Generic Bus Transaction Logging for Sim-nML Based Functional Simulator

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Kulkarni Amit Hemant

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2009

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CERTIFICATE

It is certified that the work contained in the thesis entitled "*Generic Bus Transaction Logging for Sim-nML Based Functional Simulator*" by *Kulkarni Amit Hemant* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. Rajat Moona

Department of Computer Science and Engineering,

I.I.T. Kanpur.

July 2009

Contents

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the people who helped me directly or indirectly throughout my thesis. First of all, I would like to thank **Prof. Rajat Moona** for his valuable guidance from starting to end of my thesis. This work would not have been what it is if he wouldn't have been there to guide me. I will always be thankful to him for all the things he taught me during the course of this work.

I would like to thank Nachiket Bhave who was there with me while learning about Sim-nML and other tools designed using Sim-nML, while writing the description of ARM processor in Sim-nML language, while making important changes to existing tools and during the design, implementation, testing and report writing of my thesis. He was always with me not just as a colleague but also as a friend and sometimes as a guide.

I would like to thank Hemant Shinde who taught me 'how to start my thesis'. I would also like to thank all my friends at IIT Kanpur, my family back home and my friends back home who were always supportive of what I was doing. Finally, I would like to thank my department and my institute for providing me all the resources and a good environment which I needed during my thesis.

ABSTRACT

The rapid development and increasing complexity of embedded systems has led to the use of simulators to design the correct system in a faster manner. A good embedded system design is a trade off among important constraints of system design viz. performance, flexibility, power consumption and time to market. Performance of the application programs on embedded system is considered to be an important aspect of design. Simulators help system designers to explore different designs and take design decisions to be able to meet performance criterion.

Sim-nML [8] is a processor architecture description language designed at IIT Kanpur. Several retargetable processor-centric tool generators are also developed which make use of Sim-nML description of a processor as input specification. A retargetable functional simulator generator (fsimg) [6] is one such tool generator which generates a functional simulator (fsim) to simulate functionality of the instruction set of the processor.

In this work we have designed bus transaction logger framework for the functional simulator fsim. The logger framework is used to store the information about the bus transactions that happen on different buses in the system. We have also designed a bus transaction logging library which helps the transaction loggers in logging the information about bus transactions. This information is stored in log files. The information stored in these log files can then be viewed by some viewer application which will help embedded system designers in performance analysis.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 1

Introduction

Performance analysis of a program is a process that aims at analysing the behavior of a program in terms of parameters such as time spent in execution. It is observed that the most of the execution time of any program is spent in some particular portion of the program. In the process of performance analysis, the analyser aims at finding out this portion of the program. The analyser can then explore the different ways of optimizing this portion so that the execution time of the program is reduced. There are two approaches of performance analysis. The first one is static performance analysis in which the algorithm of the program is studied for time and space requirements of the program. This analysis is based on program structure and is done without executing the program on any machine. The second approach is dynamic performance analysis in which the analyser executes the program on a machine or its simulator. While the program is being executed, the information required for performance analysis is collected and then viewed for performance analysis. The collected information is mainly dependent on runtime aspects of the program behavior. This process of collection of information for

analysing the performance is called as profiling.

1.1 Profiling

Profiling is a process of collecting the information for the purpose of performance analysis. Such information may contain the details of cache misses, memory accesses, pipeline hazards etc. A profiler is an application program that collects such information. The profiler executes along with the program. Since the profiler itself may affect the execution behavior of the program, it must be as light weight as possible to minimize its impact on performance measurement. This means the profiler should be designed in such a way that the overhead due to profiler should be as small as possible.

1.1.1 Role of profiler in performance analysis

Many times the information collected at the time of profiling contains the sequence of events that happen in the system during the program execution. A typical system contains a processor, cache and main memory and the I/O devices. These devices in the system are connected to each other using a bus. Whenever a device wants to communicate with another device, a new bus transaction is generated on the bus connecting those two devices. At time of profiling the sequence of such bus transactions that are generated on the bus along with time required to complete those bus transactions can be stored in a file. This type of profiling is called as transaction logging and the file is called as log file.

The logs that store the information about the bus transactions depict the memory access pattern, the pattern of cache misses etc. Usually the region of frequent

memory accesses or region with large number of cache misses can be a bottleneck in the program. These logs can expose such regions with the help of log viewer application. The viewer application can thus be used to find out the bottleneck in the program and programs can then be modified to achieve better performance.

1.1.2 Existing profilers

There are several profiling tools which can be used alongside the applications to collect profile information. The information collected by the profiling tools mainly contains the execution time information in the granularity varying from instructions to functions. Some profiling tools also store events occurring at the time of program execution like function calls, cache misses etc. To collect the architecture dependent information such as cache misses or time taken to execute an instruction etc. the profiler application needs to know the details about the system architecture. For this reason the profiling tools are generally designed to be architecture dependent. Architecture dependent profiling tools help in collecting architecture specific profiling information.

- VTune [4] is a performance analyser tool developed by Intel. It supports time based profiling, event based profiling etc. Time taken by each subroutine can be seen in the results which even can be taken to the granularity of instruction level if necessary. By observing the time taken by a particular instruction, one can observe time including stalls in the pipeline during its execution. VTune is developed to help the performance analysis of systems based on Intel platform.

- Gprof [3] is a performance analyser distributed as a part of the GNU binutils.

Gprof works with any language that is supported by GNU compiler collection (gcc). Visualization tools can also be used with gprof to view the profile data graphically. Gprof generates two types of profile data- flat profile table and call graph table. The flat profile table shows the time a program spent in each function, and the number of times that function was called. The call graph table shows for each function invocation the caller and callee relationship.

- CodeAnalyst [1] is a GUI based code profiler developed by AMD for x86 based machines. As in intel VTune, profiler results of CodeAnalyst contain details of time spent in each subroutine upto the granularity of the instruction level. CodeAnalyst can profile applications using various performance events available on AMD processors.

1.2 Using profilers with instruction set simulators

The existing profilers generally work with the programs being executed on the target microprocessor based system. Program behaviour can also be studied when the execution of program is simulated by an instruction set simulator. An instruction set simulator is a processor centric tool that simulates the behaviour of the processor at the level of abstraction of the instruction set. It may or may not simulate the behaviour specific to level of abstraction of the hardware depending upon implementation. Instruction set simulators usually simulate various CPU registers, program counter (PC) and in some cases the main memory.

An instruction set simulator takes an executable file as an input to simulate

the execution. It reads one instruction at a time, simulates the behaviour of that instruction, changes the values stored in the registers, modifies the state of the processor by changing the contents of PSW register if necessary, increments the program counter and continues with the next instruction. In case of jump instruction the value of the PC will be set to the appropriate value and next instruction will be read from the address stored in PC.

In general the instruction set simulators are made processor dependent by implementing processor specific behaviour. However, retargetable simulators such as fsim [6] can configure themselves using some kinds of processor configuration files such as those written in architecture description languages (ADL). The retargetable instruction set simulators therefore, can take specifications in a high level ADL, parse them and implement processor-specific instruction set simulation.

The instruction set simulators play an important role in simplifying the process of embedded system design. This can be explained with the help of the concept of hardware software co-design.

Embedded system design requires simultaneous hardware and software design. While designing hardware for embedded systems, the designer focuses on performance and power consumption. On the other hand designing software for embedded systems is inclined towards flexibility. A good design is generally a trade off among the important constraints of the system design viz. performance, flexibility and power consumption. Traditionally at first the process of hardware design is carried out. When hardware design is completed, the actual hardware is manufactured and the designed software is tested on the actual hardware. However in this process if there is some issue regarding the hardware design, the designer has to design new hardware; manufacture it; then test again along with the software.

