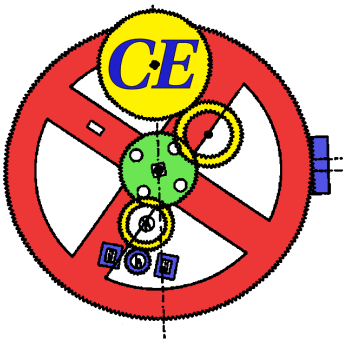

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

A Runtime Profiler for Polymorphic Computing Platforms

Hamid Mushtaq

CE-MS-2010-16

Abstract

Reconfigurable systems map the computational intensive parts of the code in hardware while less computational intensive parts are executed on general purpose processor(s), thus achieving faster execution than systems with only general purpose processor(s). If multitasking operating systems are used on such systems, a runtime system is required to perform resource management of the reconfigurable hardware, as multiple threads might be competing for the reconfigurable hardware at the same time. Such a runtime system needs to know the current configuration and load of the system to properly allocate the reconfigurable hardware. A runtime profiler is an important tool in this regard, as it can assist the runtime system by providing vital statistics about programs running on the system. Since the runtime profiler has to run in parallel with the actual code, it must be very lightweight and therefore very efficient data structures must be used to store the collected statistics. In this thesis, we present the design and implementation of such a runtime profiler. Empirical evaluation has shown that for most applications, our profiler has an overhead of less than 1.5% of the total execution time. Moreover, the information generated by the profiler is almost as accurate as that of popular design time profiler, *gprof*.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

A Runtime Profiler for Polymorphic Computing Platforms

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Hamid Mushtaq
born in Karachi, Pakistan

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Runtime Profiler for Polymorphic Computing Platforms

by Hamid Mushtaq

Abstract

Reconfigurable systems map the computational intensive parts of the code in hardware while less computational intensive parts are executed on general purpose processor(s), thus achieving faster execution than systems with only general purpose processor(s). If multitasking operating systems are used on such systems, a runtime system is required to perform resource management of the reconfigurable hardware, as multiple threads might be competing for the reconfigurable hardware at the same time. Such a runtime system needs to know the current configuration and load of the system to properly allocate the reconfigurable hardware. A runtime profiler is an important tool in this regard, as it can assist the runtime system by providing vital statistics about programs running on the system. Since the runtime profiler has to run in parallel with the actual code, it must be very lightweight and therefore very efficient data structures must be used to store the collected statistics. In this thesis, we present the design and implementation of such a runtime profiler. Empirical evaluation has shown that for most applications, our profiler has an overhead of less than 1.5% of the total execution time. Moreover, the information generated by the profiler is almost as accurate as that of popular design time profiler, *gprof*.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-16

Committee Members :

Advisor:	dr. ir. Koen Bertels, CE, TU Delft
Member:	dr. ir. Alan Hanjalic, DMIR, TU Delft
Member:	dr. ir. Zaid Al-Ars, CE, TU Delft
Member:	dr. ir. Georgi Krasimirov Kuzmanov, CE, TU Delft

Dedicated to my family

Contents

List of Figures

List of Tables

Acknowledgements

I am really grateful to Mr. Mojtaba Sabeghi for his support, encouragement, guidance and feedback throughout the Thesis. I'm also really thankful to him for proposing to me such an interesting topic for research. The topic was according to my interest and therefore I never felt overburdened in doing the research, instead I enjoyed every moment of it. I would also like to thank Mr. Koen Bertels for his valuable guidance and support throughout this project.

Moreover, I would like to thank all of my family members and my friends for all of their love, support and prayers throughout these years.

Hamid Mushtaq
Delft, The Netherlands
June 29, 2010

1 Introduction

In this thesis, we have described the implementation and performance of a runtime profiler that can be used within runtime systems of polymorphic computing platforms. This runtime profiler was implemented and tested on a machine with an x86 processor.

1.1 Motivation

Limited commercial success of Reconfigurable systems has been due to difficulty in programming them. Usually separate tools are used for writing software and designing hardware. Compilers which can partition software and hardware are rare and cumbersome to use. Furthermore, separate synthesis tools are required to program the reconfigurable logic. Therefore, the overall design and implementation cycle is difficult and demands a lot of care on part of the designers and implementors of the system. These systems become even more difficult to program if they have Time Sharing Multitasking Operating Systems running on them. In that case, multiple threads may try to use the reconfigurable hardware at the same time. Therefore, to perform proper resource management of the reconfigurable hardware in such systems, a runtime system is needed. The resource management by itself is a very complicated task and is dependent on the available information for decision making [30]. One important tool that can assist the runtime system in resource management of reconfigurable hardware is a runtime profiler. Such a profiler can give important statistics about code running on the reconfigurable computer. Those statistics can then be used by the runtime system to decide which parts of the code need to be translated into hardware at a particular instance.

1.2 Project Goals

Our task is to design and implement a runtime profiler that runs on the general purpose processor of a reconfigurable system. The main task of the profiler is to keep statistics of hotness of kernels running on the reconfigurable computer. Since the profiler is used to assist the runtime system in speeding up execution of applications, it should incur as low overhead as possible, or else its use will be unjustified. This requires it to use efficient and fast data structures. Secondly, the profiler needs to have well defined interfaces, so that it can be integrated with runtime system of any polymorphic computing platform.

1.3 Organization

In chapter 2, overview of Molen [34] polymorphic platform and its runtime system is discussed, followed by discussion about the scope of the thesis. Chapter 3, discusses

related work, while chapter 4, discusses the design and implementation of the profiler. It is followed by chapter 5, which shows the results of the empirical evaluation. The thesis is wrapped up by chapter 6, which gives conclusions and discusses future work.

2 Background

Multi-core heterogeneous systems with reconfigurable hardware are becoming common. One major hurdle in developing software for such systems is their complexity. First of all, such systems consist of different kinds of computational units. Therefore, if such systems are programmed manually, the developer needs to know how to write software for each of them. Secondly, for best performance, the applications have to be mapped in those different computational units in such a way that an optimized solution is achieved. Therefore, these characteristics of the heterogeneous multi-core systems require automatic or semi-automatic tools to ease designing software for them. However, if Timesharing Operating Systems are used in such systems, even those tools are not enough. That requirement further necessitate existence of a runtime system, which can manage the reconfigurable resources on such systems.

In this chapter, the hArtes [4] approach of Hardware/Software co-design for Heterogeneous Multi-Core platforms, is discussed. It is discussed in section 2.1 and is followed by section 2.2, which discusses the MOLEN [34] polymorphic processor. In section 2.3, the need to have a runtime system to manage reconfigurable resources and the runtime system used by MOLEN are discussed. This is followed by section 2.4, where the scope of the thesis is presented. The chapter is wrapped up by section 2.5 which gives conclusions.

2.1 Design Time Approach for Resource Management

The hArtes [4] project addresses the difficulty in programming heterogeneous multi-core platforms. It provides a semi-automatic support for hardware/software co-design of such systems. The hardware platform specified for hArtes consists of an ARM processor, a DSP and an FPGA. The design starts with an application written in C and then with the support of the hArtes toolchain, the application is modified so that it uses all the three components, that is ARM processor, DSP and FPGA, in the most efficient manner possible. In this way, the hArtes toolchain allows developers to write applications for heterogenous systems without requiring them to have intimate knowledge of the underlying hardware and its programming. Parallelism and mapping are specified in the code by using *pragmas* in the C code. These annotations can either be added manually by developers or generated by tools. To specify parallelism, standard OpenMP pragma directives are used. *pragmas* are also used to give profiling information to help the mapping tool.

The hArtes toolchains comprises of three toolboxes. Each one of them takes a C-code as input and outputs processed files, which are then fed to the next toolbox. Figure 2.1 shows the interactions among those toolboxes. The top level toolbox namely the Algorithm Exploration Toolbox is optional. Its purpose is just to generate C code from a Scilab code or from NU-Tech, which is a graphical software development tool. Next is

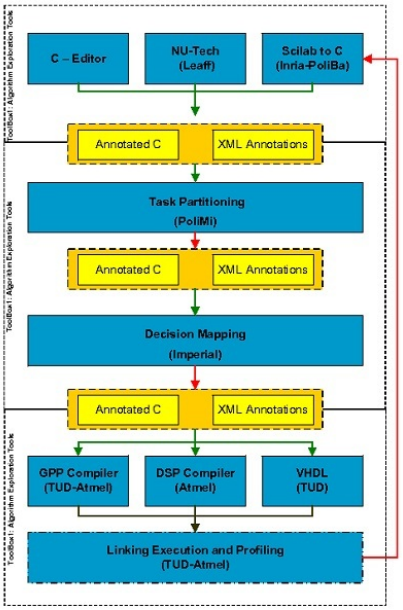

Figure 2.1: The hArtes toolchain

the Design Exploration toolbox whose task is to perform task partitioning and mapping. The task partitioning tool, known as *Zebu* generates an efficient task graph [16] with the help of static profiling to identify the most efficient paths [17]. Afterwards, it performs transformation on the task graph to take into account the overhead of managing the parallel tasks. Finally, it generates a C Code annotated with *pragmas* for parallelization and mapping. The mapping tool hArmonic [25] is used to get an optimized mapping solution. It does so by selecting which part of the code needs to run on the Main processor, DSP or FPGA and generates C source code for each of them. The input XML file here is used to provide platform specific information. The last tool in the chain is the Synthesis toolbox. It contains a compiler for each processing element. Source for each processing element is compiled separately and then linked together to form a single binary. It uses a modified gcc compiler for the general purpose processor. That compiler inserts MOLEN instructions to execute a kernel on FPGA. Those instructions are discussed in section 2.2 where we discuss the MOLEN polymorphic processor. Moreover, Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) [35] is used to perform C to VHDL translation for the FPGA.

2.2 MOLEN Polymorphic Processor

The MOLEN Polymorphic Processor [34] consists of a general purpose processor (GPP) connected to a reconfigurable processor (RP). To be compliant with hArtes specifications, a DSP can also be added as a coprocessor. The hardware organization of MOLEN is shown in figure 2.2. Communication between GPP and RP is done through shared memory and Exchange Registers (XREGs). Shared memory allows GPP and RP to

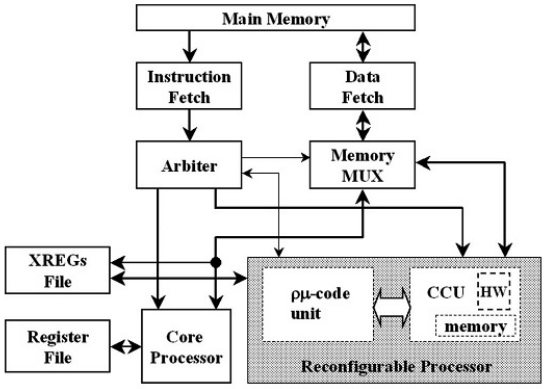

Figure 2.2: MOLEN Hardware Organization

work on the same data. The Exchange Registers are used for communicating small amounts of data between GPP and RP, like parameters to functions, returned values or pointers to memory locations.

MOLEN has specific instructions to execute tasks on the RP. Before instructions reach the GPP, they are partially decoded by the Arbiter. This is done to allow MOLEN to interpret the instructions which are used to execute programs on the reconfigurable processor. The SET instruction is used to configure the *custom configured unit* (CCU) inside the RP, to perform the required operations, while the EXECUTE instruction is used to perform the actual execution of those operations. To reduce configuration overhead, SET instruction can be issued well ahead of the EXECUTE instruction. MOVTX instruction is used to move the contents of a register in register file to an exchange register. Likewise, MOVFX instruction is used to move contents of an exchange register to a register in register file. Therefore MOVTX and MOVFX are used to pass and receive data from the RP. Code on the RP can execute in parallel with the one running on the GPP. To explicitly synchronize execution between GPP and RP, BREAK instruction is used.

