Memory Profiling in Sim-nML

A Project Report Submitted
in Partial Fulfillment of the Requirements for
CS499
by

Bhupesh Chandra
(Y3095)

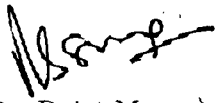
under the guidance of

Prof. Rajat Moona
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur.
May 5, 2007

Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur.

Certificate

This is to certify that the work contained in this report titled "*Memory Profiling in SimnML*" , by *Bhupesh Chandra (Y3095)*, has been carried out under my supervision.



May 4, 2007

(Dr. Rajat Moona)
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur.

Abstract

The complexity, market demands and standards have been ever increasing in the domain of embedded systems. This has led to the designers to use automated tool sets for rapid prototyping and analysis. Given a processor description, these tools facilitate the automated generation of the processor specific tools. **Sim-nML** is a processor description language used to write processor models. Modelling a new processor using Sim-nML is not enough in itself. One needs to evaluate the performance of the softwares aimed to be run on this new processor model. For example, one might be interested in analyzing average memory access per CPU cycle. Thus tools such as Debugger, Disassembler, Profiler etc. need to be developed for such an architecture. These tools are processor centric tools. In case of a processor architecture described using Sim-nML, we need to develop a generic tool, i.e. the tool-architecture depends on the architecture of Sim-nML and does not depend on the target processor architecture description.

In this work, we have developed a **Profiling Tool** which is used to evaluate the performance of the benchmarks in target environment when run with simulators based on Sim-nML description. The profiler architecture has been developed so as to keep it independent of the instruction set description. This essentially required changes to be done in the Sim-nML memory model and also changes in its language constructs so as to facilitate addition of profiling information. After adding profiling support in Sim-nML, profiler architecture was developed which consists of a data logging part and the Viewer part. Finally, profiles were obtained for few test programs for functional simulation of PowerPC architecture written in Sim-nML.

Acknowledgements

I am immensely thankful to my guide, Professor Rajat Moona, for his valuable support and guidance in completing this project. Working under him and benefiting from his expertise in the field was a valuable learning experience for me. I am also thankful to Mr. Nitin Kumar Dahra, MTech student at Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, for his support and ideas for the successful completion of the project.

Contents

1 Introduction

Instruction set simulators are indispensable tools in the present day scenario of architecture design and processor manufacturing. Simulators take the target executable (binary) and execute the instructions. They are the tools which actually let the designers to look into the performance issues without the actual hardware available on chip. Simulator is a processor centric tool. Some other processor centric tools are disassembler , debugger and profiler. They all gather information which helps one assess a particular architectural design from the application point of view. This type of analysis is a run time analysis also called as the dynamic program analysis. Profiler gathers relevant information from the instructions being executed in the binary. The size of the information collected depends primarily on the number of instructions executed, which could be very large in some cases. For collecting such a huge amount of data, profilers use various techniques such as the hardware interrupts, code instrumentation, operating system hooks, and performance counters.
Computer Architects need profilers to judge the behaviour of the application programs on the newly designed architecture. Software designers use profilers to make their programs behave efficiently up to a desired mark. Such an extensive and pervasive use of profilers in the existing hardware/software domain has been the motivation for this work. An example of one such standard profiler is the GNU profiler *gprof* [6]. GNU *gprof* allows one to learn where the program spent its time and which functions called which other functions while it was executing. This information can show which piece of code must be optimized in order to achieve faster execution. Also it would show which function is being called more or less than the expected, eventually leading one to spot bugs in the code. Another example is Seamless [7], a proprietary product from Mentor Graphics. It has many advanced features in addition to the simple profiling features.
Sim-nML is a processor description language developed at IIT Kanpur. Apart from the processor architecture description, it also provides construct to describe memory. A Sim-nML developer is the one who would write a description of a processor architecture in Sim-nML. The described processor architecture is called target architecture. This description would then be converted into an intermediate representation (IR)[8] with the Sim-nML compiler. Tools based on Sim-nML descritption for a given processor architecture take this IR as the input. Disassembler and Functional Simulator are examples of such tools. In earlier works related to Sim-nML, some major tools designed are disassembler[8], a retargetable functional simulator[10] , cache simulator[9] and Functional Simulator [3].