The traditional design methodology expects separation of hardware and software design in the initial stage making it expensive to explore other design alternatives.

As the traditional process of design is expensive and time consuming, there is a need to use alternate approaches to design embedded systems. The hardware software co-design addresses these issues and provides a solution to the problems in question. Here the behaviour of hardware is simulated with the help of hardware simulators. Simultaneously, the software is also implemented and the system is tested for the design constraints. If the system needs some hardware modifications, the hardware design is modified and simulated to study the effects of the changes. Once the hardware design is finalized, the actual hardware is manufactured. This design process has two advantages over the traditional design process. Cost of the design and the time required to design are reduced as any time before the actual hardware is manufactured, the required modifications in hardware can be done in less cost and time.

The process of hardware software co-design is driven by performance analysis of the programs on retargetable instruction set simulator. First of all, the instruction set simulator takes processor description as input. Then the simulator executes the application program. While executing the program, the profiler can collect the information required for performance analysis. The performance analysis is then carried out using the collected information. The results of performance analysis can then indicate issues in design. The design can then be modified to improve the performance of the application program. The modified design can again be simulated with the instruction set simulator and hardware simulator together.

1.3 Related work

Sim-nML [8] is an architecture description language for describing an arbitrary processor architecture. It is designed at IIT Kanpur. Sim-nML focuses on instruction set architecture (ISA) of the processor and it ignores the RTL design details. Sim-nML can be used for automated tool set generation. This tool set contains retargetable gdb [9], retargetable functional simulator [6], retargetable disassembler [7] etc.

Fsimg [6] is a functional simulator generator designed at IITK. Fsim is a retargetable functional simulator, generated using fsimg that simulates the instruction set architecture of processor. It takes specification of the processor in Sim-nML language.

1.4 Overview of this work

This work aims at providing a generic bus transaction logging facility for fsim. We designed a library to log the details about the transactions on a bus. As a part of this library, we designed a generic bus logging interface. The generic interface helps in logging bus transactions for different types of buses present in the system.

The retargetable functional simulator fsim, simulates the ISA of a processor. The controllers for a memory hierarchy that includes a couple of levels of cache memory are also simulated. The cache controllers and the memory controller communicate with each other with the help of a bus. Every time when some information is to be passed among these controllers, a new bus transaction is generated.

A set of API functions which is a part of the generic logging interface are discussed in this work along with their prototypes. The logs of the bus transactions are stored in the files (called log files). The format of the log file is also discussed in this work. The log files then can be viewed using some log viewing application and the pattern of the bus usage can be studied.

1.5 Organization of the report

The rest of the thesis is organised as follows. In chapter 2 we discuss the retargetable functional simulator generator fsimg. We also discuss the execution and memory model of fsim and explore the method of using the logger interface with fsim. In chapter 3 we discuss the design of bus logging library and discuss the generic interface for the instruction set simulators. We also present the format of the log file created and the implementation details of the library. In chapter 4 we will discuss the way of using bus logging library for an ARM processor based system. Finally we conclude and present the results in chapter 5.

Chapter 2

Functional simulator generator

A functional simulator is a simulator which simulates only the functionality of the instructions of a processor. It usually does not simulate the architectural features of the processor such as the processor pipeline, branch prediction etc. Because of this the functional simulators are relatively fast. A functional simulator maintains the state of processor resources such as registers and memory and produces the same output as the program running on an actual processor would produce.

At IIT Kanpur, there had been an effort to generate functional simulator using the architecture description of processors in Sim-nML language [8]. In this approach, simulator generator fsimg [6] takes an architecture description of a processor and a binary executable as input. The output of this simulator generator is a functional simulator fsim. This functional simulator simulates the behaviour of the processor for the given binary program destined to run on that processor.

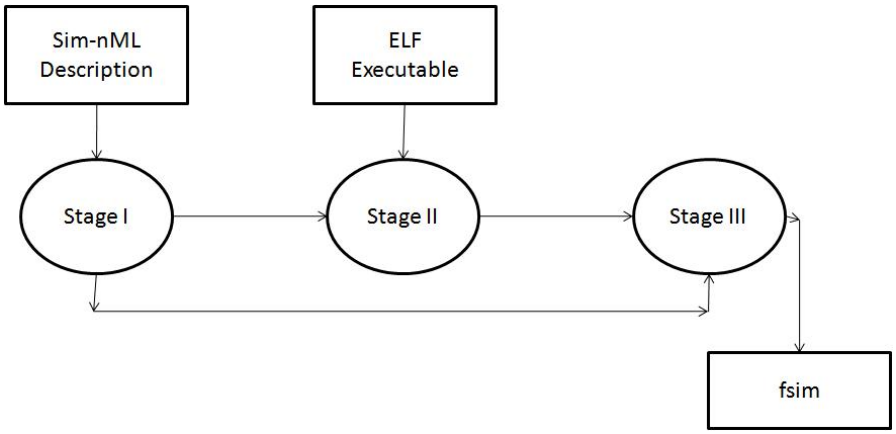

Figure 2.1: Overview of three stage fsimg

2.1 Functional simulator generator fsimg

Fsimg is a tool developed at IIT Kanpur that generates a functional simulator. Fsimg uses 'a compiled simulation' approach during simulator generation. In compiled simulation the process of simulation is divided into two phases. The first phase is a decoding phase in which the instructions that are present in the program are decoded one by one and the details about the instructions such as operands and operators are stored in a table called execution table. The second phase is the execution phase in which the instructions are executed with the help of the execution table created in the first phase. The idea of instruction fetch is realised by process of selection of next entry in the execution table. Thus the instruction fetch behaviour of the program is not necessary to simulate. Further the instructions are decoded at simulator generation time and not at simulation time. The advantage of this approach is that the process of simulation is very fast.

Because of the use of compiled simulation, the simulator generation using fsimg is done in three stages (figure 2.1). In stage I the processor model is generated

using processor description written in Sim-nML language. In stage II the binary executable is decoded using the processor model and the execution table is created. In stage III using processor model and execution table the simulator is generated. The details of these three stages are given below.

2.1.1 Stage I

Stage I (figure 2.2) of the simulator generation focuses on the details of the target processor architecture. This stage is independent of the target executable. The input to this stage is the description of the processor architecture written in Sim-nML. The output of the stage I is the processor model.

The generated processor model has two components. First component is the storage model which defines visible state of the processor including processor registers, flags, memory etc. The second component of the processor model is instruction set model which defines binary image of the processor, binary mask to extract parameters of the instruction, instruction behaviour etc. The information about binary image, binary mask etc. is required at the time of decoding and information about the instruction behaviour is required at the time of execution.

Stage I generates processor model in the form of C source and header files. Some of these files are required in stage II while others are required in stage III.