2.3 Towards Virtualization

The design time approach that we previously discussed performs mapping of code into different processing units at design time. Secondly, the SET and EXECUTE instructions in MOLEN are only useful if we have a single thread of execution. If multiple threads or applications want to access the reconfigurable logic at the same time, there must be an underlying runtime system that can manage and decide which parts of the applications need to run on the hardware at a particular instance. To solve this issue, a layer above the Operating System which can manage the hardware resources at runtime has been proposed [28]. This layer would hide platform dependant details from the programmer.

The block diagram of MOLEN's proposed runtime system is shown in figure 2.3. The *Scheduler* reads statistics from the *Profiler* about hotness of profiled kernels and

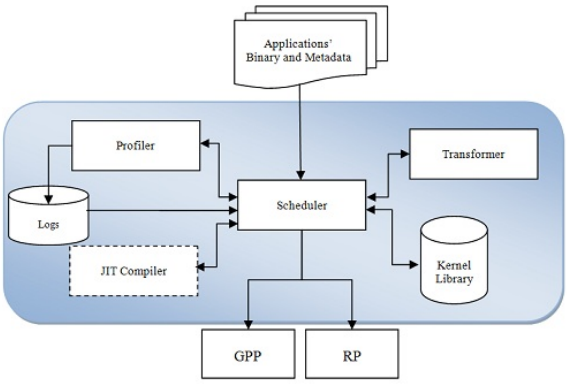

Figure 2.3: MOLEN's Proposed Runtime Environment

decides how to allocate the reconfigurable hardware. The *Scheduler* is responsible for estimating costs of hardware mapping and potential speedups gain by mapping a kernel into hardware. Therefore it tries to map those kernels into hardware, which would bring the most speed gains. By kernel, here we mean compute-intensive pieces of code. Throughout this thesis, we will refer to compute-intensive pieces of code as kernels. The *scheduler* use the services of the *Transformer* to replace the software implementation of a kernel with a call to the hardware one. The *Kernel Library* contains precompiled and already synthesized kernels. There can be different versions of the same kernel. For each version, the library contains metadata to describe characteristics of that kernel, such as power consumption, configuration latency, execution time etc. That information is used by the *scheduler* to map the most appropriate version of the kernel. The last part of the runtime system is the Just In Time Compiler that compiles and synthesizes those kernels for which there is no hardware implementation available in the kernel library.

Work has been done for using compiler generated data to map kernels to hardware at runtime, as discussed in [30]. In this scheme, the compiler, after analyzing the code and using information generated by a design time profiler, generates a configuration call graph, which is read by the scheduling algorithm. The configuration call graphs are used to find the distances to the next call and future number of calls of compute-intensive functions, known as kernels. Configuration call graphs of applications that would be running on the system, are combined into one. In this way, the *Scheduler* can choose those kernels to map into hardware which are expected to bring the most speed gains. For this purpose, the semantics of MOLEN's SET and EXECUTE instructions have been extended so that they support multithreading and mutlitasking scenarios. This is because each thread can issue its own SET and EXECUTE instructions, and if they are occurring in parallel, the runtime system must resolve the conflicts that would arise in such circumstances. For this purpose, two APIs, MOLEN SET and MOLEN EXECUTE have been added to the operating system [29]. When SET is called, the reconfigurable logic is not configured instantly as it was being done in previous implementation, but instead a request is sent to the runtime system. Then according to scheduling policy,

the intended kernel might be configured on the FPGA or rejected. Similarly, when the EXECUTE is called, it is first intercepted by the runtime system and if corresponding kernel is already configured on the FPGA, its execution will be started right away, otherwise SET has to be called again for that kernel.

2.4 Scope of the Thesis

While runtime scheduling using design time profiling information and call graphs, as discussed in previous section, might be useful for systems where such a compiler is available and where it is known beforehand which applications would be running on the system, it is not very useful for general purpose computing. In case of general purpose computing, any application could be running anytime and therefore runtime scheduling using design time information could not be applied there. Another limitation of that approach is that it assumes that access to all the source code is available. Moreover, the design time information that the scheduler would be using might not be very accurate, since input parameters at design time might be different from those at runtime. Such limitations of design time information, necessitates the need for a runtime profiler, whose design and implementation is the goal of our thesis. Although the runtime profiler would not be able to see the future frequencies of the kernels, it would be able to record the past frequencies, and empirical results in [30] have shown that accuracy of results do not vary much if past frequencies are used instead of future frequencies. Moreover, the runtime profiler would be able to give exact past frequencies unlike a design time profiler.

Here we discuss the problem specification of our thesis where we discuss the requirements for a runtime profiler. After the problem specification, we discuss the limitations of the profiler that we implemented.

2.4.1 Problem Specification

Our primary goal is to design a runtime profiler for MOLEN's runtime system. However, the techniques used by our profiler should be applicable to other similar runtime systems. The runtime profiler must be able to count the number of calls of functions in running applications. In MOLEN, both hardware and software implementation of compute-intensive functions, known as kernels is provided, as it does not support Just in time FPGA compilation yet, but to test our profiler, we try to profile every function in the running applications, assuming that a just in time FPGA compiler will be provided in the future. Secondly, the profiler should be able to give approximate time spent by each function. This is important because some kernels might be called a lot of time but might only be consuming small portion of the total execution time, while other kernels might be called lesser times but consume larger portion of the total execution time.

The profiler must also be able to perform memory profiling. This is important for two reasons. From figure 2.2, we can see that the reconfigurable fabric has an internal memory. If a kernel, when transformed into hardware, accesses all the data from the internal memory, its execution will be much faster than if it also uses data from the external memory. Therefore, the profiler should be able to give a guess about how much internal and external memory a kernel will consume when transformed into hardware.

Secondly, the profiler must also be able to tell about the data dependencies that would be there when the kernel is transformed into hardware, so as to allow the system to perform proper synchronization between the kernel running on the RP and code running on the GPP.

2.4.2 Limitations

While current implementation of our profiler performs both functions calls counting and calculation of approximate time spent by profiled functions, it does not perform memory profiling. Secondly, since it has been only tested with x86 architectures, platform specific translations need to be done if has to be ported to different processors like ARM or PowerPC. We use the local APIC timer interrupts found on x86 machines to perform sampling profiling (sampling profiling will be discussed in later chapters). To port that part to other architectures, we will need to use timers specific to those architectures. Secondly, since we also deal with the assembly and machine codes of to be profiled programs, we will need to deal with architecture dependant assembly and machine codes when porting our profiler to different architectures. Moreover, we have tested our implementation only on Linux operating system. Therefore, work needs to be done if has to be ported to other operating systems. Furthermore, we have only profiled programs whose executable files are in ELF format. Lastly, we only profile static functions of a program.

2.5 Conclusion

In this chapter, we first discussed the design time approach of hardware/software co-design, which used by the hArtes [4] project. We saw that a lot of complex tools are required to ease the task of developers to develop such systems. Things become even more complicated if such systems use time sharing and multitasking operating systems, as in the presence of multiple threads competing for limited reconfigurable hardware resources, a runtime system is required to properly manage and allocate those resources. In this regard, we discussed MOLEN's runtime system. Lastly, we discussed the scope of our thesis, which is to implement a runtime profiler to be used in MOLEN's runtime system. We also discussed the limitations of the profiler that we have implemented.

3 Related Work

In this chapter, previous research and work done on profilers is discussed besides discussion on runtime systems used by reconfigurable platforms.

Section 3.1, discusses some of the runtime systems used in reconfigurable computing platforms, while section 3.2, discusses different types of profilers. In that section, we also discuss the merits and demerits of those different kind of profilers and figure out which of them best suits our needs. Section 3.3, discusses research and implementation of profilers for Reconfigurable systems. Finally, conclusion is given in section 3.4.

3.1 Runtime Systems for Reconfigurable Computing Platforms

Besides MOLEN, there are several other reconfigurable computing platforms that use runtime systems to manage reconfigurable hardware resources. The runtime systems of such reconfigurable computing platforms is discussed next.

3.1.1 Warp Processor

Like MOLEN, Warp processor [32] also transforms compute-intensive kernels running on a general purpose processor to hardware. The block diagram of the system is shown in figure 3.1. The profiler do not run on the general purpose processor(s) but is implemented as hardware. It is used to detect kernels at runtime and its working is discussed in section 3.3.1. The Dynamic CAD tool is used to map kernels into hardware, while the binary updater is used to change the binary of the program running on the general purpose processor(s), so that it uses the FPGA to execute a kernel that was transformed into hardware by the Dynamic CAD tool. The difference between Warp processors and

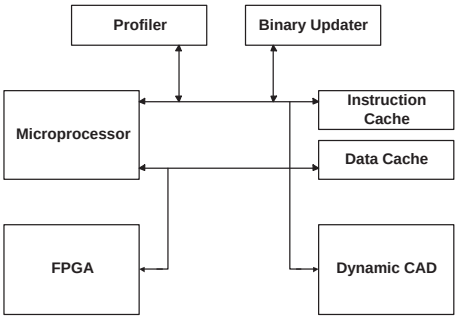

Figure 3.1: Block Diagram for Warp Processing

MOLEN is that while MOLEN can only map those kernels to FPGA for which the hardware implementation is available, Warp Processors can generate code for the FPGA on the fly by using the Dynamic CAD tool. However, this tool can only transform small loops into hardware and the hardware can only contain combinational logic circuitry.

3.1.2 ReconOS

ReconOS [23] extends embedded and standard operating systems like eCos and Linux, to support multithreading environment for reconfigurable hardware. The support is extended to include hardware threads. Hardware threads can use almost all of the operating system services that software threads can use. While the software threads execute on the CPU, hardware threads execute on the reconfigurable hardware. Almost for each POSIX function, there is an equivalent function for hardware threads. For example, the equivalent of *pthread_mutex_lock* is *reconos_mutex_lock*. Through the ReconOS API (hardware), synchronization mechanisms like mutexes and semaphores, thread signalling, message queues and shared memory are provided for hardware threads. Therefore, a software thread using POSIX API can seamlessly communicate with a hardware thread using ReconOS API.

Since hardware tasks are inherently parallel in nature, while software tasks are sequential, each hardware thread contains a sequential state machine whose job is to interact and synchronize with the operating system. Moreover, an operating system interface (OSIF) is built into the system, which serves as connection between the hardware thread and OS. Through these two mechanisms, blocking calls can be implemented in hardware. Therefore, each thread consists of at least two VHDL processes, the user logic and the synchronization state machine. Moreover, every hardware thread is associated with a software thread known as its delegate, which executes operating system calls on behalf of the hardware thread.

In current implementation of ReconOS, hardware threads are statically configured rather than dynamically loaded, but future work is planned to add this dynamic loading capability of hardware threads for more efficient utilization of the reconfigurable hardware. To achieve that, the systems's scheduler would need to be extended, so that it can properly manage resource utilization for the loadable hardware threads.

3.1.3 HybridOS

HybridOS [21] is a set of operating system extensions for multi-user reconfigurable platforms. It allows user space applications to access reconfigurable accelerators, which are implemented in a reconfigurable fabric. To achieve this, multiple data parallel kernels are mapped into accelerators, so that they can be accessed by multiple threads in an interleaved and space-multiplexed fashion. User space applications, running on a general purpose processor, use an application programming interface (API) that abstracts the hardware implementation from the programmer. To transfer data between user space applications and accelerators, different modes of communication, known as access methods, are implemented by the system. These access methods allow virtualization of the accelerator resources, thus allowing applications to treat the accelerators as black-boxes. To perform virtualization of accelerator memory and perform copying when necessary,

in-memory buffers are used. This scheme is similar to the virtualization of physical memory in modern operating systems.

