describes the way in which an application can communicate with the bus log viewing library. This includes the data structures and the API functions that can be used by the application. The interface exposes very basic and simple functionality. Therefore, the library can be used by a variety of log viewing applications.

4.1.2 Interface definition

4.1.2.1 Data Structures

Following are the data structures used by the interface.

1. The **Viewer**

This data structure contains the information needed for reading the log records from a log file. It includes the name of the log file, the name of the bus, a pointer to FILE structure corresponding to the log file, the size of the address on the bus in number of bits, number of records that are present in the log file, the data block currently present in the memory, an offset to the next log record to read from the current data block in the memory and the timestamp at the time of creation of the log file. This information is stored in structure **Viewer**.

```

typedef struct Viewer {
    FILE *logFile;                /* file pointer for the */
    char *fileName;               /* name of the log file */
    char *busName;                /* name of the bus */
    unsigned char sizeOfAddress;  /* size of the bus address in
                                number of bits */
    unsigned long long int numRecords; /* number of log records
                                present in the log file */
    unsigned char currentBlock[BLK_SIZE]; /* buffer to store the cur-
                                rent data block */
    unsigned int nextRecordOffset; /*offset to the next log
                                record to read from the cur-
                                rent data*/
    unsigned long long int timestamp; /* the timestamp at the
                                time of creation of the log
                                file */
}Viewer;

```

2. The LogRecord

This data structure is used to store the information of a log record. It includes the type of bus transaction, the cycle counter, the bus transaction duration, a pointer to the buffer for storing the bus address, the size of the data in number of bytes and a pointer to the buffer for storing the data. The log record is read (from the log file) into the following structure.

```

typedef struct LogRecord {
    unsigned char transactionType;      /* type of the bus transac-
                                         tion */
    unsigned long long int cycleCounter; /* the cycle counter */
    unsigned char transactionDuration;   /* number of CPU clock cy-
                                         cles in which the transaction
                                         completes*/
    unsigned char *address;              /* pointer to the buffer stor-
                                         ing the bus address */
    unsigned int sizeOfData;             /* size of the data in num-
                                         ber of bytes */
    unsigned char *data;                 /* pointer to the buffer for
                                         storing the data */
}LogRecord;

```

4.1.2.2 API Functions

The API functions provided by the interface are as follows.

```

1. Viewer *initViewer(
    char *fileName,
    int *errCode
)

```

This function is used to open a log file for reading logs. It takes the log file name and the address of an integer to store error code as input.

On success this function initializes and returns a pointer to the variable of

type **Viewer**. The fields of this variable are initialized to appropriate values from the log file header. In case of failure, a **NULL** is returned and the specific error code is set in an integer variable, pointer to which is given by **errCode**. Following is the list of error codes.

- **EMEM**: Out of memory
- **EFNEXISTS**: File does not exist
- **EOPEN**: File cannot be opened

```
2. int readLog(  
    Viewer *viewer,  
    LogRecord *logRecord  
)
```

This function is used for reading log records one at a time. It reads the log record for the corresponding **Viewer**. It takes the following parameters.

- **viewer**: The pointer to a variable of type **Viewer** previously returned by a call to **initViewer** for the reading the required log file.
- **logRecord**: A pointer to a variable of type **LogRecord** in which the current log record read from the log file is stored.

Upon success, this function returns a constant **SUCCESS**. In case of failure, an error code indicating the type of error is returned. Following is the list of error codes.

- **EVIEWER**: Viewer not initialized.
- **ELAST**: No more records are available.

- ETRUNC: The data is truncated due to insufficient buffer size.
- EREAD: Unable to read record because of file I/O problem.

```
3. int closeViewer(
    Viewer *viewer,
)
```

This function takes a pointer to a variable of type **Viewer** as input and closes the associated log file and frees up any allocated memory. This function returns the value **SUCCESS** on success. In case of failure, an error code indicating the type of error is returned. Following is the list of error codes.

- EVIEWER: Viewer not initialized.
- ECLOSE: Unable to close the file.

```
4. int seekByCycleCounter(
    Viewer *viewer,
    unsigned long long int cycleCounter
)
```

This function takes a pointer to a variable of type **Viewer** and cycle counter as input. After successful call to this function, the values **currentBlock** and **nextRecordOffset** from viewer are modified such that the next record that is read will have the value of cycle counter equal to the value of input cycle counter. If such a record is not present **currentBlock** and **nextRecordOffset** are set so that the value of the cycle counter in the next record that is read will be the highest possible value less than the required value of cycle counter.

This function returns the value **SUCCESS** on success. In case of failure, an error code indicating the type of error is returned. Following is the list of error codes.

- **EVIEWER**: Viewer not initialized.
- **EREAD**: Unable to read the log file.

4.2 Viewer interface implementation details

4.2.1 `initViewer` implementation

The function `initViewer` takes name of the log file as input and opens it. The steps performed by `initViewer` are as follows.

1. A new variable of type **Viewer** is created. The log file is opened and the `logFile` variable is set accordingly. The fields `busName`, `sizeofAddress`, `timestamp` and `numRecords` are initialized from the header block. The value of the `nextRecordOffset` field in the **Viewer** variable is set to 0 to indicate that the current data block is empty.
2. The pointer to the created **Viewer** variable is returned if all the above mentioned steps are executed properly. Otherwise **NULL** is returned to indicate to the calling function that some error has occurred. An error code is set in the variable whose address is stored in `errorCode` indicating the type of error that has occurred. The error codes are explained in the interface definition.

4.2.2 readLog implementation

`readLog` function takes a pointer to a variable of type `Viewer` and a variable of type `LogRecord`. The steps performed by `readLog` are as follows.

1. If the data block in the memory does not have any more records to be read, `nextRecordOffset` is set to 0 to indicate empty data block.
2. If the associated data block is empty (`nextRecordOffset = 0`), the next data block is read from the log file into the memory and the value of the `nextRecordOffset` is adjusted accordingly.
3. The log record is read from the `nextRecordOffset` and the `nextRecordOffset` is adjusted accordingly. The cycle counter for this record is compute by adding the offset to the cycle counter present at the start of the current data block. The information from the record that is read is stored in the corresponding fields in the variable `logRecord`.
4. On success, this function returns the value `SUCCESS`. In case of failure, an error code is returned indicating the type of error as explained earlier.