2.1.2 Stage II

Stage II (figure 2.3) deals with the part of the simulator which is dependent on target executable. Stage II takes the target executable as input. The target executable can be obtained by compiling the program using a compiler for target

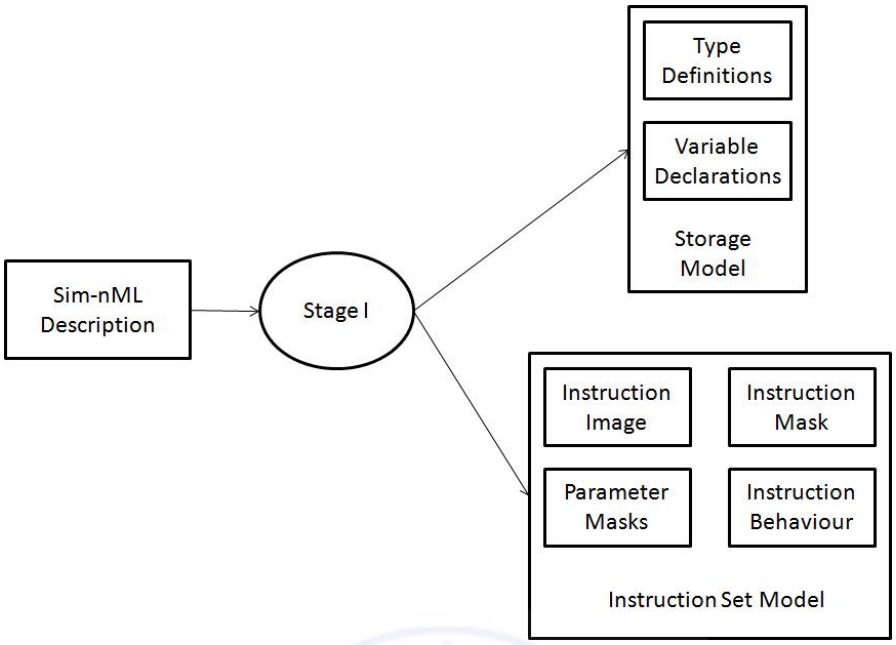

Figure 2.2: Stage I of fsim generation

processor architecture. The target executable program should be in Executable and Linking format (ELF) [2]. This stage also takes as input, some of the files that are generated in stage I.

In this stage the target executable is decoded. During the process of decoding, the sequence of instructions from the target executable is taken as input and a sequence of function calls corresponding to these instructions along with the parameters for the instructions is produced as output. This output is stored in the table known as execution table. Decoder makes use of the instruction set model generated in stage I for decoding purpose and generates the execution table. The image of the main memory containing the loaded image of the target program is also generated in this stage. This memory image is stored in a file in Intel hex format which becomes the input during the run time of simulation to initialize the

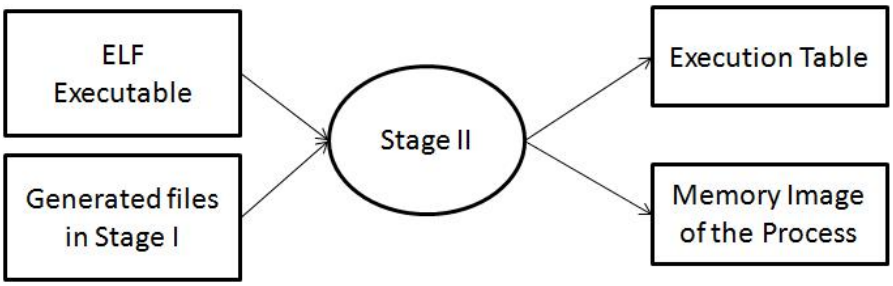

Figure 2.3: Stage II of fsim generation

memory contents.

The files that are produced as output in this stage are required in stage III.

2.1.3 Stage III

Input to the stage III (figure 2.4) is the code generated in stage I and stage II, and static source code of stage III. Static source code of stage III contains the execution logic of the simulator that is independent of the target processor architecture and the executable program. This code also contains the implementation of OS dependent part of the simulator where the system calls in the target program are handled.

The output of this stage is a functional simulator which on execution mimics the behaviour of the target program on target processor.

In stage III, the instruction behaviour captured in stage I is compiled to be used at the run time of execution of functional simulator fsim. The information about the instruction to be executed is taken from the execution table generated in stage II. The program counter behaviour during the instruction fetch is implemented by maintaining an index into the execution table during the runtime of fsim.

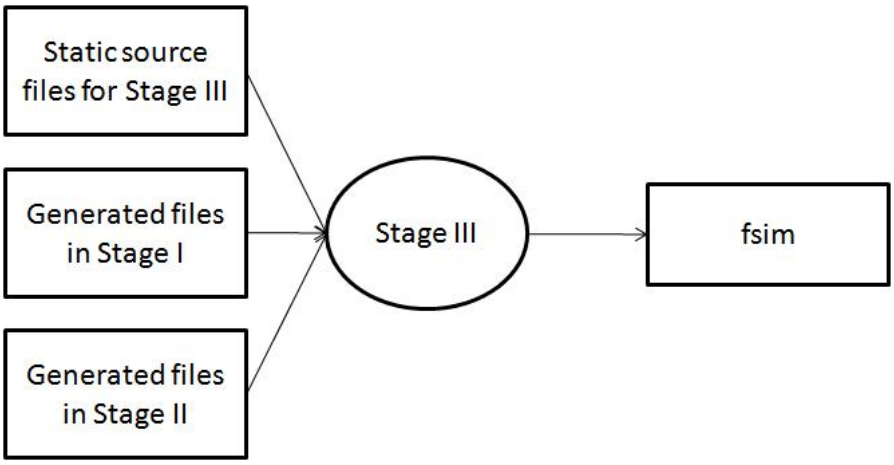

Figure 2.4: Stage III of fsim generation

2.2 The generated functional simulator fsim

The functional simulator is a compiled binary program destined to run on the host machine but simulating the behaviour of the target machine. It encodes the target program and requires the file containing memory image generated in the second stage for initialization of target memory. The values of general purpose registers, PC and Stack Pointer are initialized from the ELF program image of the target.

2.2.1 Algorithm for fsim execution

Step 1: If the memory image file does not exist then goto step 10

Step 2: Read the memory image file

Step 3: Initialize the memory of the target program

Step 4: Initialize the stack pointer and the program counter

Step 5: If stack overflow then goto step 10

Step 6: If PC is invalid then goto step 10

Step 7: Get the parameters for the instruction from execution table

Step 8: Simulate the instruction behaviour

Step 9: Update the processor state and go to step 5

Step 10: Stop.

2.3 Memory model of fsim

Fsim has two different interfaces to access the target processor's memory. First interface provides functions to access only the data from the memory. This interface is built for debuggers and other similar tools to be able to see the data for the program being simulated. The second interface simulates the bus transactions needed to access the data from memory hierarchy. This interface is aimed for implementation of several different memory architectures in the target system. These two different interfaces form two different levels of abstraction.

2.3.1 getMem and storeMem functions

The functions getMem and storeMem provide the first level of abstraction. These functions simply provide a mechanism for host processor to access data from the simulated memory of the target machine. It is useful for tools like retargetable debugger where simulation of bus transactions is not required in order to show the contents of a memory location. getMem is used to get the memory contents of the target machine while storeMem function is used to modify memory contents of the target machine.

2.3.2 writeMem and readMem function

The functions readMem and writeMem provide the second level of abstraction. These functions simulate the bus transactions that take place during the memory access by the processor being simulated. These functions are used by the simulator when data is written in or read from the memory through the instructions being executed by the target processor during program execution. This level of abstraction is also used during instruction fetch operation. readMem function is used to simulate data or instruction read from the memory by target machine and writeMem functions is used to simulate data being written into the memory by target machine.

2.4 Interfacing fsim with bus transaction logger

The simulator uses readMem and writeMem functions to access memory. Here the bus transactions and the behaviour of memory controllers also gets simulated. The readMem and writeMem functions result in logger interface to be activated for recording of the bus transactions. These functions take the address space descriptor as argument and initiate the bus cycles for that address space. The bus interface in turn simulates the artefacts such as cache etc. and may cause the bus cycles on one or more system buses.