3.1.4 An Execution Environment for Reconfigurable Computing

The authors in [18] discuss an execution environment for reconfigurable computing which uses scheduling algorithms to allocate reconfigurable hardware resources to threads in a multithreaded environment. Like Molen, the reconfigurable hardware is used as a co-processor of a general purpose processor. The reconfigurable hardware is used to map the compute-intensive parts of applications, known as kernels, into the reconfigurable hardware. These kernels are usually loops which can partially or fully execute in parallel. Like Molen, the system contains both software and hardware versions of kernels. Moreover, a kernel may have several hardware versions, each one of them optimized for different requirements. For example, one kernel could be optimized for speed while the other may be optimized for area. The system deals with the situation in which multiple threads are competing for the reconfigurable hardware, by using a hardware based scoreboard. The scoreboard is used to count the time a kernel is executed. The count is incremented regardless of whether a kernel is executing in hardware or software. Moreover, the scoreboard is periodically flushed to throw away old counts. By looking at the scoreboard, the scheduler decides which kernels to map into the hardware.

3.2 Profiling Techniques

Profilers can be used to profile CPU usage as well as memory usage, but most profilers deal with profiling CPU usage. A Call-graph profiler keeps statistics for frequencies of function calls, function call times and even call chains. A flat profiler on the other hand only computes the average call times.

Based on methods of gathering information, profilers can be categorized into two major groups, Instrumentation-based and Sampling-based. These two types of profilers are discussed below.

3.2.1 Instrumentation-based Profilers

Instrumentation-based Profilers work usually by inserting instrumentation code into the application to be profiled. The instrumentation code can be injected either at compile time, link time or run time. The advantage of such profilers is that they are very accurate, but on the downside they have relatively large overheads.

Our profiler would require Instrumentation-based Profiling to record the number of times each kernel is called in the system. The instrumentation technique that we use should have the minimum possible overhead. Below we will discuss some existing Instrumentation profilers and the techniques used by each of them. We will also discuss the relative merits and demerits of those techniques for our profiler.

Instrumentation File

```
1  #include <stdio.h>
2  #include <instrument.h>
3  Instrument() {
4     Proc *p;
5     AddCallProto("RecordRead(REGV)");
6     AddCallProto("PrintResults()");
7     p = GetNamedProc("read");
8     if (p != NULL) {
9        AddCallProc(p,ProcBefore,"RecordRead", REG_ARG_3);
10       AddCallProgram(ProgramAfter,"PrintResults");
11    }
12 }
```

Analysis File

```
1  #include <stdio.h>
2  long bytes = 0;
3  void RecordRead(long size) {
4     bytes = bytes + size;
5  }
6  void PrintResults() {
7     FILE *file = fopen("read.out","w");
8     fprintf(file, "%ld\n", bytes);
9     fclose(file);
10 }
```

Figure 3.2: ATOM Read Tool Implementation [15]

3.2.1.1 ATOM

An example of an Instrumentation-based Profiler that modifies code at link time is ATOM[15]. It uses OM[31], a link time modification system. To perform profiling using ATOM, one needs to write an instrumenting code file and an analysis code file. The instrumenting code file is used to specify places in the original code where instrumentation needs to performed and the profiling routines that will be used to perform profiling, whereas the analysis code file contains the implementation of those profiling routines. The OM generic object modification library is linked with the afore mentioned instrumentation code file to produce a tool which reads the user application and modifies it by adding calls to the routines specified in the analysis file. Figure 3.2 shows the instrumentation and analysis files for instrumenting an application to count the total number of bytes read during execution of the application and write the results in a file at the end. In the instrumentation file, we have specified that whenever **read** function is called, prior to that, *RecordRead* function should be called to update the bytes read, from the argument that specifies size in the **read** function. Furthermore, we have specified that function *PrintResults* should be called at the end of user application to print the total number of bytes read. Similarly, many custom instrumentation tools can be made by using ATOM. Since ATOM is not dependant upon the underlying target platform's architecture, it is quite portable. On the other hand, by virtue of being an Instrumentation-based profiler, there are considerable overheads of using ATOM. It can slow down an application to as much as greater than 8 times[15].

Since ATOM performs injection of instrumentation code at link time using the object files, it is not useful for our profiler because our profiler is required to work with executable binaries. Pin[24] is a similar tool but it performs code injection at runtime rather than link time. That tool might be of interest to us and therefore is discussed next.

3.2.1.2 Pin

Pin[24] follows ATOM's model, but unlike ATOM, to perform instrumentation, it modifies applications at runtime. It uses a Just-in-time compiler to insert instrumentation code and optimize code. Pin attaches to a running process and inject code into it by using the *Ptrace* API. In this way it can insert code into an application to perform jitting. Through the Pin API, it is possible to observe all architectural state of a process, such as memory, control flow or contents of registers. Just like ATOM[15], one writes analysis routines and instrumentation routines to determine where analysis functions need to be called in the user application. Pin can attach to a process for instrumentation and then detach from it, thus saving instrumentation costs when not needed. The instrumentation is performed by a JIT compiler whose input is the native code. Pin intercepts first instruction and compiles straight line sequence from that instruction onwards. The control is then transferred to the generated sequence, but Pin makes sure that it regains control when a branch exits the sequence. After regaining control, Pin generates more code from the branch target onwards. At each fetch of code by JIT, Pin has the ability to instrument it before translation. To speed up execution, the translated and instrumentation code is kept in a code cache.

Since Pin injects code at runtime, it might be useful for our profiler. One major advantage of using Pin is that any profiling tool can be created from it. This is possible because it performs interpretation of native binary code and just in time compilation. Therefore it is able to insert code anywhere in the program. However, the interpretation of native code also means that it has large overheads. To check its performance, we build a pin tool which just counts the number of times each function is called in an application and found out that for each benchmark application that we tested it with, the overhead was at least 20%. Another disadvantage of using Pin is that it is difficult to port. To port it to an architecture for which it is not available, such as PowerPC, the whole native binary code interpreter and Just in Time compiler would need to be written from the scratch for that particular architecture. However, Pin gives us a useful hint about how to attach to a running process and inject code into it. Like Pin, we could use the *Ptrace* API to do that.

3.2.1.3 Dynamo

Dynamo[10] is a dynamic optimization system that interprets native binary code and optimizes it by using a runtime profiler. Dynamo interprets instructions until it encounters a backward taken branch. At that point, it increments a counter associated with the target address of that branch. If the counter value exceeds a preset hot threshold, Dynamo switches to code generation mode and starts recording the instructions in a hot trace buffer until an *end of trace* condition is reached. At that point, Dynamo performs optimization on the code recorded in the hot trace buffer and put that optimized code in a code cache. The *end of trace* condition is satisfied when a backward taken branch or a taken branch whose address is already present in code cache, is encountered. At *end of trace*, if branch target is not found in code cache, control is taken back by Dynamo, until it once again finds a hot trace or it hits a branch whose target is already present in the code cache. In case of a branch target being already present in the code cache, Dynamo

gives up control and code is directly executed on the processor through the code cache. When control is exited from code cache, Dynamo resumes interpreting instructions and the whole process is repeated again. Dynamo performs pretty well. On the SpecInt95 benchmarks, the average overhead is less than 1.5 % of execution time[10].

Regarding the usability of the techniques used by Dynamo for our profiler, Dynamo suffers the same drawbacks as Pin because like Pin, it also performs interpretation of native binary code and Just in Time compilation. The reason it has a low overhead is because it only performs profiling for a piece of code until it is optimized and translated to execute from code cache. On the other hand, our profiler needs to continuously profile all the kernels in the system.

3.2.1.4 Detours

Detours [20] is a library which is used for instrumenting *Win32* functions on x86 machines. Let us consider a source function that calls a target function. At runtime, Detours replaces prologue of the target function with a jump to a user-provided *detour* function. The replaced prologue is saved in a trampoline function. That trampoline function also contains an unconditional jump to the rest of the target function. Therefore, when the source function calls the target function, the call is redirected to the user-supplied *detour* function. The *detour* function performs interception preprocessing and then either returns to the source function or calls the target function through the trampoline. The target function, in the end, returns control to the *detour* function. At that point, the detour function does interception postprocessing and returns control to the source function. Igprof [14] uses a very similar technique to instrument programs.

The technique used by Detours to replace the prologue of target functions with a jump to a *detour* function that contains the instrumentation code, can be very useful to create a low overhead profiler. The only requirement to achieve that is to have the *detour* function perform minimal work. One disadvantage of the technique used by Detours is that the *detour* functions can only be called from specific areas in the code, like function prologues and function returns and not from anywhere in the code, like in case of Pin. Secondly, a prologue of a function must be of a certain minimum size to be able to be replaced by a jump instruction.

3.2.1.5 Dyninst

Dyninst [11] is a library for program instrumentation that allows insertion of code into running programs. Dyninst provides an API (Application Program Interface) to achieve that. By using this API, one can write a program that attaches to a running application and inserts code into it. After insertion of the new code, the modified application also executes the injected code. In this way, code can be added to a running program for profiling, debugging or any other purpose. Dyninst internally uses *Ptrace* to attach to and inject code into running applications.

We have seen that both *Dyninst* and *Pin* use *Ptrace* to inject instrumentation code into applications to be profiled. Therefore it is a standard way to do such tasks in Linux and due to that we would also use *Ptrace* for our profiler.

3.2.1.6 A Framework for Reducing the Cost of Instrumented Code

The authors in [9] combine instrumentation and sampling techniques to build a low overhead profiler for runtime optimizations. The sampling is not done through any hardware mechanism but is done through software instead. It can collect quite accurate profiles(93-98%) at low overhead (average of 3%). This framework is implemented and tested with the Jikes[1] Java Virtual Machine. An instrumented method contains the original code and its duplicated copy modified by instrumentation code. The original code is called checking code and is only slightly modified to swap between itself and duplicated code. The checking code has conditional branches inserted, known as checks, to check a sample condition. If the sample condition is true, control jumps to the duplicated code. Checks are placed at method entries and backward branches, known as backedges. All backedges (backward branches) in duplicated code are modified to transfer control back to the checking code. The sampling condition can be a simple condition for checking if value of a counter has become zero after decrementing. When control is transferred back to checking code, the value of the counter is reset. By changing the reset value of the counter, the sampling frequency can be modified. The benefit of lowering the sampling rate is to have the application spent most of its time in original code, rather than the duplicated code with instrumentation. However, the downside of lower sampling rate is loss of accuracy. Therefore, a compromise has to be done. We may put any type of instrumentation code in the duplicated code, for example, event counting of a function or a loop.

With the method of instrumentation discussed above, we can see that one can still get quite accurate instrumentation by not incrementing calls count every time the piece of code being instrumented is executed. Instead, the count can be incremented at certain intervals. For example, we can increment the calls count for a function on each 10th hit. In that way, we can minimize the instrumentation overhead by 10 at the cost of loss of a little accuracy.

3.2.1.7 GProf

GProf[19] is a profiler for Linux. It uses both instrumentation and sampling techniques. To profile a program with GProf, one has to compile and link it with profiling enabled, so basically instrumentation code is injected at compile and link time. At entrance, the profiled functions call a runtime routine for profiling. The profiling routine counts the number of time a function is called. It also records the caller of a function by storing the return address of the call. In this way, it can build a Call Graph. Besides that, GProf also reads the Program Counter value at each timer interrupt. In this way, it maintains a Program Counter Histogram. A postprocessing program combines the Program Counter Histogram data and Call Graph arcs to show the time spent in each function as well as time spent by functions called from other functions.

Since GProf injects code at compile and link time, the method used for instrumentation by it cannot be applied to our profiler because our profiler should be able to profile executable binaries.

3.2.2 Sampling-based Profilers

Sampling-based profilers periodically take samples of Program Counter's value. Therefore, the gathered data is a statistical approximation. They are not as accurate as Instrumentation-based profilers, but have much lower overheads because no extra instruction is inserted in original code. As a result, they are relatively non-intrusive and allow the profiled program to run almost at its normal speed. Normally, Sampling-based profilers also exploit the hardware performance counters of modern processors, to look for instructions causing more instruction cache misses, data cache misses, pipeline stalls or any other event supported by those performance counters.