4.2.3 closeViewer implementation

`closeViewer` function takes a pointer to a variable of type `Viewer` as input. The Log file associated with this variable is closed.

On success, this function returns the value `SUCCESS`. In case of failure, an error code is returned indicating the type of error as explained earlier.

4.2.4 seekByCycleCounter implementation

`seekByCycleCounter` function takes a pointer to a variable of type `Viewer` and `cycleCounter` as input. The steps performed by `seekByCycleCounter` are as follows.

The log files are typically large. A binary search is carried out on the file to seek to the data block whose current cycle counter is smaller than the argument given and there is no other block whose cycle counter is bigger than this.

Once a data block is found, following steps are performed.

1. The data block is read into the `currentBlock` field of the `Viewer` variable.
A sequential search is performed in this data block. If a record with specified cycle counter exists, then the `nextRecordOffset` field in the `Viewer` variable is set to the offset of the first byte of that record in `currentBlock`.
2. If such a record is not present `nextRecordOffset` field is set to the value of the cycle counter in the record which is the highest possible value less than the required value of cycle counter.
3. On success, this function returns the value `SUCCESS`. In case of failure, an error code is returned indicating the type of error as explained earlier.



Chapter 5

The Log Viewer Application

In this chapter we describe the log viewer application. The log viewer is a GUI based tool which uses the log viewing library described earlier to read log files generated by the bus transaction logger[6]. The log files contain the information regarding the bus transactions that take place whenever the execution of certain program on a target system is simulated using functional simulator fsim. By providing suitable representations of the bus transaction related data in the log file, the log viewer aids in the performance analysis of the system. The application is developed in C++ and uses the Qt 4.0[7] cross platform GUI application development library.

5.1 Organization of the application

The log viewer (figure 5.1) consists of a fixed size rectangular window which is used to display the information in the log file.

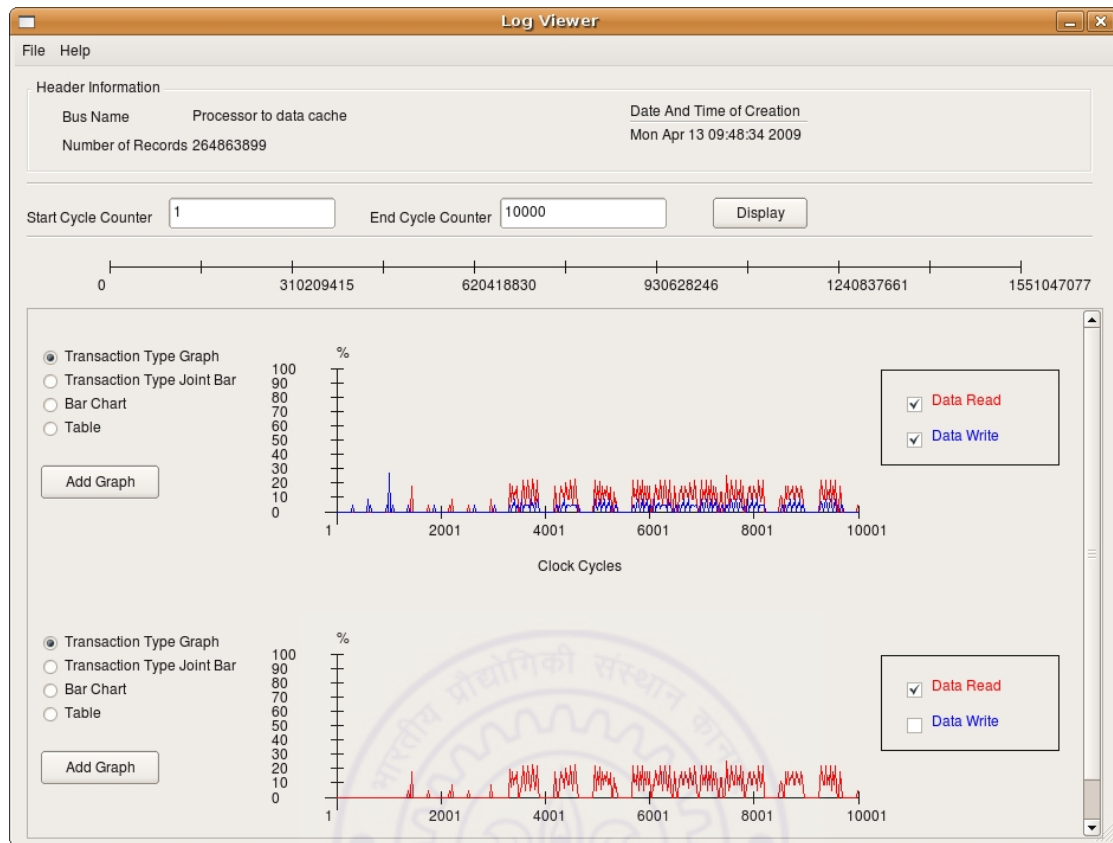


Figure 5.1: The log viewer application window

The application window can be divided into different parts as follows.

5.1.1 The menu

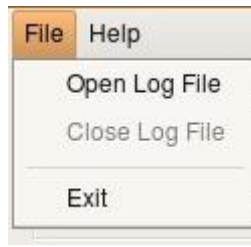


Figure 5.2: The menu

The menu (figure reffig:menu) consists of ‘File’ and ‘Help’ sub-menus. The ‘File’ sub-menu consists of options to open a log file, close an already opened log file and to exit the application. The ‘Help’ sub menu consists of the ‘About’ option, which gives the information about the application.

5.1.2 The header section



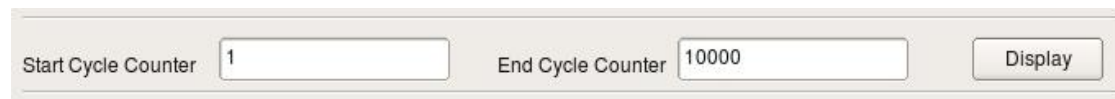
Header Information		Date And Time of Creation
Bus Name	Processor to data cache	Mon Apr 13 09:48:34 2009
Number of Records	264863899	

Figure 5.3: The header section

The header section (figure 5.3) is used to display the information contained in the header of the log file. The information from the log file header that is displayed is the name of the bus , number of log records in the log file and date and time of creation of the log file.