The loggers for all the buses should be initialized at the time of initialization of the simulator. This operation creates logger for each bus. Whenever instruction is being fetched, the memory is accessed. So, this memory access should be logged. For the instructions of type load/store the memory is accessed, so those memory accesses should be logged. These memory accesses may lead to cache miss. In

such case, the bus transactions that are happening on the bus connecting second level cache should be logged. At the end of the simulation the log files are closed and the allocated memory is freed.

Chapter 3

Design and implementation of bus transaction logging library

When program execution is simulated using *fsim*, one would also like to know performance characteristics and memory access pattern. For this purpose we need to log bus transactions during simulation. These logs can then be read and analysed in order to gain knowledge about memory access pattern and performance characteristics of the program.

A typical microprocessor based system can have multiple different kinds of buses which support different kinds of transactions. For a system design, we may monitor the bus transactions by logging them. Therefore we need a logging facility which is generic enough to support all kinds of buses and transactions. The functionality provided by our logger interface is very basic and simple. The loggers may define new transactions as well as create different log files for different buses. This makes the logger interface generic enough to be able to capture the bus transactions of the system being simulated.

3.1 Logger interface definition

The logger interface presented here provides a set of functions which help in logging bus transactions. The behaviour of these functions, along with their prototypes, is explained below.

3.1.1 Function `initLogger`

SYNOPSIS

```
#include "logger.h"
LoggerId initLogger(
   char *fileName,
   char *busName,
   unsigned char sizeOfAddress,
   unsigned char endianity,
   int *errCode
)
```

DESCRIPTION

This function creates a new logger for a bus. It takes as input the name of the log file, the name of the bus, the size of address on the bus in bits, endianity of the machine being simulated and the address of an integer to store error code and creates a new log file by the given name. Only the first 31 characters in the name of the bus are meaningful. For a name longer than 31, other characters are ignored. For a name smaller than 31, the name is padded with '\0' before storing in the header of the log. The details of the header of the log are explained later. If the parameter `endianity` is LITTLE, then the *endianity* field in the header of the

log is set to *little endian*. If the parameter endianity is BIG, then the *endianity* field in the header of the log is set to *big endian*.

Upon success the function returns a non-negative integer, specifying the logger id associated with the newly created logger. In case of failure, the value returned is NULL, and the specific error code is set in a variable whose pointer is provided as `errCode`. Following is the list of error codes.

- EMEM: Out of memory
- EFEXIST: File already exists
- ECREATE: File cannot be created
- EWRITE: Cannot write to the log file
- EENDIAN: The specified endianity is invalid.

3.1.2 Function `writeLog`

SYNOPSIS

```
#include "logger.h"
int writeLog(
   LoggerId loggerId,
   unsigned char transactionType,
   unsigned long long int cycleCounter,
   unsigned char transactionDuration;
   unsigned char *address,
   unsigned int sizeOfData,
   unsigned char *data
)
```

DESCRIPTION

This function is used by the bus transaction loggers to write a log record for a bus transaction in the file associated with the specified logger. Following parameters are passed as input to this function.

- `loggerId`: This is the variable of type `LoggerId` previously returned by `initLogger` that specifies the logger to write log record into.

- `transactionType`: The logger interface provides a facility to bus logging interface developers to be able to define new bus transaction types based on the kind of bus transactions. These transaction types are defined by assigning a unique integer identifier to every transaction type specific to the bus. For example, a particular bus may support WRITE, WRITE BURST and READ transactions each of which may be assigned a unique integer identifier. The

bus logger software provides the transaction type for recording in the log file through `transactionType`. Transaction type can have a value between 1 and 255. The value 0 is reserved and is not used for any specific transaction type.

- `cycleCounter`: Whenever, program execution is simulated by fsim, the CPU clock cycle number is set to zero in the beginning. It is incremented after the execution of every instruction by the time steps taken by the instruction. This helps us in maintaining an ordering over the instructions being executed. This parameter specifies the CPU clock cycle number at the start of the current transaction being recorded.

- `transactionDuration`: It is the number of CPU clock cycles taken by the current bus transaction to complete.

- `address`: This is a pointer to array of bytes containing the address on the bus. The bytes are stored in little endian order if the target machine is little endian whereas they are stored in big endian order if the target machine is big endian.

- `sizeOfData`: This parameter gives the size of data being transferred in the current bus transaction in bytes.

- `data`: This is a pointer to the array of bytes in which the data being transferred in the current bus transaction is stored. The bytes are stored in little endian order if the target machine is little endian whereas they are stored in big endian order if the target machine is big endian.

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

- ELOGGER: Logger not initialized
- EMEM: Out of Memory
- EWRITE: Unable to write the record

3.1.3 Function `closeLogger`

SYNOPSIS

```
#include "logger.h"
int closeLogger(
   LoggerId loggerId
)
```

DESCRIPTION

This function takes logger id as input and closes the associated log file. After a successful call, the logger associated with the logger id is not available for logging the transactions. 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.

- ELOGGER: Logger not initialized
- ECLOSE: Unable to close log file
- EWRITE: Unable to write the record

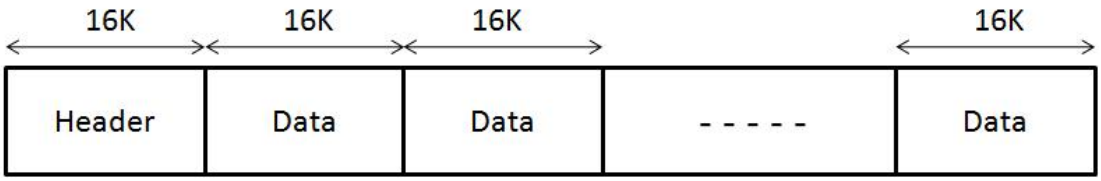

Figure 3.1: Format of the log file

3.2 Log file format

The log file contains several fixed sized blocks. The size of each block is 16 KB. To have 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. The header block is the first block in the file. The format of the log file is shown in figure 3.1.

3.2.1 The header block

The header block (figure 3.2) is the first block of the log file used to store the following information.

1. Bus Name: It is a sequence of 32 bytes which is used to store the name of the bus. First 31 bytes in this sequence are taken from the bus name of the `initLogger` function. The last character is set to null character ('\0'). If the length of the name of the bus is less than 31, then unused bytes are also set to 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