3.2.2.1 GProf

GProf[19] also uses sampling besides instrumentation for profiling. It regularly takes samples of the Program Counter. By default, GProf uses kernel's timer interrupt to collect value of the Program Counter. Therefore, very little overhead is added to the system, because the kernel has to service the timer interrupt anyway. Since Linux kernel's timer interrupt typically occurs after each 10 ms, this sampling technique will not give proper figures for a code which consumes a lot of execution time but is seldom hit at the occurrence of the timer interrupt.

Our profiler can use Sampling-based profiling just like GProf to calculate the average approximate spent by each kernel running on the GPP of the reconfigurable system. This method is quite portable since use of timers with general purpose processors is commonplace.

3.2.2.2 OProfile

OProfile[3] is a sampling-based profiler available for Linux Operating systems. Just like GProf, it uses timer interrupts to take samples to find time spent by different parts of the code. Furthermore, It also allows optional use of Hardware Performance Counters to aid profiling. One can can set counts of Hardware Performance Counters at which interrupts are generated. If there are more than one type of Hardware Performance Counters, then events corresponding to those different types of Performance Counters can be used to profile different types of events, like cache misses or dynamic stalls, but if a processor does not have any Hardware Performance Counter, only the kernel timer interrupt is used. At the occurrence of an interrupt, Program Counter's value and Process ID of the process which was running at that time is read. The gathered profiled data is used offline for analysis.

For each CPU in the system, OProfile maintains a small buffer to log samples. A *head* iterator is used to dictate placement of next sample in the buffer. Since Program Counter's value is meaningless without knowing the task which was interrupted at the time of taking a sample, the task switches are also logged. This is done by storing the address of the last task in a data structure and writing the task switch entry into the buffer, if the current task is different than the previous. Similarly, information about whether the interrupted task was in user space or kernel space is maintained. Besides

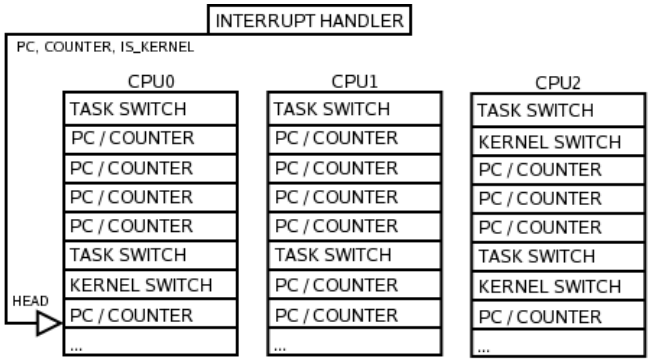

Figure 3.3: per-CPU Buffers used by OProfile [3]

the small buffer in each CPU, there is also a large buffer, to which logged data is copied, to be read by the OProfile daemon. The per-CPU buffers are shown in Figure 3.3.

We have seen that OProfile uses small buffers for each CPU to hold the samples and also a large buffer to copy those samples to when the small buffers are overflowed. It is done like that because the data logged by OProfile is used offline for analysis, so each and every sample should be there to perform the offline analysis. On the other hand, the statistics generated by our profiler will be periodically read by the scheduler. Therefore, we do not require a separate large buffer. For our purpose, it is enough to maintain a small buffer of samples that can be continuously read by the profiler to calculate approximate time spent by each kernel running on the GPP of the reconfigurable system, and provide that information to the scheduler.

3.2.2.3 Continuous Profiling

The authors in [8] describe a low overhead (1-3%) sampling profiler, designed to run continuously on multi-processor production systems. The sampling is done at a high rate to increase the accuracy of the profiler. The profiler profiles the entire system (user programs, OS kernel etc...) and not just a single program. Samples of Program Counter are periodically taken on each processor. The sampling uses Processor's Hardware Performance Counters for counting different. A processor generates interrupt when a specified number of events occur, at which point Program Counter's value and hence interrupted instruction is read along with the Processor ID of the active process and the event that caused the interrupt. The Profiler resets Performance Counter with a random value at each interrupt to lessen biasedness towards hitting certain instructions at occurrence of an interrupt. The profiler's data is used offline to analyze time spent in each function, cycles per instruction and variations of data from different profile runs. The data structures used for profiling are mainly modified by the interrupt routine of a processor. On each processor, A hash table is maintained which has (Process ID, Program Counter, Event which caused the Interrupt) triple, as the key, and count as the value. The hash table uses an array of fixed size buckets, where each bucket can store maximum of four entries. The entries which could not be fitted into the hash table are placed in a pair

of overflow buffers. The purpose of using two overflow buffers is to allow an entry to be copied into one of them, while the entry found in the other one is being copied to user space. The purpose of keeping the hash table' size small, is to reduce data cache misses. Each entry in a bucket occupies 16 bytes, and therefore the whole bucket occupies 64 bytes, which is equal to size of cache line of Alpha 21164 processor. Therefore, at most data cache is missed only once when accessing the hash table.

The profiler discussed above is very similar to OProfile. During its discussion, we observed a very important point which is that special attention has to be paid for the data cache misses that can occur while reading and writing to the data structures maintained by the profiler. For our profiler, this can be done by keeping the size of the buffer that holds the samples, as small as possible.

3.2.2.4 ADORE

ADORE(Adaptive Object code RE-optimization)[22] is a trace based user-mode dynamic optimization system, which uses a sampling profiler based on hardware performance counters of IA64. The system is able to apply runtime optimizations such as data cache prefetching, which greatly reduces data cache misses, while the overhead of the system is only 1-2%. It is implemented as a shared library on Linux, which is linked to the application at startup. Two threads are created. One contains the original program, while the other is the dynamic optimization thread. At start up, Perform[2] is initiated, which is a performance monitoring interface for Linux developed by HP. Perfmon is used to read, write and reset the Hardware Performance Counter in the Performance Monitoring Unit of the Processor. It also determines the sampling rate and creates a kernel buffer known as System Sampling Buffer (SSB). A call back function is provided by the system, to copy events logged by SSB to a user buffer when the SSB overflows. The four Hardware Performance Counters are used to monitor various performance metrics like CPU cycles, Cache Misses, Memory Stalls and Pipeline flushes besides others. The Program Monitoring Unit of Itanium 2 have registers that can be used for event based sampling, that is generating of an interrupt when some Hardware Performance Counter overflow, so that a sample could be taken by the profiler. More detail about these Hardware Performance Counters can be found in [12].

We can see that the profiling technique used by ADORE is very similar to OProfile and the profiler discussed in the previous section.

3.2.3 Comparison of Instrumentation-based and Sampling-based Profilers

We have seen that Instrumentation-based profilers are more accurate than Sampling-based profilers, but have higher overheads. Sampling-based profiling can be effectively used to profile an application continuously, like in case of [8]. Another great benefit of Sampling-based profiling is that it is non-intrusive and one does not have to modify any code at either compile time or load time. The downside of Sampling-based profilers however is that the information they give is a statistical approximation. Also, it can only perform flat profiling, that is, we can not use it to count the number of function calls. Therefore the best way is to combine combine good aspects of Sampling and

Instrumentation profiling to achieve better results. This is done by several profilers, like [9], [19] and [33]. For this reason, our profiler also employs both sampling and instrumentation profiling.

3.3 Runtime Profilers for Reconfigurable Systems

All the profilers that we have been discussing in the previous section had one thing in common. They are all meant to speed up programs running on general purpose processors. But, In this section, we will describe research and implementation of profilers which are used to dynamically partition applications among software running on general purpose processor(s) and a reconfigurable fabric.

3.3.1 Frequent Loop Detection Profiler

The Warp processors discussed in section 3.1 use a frequent loop detection profiler to help perform dynamic hardware/software partitioning. The profiler works by monitoring instructions on the memory bus. Whenever it reads a backward branch, it updates a cache entry which stores the branch frequency. The cache can hold maximum 16 entries, and each entry is of 16 bit. The profiler only considers loops whose branch offsets are less than 256, or 64 instructions[27]. The purpose of considering only branches with small offsets is to limit the size of configurable logic and thus save power and area. Due to this and the small cache size, the overhead of the profiler is low at less than 1% of power and area on a MIPS processor. The information from the profiler is read by a Partitioning Co-Processor, found in the Dynamic Partitioning Module, to select most hot branches and transform them into hardware through the configurable Logic.

The advantage of the frequency loop detection profiler is that since it is built as hardware, it is very fast. However, the technique used by it is not suitable for our profiler due to several reasons. First of all it only profile loops, while we are interested in profiling whole functions. Secondly, it deals with addresses at physical level, while we need to deal with virtual addresses as we want to profile programs running in user space. Furthermore, it only keeps counts of 16 loops at a time, while we want to keep calls counts for all the kernels in profiled applications. Lastly, since it is hardware-based, it cannot be easily ported to different reconfigurable systems.

3.3.2 Dynamic Application Profiler

DAProf[26] improves upon the frequent loop detection profiler. Like the frequent loop detection profiler, it also looks for backward branches on microprocessor's instruction bus and is also implemented in hardware. While the frequent loop detection profiler only consider loops with branch offset of less than 256, or 64 instructions, DAProf considers loops whose branch offsets are less than 1024, or 256 instructions. Secondly, while the frequency loop detection profiler only counts iterations of loops, DAProf counts both number of executions and average number of iterations per execution for a loop. The reason of doing so is because it is preferable to map those loops into hardware which have fewer executions and higher iterations per execution. For example, a loop that

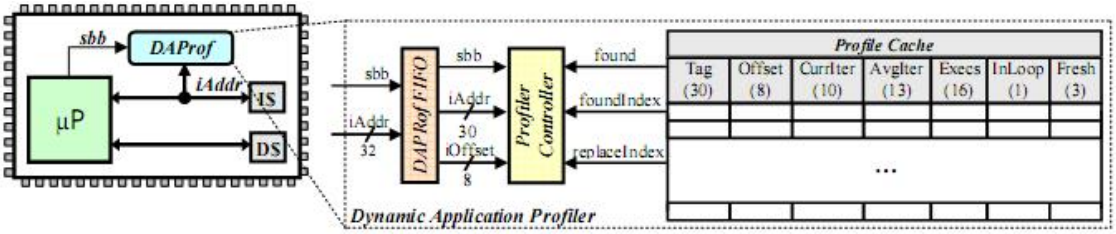

Figure 3.4: Dynamic Application Profiler System [26]

executes only twice and has 500 iterations per execution would be preferred over the one which executes twenty times and have 50 iterations per execution. This is because the communication requirements for the later loop would be greater.

The block diagram of the system is shown in Figure 3.4. Although DAProf can recognize backward branches by sniffing the instruction bus, for simplicity an sbb pin is added to the microprocessor. That pin indicates that a short backwards branch is executed. DAProf contains a FIFO to synchronize with the Microprocessor. This is because the Microprocessor has a greater clock frequency than the profiler hardware. DAProf stores much more information than the frequency loop detection profiler. The tag entry contains the address of a profiled loop, while the offset contains the branch offset of that loop. Both tag and offset are determined by the profiler FIFO. Besides tag and offset, current number of iterations of an executing loop, average iterations per loop execution, and total number of executions of the loop are also stored in in the profiler cache. Also a flag *InLoop* is used to determine if the loop is currently executing. Lastly, a freshness count is maintained to determine how recently the loop has been executed. A higher freshness count implies that the loop has been executing more recently. The total number of iterations of a loop is calculated by multiplying average iterations by total number of executions of that loop. This information, along with the freshness count is used to replace an entry in the profiler cache when the profiler cache is full. The entry with lowest total iterations and freshness count of zero is replaced. DAProf performs slightly better than the frequency loop detection profiler as it has profiling accuracy of 95% as compared to 94% of the former. However, it consumes more power and area than the frequency loop detection profiler.