The information displayed in the header section remains unchanged for the entire period during which the log file is open.

5.1.3 The clock cycle input section



The screenshot shows a user interface for selecting a range of CPU clock cycles. It features two text input fields: 'Start Cycle Counter' with the value '1' and 'End Cycle Counter' with the value '10000'. To the right of these fields is a button labeled 'Display'.

Figure 5.4: The clock cycle input section

The clock cycle input section (figure 5.4) consists of two boxes. These boxes are used to take a range of CPU clock cycle numbers as input. Once this range is selected, the log viewer displays the information only during the selected range.

5.1.4 The timeline

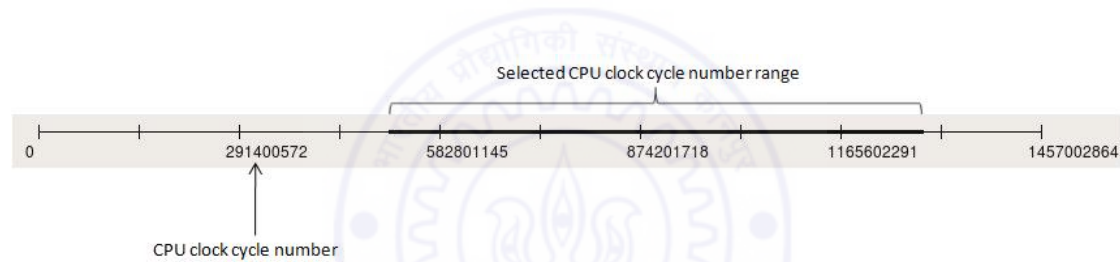


Figure 5.5: The timeline

The timeline (figure 5.5) is another way of selecting the range of CPU clock cycles for bus transaction display. It consists of a line segment with CPU clock cycle markers. By clicking and dragging the mouse pointer on this line segment a range of CPU clock cycles can be selected. The selected range is also displayed in clock cycle input section (figure 5.4) and the log display is restricted to this range only.

5.1.5 The display panel

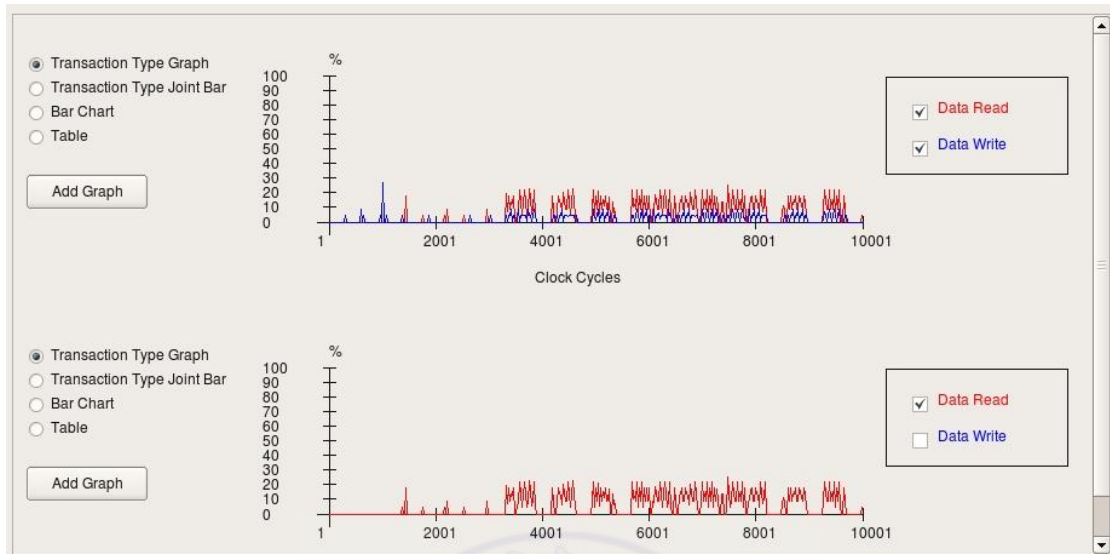


Figure 5.6: The display panel

This panel (figure 5.6) is the most important part of the application. It is used to display the bus utilization time within the selected CPU clock cycle range. There are several ways in which the data can be displayed as follows and discussed later in section 5.2.

- Transaction type graph
- Transaction type joint bar chart
- Transaction type bar chart
- Table

It is possible to view multiple visual representations of the data at the same time using the display panel. This can be done by dynamically adding another data representation below a given data representation. Thus it is possible to

compare different representations of the available bus transaction data. This helps in effectively analyzing the performance of the system. If the area of the window is not enough for displaying the data, the user can use a scroll bar that is provided with the display panel.

5.2 Types of visual data representations

Following are the different types of visual data representations provided by the log viewer application.

5.2.1 The transaction type graph

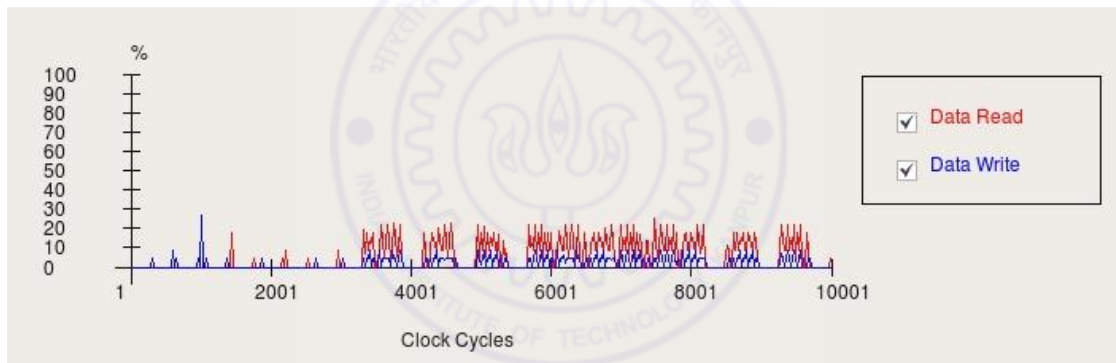


Figure 5.7: The transaction type graph

The transaction type graph is a graph of the percentage bus transactions duration against the CPU cycles for the types of bus transactions supported by the target bus. The user can select bus transaction types using checkboxes. Once the bus transaction types are selected, the information about only the selected transaction types is displayed in the graph. The user can also zoom into certain areas of the graph by either clicking and dragging the mouse in the desired area