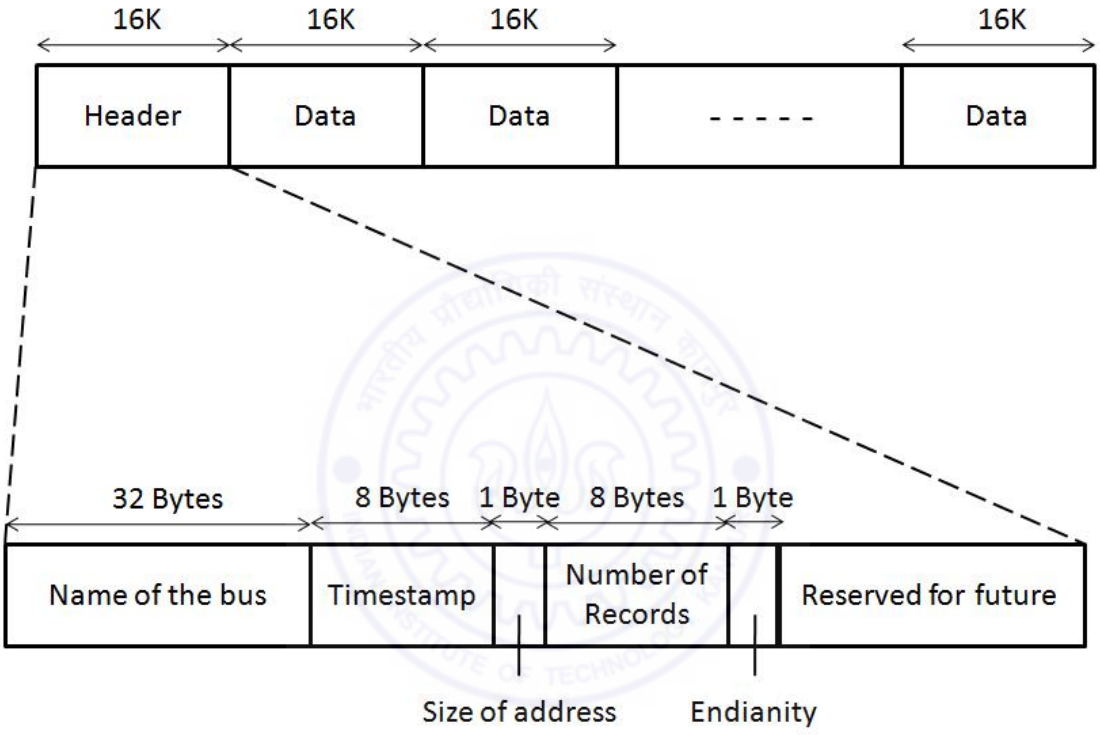

Figure 3.2: Format of the header block

endianity of this field is the same as the endianity of the machine being simulated.

3. Size of the Address: The size of the address on the bus in number of bits may vary from one bus to another. For a given bus, the size of the address is always constant. This value is stored as one-byte unsigned integer in this field.

4. Number of Records: This field is an 8-byte integer and it stores the number of records present in the log file. The endianity of this field is the same as the endianity of the machine being simulated.

5. Endianity: It is a one byte integer and it stores the endianity of the machine being simulated. The value 0 is used to specify that the machine is little-endian and the value 1 is used to specify that the machine is big-endian.

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

3.2.2 The data block

The data block (figure 3.3) is used to store the log records. The first 8 bytes in a data block contain the cycle counter associated with the first log record in this block. Endianity of this field is the same as the endianity 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 counter of the CPU being simulated at the start of the bus cycle.

The cycle counter is followed by sequentially stored variable length log records. Each log record (figure 3.3) contains following data items. The data items are

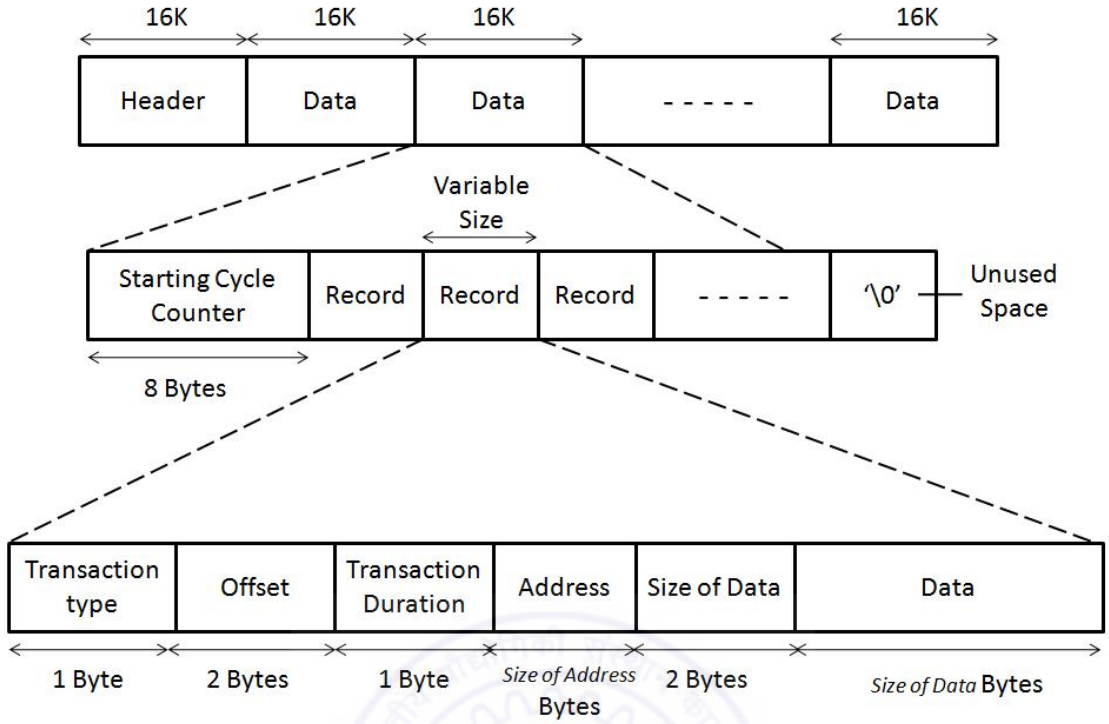

Figure 3.3: Format of the data block

explained in the order in which they are stored in the log records.

1. Transaction Type: This is the bus transaction type of the currently logged transaction stored as a single byte unsigned integer.

2. Offset: The offset is the difference between the cycle counter associated with this record and the cycle counter associated with the first record in this block. The offset of the first record in a data block is always zero. The offset is stored as a two byte unsigned integer. The endianity of this field is same as the endianity of the processor being simulated.

3. Transaction Duration: It is a one byte unsigned integer used to store the number of CPU clock cycles taken by the current bus transaction to complete.

4. Address: This is a sequence of bytes containing the address on the bus. The endianity of this field is same as the endianity of the processor being simulated. The size of address is specified in call to the `initLogger` for this bus which is also stored in the header block.

5. Size of Data: This is 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 endianity of this field is same as the endianity of the processor being simulated.

6. Data: This is a sequence of bytes in which the data being transferred in the current bus transaction is stored. The endianity of this field is same as the endianity of the processor being simulated.

3.3 Implementation of API function

We create a separate logger for every bus. The information needed by a logger includes the name of the log file, the name of the bus, a pointer to the FILE structure corresponding to the file in which the log records are stored, the number of records already logged, a pointer to the current data block in the memory and the size of the used portion of the current data block in memory.

The structure Logger is implemented as given below.

```
typedef struct Logger {
  FILE *logFile;
  char *fileName;
  char *busName;
  unsigned long long int numRecords;
  unsigned char currentBlock[BLK_SIZE];
  unsigned int freeRecordOffset;
  unsigned int sizeOfAddress;
  unsigned char endianity;
} Logger;
```

The pointer to this structure is used to define the type `LoggerId` as follows.

```
typedef Logger* LoggerId;
```

3.3.1 `initLogger` implementation

This function creates a new logger for a bus. It takes the name of the log file, the name of the bus, the size of address on the bus in bits and the address of an integer to store error code as input. The steps performed by `initLogger` are as follows.

1. A new variable of type `Logger` is created. The fields in this `Logger` variable `fileName` and `busName` are set using the values from input parameters to `initLogger` function. A new log file is created and the field `logFile` is set corresponding to this log file. The field `freeRecordOffset` is set to zero to indicate an empty data block. The field `numRecords` is set to zero.

2. The header block is created and the header block fields *bus name* and *size*

of the address are set using the values from the input parameters. The *timestamp* field in the header block is set using the system time of the host machine. The *number of records* field in the header block is set to zero.

3. The pointer to the created `Logger` variable is returned if all the above mentioned steps are executed properly. Otherwise a 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 `errCode` indicating the type of error that has occurred. The error codes are explained in the interface definition.