3.4 Conclusion

We have seen that Instrumentation-based Profiling can be used to record information such as the number of times each function is called in the application, while Sampling-based profiling can be effectively used to approximate the average time spent by functions or any part of the code in running applications. Almost all of the Sampling-based profilers apply the same technique to calculate the approximate time spent by different parts of the code, that is to keep a histogram of program counter values.

For Instrumentation-based profiling, unlike Sampling-based profiling, different profil-

ers use different techniques. For code injection, both Pin and Dyninst use the *Ptrace* API in Linux. Therefore, we can also choose *Ptrace* for our profiler. However, the technique to perform instrumentation differs for different profilers. We have seen that Pin and Dynamo use interpretation of native binary code and JIT compilation, while Detours can perform instrumentation for a function by replacing its prologue with a jump to an instrumentation code. Furthermore, we have hardware based profilers like the frequency loop detection profiler used in Warp Processor. The relative merits and demerits of all these different techniques are given in table 3.1. We have designed our profiler on basis of this table. Our design choices are discussed in detail in the next chapter, where we discuss the design and implementation of our profiler.

Table 3.1: Comparison of different types of profilers

Profiling Technique	Method used	Advantages	Disadvantages	Examples
Instrumentation	Instrumentation Code Insertion at Compile and Link time	Platform Independent	Cannot work with executable binaries	GProf
	Instrumentation Code Insertion during linking of object files	Easy to port	Cannot work with executable binaries	ATOM
	Interpretation of Native Binary Code and JIT Compilation	Instrumentation Code can be inserted anywhere in the code	Has relatively large overhead. Secondly, it is very complicated and time consuming to create such a tool.	Pin, Dynamo
	Modification of Function Prologues to jump to an Instrumentation Code	Has relatively low overhead. Secondly, it is easy to implement and port.	Instrumentation Code cannot be called from anywhere in the code. Secondly, a function prologue must be of a certain minimum size, so that it can be replaced with a jump to the Instrumentation Code.	Detours and IgProf
Sampling	Using Timer Interrupts	Has low overhead and is non-intrusive	Can only give statistical approximation about time spent by different parts of the code	GProf
	Using Timer Interrupts and Hardware Performance Counters	Has low overhead and is non-intrusive. Secondly, it also gives additional information like the parts of code causing more cache misses, pipeline stalls etc.	Same as above	OProfile
Hardware-based	Using custom-designed hardware	Has very low overhead and is non-intrusive	It is difficult to port a design to another system. Secondly, consideration has to be given to handle virtual addresses.	Frequency Loop Detection Profiler and DAProf

4 Design And Implementation

In this chapter we discuss the design and implementation of our profiler. Section 4.1, discusses the overall design of the profiler while section 4.2, discusses its implementation. Finally, conclusion is given in section 4.3.

4.1 Design of the Profiler

Reconfigurable computing platforms with multitasking operating systems running on them, might have multiple threads competing for the limited reconfigurable hardware resources. To manage these hardware resources properly, a runtime system is required. The runtime system has to know the current configuration and load of the system to properly allocate the reconfigurable hardware to competing threads in the system. The runtime system for MOLEN was shown in figure 2.3. The proper allocation of hardware resources there is done by the *Scheduler*. The *Profiler* assists the *Scheduler* in performing its task by providing vital statistics about programs running on the system. The *Profiler* records these statistics in the *Logs* shown in figure 2.3. Since, like the *Scheduler*, the *Profiler* also needs to continuously run on the system, it must have as less overhead as possible. Moreover, writing to and reading from the *Logs* should also impose very low overhead.

In this section, first we will discuss the design choices that we made and then discuss the basic design of our profiler.

4.1.1 Design Choices

Our profiler performs both Instrumentation and Sampling. The design choices that we made for both of them are discussed next.

4.1.1.1 Sampling Profiling

We are not only interested in the number of times different functions are called, but also approximate time spent per function, because it is preferable for the *Scheduler* to map those functions into hardware which take longer times to execute. The basic idea of how sampling can be used to find out approximate time spent by a function is that if function A takes ten times longer to execute than function B, then likelihood of a timer interrupt occurring in function A would be ten times more than the likelihood of it occurring in function B. However, this method only gives a statistical approximation and not the exact value, but the accuracy can be increased by increasing the frequency of timer interrupts.

We have seen in the last chapter that all Sampling-based profilers use the same technique to find out the approximate time spent by different parts of the code, that

is by recording program counter's value at each timer interrupt. Therefore, there is a standard way of doing this. For this reason, we also used the same technique. However, we need to be careful about one thing, which is that we should be able to take samples from each core of a multicore general purpose processor. Since, we are testing our profiler with an x86 general purpose processor, we used the local APIC timers, which are found for each core of the processor. At each interrupt of a local APIC timer, the program counter, process ID of the interrupted process and the number of the core on which the interrupt occurred is read and stored in a buffer that we call the *samples buffer*. This *samples buffer* is periodically read to find out the approximate time spent by the profiled functions. The reading of the *samples buffer* by the *daemon* of the profiler (the *daemon* is discussed later in this chapter) is done through a file in */proc* directory. The */proc* file system is basically a virtual file system, in which the data in files do not reside in physical drives but in the memory. It is standard way of communicating data from kernel space to user space in Linux. The code to handle the APIC timer interrupts actually runs in kernel mode, while the *daemon* of our profiler runs in user mode.

4.1.1.2 Instrumentation Profiling

Through instrumentation profiling, we want to record the number of times different functions have been called. To instrument programs, we need some mechanism to inject code into those programs. In the last chapter, we saw that it is a standard practice for profilers that run on Linux to use the *Ptrace* API. We saw that both *Pin* and *Dyninst* use this technique to inject code into and modify code of the running applications. Therefore, we also use the *Ptrace* API for this purpose.

However, the technique used to perform instrumentation is different for different profilers. In our case, we have to work with executable binaries and therefore, we cannot perform compile or link time code injection like *GProf* or *ATOM* do. We need to either modify the executable binary, so that it can contain and use an instrumentation code or use a hardware approach as used by the *frequency loop detection profiler* and *DAProf*.

One way to count the times each kernel is called is by using a custom-made hardware. Although the hardware approach has a very low overhead, it has several disadvantages, as were discussed in table 3.1. One obvious disadvantage is that it is not very portable. A hardware design made for one system might not fit into another one. The second disadvantage is that we will also need to take care of the virtual addresses in the system, since at hardware level we will only be seeing the physical addresses. Since the Warp Processors only deal with small loops that they can see from the instruction bus, they don't require to know the virtual addresses. On the other hand, for MOLEN, we will be dealing with software implementations of the kernels and therefore would require to know the virtual addresses of the instructions in the programs that we will profile. Therefore, it is advantageous for us to implement this part of the profiler in software. For that purpose, we have two choices. Either we should perform native binary code interpretation and JIT compiling, like *Pin* and *Dyanmo*, or we can use the approach used by *Detours* and *IgProf*, which is to modify a prologue of a function with a jump to an Instrumentation Code.

The native binary code interpretation and JIT compiling approach, although very

useful to create versatile profiling tools like *Pin*, has several disadvantages for our purpose. The first of which is that it is slow. We created a *Pin* tool which just counts the number of time each function is called in an application and observed that the overhead was never less than 20%. The code for that *Pin* tool is given in appendix A. Another disadvantage of this approach is that it is very complicated and time consuming to write a native binary code interpreter and JIT compiler for any processor. While it is true that we can use the source code of *Pin* for platforms for which *Pin* is available, but to port such a tool to a processor for which it is not available is a very complicated task. For example, at this moment *Pin* does not support PowerPC, while the general purpose processor used in MOLEN uses PowerPC as the GPP.

We are finally left with the technique used by tools like *Detours* and *IgProf*, which is to replace a function prologue with an unconditional jump to an instrumentation code and having the instrumentation code contain the removed prologue as well as an unconditional jump to the remaining part of the instrumented function. This technique also has its own disadvantages, the foremost of which is that the instrumentation code cannot be inserted anywhere in the code, but since for this thesis, we are only interested in instrumenting the number of times each function is called, it is not a limitation for us. Another disadvantage of this approach is that an unconditional jump can only be inserted in place of prologue of a function, if the prologue of that function consists of at least the number of bytes an unconditional jump instruction takes. Experiments have shown that most of the functions in applications do have required prologue sizes (See section 5.6 for detailed discussion on this). Secondly, currently MOLEN do not support Just In Time FPGA compilation and therefore can only transform those kernels into hardware for which there is a hardware implementation available. Those kernels also have corresponding software implementations. In cases where the prologue of a kernel do not have enough instructions to replace with a jump instruction, we can deliberately put extra instructions to allow them to be instrumented by our profiler. For x86, these extra instructions can be NOP instructions.

For our purpose, the scheme used by *Detours* and *IgProf* has several benefits. Firstly, with this approach, low overhead instrumentation can be achieved by using an efficient instrumentation code. Secondly, this approach is much less complicated than the native binary code interpretation approach. Lastly, this technique is very portable, as one only has to deal with function prologues and not the rest of the code inside the functions. Special attention has to be paid to the mechanism for writing the function counts, as it can have huge impact on the performance. We tried two different approaches for that purpose. First we tried to keep the counts in a file in */proc* directory. As discussed previously, files in */proc* directory are not located in physical drives but reside in the memory, therefore writing to them is much faster than writing to a normal file. However, we still saw that this method imposed a lot of overhead. Therefore we had to shift to the shared memory approach, in which we kept the count in a shared memory. The detailed discussion on the */proc* file and shared memory approaches is discussed in section 4.2.1.1.

Table 4.1: Logical View of the Interface between *Profiler* and the *Scheduler*

Number of Functions = n		
0	Calls Count for Function *0*	Approximate Time Spent in Function *0*
1	Calls Count for Function *1*	Approximate Time Spent in Function *1*
2	Calls Count for Function *2*	Approximate Time Spent in Function *2*
.	.	.
.	.	.
n-1	Calls Count for Function *n-1*	Approximate Time Spent in Function *n-1*

4.1.2 Basic Design

Our profiler can either perform *global profiling* or *per process profiling*. *Per process profiling* means that the profiler would be keeping profiled statistics for each process separately, while *global profiling* means that the profiler would be keeping profiled statistics for all processes in one place. For *global profiling*, we have made a simplifying assumption that functions in two different processes with the same names are considered the same functions. Therefore, for example, if we have a function by the name of *functionA* and it is found in both process *A* and process *B*, then in case of *per process profiling*, statistics for *functionA* in process *A* would be kept separately from those for *functionA* in process *B*, while for *global profiling*, they would be kept in the same place. In future, when the *Profiler* will be integrated with the *Scheduler*, experiments could be carried out to know which scheme is better. *Per process profiling* might be more useful in cases where different processes have different priorities, while *global profiling* would be more useful when all processes have the same priority.

The logical view of a log maintained by the *Profiler* for the *Scheduler* is shown in table 4.1. In our implementation, the logs are actually memory areas which are shared between the *Profiler* and the *Scheduler*. We have used this approach because it is the fastest possible mechanism of communicating data. The *Profiler* records counts of function calls as well as approximate time spent per function. In future, statistics produced by memory profiling would also be kept in that log. Each profiled function has a corresponding index, through which statistics related to it are updated and read from the log. Since indexes are used to read information about the profiled functions, the *Scheduler* requires a mechanism to get corresponding values of function indexes through function IDs, where depending upon the configuration of the system, function ID can be either the function's name or its address or any other parameter. This can be achieved by using a Hash Table. For this reason, the *Profiler* also maintains a Hash Table. The Hash Table is keyed by function IDs and its values are the function indexes used to index its corresponding statistics in the log.