of the graph or by double clicking on the required point the graph. It is also possible to zoom out by double clicking the right mouse button. Using this data representation, it is possible to find out the pattern in which different types of bus transaction take place during the selected range of CPU clock cycles. It is also possible to find out the pattern in which the bus stays busy or idle during the selected range of CPU clock cycles. For example if it is found that the bus from the CPU to the L1 instruction cache stays idle for large periods of time, it may indicate large number of cache misses causing the CPU to stall . The exact problem can be found out by corroborating the other bus views such as on the bus from L1 data cache to L2 cache or the bus from L2 cache to main memory.

5.2.2 The transaction type joint bar chart

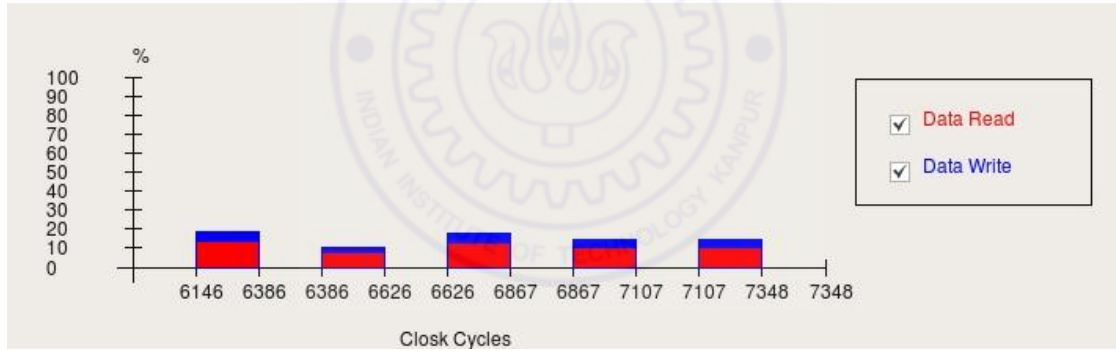


Figure 5.8: The transaction type joint bar chart

Like the transaction type graph (figure 5.8), the transaction type joint bar chart also represents the percentage bus transaction duration against the CPU cycles for all types of the bus transactions supported by the target bus. However, unlike transaction type graph which is a continuous representation of data, the collective data for a certain part of the CPU clock cycle number range is represented in the

form of a joint bar chart. The CPU clock cycle number range is divided into five or fewer parts based on the number of clock cycles in the range and a joint bar chart is shown for each part in the range. This representation is useful when a discrete summery view of the bus transaction data is needed rather than the continuous view. It can be used to get an idea of performance issues at a larger granularity which can be further investigated using the transaction type graph or the table.

5.2.3 The transaction type bar chart

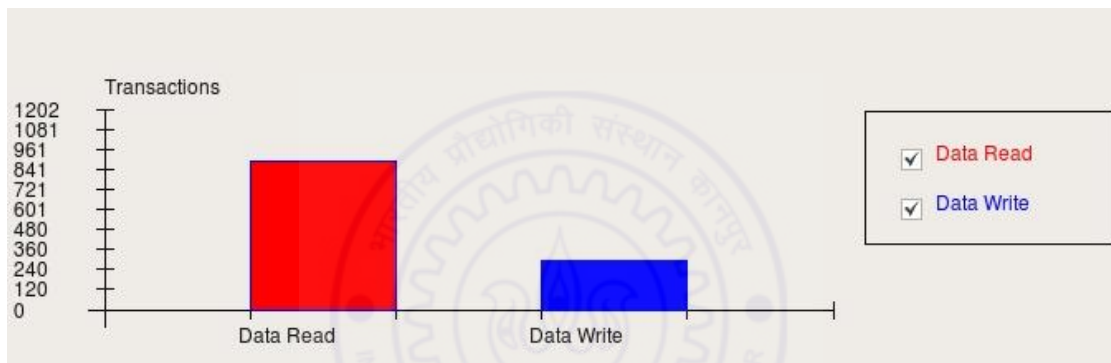


Figure 5.9: The transaction type bar chart

The transaction type bar chart (figure 5.9) is useful when summery of the bus transactions for the entire range of CPU clock cycle numbers is needed. It is simply a bar chart of duration of the different types of bus transactions. Based on the summary,a designer may decide to further investigate the given range using other representations.

5.2.4 The table view



	Transaction Type	Cycle counter	Transaction Duration	Address	Size of Data	Data
1	Instruction Fetch	0	1	80a8	4	0d c0 a0 e1
2	Instruction Fetch	301	1	80ac	4	00 d8 2d e9
3	Instruction Fetch	656	1	80b0	4	04 b0 4c e2
4	Instruction Fetch	657	1	80b4	4	43 00 00 eb
5	Instruction Fetch	658	1	81c8	4	0d c0 a0 e1

Figure 5.10: The table view

The table view (figure 5.10) is the most detailed representation of the bus transaction data. It displays the textual contents of the bus transaction logs for the given range of CPU clock cycles in the form of a table. The fields of the log record (section 3.2.2) that are displayed include transaction type, the cycle counter, the transaction duration, the address and the data including the size of the transaction. This data representation could be used to investigate at finer level the performance related problems once they have been identified by using other data representations. For examples, the table view can be used to find out the details about an instruction whose execution caused cache misses. Such instructions can be located by using transaction type graphs as described in section 5.2.1.

5.3 Features of the log viewer application

The log viewer application provides several features as follows.

- The application is developed in C++ using the Qt 4.0[7] library for cross

platform GUI application development. Thus this application can be built on multiple platforms including Linux and Windows.