3.3.2 `writeLog` implementation

This function writes a log record in the file associated with the specified logger. It takes `loggerId` and the other information about a log record as input. The steps performed by `writeLog` are as follows.

1. If the data block does not have enough empty space to store the log record, it is flushed to the log file and the `freeRecordOffset` is set to zero to indicate empty data block. While flushing, unused bytes in the data block are set to zero.

2. If the associated data block is empty (`freeRecordOffset` = 0), the current cycle counter is stored in the beginning and the `freeRecordOffset` is adjusted accordingly.

3. The log record is stored at the `freeRecordOffset` and the `freeRecordOffset` is adjusted accordingly. While storing this `offset` is computed for the value stored in the data record.

4. If these steps are performed correctly, the value of variable `numRecords` is incremented and the value SUCCESS is returned. In case of failure, an error code is returned indicating the type of error as explained earlier.

3.3.3 `closeLogger` implementation

`closeLogger` function takes `loggerId` as input. The steps performed by `closeLogger` are as follows.

1. This function closes the log file associated with the specified logger. It checks if the associated data block in memory is empty or needs to be written in the log file. If it is not empty then the data block is written into the log file after setting the unused bytes to zero. The value of the field `numRecords` of the retrieved `Logger` variable is written in the header of the log file.

2. The log file is closed. If these steps are performed correctly, the value SUCCESS is returned. In case of failure, an error code is returned indicating the type of error as explained earlier.

Chapter 4

Bus transaction logger for a ARM processor based functional simulator

A bus transaction logger is used for logging the bus transactions that take place on a bus in the system being simulated. We have implemented bus transaction loggers for a ARM processor based system being simulated using fsim. Our system supports multiple buses as described later. For each bus in the system, there is a separate bus transaction logger. These loggers make use of the bus transaction logging library to capture the information regarding the bus transactions that happen during the simulation of a program. Wherever required, we have made some assumptions about the design of the system being simulated.

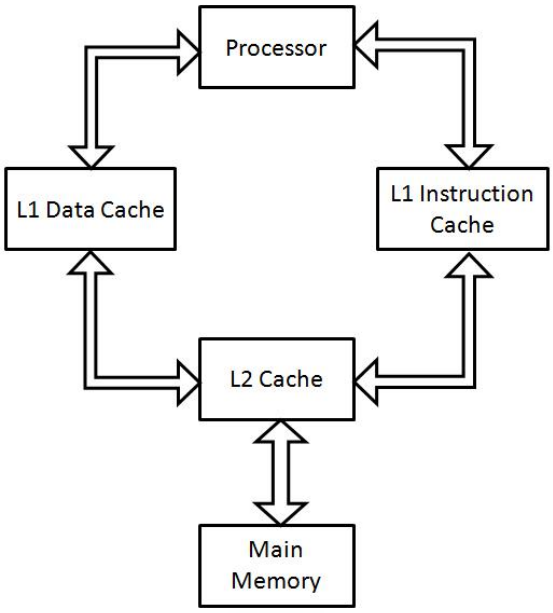

Figure 4.1: Design of the hypothetical ARM based system

4.1 Details of the system

The details of the system that is being simulated are given below (figure 4.1).

- The target system is a system based on ARM architecture. The simulator is generated through the 3-stage process of fsim generation using functional simulator generator (fsimg). The input Sim-nML description is written for ARM v5 architecture based processor.

- ARM processor supports both little endian and big endian memory systems. Our Sim-nML description caters only for little endian architecture and hence the generated simulator for ARM processor supports only little endian architecture.

- Total addressable memory is 2^{32} bytes. We have implemented only one

address space covering the whole main memory.

- The system has two levels of cache. At level 1, data and instruction cache are separate but the level 2 cache is unified.
- The size of L1 data cache is 32 KB. The size of one cache line is 32 bytes resulting in the total number of data cache lines being 1024.
- Similarly the size of L1 instruction cache is 32 KB with 32 byte cache line size and 1024 cache lines.
- L2 cache is a unified cache of size 256KB. It is 2-way set associative cache with 1024 sets. The line size is 128 bytes. LRU policy is used for replacement of a line from the set in L2 cache.
- The processor communicates with L1 data cache and L1 instruction cache with two separate buses.
- L2 cache communicates with L1 data and instruction caches using two separate buses. L2 cache communicates with the memory using another bus connecting L2 cache to memory.
- The bus connecting processor and L1 instruction cache supports only one kind of transaction i.e. 'Instruction Fetch'. In this transaction the processor specifies the address of the next instruction which it has to fetch from L1 cache and corresponding instruction is transferred to the processor from the L1 instruction cache.
- The bus connecting processor and L1 data cache supports two kinds of transactions viz. 'data read' and 'data write'. Both of these transactions are used

for transferring a word. The transaction 'data read' is generated when processor executes a load instruction and the L1 data cache provides the required data. Similarly, the transaction 'data write' is generated when processor executes a store instruction and the data is stored in L1 data cache.

- The bus connecting L1 data cache and L2 cache supports two kinds of transactions viz. 'burst read 32' and 'write back 32'. Both of these transactions are used for burst transfer of the data between the caches. The size of data transferred in one bus transactions is 32 bytes. This is the size of one L1 data cache line. The transaction 'burst read 32' is generated when one L1 cache line is to be transferred from L2 cache to L1 data cache. The transaction 'write back 32' is generated when one L1 cache line is to be written back from L1 data cache to L2 cache.

- The bus connecting L1 instruction cache and L2 cache supports two kinds of transactions viz. 'burst read 32' and 'write back 32'. Both of these transactions are used for burst transfer of the data between the caches. The size of data transferred in one bus transactions is 32 bytes. This is the size of one L1 instruction cache line. The transaction 'burst read 32' is generated when one L1 cache line is to be transferred from L2 cache to L1 data cache. The transaction 'write back 32' is generated when one L1 cache line is to be written back from L1 data cache to L2 cache.

- The bus connecting L2 cache and the main memory supports two kinds of transactions viz. 'burst read 128' and 'write back 128'. Both of these transactions are used for burst transfer of the data between the cache and the main memory. The size of data transferred in one bus transactions is 128

bytes. This is the size of one L2 cache line. The transaction 'burst read 128' is generated when one L2 cache line is to be transferred from main memory cache to L2 cache. The transaction 'write back 128' is generated when one L2 cache line is to be written back from L2 cache to main memory.

- Every transaction on the bus connecting processor and L1 data cache takes 1 clock cycle. Similarly, every transaction on the bus connecting processor and L1 instruction cache takes 1 clock cycle.
- Every transaction on the bus connecting L1 data cache and L2 cache takes 50 clock cycles. Similarly, every transaction on the bus connecting L1 instruction cache and L2 cache takes 50 clock cycles.
- Every transaction on the bus connecting L2 cache and main memory takes 250 clock cycles.
- All caches implement write back protocol.

This system is simulated by fsim. We used the bus transaction logging library to log information about the bus transactions.

4.2 Initializing the loggers

The system being simulated contains three different types of buses. For each bus we create a separate logger. The declaration for the logger of the bus from processor to L1 instruction cache is as follows.

```
LoggerId PtoL1iLogger;
```

Similarly the loggers for other buses are also declared.