In figure 2.3, there is also an arrow from the *Scheduler* to the *Profiler*. It signifies the fact that the *Scheduler* can change parameters of the *Profiler* at runtime. For example, it can ask the profiler to either perform *per process profiling* or *global profiling*. But to test our profiler at this moment, we read these parameters from a configuration file, so

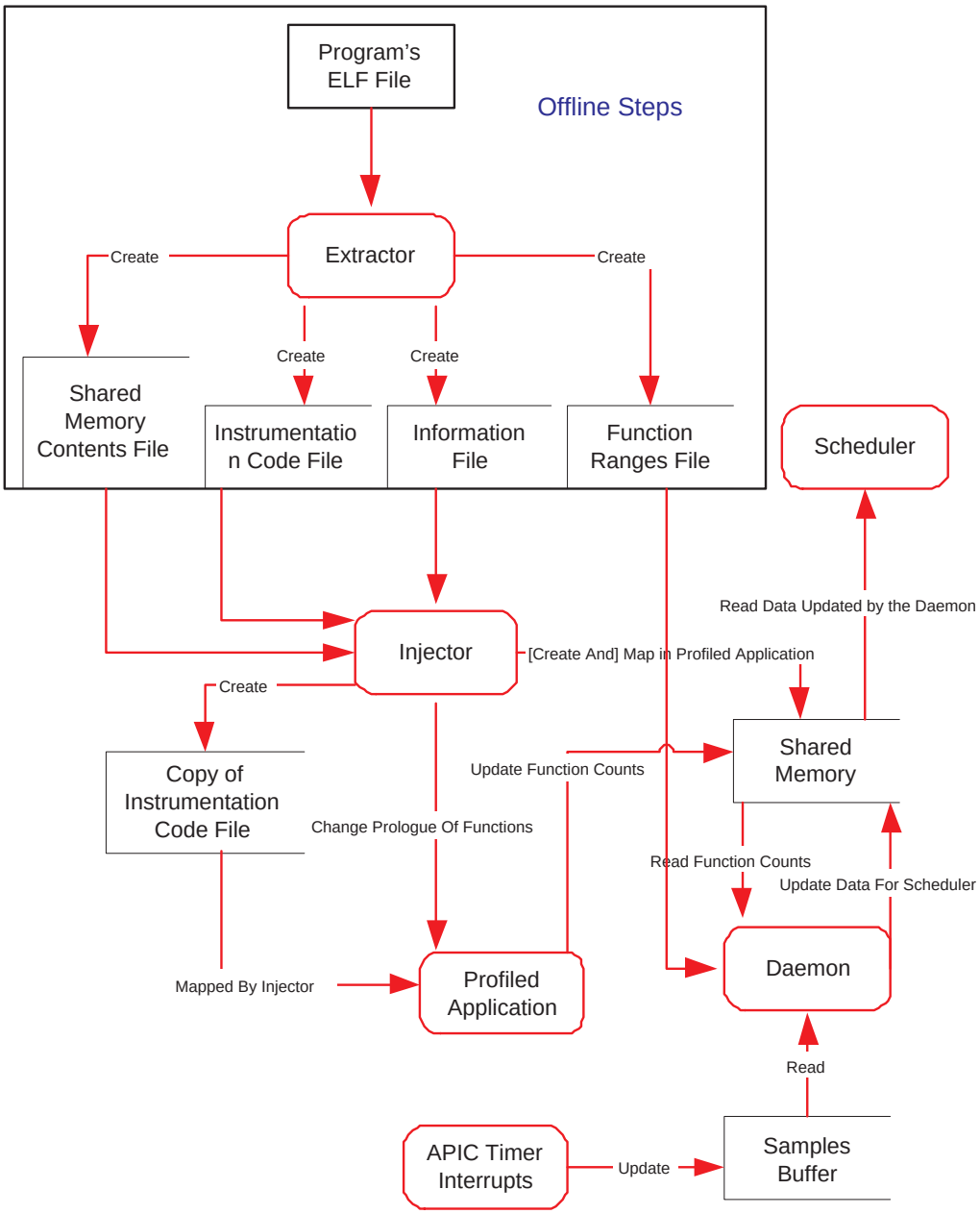

Figure 4.1: Interaction of different parts of the *Profiler* with the *profiled application* and the *Scheduler*

that we can test our profiler in a stand alone fashion.

4.1.2.1 Components of the Profiler

Our profiler consists of several different parts. The interaction of those different parts with the *profiled application* and the *Scheduler* is shown in figure 4.1. For each program that has to be profiled in our system, an instrumentation code file and a shared memory contents file is created. Moreover, for *global profiling*, one global shared memory contents file is also created.

Instrumentation code file is used to update function calls counts for profiled functions and those counts are placed in a shared memory file, so that the *Daemon* (discussed later) can read those counts. Both instrumentation code and shared memory contents files are created offline, but their copies are injected into the profiled programs at runtime. These files are created by using the *Extractor* utility. The *Extractor* reads ELF file of a program and generates instrumentation code and shared memory contents files from it. Besides, it also creates several other files which are used to inject and modify code at runtime. The *Extractor* utility and the files created by it are explained in detail in section 4.2.1.

Copies of the instrumentation code and shared memory contents files are injected into the address space of a profiled process at runtime, by using the *Injector* utility. The *Injector* utility uses the *Ptrace* API in Linux to inject and modify code at runtime. The copy of instrumentation code file contains the process ID in its name, so that it can be easily identified by process ID. The *Injector* utility is explained in detail in section 4.2.2.

As discussed previously, *samples* containing information about current running process, core number of the processor and program counter value, are collected at each local APIC timer interrupt. These samples are placed in a *samples buffer*. The *Daemon* periodically reads *function call counts* from the *Shared Memory* and samples from the *samples buffer*. From the samples, for each profiled function, it estimates the time spent in those function. It can do so because for each profiled process, it first reads a file that contain starting and ending addresses of profiled functions in that process. The file containing those function ranges is also created by the *Extractor* utility. The *Deamon* compares the process ID and program counter value saved in the sample to see which function was being executed when the sample was taken. These samples counts are used to know the approximate time spent by each function. We refer to these samples counts as *function samples counts*. It is also the responsibility of the *Daemon* to update logs for the *Scheduler*. Sampling is discussed in section 4.2.4, while the *Daemon* is discussed in section 4.2.5.

The *Shared Memory* is used to keep the *function calls counts* as well as to provide data to the *Scheduler*. For *per process profiling*, a separate *Shared Memory* is created for each process and the name of each shared memory file contains the process ID in its name, so that it can be easily identified. However, for *global profiling*, one global shared memory is used. Since the profiled applications increment the function calls counts directly in the *Shared Memory*, the *Shared Memory* file needs to be mapped into the address space of the profiled applications. Besides the profiled applications, the *Daemon* and the *Scheduler* also map the *Shared memory*. The *Daemon* does so to be able to read the *function calls counts* updated by the profiled applications, while the *Scheduler* needs it to read the logs updated by the *Daemon*, which are also placed in the *Shared Memory*. The *Shared Memory* is discussed in detail in section 4.2.3.

4.2 Implementation of the Profiler

Our profiler consists of several components. In this section, each one of them is discussed in detail. Section 4.2.1 discusses the *Extractor* utility while section 4.2.2 discusses the *Injector*. *Shared Memory* is discussed in section 4.2.3, followed by a section about the *Sampler*. Finally, the *Daemon* is discussed in section 4.2.5.

4.2.1 Extractor

The purpose of the instrumentation code is to record the function calls counts. The instrumentation code is generated offline but is injected into the profiled program at runtime. The reason for generating instrumentation code files offline is to speed things up, because doing so at runtime would impose considerable overhead. The *Extractor* utility reads an ELF executable file and generates an instrumentation code file from it. Besides that, it also creates several other files. One of these contains information about the to be profiled program and its instrumentation code. We refer to this file as the *information file*. Another is used to contain the contents of the shared memory. This file will be referred as *shared memory contents* file. The third file is used to hold the address ranges of the profiled functions. The purpose of this file is described in section 4.2.5, where we discuss the *Daemon*. We refer to this file as *function ranges file*. Besides these three files and the instrumentation code file, two other files are created. One of these is copy of the *information file* and the other is copy of *function ranges file*. These copies are same as original files but with different function indexes. Instead of local function indexes, they contain global function indexes. Local function indexes are used for *per process profiling*, while global function indexes are used for *global profiling*. Note that for *global profiling*, one global *shared memory contents* file is used, while for *per process profiling*, for each program, one separate *shared memory contents* file is created by the *Extractor*. We will refer to the files generated by the *Extractor* for a program, collectively as *profiling information files*.

The full path of the program is used to create the path for *profiling information files*. For example, the files generated for an executable file */usr/home/hmushtaq/prog*, will be placed in the folder named */prof_files_path/usr/home/hmushtaq/prog*, where *prof_files_path* is some predefined folder to contain the folders and files created by the *Extractor* utility. The reason for putting these files in such paths is to make locating these files easy and fast, as all that is required is to know the absolute path of the profiled program. In *Linux*, one just needs to use the *realpath* function to know the absolute path of a program from a relative path and use *readlink* to know the absolute path from a process ID. The instrumentation code file is named as *injected.o*, the two *information files* are named as *data.bin* and *datag.bin*, where *datag.bin* is the one which contains global function indexes. The shared memory contents file for the program is named *shmem.dat* and the two function ranges file are named as *funcs.bin* and *funcsg.bin*. Besides, there is only one *shared memory contents* file for *global profiling*, whose full path is */prof_files_path/shmemg.dat*.

In current implementation, we only extract information from a program's ELF file and therefore are only able to profile static functions.

4.2.1.1 Instrumentation Code file

A copy of the instrumentation code file generated by the *Extractor* utility, is mapped into the address space of the profiled program by the *Injector*. The instrumentation code is used to update calls counts for profiled functions. For each function that has to be profiled, the *Injector* replaces its prologue with a jump to a part in the instrumentation code that updates the calls count for that function. After executing the code to increment calls count, the instrumentation code executes the replaced prologue of the function before jumping back to the remainder of the profiled function. To increment function calls count, we tried two different approaches. One was to use a file in */proc* directory to hold the function counts, while the other was to use the *Shared Memory*. Since the shared memory approach is much faster, we sticked to that approach. Below, first we describe the */proc* file approach and its disadvantages, and then we describe our current implementation with shared memory in detail.

/Proc File Approach In this case, we maintain a file in */proc* directory to keep the *function calls counts*. The */proc File* system is actually a *virtual file system* used in Linux to communicate data between user and kernel space. The files do not reside on a physical storage, but are kept in memory and therefore reading and writing to them is much faster. In this case, the file that we used consisted of an array of function calls counts indexed by the function number. The instrumentation code for a function, uses the *write* system call to update the function calls counts in the file. The drawback of this method is that the system call is very expensive. The system call is done through an interrupt instruction and therefore there is a context switch for each such call. To mitigate this overhead, we updated the function calls count at each 50th hit of a function. That technique lessened the overhead significantly, but still the overhead was in the range of 10-15% for medium sized functions. Moreover, we had the disadvantage of only having counts at intervals of 50. The sampling part of the profiler has to be combined with the instrumentation part to find out the approximate average time spent per each function. With this scheme, we would have a lot of error in calculating that time. To overcome these problems, we devised a new scheme, which is to use shared memory to keep the function calls counts. That scheme is described below.

Shared Memory Approach In this approach, a file of fixed size is shared in the memory, so that different parts of the profiler can write and read from it. The interaction of shared memory with different parts of the profiler was shown in figure 4.1. A portion of that shared memory file contains array of function calls counts indexed by function numbers, so that the instrumentation code for a function can directly increment the calls count for that function in the shared memory.

To illustrate this procedure of keeping calls counts in the shared memory, let us take a program which contains two functions that need to be profiled. Listing 4.1 shows the disassembly of two such functions. Function *mul* returns product of two numbers, while *sqr* returns the square of a number. The instrumentation code for this program is shown in listing 4.2. The prologue of a to be profiled function is replaced by the *Injector*, with an unconditional jump to the instrumentation code for that function. So in this case, it will be replaced by jump to *lbl_start1* for *mul* function and *lbl_start2* for *sqr* function. More on this is discussed in section 4.2.2, where we discuss the *Injector*.