- It uses the bus log viewing library described in chapter 4 for reading log files.
- It provides a variety of visual representations of the bus transaction related data from the log file. Some representations provide a summary of data while other representations give a more detailed view of the data. Thus the user can choose a data representation that suits his needs in the process of performance analysis.
- It supports simultaneous viewing of multiple visual data representations. This facilitates the process of performance analysis by allowing comparison between different representations.
- Selecting the range of CPU clock cycles for which the bus transaction related data is to be represented is easy and intuitive since a variety of graphical and text input based methods are provided for this purpose.
- Since the search in a log file as provided by the log viewing library is fast, the application has faster response time when user performs operations involving the selection of the range of CPU clock cycles.
- The log viewer can act as an effective performance analysis tool in the process of hardware software co-design[2].

Chapter 6

Results and Conclusion

The log viewer application is intended to be used as a performance analysis tool in the process of embedded system design. If a program running on an embedded system is unable to meet its performance specifications, we must be able to identify the problem using this application. To test the usefulness of the log viewer application as a performance analysis tool, we analyzed some programs with the help of the log viewer. These programs access data in such a way that they make inefficient use of cache memory in the target system. The target system on which the programs execute is an ARM processor[15] based system which was simulated using the functional simulator FSim[2] interfaced with the bus transaction logger[6]. Once the execution of these programs on the target system was simulated and the corresponding log files were generated, the log files were analyzed using the log viewer application.

6.1 Experimental setup

6.1.1 The host machine

The host machine is the machine on which the simulation of the target system takes place. The functional simulator FSim, the bus transaction logger and the log viewer run on this system. The specifications of the host machine are as follows.

- Processor: Intel Pentium-4 2.4 GHz
- RAM: 512 MB
- Operating system: Ubuntu 8.04 with Linux kernel 2.6.24

6.1.2 The target system

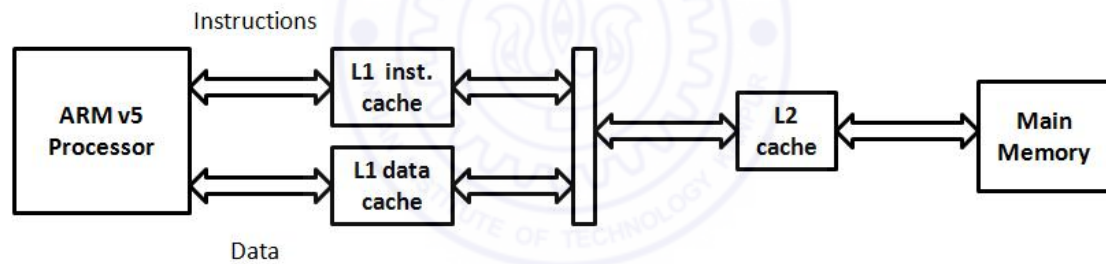


Figure 6.1: The target system

The target system is a hypothetical system designed around ARM processor [15]. It consists of a processor with the ARMv5 architecture. There are two levels of cache memories in the system. The L1 cache is split cache for instructions and data. There are two separate buses connecting the L1 instruction and data caches to the processor. Both instruction and data caches are 32 KB direct mapped caches with line size of 32 bytes. The L2 cache is a unified instruction and data

cache. This cache is 256 KB, 2 way set associative cache with line size of 128 bytes. Total addressable memory is 2^{32} bytes and the available main memory is also of the same size. There are separate buses connecting the L1 caches with the L2 cache and there is a separate bus connecting the L2 cache with the main memory.

6.2 Results

We have analyzed the matrix copy program using the log viewer. The matrix copy program simply copies the contents of one matrix into another. There are two versions of this program. One of them copies the matrix row-wise, and the other one copies the matrix column-wise. The matrices are stored in row major format. The program that copies the matrix row-wise would utilize the cache in a better way as compared to the program that copies the matrix column-wise because the former takes advantage of spatial locality of reference. Therefore the program that copies a matrix row-wise executes faster. We try to demonstrate this fact using the log viewer application.

When the execution of row-wise matrix copy program on the target system is simulated using functional simulator fsm, it takes 37287748 CPU clock cycles while for column-wise matrix copy program it takes 62447048 CPU clock cycles. Following is the 'C' code block that has been used to copy a matrix row-wise.

```

for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
    {
        B[i][j] = A[i][j];
    }
}

```

As we can see, contents of matrix A are copied into matrix B row-wise. Matrices A and B are both integer matrices. Both matrices are square matrices and each row or column contain 500 elements.

Similarly, the following 'C' code block has been used to copy contents of matrix A into matrix B column-wise.

```

for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
    {
        B[j][i] = A[j][i];
    }
}

```

Figure 6.2 shows the contents of the bus transaction log file for the bus connecting the processor and the L1 instruction cache viewed using the log viewer application. This log file was generated during the simulation of column-wise matrix copy program. The CPU clock cycle number range selected for viewing is from 15363246 to 19202140. This range is used consistently during the entire analysis in order to facilitate comparison of bus transactions on different buses during the selected time frame. As we can see, the transaction type graph and bar chart