The loggers are initialized when the *fsim* is initialized i.e. before starting the simulation of program execution. The bus loggers are initialized by successfully calling `initLogger` separately for each of the declared logger. The call to `initLogger` function for the bus between processor and L1 instruction cache is as shown below.

```
PtoL1iLogger = initLogger(
   "PtoL1i.log",
   "Processor to instruction cache",
   32,
   &(errCode),
   endianity
);
```

Here, `PtoL1i.log` is the name of the log file to be created, `Processor to instruction cache` is name of the bus, 32 is the size of the address on the bus in bits, `&(errCode)` is the address of the variable to store the error code and `endianity` is the endianity of the system being simulated. The endianity of the target system is taken from the Sim-nML description of the target machine.

Similar to the bus logger for processor to instruction cache bus, other bus loggers are also initialized. Cycle counter value is also initialized to 0 before starting the simulation.

4.3 Writing the log entries to the log files

Whenever there is a need of data transfer from one device to the other, a new bus transaction is generated on the bus connecting those two devices. The information

to be stored in the log file about the newly created bus transaction contains the type of the transaction generated, the current value of the cycle counter, address on the bus, size of the data to be transferred and the actual data.

When processor wants to fetch an instruction, a transaction is generated on the bus connecting processor and the L1 instruction cache. L1 instruction cache provides the required instruction to the processor and the information about this transaction is stored in the log file. To store this information in the log file, `writeLog` function is called. So, for each instruction to be fetched by *fsim*, the `writeLog` function is called.

```
errCode = writeLog(
  PtoL1iLogger,
  INST_FETCH,
  cycleCounter,
  1,
  address,
  size,
  data
);
```

Here, `PtoL1iLogger` is the LoggerId of the logger for which we want to record the log entry. `INST_FETCH` is the type of the transaction that is being generated on the bus. `cycleCounter` is the current value of the cycle counter. Number of clock cycles required for this transaction is 1. The address on the bus is given by `address`. This is the address from which we want to fetch the instruction. '`size`' in this case, will be 4 as all the instructions supported by ARM are 4-byte long and '`data`' will be the byte stream containing actual instruction. Error, if any will be

returned to the variable `errCode`. The value of `cycleCounter` will be incremented by 1.

If the required instruction is not present in the L1 instruction then it results in a L1 instruction cache miss. This will require the L1 instruction cache controller to get the required instruction from L2 cache and thus a new transaction will be generated on the bus connecting L2 cache and L1 instruction cache. The information regarding this newly generated transaction will be stored in the log file by a successful function call to `writeLog` as shown below.

```
errCode = writeLog(
  L1ItoL2Logger,
  BURST_READ_32,
  cycleCounter,
  50,
  addrBuffer,
  32,
  arr32
);
```

Here, `L1itoL2Logger` is the LoggerId of the logger for which we want to record the log entry. `BURST_READ_32` is the type of the transaction that is being generated on the bus. `cycleCounter` is the current value of the cycle counter. Number of clock cycles required for this transaction is 50. The address on the bus is given by `addrBuffer`. The size of the data transferred will be 32 and the array '`arr32`' will contain the actual data transferred. Error, if any will be returned to the variable `errCode`. The value of `cycleCounter` will be incremented by 50.

Similar to L1 instruction cache miss, L2 cache can also have the required in-

struction missing. This will lead to L2 cache miss. L2 cache miss is also handled in the same way as l1 instruction cache miss and the required instruction is brought in from main memory. The only difference is if the L2 cache line is dirty, it is written back to the main memory. So, in this case a separate bus transaction is generated for write back operation. In this case also, the different transactions are logged in the respective log files. The transactions that take place on these buses contain of 128-byte bursts. Transaction type for write back from L2 cache is specified as `WRITE_BACK_128`. For all the transactions on the bus from L2 to main memory 250 clock cycles are required so the `cycleCounter` is incremented by 250.

Once the instruction is fetched successfully, *fsim* simulates the execution of the instruction. If the instruction is of type load/store then the memory is accessed. The read proceeds through the cache memories and the main memory in the similar way as that of instruction fetch. In case of write, the dirty bit associated with the cache line is set. Depending upon the type of the instruction (load or store), the bus transaction is generated and corresponding information is stored in the log files by successful call to the `writeLog` function for appropriate loggers.

4.4 Closing the loggers

At the end of the simulation, the loggers are closed by successful calls to `closeLogger` function. The `closelogger` function takes the loggerId of the logger to be closed. The logger for the bus from processor to L1 instruction cache is closed by following function call.

```
errCode = closeLogger(PtoL1iLogger);
```

Here, `PtoL1iLogger` is the logger that is being closed. Error, if any will be

returned in the variable `errCode`. Similarly, other loggers are also closed by successful calls to `closeLogger`.

Chapter 5

Results and Conclusion

In this chapter we discuss the experiments performed to test the bus transaction logging library and for measurements of its performance. While testing for correctness, we verify the contents of the generated log files. We make use of viewer application [5] to read the information stored in log files. While testing for performance, we compare the time spent during the simulation with and without without logging.

5.1 Experimental setup

We tested our bus transaction logging library by interfacing it with functional simulator fsim. The functional simulator for our purpose was generated for ARM v5 architecture. We also simulated a ARM processor based system as explained in chapter 4. The simulations were carried out on an Intel processor based host machine with the following configuration.

- Intel Pentium 4, little endian, 32 bit, 2.4 GHz processor

```
bubblesort.arm:     file format elf32-littlearm

Disassembly of section .init:

00008094 <_init>:
    8094:       e52de004        push    {lr}            ; (str lr, [sp, #-4]!)
    8098:       eb000009        bl      80c4 <call_gmon_start>
    809c:       eb000031        bl      8168 <frame_dummy>
    80a0:       eb0000be        bl      83a0 <__do_global_ctors_aux>
    80a4:       e49df004        pop     {pc}            ; (ldr pc, [sp], #4)
Disassembly of section .text:

000080a8 <_start>:
    80a8:       e1a0c00d        mov     ip, sp
    80ac:       e92dd800        push    {fp, ip, lr, pc}
    80b0:       e24cb004        sub     fp, ip, #4      ; 0x4
    80b4:       eb000043        bl      81c8 <main>
    80b8:       e3a03000        mov     r3, #0  ; 0x0
    80bc:       e1a00003        mov     r0, r3
    80c0:       e89da800        ldm     sp, {fp, sp, pc}

000080c4 <call_gmon_start>:
    80c4:       e59f3014        ldr     r3, [pc, #20]   ; 80e0 <call_gmon_start+0x1c>
    80c8:       e59f2014        ldr     r2, [pc, #20]   ; 80e4 <call_gmon_start+0x20>
```

First Instruction

Second Instruction

Third Instruction

Fourth Instruction

Figure 5.1: Disassembly for start function of *bubblesort.c*

- 512 MB RAM
- Ubuntu 8.04 Linux distribution with Linux kernel version 2.6.24

5.2 Testing for correctness

We simulated different programs using the functional simulator fsim with an interface to our bus transaction logger. Information about various bus transactions carried over different buses in the system during the execution was stored in various log files. We viewed contents of the log files with the help of viewer application [5]. We discuss the simulation of the program *bubblesort.c* and the log files created during its simulation.

In figure 5.1, we can see the first four instructions to be simulated. As the fourth instruction is an unconditional branch to the main function, the next instruction

	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 5.2: Output of the viewer application for processor to L1 instruction cache bus for *bubblesort.c*

fetched will be the first instruction of the main function. In the figure 5.2, we can see the information stored in the first log record. The type of transaction is INST_FETCH. The value of cycle counter is 0. Transaction duration is 1 as only 1 CPU clock cycle is required to transfer data from L1 instruction cache to processor. Address is the address of the first instruction. Size of data is 4 as we are transferring a 4-byte instruction and the data represents stream of bytes of the fetched instruction. We can also see that the opcode bytes are fetched in little endian order. Similarly we can verify other instructions that are fetched.