2 Profiler Design

Profiler should have certain desirable features as

- Profilers should be as efficient as possible. They should in themselves not become an overhead for the simulation process which may lead the simulation to run for larger number of hours.
- Typical simulation of benchmarks may take several days to execute. Normally, designers do not want to wait for that duration and would want to continuosly monitor the performance of the program during it's execution rather than at the end.

Our profiler consists of two functionalities:

- Data Logging

- Log Viewing

We discuss the above two functionalities in detail along with the implementation issues.

2.1 Data Logging

Execution of a binary on a processsor could be abstracted as performing three basic tasks of **instruction fetch**, **instruction decode** and **instruction execution**. The data logger part of the profiler can log suitably during the execution of the binary so as to collect the data which would be used to construct more meaningful parameters which ultimately help evaluate the performance of the program on the processor architecture.
As discussed above, the log file is generated as the binary of a program is executed. Thus the only inputs to the profiler are the architecture and the binary that is being executed. The data contained in this log file could be used to reconstruct some meaningful parameters of the program execution. The parameters are decided first and suitable data is logged. Our profiler seeks to capture only the memory operations during the execution. We are mainly concerned with the following events in the binary

1. Instruction Fetch
2. Data Read
3. Data Write

The data read/write may be a memory read/write or it may be cache read/write (if there is one). The same is true for the instruction read/write. Such an information can be ultimately used to construct metrics like number of read/write misses which are a measure of the performance for the binary being executed.

2.2 Log Viewing

Viewer is that part of the profiler which would take as input the log file generated by the data logger and would then construct performance parameters for the binary in the form of graphs and statistical data. For example, it could tell the total percentage of memory reads/writes for a particular region of execution. If this percentage is highly undesirable, the binary could then be optimized for desired percentage level. Infact, if this percentage is highly unexpected, then a possible bug like memory leak might be detected. A graphical representation makes it convenient to analyse the performance issues.

3 Tools based on Sim-nML

As discussed earlier, major processor centric tools like Disassembler and Functional Simulators have been generated using Sim-nML. Profiler is one such tool to be generated using a target Sim-nML description. Since the data logging part of profiler works at the time of simulation of the target binary, Functional Simulator is the tool of our interest.

3.1 Functional Simulator

Fuctional Simulator tool was developed in the work done by Surendra Vishnoi [3]. Functional Simulator simulates the execution of a program on the target architecture. The host architecture is the one on which the simulator executes. So apart from the IR, we have another input i.e. the binary of the program for the target architecture. The type of simulation done by the

Functional Simulator is compiled simulation. That is, it decodes the binary for it's instructions prior to the start of the execution. The advantage here is that the same instruction need not be decoded again and again if it is executed more than once. Finally the Functional Simulator simulates the execution of the binary. Functional simulator provides memory management and system call handling for the simulation of the binary, as discussed in the next section.

In short, functional simulation requires the following three stages

- Conversion of the Sim-nML description to Intermediate Representation . This generates the C-code corresponding to the target architecture instructions.
- Decoding the ELF[11] binary and reading the ELF sections. The ELF information is then dumped into a different set of files which would be required by the next stage. Currently, the Functional Simulator supports decoding of only ELF binaries.
- Simulating the binary. This is done by reading the information that has been generated in the last two stages.

4 Memory Model

As stated earlier, Functional Simulator provides memory management and system call handling to simulate the execution of the target binary. The system calls from the target are redirected to the host operating system. To provide memory management support, various memory models could be implemented. One such model is a linear fixed sized array. This, however has following disadvantages:

1. It is always not possible to know the memory usage of the target binary prior to execution. One cannot declare an array of the size of the target address space which would be typically 2^{32} bytes for a 32 bit processor architecture. Thus some value for the size of the target memory has to be fixed. This could lead to both wastage of space in case the value is too large or memory exhaustion in case the value is too small.
2. This model requires relocation of target address to the memory index. Thus the processor instructions call the memory read and write with the relocated address and not the real address which is not correct. In Sim-nML, the relocation is done using a value called DATA_BASE which is the starting address of the ELF sections. Thus a byte at address EA was stored in memory at M[EA - DATA_BASE], where M is the name of the memory array.