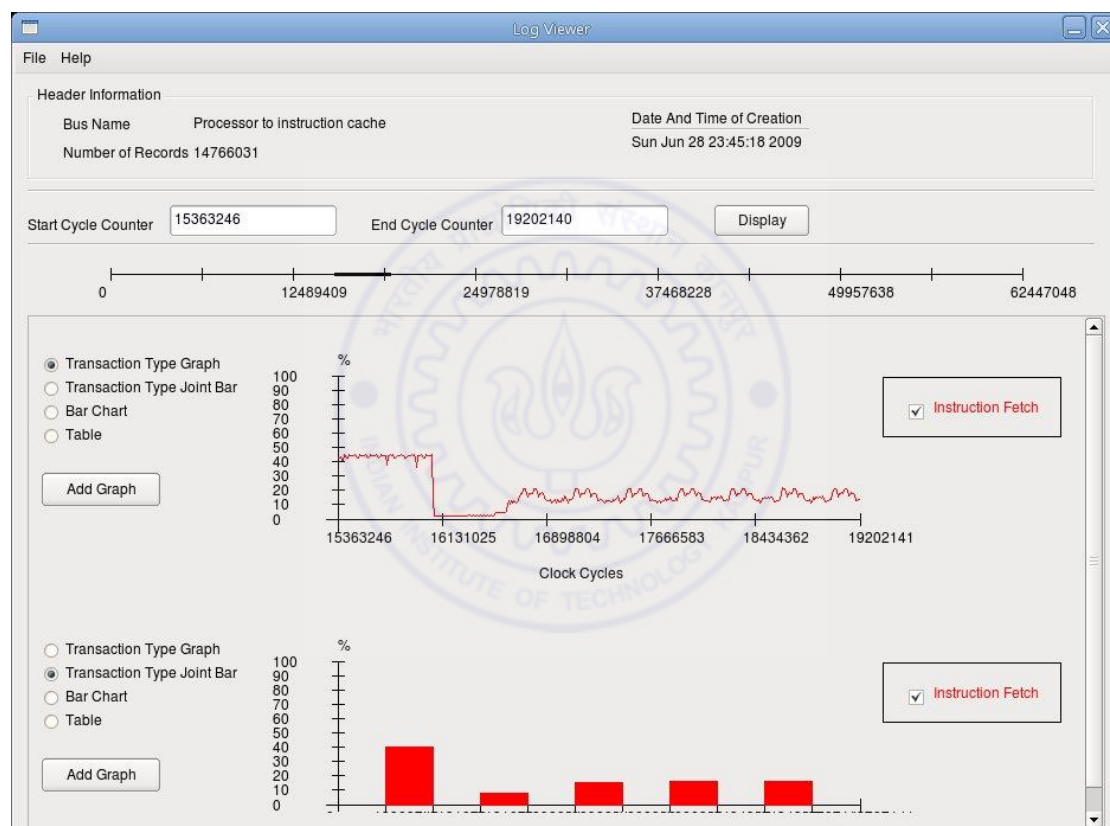


Figure 6.2: Log viewer output for the bus connecting processor and L1 instruction cache during simulation of column-wise matrix copy

shows a drop in the rate at which instructions are fetched by the processor at certain point. This is the point during simulation when the source matrix has

been initialized and the copy operation has begun. This means that the rate of instruction execution is reduced during the matrix copy operation.

Figure 6.3 shows similar output for the row-wise matrix copy program. In



Figure 6.3: Log viewer output for the bus connecting processor and L1 instruction cache during simulation of row-wise matrix copy

this case, we can infer from the transaction type graph and bar chart that the rate of instruction execution during the row-wise matrix copy operation is twice as compared to the rate of instruction execution during column-wise matrix copy operation.

Now we compare the contents bus transaction log file for the bus connecting the L1 data cache and the L2 cache for both the programs (figure 6.4 and figure 6.5).

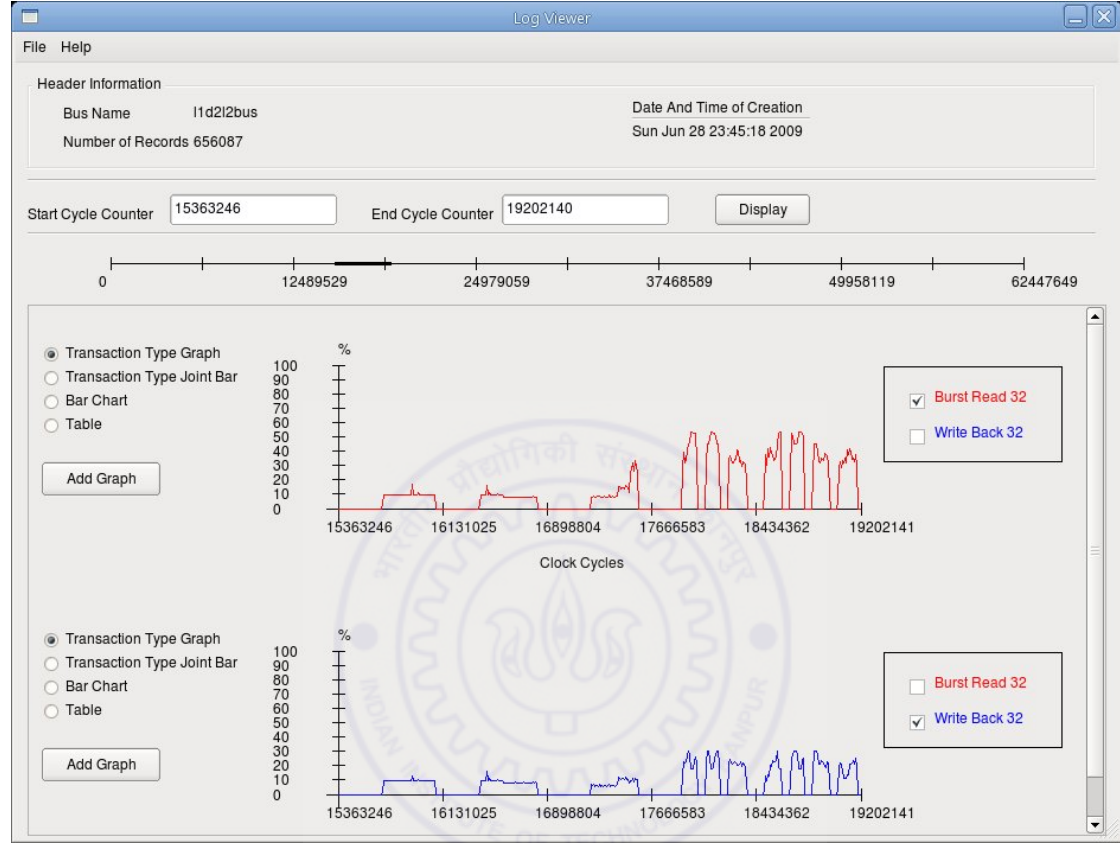


Figure 6.4: Log viewer output for the bus connecting the L1 data cache and the L2 cache during simulation of column-wise wise matrix copy