First instruction fetch misses in both the caches taking 50 clock cycles penalty for L1 cache miss and 250 clock cycles penalty for L2 cache miss. One clock cycle is required to execute the instruction. Therefore the cycle counter of second log record is 301.

The second instruction is a push instruction which causes memory write from processor. This therefore generates a transaction on the bus from processor to L1 data cache. In this instruction, values of four registers are stored in the memory. First one of these four data write operations misses in both the caches. Next two are written into the memory in one clock cycle due to a hit in L1 cache. The fourth

	Transaction Type	Cycle counter	Transaction Duration	Address	Size of Data	Data
1	Data Write	302	1	26fb8	4	00 00 00 00
2	Data Write	603	1	26fbc	4	c8 6f 02 00
3	Data Write	604	1	26fc0	4	00 00 00 00
4	Data Write	655	1	26fc4	4	b0 80 00 00
5	Data Write	960	1	26fa8	4	c4 6f 02 00

Figure 5.3: Output of the viewer application for processor to L1 data cache bus for *bubblesort.c*

data write operation experiences a miss in L1 cache and hit in L2 cache (as can be seen from figure 5.3). In a similar manner several other instructions and their corresponding logs were verified.

5.3 Testing for performance

We also used our bus transaction logger for measurement of its performance. For this purpose we observed the time taken by different programs with and without enabling the bus transaction logging during simulation. The programs that we have tested and the corresponding results are as below.

1. *bubblesort.c* is a program which sorts an array of 2000 integers in ascending order. Table 5.1 provides the performance related measurements of the simulation of this program. Table 5.2 provides the sizes of the log files created during the simulation.

Number of instructions simulated	130005023
Simulator clock cycles taken	168047038
Simulation time without logging	78.814 sec
Number of instructions per second without logging	1649516
Simulation time with logging	109.527 sec
Number of instructions per second with logging	1186967
Time spent in file system calls	10.522 sec

Table 5.1: Information about simulation of bubblesort.c

Log file for the bus from	*Size of the log file*
Processor to L1 instruction cache	1.8 GB
Processor to L1 data cache	510 MB
L1 data cache to L2 cache	32 KB
L1 instruction cache to L2 cache	32 KB
L2 cache to main memory	32 KB

Table 5.2: Sizes of the log files for bubblesort.c

2. *heapsort.c* is a program which sorts an array of 1000 integers in ascending order using the heap sort algorithm. Table 5.3 contains the performance related measurements of the simulation of this program. Table 5.4 contains the sizes of the log files created during the simulation.

Number of instructions simulated	27562512
Simulator clock cycles taken	42110370
Simulation time without logging	21.784 sec
Number of instructions per second without logging	1265264
Simulation time with logging	29.158 sec
Number of instructions per second with logging	945281
Time spent in file system calls	3.159 sec

Table 5.3: Information about simulation of heapsort.c

Log file for the bus from	*Size of the log file*
Processor to L1 instruction cache	370 MB
Processor to L1 data cache	195 MB
L1 data cache to L2 cache	32 KB
L1 instruction cache to L2 cache	32 KB
L2 cache to main memory	32 KB

Table 5.4: Sizes of the log files for heapsort.c

3. *matrixmult.c* is a program which multiplies two integer matrices of size 100x100. Table 5.5 contains the performance related information about the simulation of this program. Table 5.6 contains the sizes of the log files created during the simulation.

Number of instructions simulated	49672427
Simulator clock cycles taken	74305449
Simulation time without logging	34.595 sec
Number of instructions per second without logging	1435826
Simulation time with logging	45.460 sec
Number of instructions per second with logging	1092662
Time spent in file system calls	4.433 sec

Table 5.5: Information about simulation of matrixmult.c

Log file for the bus from	*Size of the log file*
Processor to L1 instruction cache	666 MB
Processor to L1 data cache	258 MB
L1 data cache to L2 cache	4.2 MB
L1 instruction cache to L2 cache	32 KB
L2 cache to main memory	164 KB

Table 5.6: Sizes of the log files for matrixmult.c

4. *matrixcopy.c* is a program which copies elements of one 500x500 matrix of integers to another. Table 5.7 contains the performance related information about the simulation of this program. Table 5.8 contains the sizes of the log files created during the simulation.

Number of instructions simulated	59032031
Simulator clock cycles taken	387478351
Simulation time without logging	40.776 sec
Number of instructions per second without logging	1447715
Simulation time with logging	57.604 sec
Number of instructions per second with logging	1024790
Time spent in file system calls	6.605 sec

Table 5.7: Information about simulation of matrixcopy.c

Log file for the bus from	*Size of the log file*
Processor to L1 instruction cache	792 MB
Processor to L1 data cache	269 MB
L1 data cache to L2 cache	129 MB
L1 instruction cache to L2 cache	32 KB
L2 cache to main memory	79 MB

Table 5.8: Sizes of the log files for matrixcopy.c

5.3.1 Analysis of the results

We observed the performance of the simulator with and without bus transaction logging. The overhead due to logging is the extra time taken by the procedure of logging the bus transactions. For example in table 5.1 we can see the time spent in simulation of *bubblesort* with bus transaction logging is 109.527 seconds and time spent in simulation without bus transaction logging is 78.814 seconds. So, the extra time spent due to logging is 30.713 seconds or 28.04% of the total time. We also observed that out of 30.713 seconds of overhead, 10.522 seconds are spent

in file system calls for opening, writing to the and closing the log files.

As discussed in chapter 3, we write the information in the log files block by block. Therefore only a small part of total time is spent in file system calls. For different programs, time spent in file system calls varies from 9% to 12% of total simulation time. Given the size of the log files that are created during the simulation, the overhead due to file system calls can be considered acceptable.

In general different programs spent from 23% to 30% of total simulation time in logging. Since the logging API functions are called frequently during simulation (at least one call per simulated instruction) and since the API provides detailed functionality, the overhead due to logging can be treated as acceptable.

5.4 Future Work

Currently the log file format is designed in such a way that one can efficiently access the log records for given value of cycle counter. In future, the log file format can be modified (if necessary) to access the log records efficiently for the queries on transaction type, address ranges etc. Also, an attempt can be made to reduce the overhead due to logging further.

Bibliography

[1] *Amd codeanalyst performance analyzer for windows.* `http://developer.amd.com/cpu/CodeAnalyst/codeanalystwindows/Pages/default.aspx`.

[2] *Elf, bsd file formats manual.* `http://www.linuxmanpages.com/man5/elf.5.php`.

[3] *Gnu gprof.* `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`.

[4] *Intel vtune pefromance analyzer 9.1 for windows and linux.* `http://software.intel.com/en-us/intel-vtune/`.

[5] N. Bhave, *Bus transaction log graphing tool for hardware software co-design applications*, tech. report, Department of computer Science, IIT Kanpur, July 2009.

[6] S. Bishnoi, *Functional simulation using fsim*, tech. report, Department of computer Science, IIT Kanpur, May 2006.

[7] N. Dahra, *Disassembly and parsing support for retargetable tools using simnml*, tech. report, Department of computer Science, IIT Kanpur, May 2007.

[8] V. Rajesh and R. Moona, *Processor modeling for hardware software codesign*, in International Conference on VLSI Design, 2000, pp. 132–137.

[9] H. Shinde, *Debug support for retargetable simulator fsim using gdb*, tech. report, Department of computer Science, IIT Kanpur, July 2008.