Another efficient memory model is a trie based organization. This is the model that has been implemented currently in the Functional Simulator. The description of this implementation in the Functional Simulator can be found in the work done by Rishabh Uppal [5]. In this model, the simulator manages the virtual address space of the target using a *trie* of memory pages. Each memory page is a 4KB block. This model is more efficient than the linear model as it allocates memory dynamically and the allocation depends on memory requirement of the binary. When the simulation starts, the virtual address space of the target has to be initialized with the data that must be present before the execution of binary. This data is precisely obtained from the decoding stage. One of the tasks of decoding of the binary is to read and store the data of the various ELF sections store in a memory image file in the *Intel Hex format*[4]. When the simulation starts, the data is read from the memory image file into the trie at exact virtual address of the target.

4.1 Generation of Memory Image

The functional simulator decodes all the instructions in the target binary prior to execution. Presently it can decode binaries which are in ELF format only. It reads all the ELF sections present in the target binary and stores them as a sequence of intel hex records. A general intel hex record format is as follows:

The type of intel hex records being used in the present implementation are **Extended Linear Address Record**, **Data Record**, **End of File Record** and the **Start Linear Address Record**. The Start Linear Address Record stores the address of the first instruction to be executed.

4.2 Trie

As discussed earlier, the current memory model implementation in Functional Simulator is a trie based organization . The trie has following three interface operations

Initialize Memory intialize (MemDesc, image file): *MemDesc* is the descriptor of the address space. It reads the data in the intel hex format from the image file and writes into the target space.

Memory Read getmem (MemDesc, address, size, data) : *address* is the target space address from which *size* bytes have to be read in to the buffer *data*.

Memory Write storemem (MemDesc, address, size, data) : It writes *size* bytes from buffer *data* to the target space address *address*.

These trie functions are called whenever memory read or write has to take place. The data stored in the trie is always in the target endianity. However, the processor does not directly call these functions *getmem* or *storemem* for reading and writing into the memory. The processor uses it's own version of read and write functions as follows:

Memory Read readMem (MemDesc, address, size, data, flag):*flag* is a boolean value which tells the manner in which data obtained has to be stored into the target memory. Normally, the first byte of *data* is stored at *address*, second at *address* + 1 , and so on and last byte at *adddress* + *size* - 1. If the *flag* is reversed, the first byte of *data* is stored at *address* + *size* -1, first at *address* + *size* - 2 and so on till the last byte at *address*. *readMem* calls *getMem* with appropriate arguments internally.

Memroy Write writeMem (MemDesc, address, size, data, flag): *flag* means the same as in *readMem*. *writeMem* calls *storemem* with appropriate arguments internally.

readMem and the writeMem are seen by the application. Any other interacting agents like the interactive debugger, system call redirection would call *getmem* and *storemem* for reading and writing from/into memory. Thus profiler must collect it's data within the *readMem* and the *writeMem* functions.

5 Memory Profiling in Sim-nML

With Sim-nML an arbitrary processor architecture can be described. Sim-nML has no constructs to identify the type of the instruction, ie. one cannot know from the bit pattern of the instruction what the instruction does. The following shows part of the action of a load word instruction in Sim-nML for a PowerPC architecture.

```
GPR [ rd ]<31..24> = M [ EA ];
GPR [ rd ]<23..16> = M [ EA + 1 ];
GPR [ rd ]<15..8>  = M [ EA + 2 ];
GPR [ rd ]<7..0>   = M [ EA + 3 ];
```