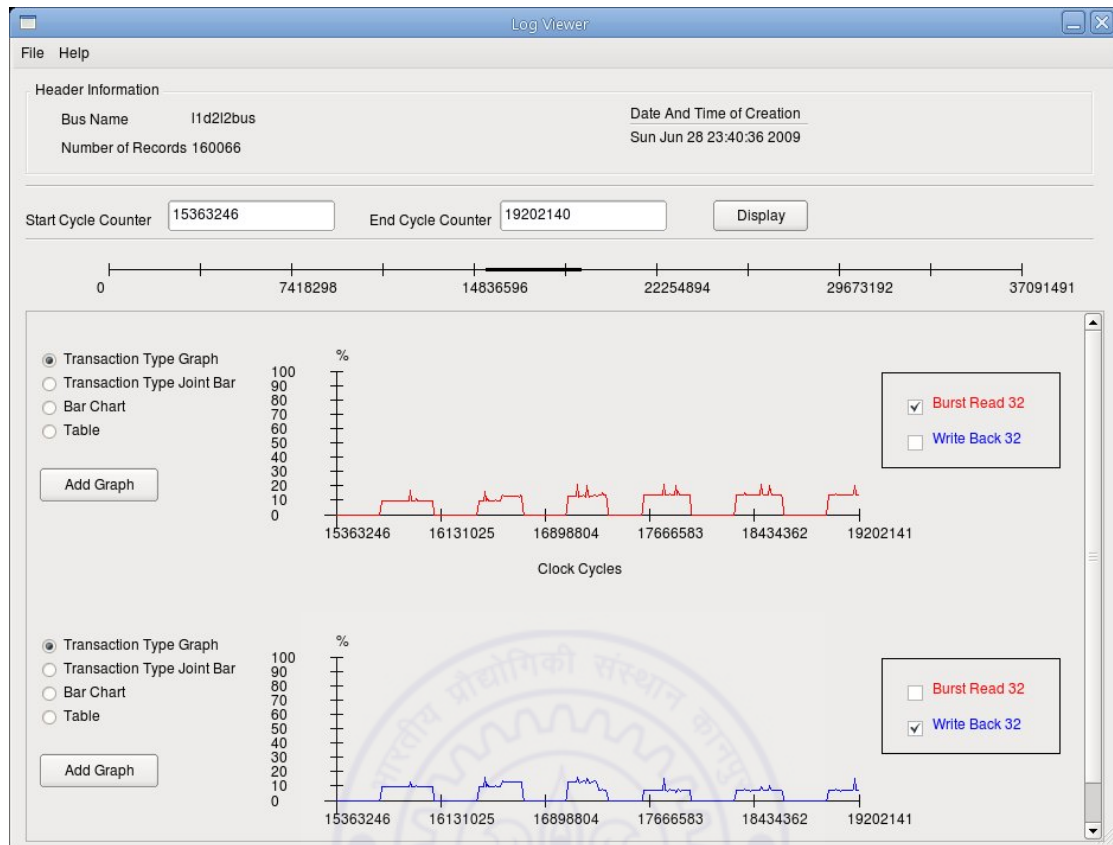


Figure 6.5: Log viewer output for the bus connecting the L1 data cache and the L2 cache during simulation of row-wise wise matrix copy

As we can see, the bus is busier in case of column-wise matrix copy. This means, that there are more L1 cache misses during the simulation of column-wise matrix copy program. This is one of the reasons why the column-wise matrix copy is slower than the row-wise matrix copy program.

Similarly, we also compare the contents of the bus transaction log file for the bus connecting the L2 cache and the memory for both the programs (figure 6.6 and 6.7).

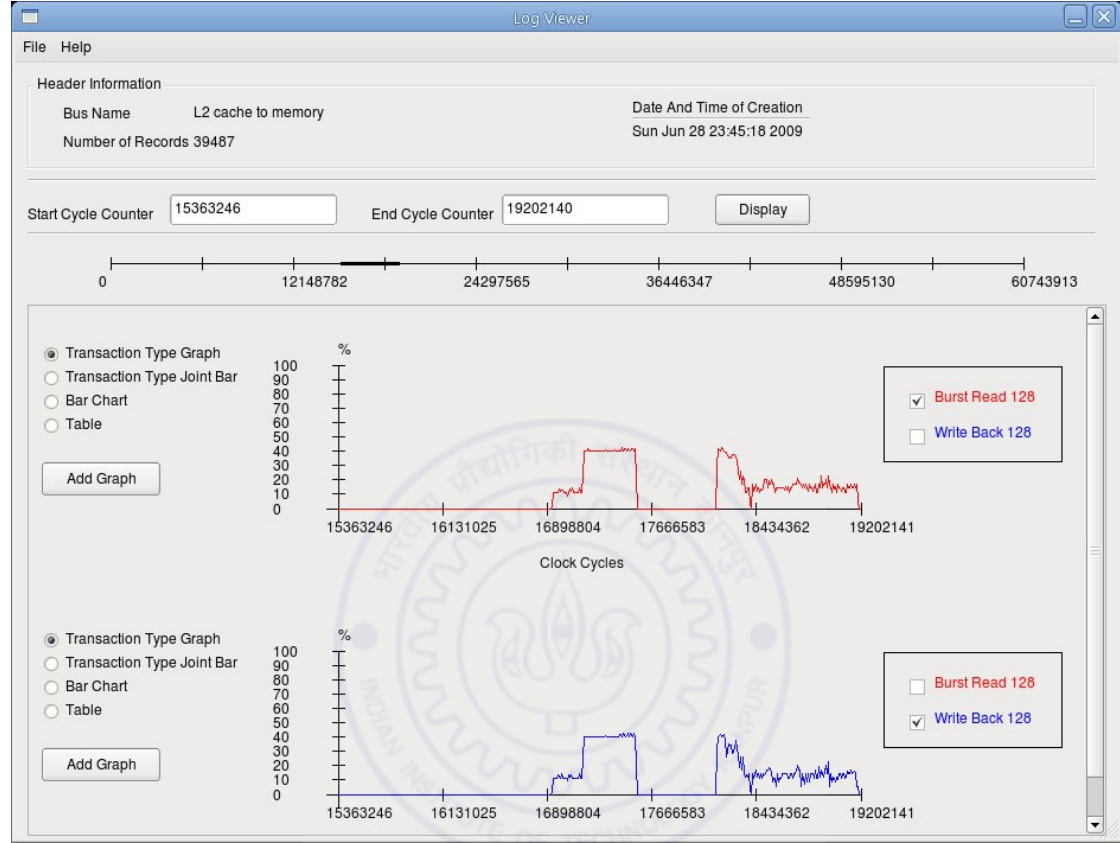


Figure 6.6: Log viewer output for the bus connecting the L2 cache and the main memory during simulation of column-wise wise matrix copy