Listing 4.1: Sample functions to be instrumented in original form

```
1  080483b4 <mult>:
2   80483b4:        55                              push   %ebp
3   80483b5:        89 e5                           mov    %esp,%ebp
4   80483b7:        83 ec 10                        sub    $0x10,%esp
5   80483ba:        8b 45 08                        mov    0x8(%ebp),%eax
6   80483bd:        0f af 45 0c                     imul   0xc(%ebp),%eax
7   80483c1:        89 45 fc                        mov    %eax,-0x4(%ebp)
8   80483c4:        8b 45 fc                        mov    -0x4(%ebp),%eax
9   80483c7:        c9                              leave
10  80483c8:        c3                              ret
11
12 080483c9 <sqr>:
13  80483c9:        55                              push   %ebp
14  80483ca:        89 e5                           mov    %esp,%ebp
15  80483cc:        83 ec 10                        sub    $0x10,%esp
16  80483cf:        8b 45 08                        mov    0x8(%ebp),%eax
17  80483d2:        0f af 45 08                     imul   0x8(%ebp),%eax
18  80483d6:        89 45 fc                        mov    %eax,-0x4(%ebp)
19  80483d9:        8b 45 fc                        mov    -0x4(%ebp),%eax
20  80483dc:        c9                              leave
21  80483dd:        c3                              ret
```

Listing 4.2: Instrumentation code for sample program

```
1          .text
2  lbl_start1:
3          incl 0x0
4          push %ebp
5          mov %esp,%ebp
6          sub $0x10,%esp
7  lbl_jmp1:
8          jmp 0x7fffffff
9  lbl_start2:
10         incl 0x0
11         push %ebp
12         mov %esp,%ebp
13         sub $0x10,%esp
14 lbl_jmp2:
15         jmp 0x7fffffff
```

The *incl* instructions at line 3 and 10 of the instrumentation code take the addresses of the shared memory locations where the count of function calls are placed. The addresses of those memory location is only available at runtime and therefore *0x0* is replaced by the actual addresses of shared memory locations by the *Injector*. Since, the increment is done through only a single instruction, In case of single-core processors, it is atomic

and does not pose a problem of race condition in cases where different threads are trying to write to the same memory location in the *Shared Memory*. In case of multi-core processors, the *incl* is preceded by *lock* instruction, to avoid different processors writing to the same memory location at the same time. After the *incl* instruction, there is the replaced prologue of the profiled function followed by a jump to the remainder of that function. In x86, the jump addresses are relative, and since the starting address of the instrumentation code is only available at runtime, therefore *0x7fffffff* at line 8 and 15 are replaced by appropriate values by the *Injector* at runtime.

4.2.1.2 Information file

The contents of the *information file*, generated by the *Extractor* for a program, are shown in table 4.2. The flag *StopProgramAtBeginning* is used to specify if the first instruction(s) of the program has to be replaced by a jump instruction, which just keeps on jumping back to the same location and therefore stops the program execution at the beginning. To do that, the *Extractor* utility is run with *-s* flag. Actually, a profiled program can be run in two ways. Either it can be run as a child process of the *Injector* program or it can be run in a normal fashion. When the program is run as a child process, there is no need to insert the stopping jump instruction at the beginning of the program because the *Injector* program gets hold of the to be profiled program from the very beginning, but in cases where that program does not run as a child of the *Injector*, the instrumentation code is injected into it and shared memory mapped into it by attaching to it while it is running. The *Daemon*, which is described fully in section 4.2.5, actually receives signals from kernel whenever a new program is loaded to execute. If the program has corresponding *profiling information files*, then the *Daemon* performs injection and mapping by calling the *Injector* utility with -i flag and Process ID as parameter. When the *Injector* is called that way, it attaches to the running program while it is running and detaches form it as soon as it has inserted the instrumentation code and mapped shared memory file into it. Now when the program is stopped in that way, that is while it is running, it might crash. For this reason, the *Extractor* utility has this option of stopping the program at the very beginning. But as discussed previously, it is only required if the program has to be run in a normal fashion and not in supervision of the *Injector* program. The next two contents of the information file are Starting Address of the program and first 4 bytes of the program. This information is needed if one wants to revert the program back to its normal form, that is, replace the jump instruction which stops the program at the beginning with the original instruction(s).

After that, we have information about program's size and its last modification date. The purpose of keeping this information is that the *Injector* can check the current size and last modification date of the to be profiled program and compare it with that in the *information file*. If the values differ, then it means that the instrumentation files were created from some older version of that program. In that case, the *Injector* let the program continue without profiling. Next, we have the values of Code and Data segment offsets of the instrumentation code file. The code segment offset is used to locate the labels and variables in the instrumentation code file. The data segment offset is not used and is only kept as a reserve. After that, we have number of to be profiled functions

Table 4.2: Information File generated by the *Extractor*

StopProgramAtBeginning flag
Starting Address of Program
First 4 bytes of Program
Instrumentation Code File's Size
Instrumentation Code File's Modification Time
Instrumentation Code File's Code Segment Offset
Instrumentation Code File's Data Segment Offset
Number of Profiled Functions
Information about the First Profiled Function
Information about Second Profiled Function
.....
Information about the Last Profiled Function
Number of labels in Instrumentation Code File
Information about the First Label in Instrumentation Code File
Information about the Second Label in Instrumentation Code File
.....
Information about the Last Label in Instrumentation Code File

followed by information about each such function.

The information of a function is shown in table 4.3. We have starting address of the function, followed by the address to which the instrumentation code for that function will return. Next, we have the index of that function, followed by the size of the new prologue. Lastly, we have the new prologue of the function. By new prologue, we mean the jump instruction which jumps to the instrumentation code. That jump instruction may be followed by one or more *nop* instructions. This happens if the prologue of the function is more than 5 bytes, keeping in mind that an unconditional jump takes 5 bytes. So if for example the original prologue of a to be profiled function was six bytes, the new prologue will consist of a jump instruction followed by a *nop* instruction to make up full six bytes.

The last thing in the information file is the number of labels in the Instrumentation code followed by the information about those labels. The labels in the instrumentation code for the sample program, as shown in listing 4.2 are *lbl_start1*, *lbl_jmp1*, *lbl_start2* and *lbl_jmp2* respectively. The information about a label consists of its name and address in the instrumentation code file. This information is used by the *Injector* to find relative jump addresses and the address of references to shared memory locations.

4.2.2 Injector

The *Injector* utility is used to map *Instrumentation Code* and *Shared Memory* files into the address space of a process that has to be profiled. It can be used in two ways, as was discussed in the previous section. It attaches to a process that has to be profiled by

Table 4.3: Information about a Function in Information File

Starting Address of the Function
Address to which Instrumentation Code will return
Index of the Function
Size of New Prologue of the Function
New Prologue of the Function

Table 4.4: Information about a Label in Information File

Name of the Label
Address of the Label

using *ptrace*. We have used the *mmap* system call to map both *Instrumentation Code* and *Shared Memory* files into the address space of a process that has to be profiled. The *Injector* utility copies the instrumentation code file */prof_files/[program_path]/injected.o* to */tmp_files/injected[pid].o*, where *tmp_files* is a predefined folder used to hold all such temporary files. The process ID is used in the name so that no instrumentation code file is accidentally shared by different processes. The code to perform *mmap* system call is shown in listing 4.3. 00000 at line 25 is replaced by the process ID padded with zeros by the *Injector* utility, before executing the *mmap* code. Note that the *mmap* code also has a breakpoint instruction at line 22, the purpose of which is to stop the execution in to be profiled process, so that the *Injector* can take the control back. Also note that at line 14, the file length is calculated at runtime and therefore *0* is replaced by the actual file length. The *Injector* backs up some data from the to be profiled process, and also backs up processor registers and injects the *mmap* code in place of the backed up data. It then executes the injected *mmap* function on behalf of the process that has to be profiled, by using *ptrace*. After executing the *mmap* function and regaining control, it puts back the backed up data and saved processor registers. By that time, the *Instrumentation Code* file has been attached to the process that has to be profiled.

The *Injector* attaches *Shared Memory* file to the process, in exactly the same way as the *Instrumentation Code* file, that is by executing *mmap* function from within that process. For that purpose, it uses the same code as given in listing 4.3 except for the file name at line 25. If *global profiling* is done, than all processes write to the same *Shared Memory*, but if *per process profiling* is done, then for each process, a separate *Shared Memory* file is created by the *Injector*. In that case, like the *Instrumentation Code* file, the *Shared Memory* file is also named according to the process ID of the profiled process as */tmp_files/[pid].dat*. For example, if the profiled program's process ID is 3478, then the name of the *Shared Memory* file would be */tmp_files/03478.dat*. Like in the case of *Instrumentation Code* file, here the file name is also padded by zeros. For *global profiling*, the *Shared Memory* file is just named as */tmp_files/00000.dat*.

Listing 4.3: Code that performs mmap system call

```
1  void main()
2  {
3  __asm__("jmp forward\n"
4  "backward:\n"
5           // open
6          "popl   %esi\n"
7          "movl   $5, %eax\n"        // syscall_open = 5
8          "movl   %esi, %ebx\n"
9          "movl   $2, %ecx\n"        // O_RDWR = 2
10         "int    $0x80\n"
11         // mmap
12         "subl   $24, %esp\n"
13         "movl   $0, (%esp)\n"
14         "movl   $0, 4(%esp)\n"     // filelen
15         "movl   $3, 8(%esp)\n"     // readwrite
16         "movl   $1, 12(%esp)\n"    // MAP_SHARED = 1
17         "movl   %eax, 16(%esp)\n"// fd = %eax
18         "movl   $0, 20(%esp)\n"
19         "movl   $90, %eax\n"       // syscall_mmap = 90
20         "movl   %esp, %ebx\n"
21         "int    $0x80\n"
22         "int3\n"                   // breakpoint
23 "forward:\n"
24         "call   backward\n"
25         ".string \"/tmp/mprof/injected00000.o\"");
26 }
```

The whole process of injecting *Instrumentation Code* and *Shared Memory* files into a process that has to be profiled is shown in figure 4.2, in the form of a sequence diagram. That figure also shows further steps that are done by the *Injector*. Those steps are discussed next.