This instruction has not been marked as a memory read instruction but the semantics of the instruction shows that it is infact doing memory read. This also amounts to the fact that during the simulation of the binary one cannot identify a memory read/write instruction by looking at the binary string of the instruction. But to do memory profiling we must identify the memory read/write instruction. One way to do this was to identify them during the compilation of the Sim-nML description in the Intermediate Representation generation.
Second issue was that of the multiple byte memory read/write instructions. A sample code generated for the integer load word instruction is as follows.

```
{
int64 T_RANG;
(T_RANG=M[EA-DATA_BASE]);
(GPR[ rd ]=(RANGE_ONR_FUNC_I32(GPR[ rd ], 31, 24, T_RANG)));
}
;
{
int64 T_RANG;
(T_RANG=M[((0xffffffff&(uint64)EA)+1)-DATA_BASE]);
(GPR [ rd ]=(RANGE_ONR_FUNC_I32( GPR[ rd ], 23, 16, T_RANG)));
}
;
{
int64 T_RANG;
(T_RANG=M[((0xffffffff&(uint64)EA)+2)-DATA_BASE]);
(GPR[ rd ]=(RANGE_ONR_FUNC_I32(GPR [ rd ], 15, 8, T_RANG)));
}
;
{
int64 T_RANG;
(T_RANG=M[((0xffffffff&(uint64)EA)+3)-DATA_BASE]);
(GPR[ rd ]=(RANGE_ONR_FUNC_I32(GPR[ rd ], 7, 0, T_RANG)));
}
;
```

Several instructions like the load word in the figure above is reading multiple bytes from the memory into the processor registers. With the earlier description as shown above, the processor is generating four one byte memory reads rather than one four byte memory read. Thus a construct of the language was modified to identify such multiple byte memory operation. Taking both the issues related to memory profiling in account, the modified Sim-nML description for the same instruction is

```
 GPR [ rd ]<31..0> = M [EA ] :: M [ EA + 1] :: M[ EA + 2] :: M[ EA + 3];
```

We use the concatenation operator (::) to concatenate all the bytes that have to be read in the same cycle rather than reading one by one. The corresponding code generated is

```
int32 T_CON ;
readMem( getMemDesc( "M"),(EA),4,(uint8*)(&T_CON), 1 );
GPR[ rd ]=RANGE_ONR_FUNC_I32(GPR[ rd ],31, 0,T_CON);
```

6 Implementation of Data Logging

With the *readMem* and the *writeMem* instructions, we have identified the memory read and the write instructions respectively. Thus logging for the information needed for the profile can be done in these instructions. Following assumptions have been made for the profile:

1. Each Memory read and write requires one cycle. Thus cycle counter is increased by one after every read or write operation.
2. Each instruction fetch operation requires one cycle. Once again, cycle counter in increased by one after the instruction fetch operation.

7 The structure of Profiler Log File

The design of the log file has been adopted from done by Luvish Satija[2]. A brief description, along with current implementation details, is as follows:

The Profiler log file consists of a sequence of records. A record can be a Register Record (RR) or a Memory Record (MR).The log file structure is as follows:

Figure 1: Log File Structure[2]

7.1 Register Record

The purpose of a register record is to reduce the storage size by not storing the same information twice, i.e. to store when the information changes.

A Register value is an N bit value. In the current implementation, N is a 32 bit value. There are 7 registers each of which store information of one of the two: Cycle Counter or the Memory Address.

The first 3 bits of the register record are 0. Next 3 bits tell the number of the register for which this record has been written. Len field is the length of the value in bytes. It is assumed that value is 2^{Len} bytes. In the current implementation, memory address is a 32 bit value and

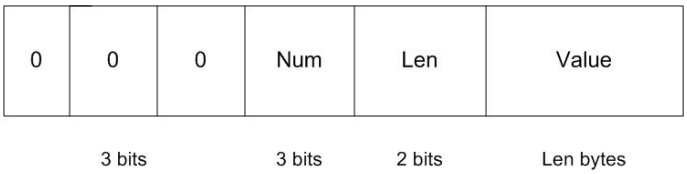


Figure 2: Register Record[2]

we are storing most significant 19 bits of the memory address in the value field of register record. The rest 13 bits are stored in the corresponding memory record. The same is true of the cycle counter too.

7.2 Memory Record

A Memory Record holds the actual log information. Following is the structure of a Memory Record.