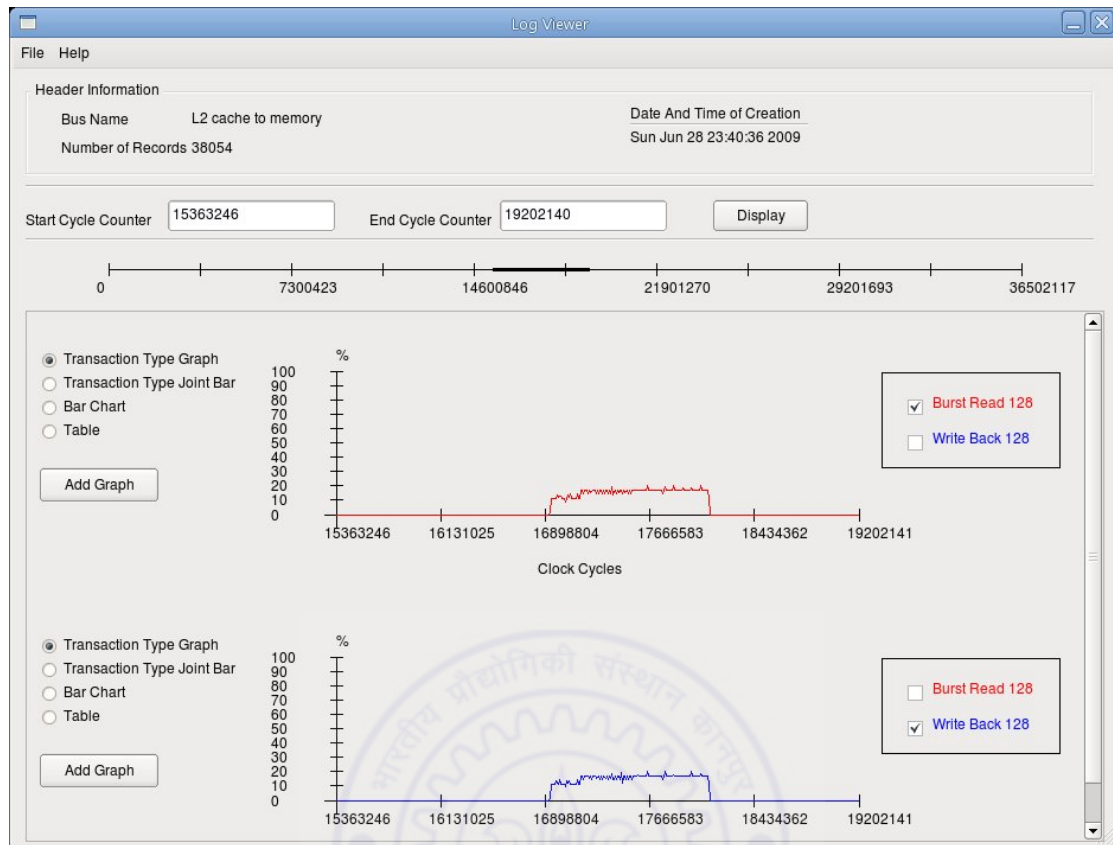


Figure 6.7: Log viewer output for the bus connecting the L2 cache and the main memory during simulation of row-wise wise matrix copy

Even in this case, we can see that the bus is busier in case of column-wise matrix copy. Thus, we can say that there are more L2 cache misses during the simulation of column-wise matrix copy program. This is another reason why the column-wise matrix copy is slower than row-wise matrix copy program.

Therefore the log viewer application was used successfully to find out that column-wise matrix copy program is slower as compared to the row-wise matrix copy program because of inefficient cache utilization.

6.3 Conclusion

In this thesis, we have developed a log viewer application for viewing bus transaction log files. These files are generated by the bus transaction logger during the simulation of execution of a program on a target system using the functional simulator FSim. We have developed a log viewing library which is used by the log viewer application for reading the log files. The log viewer application displays the bus transaction data using a variety of visual data representations to facilitate a better understanding of the program execution. This application can be used as a performance analysis tool in embedded system design. If an embedded system does not meet its performance specification then the designers can identify the problems in the system using this application and make changes accordingly.

6.4 Future Work

Following enhancements can be made to this work in the future. In order to facilitate run time analysis of a system, facility to display the bus transaction data during simulation can be added to the log viewer. This can be done by declaring the log file as a named pipe. This will help us in analyzing performance of programs that execute for a long time such as operating systems. The user interface can be further enhanced by adding facilities like viewing data in multiple log files in the same window in order to facilitate comparison.



Bibliography

- [1] V. Rajesh and R. Moona, "Processor modeling for hardware software code-sign," in *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pp. 132–137, 1999.
- [2] S. Bishnoi, "Functional simulation using sim-nml," Master's thesis, Department of Computer Science, IIT Kanpur, 2006.
- [3] H. Shinde, "Debug support for retargetable simulator fsim using gdb," Master's thesis, Department of Computer Science, IIT Kanpur, 2008.
- [4] "Gprof." <http://www.linuxmanpages.com/man1/gprof.1.php>.
- [5] "Valgrind." <http://valgrind.org>.
- [6] A. Kulkarni, "Generic bus transaction logging for sim-nml based functional simulators," Master's thesis, Department of Computer Science, IIT Kanpur, 2009.
- [7] "Qt 4.0." <http://doc.trolltech.com/4.0/>.
- [8] "Gnu compiler collection." <http://gcc.gnu.org>.
- [9] "Intel vtune analyzer." <http://software.intel.com/en-us/intel-vtune/>.
- [10] "Memcheck." <http://valgrind.org/docs/manual/mc-manual.html>.
- [11] "Chachegrind." <http://valgrind.org/info/tools.html#cachegrind>.
- [12] "Kprof." <http://kprof.sourceforge.net/>.
- [13] "Function check." <http://www710.univ-lyon1.fr/~yperret/fnccheck/profiler.html>.
- [14] "Palm os emulator." <http://sourceforge.net/projects/pose/>.
- [15] "The arm architecture." <http://www.arm.com>.