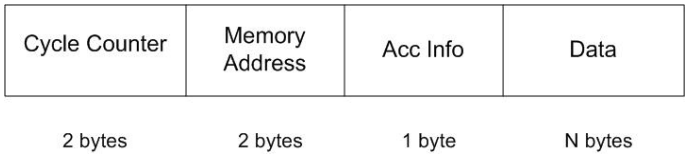


Figure 3: Memory Record[2]

CycleCounter

This field keeps the information about the cycle counter. Cycle Counter is a sequencing information of the memory records. It is stored in 2 bytes in the following pattern.

Figure 4: Cycle Counter or Memory Address[2]

The first 3 bits are for the register number which keeps the base value, ie. the first (N-13) most significant bits for a N bit value. Thus first 3 bits can never be zero as the register numbering starts from 1 to 7. Thus the first 3 bits of a record tell whether it is a register record or a memory record. Register Record has the first 3 bits as 0. The next 13 bits are the offset

value for the cycle counter. Consequently, knowing the register number and this offset value, the actual value can be obtained by concatenation.

Memory Address

Memory Address is the address of the memory which was accessed. For instruction fetch, it is the value of the program counter. The value is stored exactly in the same way as is stored for the cycle counter.

Access Information

The Access Information field is a one byte value containing the access information of the memory access. Access information is broken as follows:

Figure 5: Memory Access Information[2]

The *size* denotes the number of bytes of memory accessed. In the current implementation, **1, 2, 4, 8, 16, 32, 64, and 128 byte** memory read, memory write and instruction fetch are supported. The *type* field keeps information about the access type. With the current Functional Simulator implementation, only **Memory Read, Memory Write and Instruction Fetch** type of access are supported.

Data

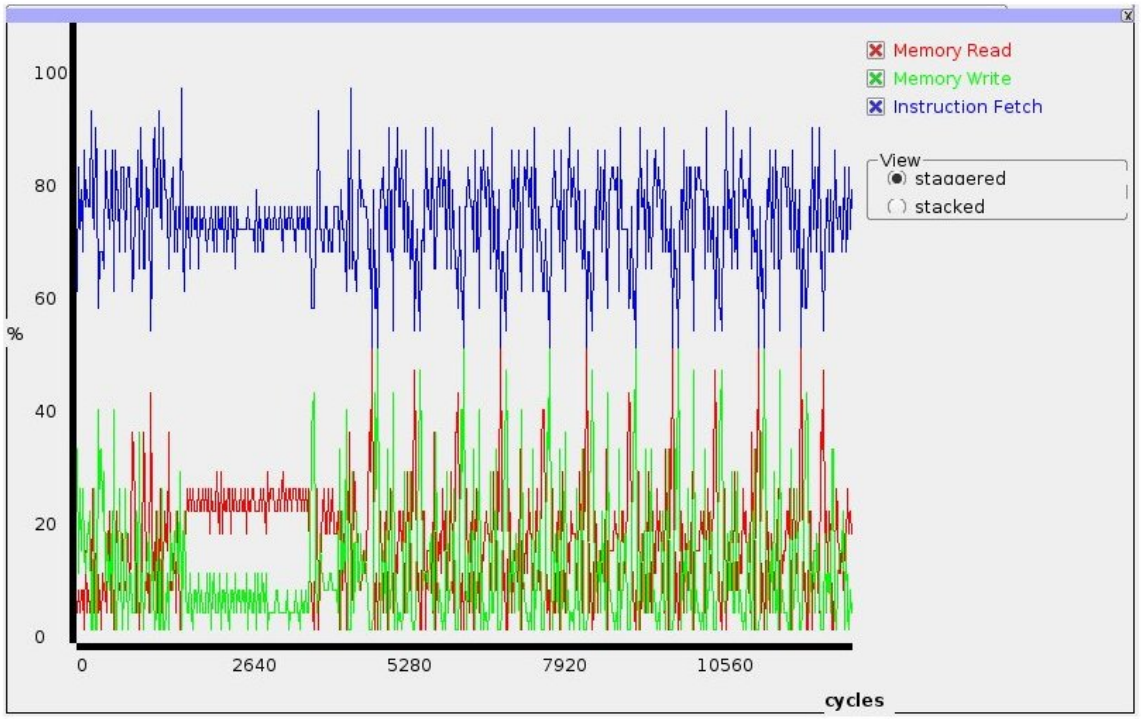
The *data* field is the actual data that was accessed. In case of memory read it is the data that was read. In case of write, it the data that was written. For instruction fetch, it is the binary image of the instruction. For all the three, data is stored in the **target endianity**, ie, endianity of the target processor.

7.3 Register Allocation Algorithm

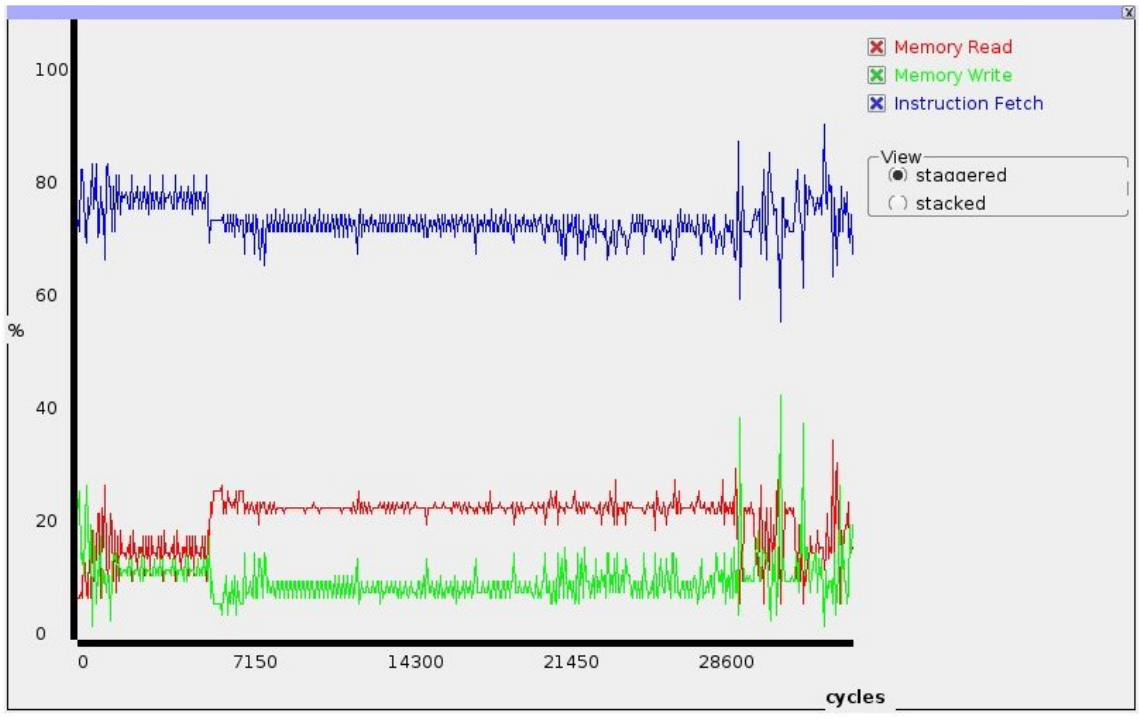
The cycle counter and the memory addresses keep their base values in registers. Every time a new memory record has to be written, we have to first find register numbers that will store the base values of this cycle counter and the memory addresses. Register allocation is done in the same manner for both the cycle counter and memory address. First a register with (N-13) most significant bits same as the value to be stored is sought for. In case we get such a register, this is allcoated. If such a register is not found, we have to replace a register. The register taken for replacement is the **least recently used** register. The register is updated and this updated value is stored as a register record in the log file.

8 Implementation of Profile Viewer

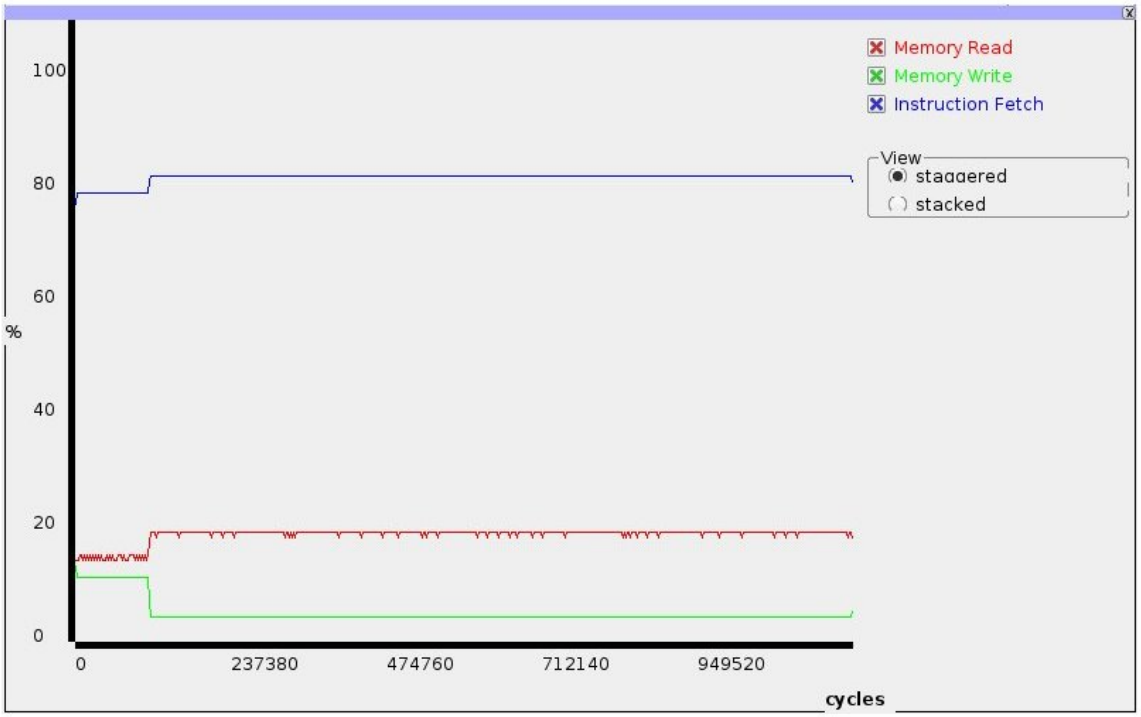
Profile Viewer is developed to project the logged data in a required fashion. The Viewer draws a graph of percentage memory access against the memory bus cycles for all the three types of accesses. This essentially projects the behaviour of the memory access as the program moves ahead. The type of graph obtained depends on the logging policy. Currently, the memory records are being logged with consecutive numbers for the cycle counter and thus the sum of percentage access for read, write and fetch is always 100 for any set of consecutive cycles. Also because of such a logging policy, we do not have areas where bus load is zero. However, viewer algorithm is such that it would show gaps (areas of no memory access) in case the cycle counters in the log file are not consecutive. This is done so as to incorporate future changes to the logging model. The Profile viewer has facility to zoom in between any portion of the profile to get more exact information about the memory accesses. Following are some profiles obtained using the profiler.



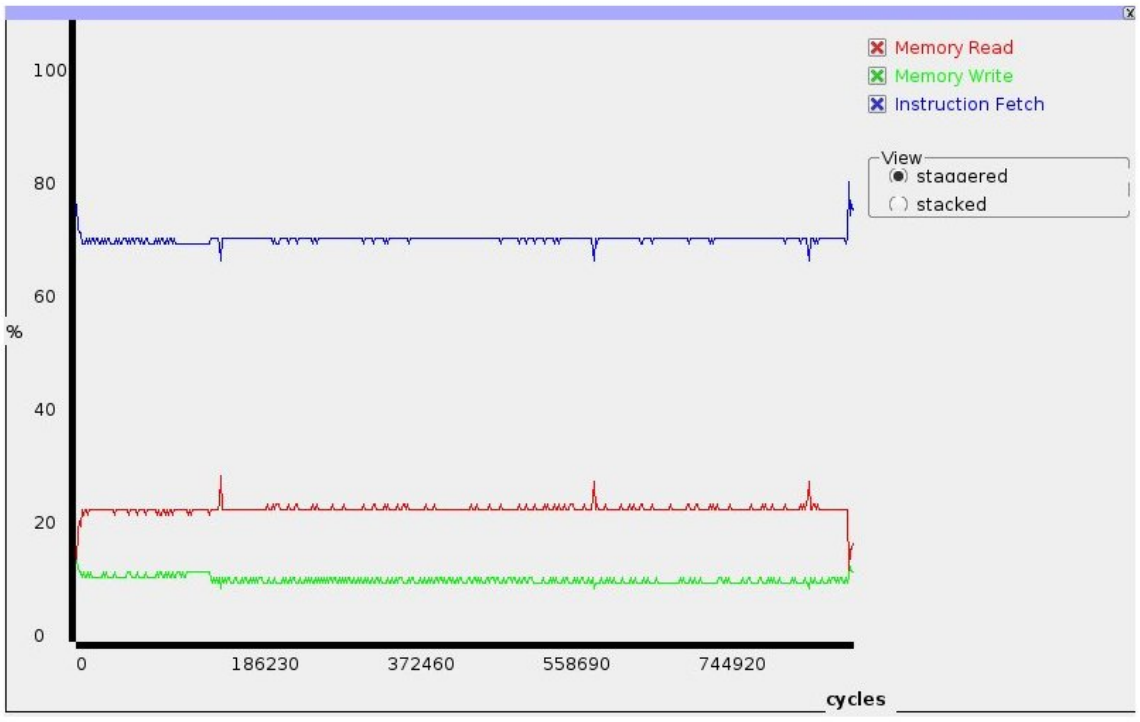
Profile for Bubble Sort Program



Profile for Quick Sort Program



Profile for Integer Matrix Multiplication Program



Profile for the NQueens Program

9 Future Work

In the current implementation for the profiler, the data logging is done by incrementing the cycle counter by one after every read/write/fetch operation. A construct can be added to Sim-nML where user can specify the exact number of cycles taken in a read/write/fetch operation. Profiler design could be enhanced to view the profile as the simulation goes on. This would prove extremely useful for simulations that have very large execution time. Also design of the Profile Viewer could be enhanced to provide more information like line number and other source code information of target binary from the profile.

References

[1] J. L. Hennessey and D. A. Patterson, Computer Architecture: A Quantitative Approach, 3rd Edition, Morgan Kaufmann Publishing Co., Menlo Park, CA. 2001.

[2] Luvish Satija, Profiling and Memory Bus Load Speculation, CS397 course work report, 2006, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur,

[3] Surendra Kumar Vishnoi, Functional Simulation Using Sim-nML, M.Tech Thesis, 2006, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, http://www.cse.iitk.ac.in/gsdl/collect/cse/index/assoc/HASH95ad.dir/doc.pdf

[4] Intel Corporation, Intel Hexadecimal Object File Format Specification, Revision A, 1/6/88, http://pages.interlog.com/speff/usefulinfo/Hexfrmt.pdf

[5] Rishabh Uppal, Implementing Memory Management and Performance Enhancement in Instruction Set Simulator based on Sim-nML Description, CS497 course work report, 2006, Indian Institute of Technology, Kanpur, http://www.cse.iitk.ac.in/gsdl/collect/cse/index/assoc/HASHf65e.dir/doc.pdf

[6] GNU gprof, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

[7] Seamless, Mentor Graphics Corporation, http://www.mentor.com/products/fv/hwsw_coverification/seamle

[8] Nihal Chand Jain, Disassembler using High Level Processor Models, M.Tech Thesis, 1999, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.

[9] Rajiv A.R, Retargetable Profiling Tools and Their Application in Cache Simulation and Code Instrumentation, MTech Thesis, 1999, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.

[10] Y. Subhash Chandra, Retargetable Functional Simulator, MTech Thesis, 1999, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.

[11] TIS Committee, Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2, May 1995, http://www.cs.princeton.edu/courses/archive/spr07/cos217/reading/elf.pdf