After mapping *Instrumentation Code* and *Shared Memory* files into the process that has to be profiled, the *Injector* also modifies that process and its instrumentation code. The prologue of to be profiled functions are replaced by a jump instruction to the *Instrumentation Code*, the addresses of the shared memory locations are found and inserted into the *Instrumentation Code* as well as the jump addresses. For the sample functions shown in listing 4.1, the functions look like what is shown in listing 4.4 after modification. Note that there are *nop* instructions after the jump instructions on line 3 and 13. This is due to the fact that the jump instruction takes 5 bytes, while the replaced prologue for both functions took 6 bytes. Therefore, to make the new prologues equal to 6 bytes, an extra *nop* instruction is added. Likewise, the instrumentation code is changed from what is shown in listing 4.2 to listing 4.5, assuming that function indexes of *mul* and *sqr* functions are 0 and 1 respectively. The *mmap* function actually returns the starting address of the file mapped and we use that starting address in case of both

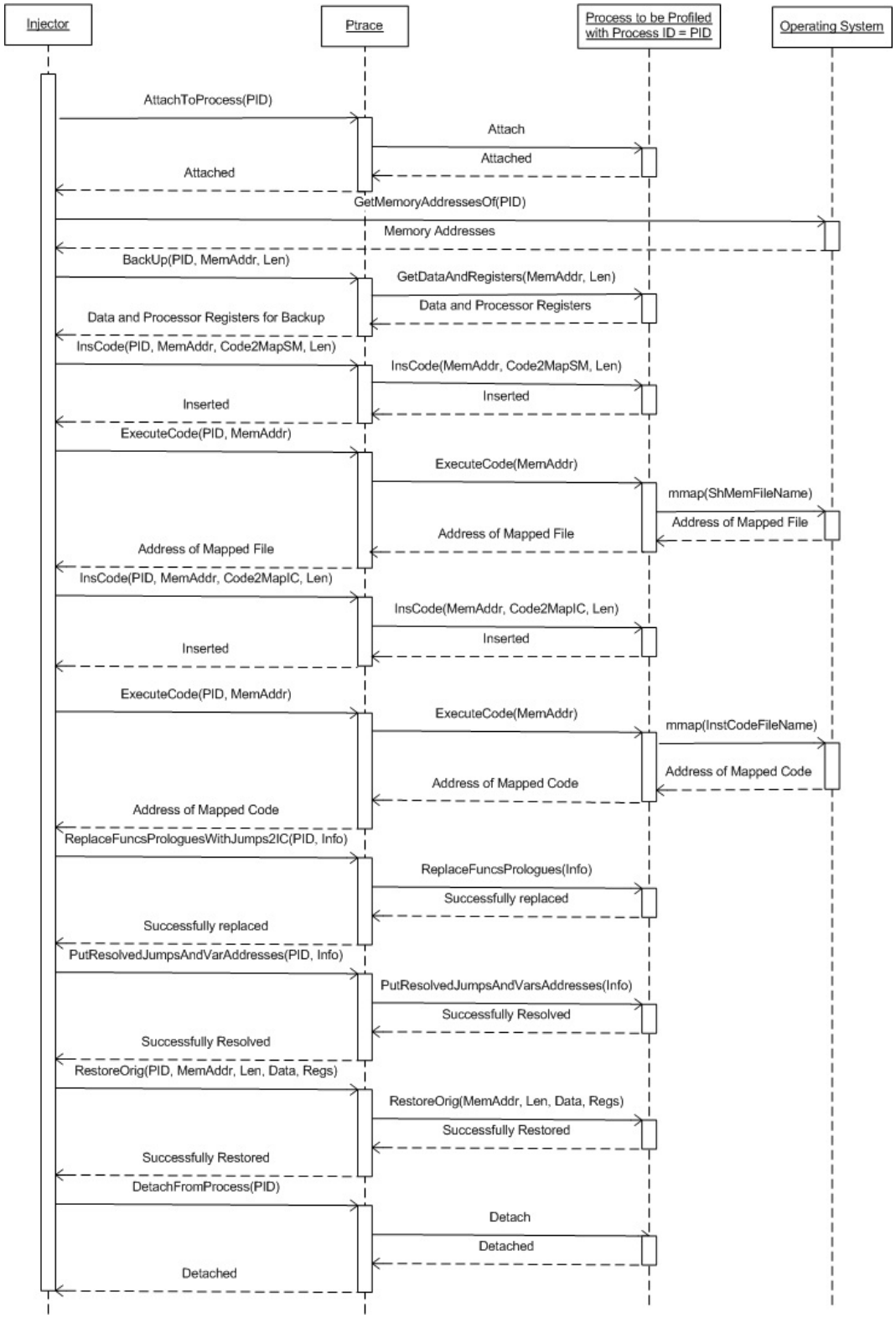

Figure 4.2: Interaction between the *Injector* and the process that has to be profiled

Instrumentation Code and *Shared Memory* files. The starting address of mapped *Instrumentation Code* file and the code segment offset along with the relative addresses of the labels defined in the *information file*, are used to find out the addresses of references to the shared memory locations on lines 3 and 10 of the *Instrumentation Code* as well as the jump addresses on lines 8 and 15. Note that the opcode for jump instruction takes relative addresses, so when we say *0x80483ba* or *0x80483cf*, that does not mean that the machine code will contain that value, but it actually contains the difference between the referral point and that address, and since the starting address of *Instrumentation Code* file is only known at runtime, this value has to be calculated and inserted by the *Injector* as well. Note that besides the jump instructions in line 3 and 8 of the *Instrumentation Code*, the jump instructions inserted in place of function prologues also use information from the information file and starting address of the *Instrumentation Code* to calculate the relative addresses. Similarly, the starting address of mapped *Shared Memory* and function indexes are used to find out the addresses of shared memory locations in the *Instrumentation Code*.

Listing 4.4: Sample functions after modification for Instrumentation by the *Injector*

```
1  080483b4 <mult>:
2   80483b4:    jmp    lbl_start1
3   80483b9:    nop
4   80483ba:    mov    0x8(%ebp),%eax
5   80483bd:    imul   0xc(%ebp),%eax
6   80483c1:    mov    %eax,-0x4(%ebp)
7   80483c4:    mov    -0x4(%ebp),%eax
8   80483c7:    leave
9   80483c8:    ret
10
11 080483c9 <sqr>:
12  80483c9:    jmp    lbl_start2
13  80483ce:    nop
14  80483cf:    mov    0x8(%ebp),%eax
15  80483d2:    imul   0x8(%ebp),%eax
16  80483d6:    mov    %eax,-0x4(%ebp)
17  80483d9:    mov    -0x4(%ebp),%eax
18  80483dc:    leave
19  80483dd:    ret
```

Listing 4.5: Instrumentation Code of sample program after modification by the *Injector*

```
1         .text
2 lbl_start1:
3         incl shared_mem[0]
4         push %ebp
5         mov %esp,%ebp
6         sub $0x10,%esp
7 lbl_jmp1:
8         jmp 0x80483ba
9 lbl_start2:
10        incl shared_mem[1]
11        push %ebp
12        mov %esp,%ebp
13        sub $0x10,%esp
14 lbl_jmp2:
15        jmp 0x80483cf
```

4.2.3 Shared Memory

Shared Memory is accessed by the profiled application, *Scheduler* and the *Daemon* as shown in figure 4.1. The reason we used a shared memory is because it is the fastest mechanism to access and communicate data. The contents of the shared memory are shown in figure 4.3.

From that figure, we can see that the upper area of the shared memory is used to hold the function calls counts. The data in that area is written by the profiled application. If *per process profiling* is being performed, than each profiled process will have its own *Shared Memory* file, while in case of *global profiling*, only one *Shared Memory* file is used. For *per process profiling*, the *Shared Memory* file is created by the *Injector* and the *Daemon* only requires to map that file.

The middle portion of the *Shared Memory* is used to present data to the *Scheduler*. That data contains the difference of function calls and samples counts for a given span time. For example, if a process has been running for 7 seconds and the *span time*, which is read from a configuration file, is 5 seconds, then that data will contain the function calls and sample counts since the last 5 seconds. That data is updated every 1 second by the *Daemon*. We need to make sure that the *Scheduler* does not read the data at the same time when the *Daemon* is writing to it. To achieve that, one option was to use some synchronizing mechanism, but since the *Daemon* is running in user space, any synchronization between it and the kernel space would be very inefficient. For example, there may be a situation when the *Daemon* has acquired the lock but is interrupted in between. Now if the *Scheduler* runs at that time, it won't be able to acquire that lock and therefore read data from the *Shared Memory*. For this reason, we have used the concept of double buffering, that is, the *Daemon* writes data to one buffer and the next time it writes to the other. When it completes writing data to one buffer, it changes the *index* to point to that buffer, so that the *Scheduler* can read it from there. Now even

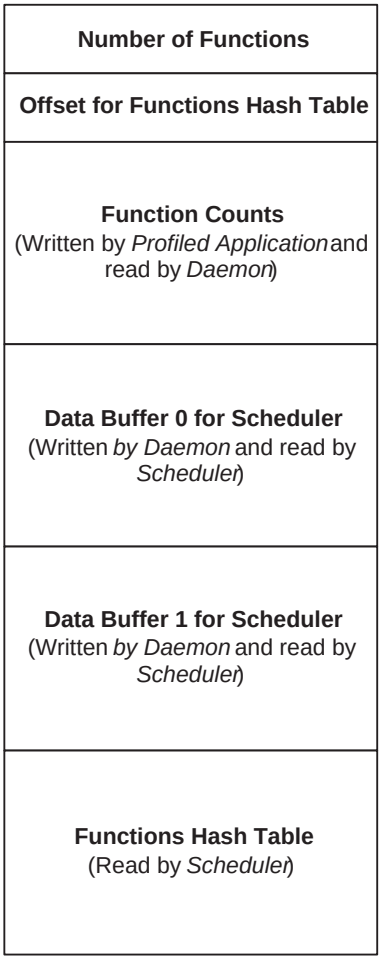

Figure 4.3: Contents of Shared Memory

in this case, it might happen that the *Daemon* was modifying the index value when the *Scheduler* was reading from it. However, that does not pose a problem, because it only happens when the *Daemon* has already written the new data to the buffer. In worst case, the *Scheduler* will read data from the other buffer, which was updated one second ago. Therefore, if the *Scheduler* can read data within one second, which is a fair assumption, then this schemes works fine and is quite efficient, albeit at the cost of a few bytes of more memory.

The lower portion of the *Shared Memory* contains a hash table. For MOLEN, the keys to that hash table are profiled functions' names and values are indexes of those functions. The SET and EXECUTE APIs which are used at runtime and which were discussed in section 2.3, are just hints to the runtime system and it is the responsibility of the runtime system to decides which kernel to map. The SET API takes the name of the kernel to be mapped as the parameter. Now since in our system, the data written by the *Daemon* is contained in an array form, where functions are identified by the indexes

of that array, the *Scheduler* requires this hash table to get the indexes from function names. It has to be noted here that the form of this hash table is dependant on the current scheduler of MOLEN and therefore will be different for different schedulers.

4.2.4 Sampler

For Sampling, we have used the local APIC Timers' interrupts. The reason of using this approach is that each core in an x86 Multi-core processor has its own local APIC Timer and hence through this approach, it is possible to perform sampling profiling for multi-core processors besides single-core processors. At each APIC Timer interrupt, besides the program counter value and process ID of the current process, we also note down the core number on which the interrupt is occurring. These samples are saved in a circular buffer. We use a *spin-lock* to avoid different cores writing to that buffer at the same time. The data in the circular buffer is read by the *Daemon* periodically through a file in */proc* directory. The size of the circular buffer is actually kept a few times larger than the maximum number of samples the *Daemon* can read in a period, so that no overwriting of buffer locations occur within one period. More detail about how these samples are used is discussed in the next section, where we discuss the *Daemon*.

4.2.5 Daemon

The task of the *Daemon* is to collect information from different places and present it to the *Scheduler* in the most efficient manner. The *Daemon* consists of two threads and a *signal handler*. Actually, we have modified the *do_execve* function in the kernel so that it sends a signal to the *Daemon* whenever it starts executing a new program. For this the kernel code has to know the process ID of the *Daemon*. For that purpose, we have used a file in */proc* directory. When the *Daemon* writes its *process ID* to that file, the kernel starts sending signals to the *Daemon* whenever a new program is loaded in memory to execute. The sent signal's information contain the *process ID* of the newly loaded program, which is then used by the *Daemon*.

4.2.5.1 Data Structures

The *Daemon* code is written in C++ in an object oriented fashion. The UML Class diagram of the data types used are shown in figure 4.4. Here *Sample* is a structure which is used in both *kernel* and the *Daemon* to represent a sample. Since our profiler is also able to profile with multi-core general purpose processors, it also contains *core* member variable which holds the core number on which the APIC Timer interrupt occurred when the sample was recorded. Note that the *core* member variable is redundant for single-core general purpose processors. The other data types shown in figure 4.4 are implemented as C++ classes and are described below.

FuncRange and FuncsInfo *FuncRange* class is used to represent range of a function. Its member variables are starting address and ending address of a function and its *function index*. As discussed previously, all the functions that have to be profiled in our system will be given an index, so that those indexes can be used to index function related values in the shared memory. The *functionIndex* member here actually represents that

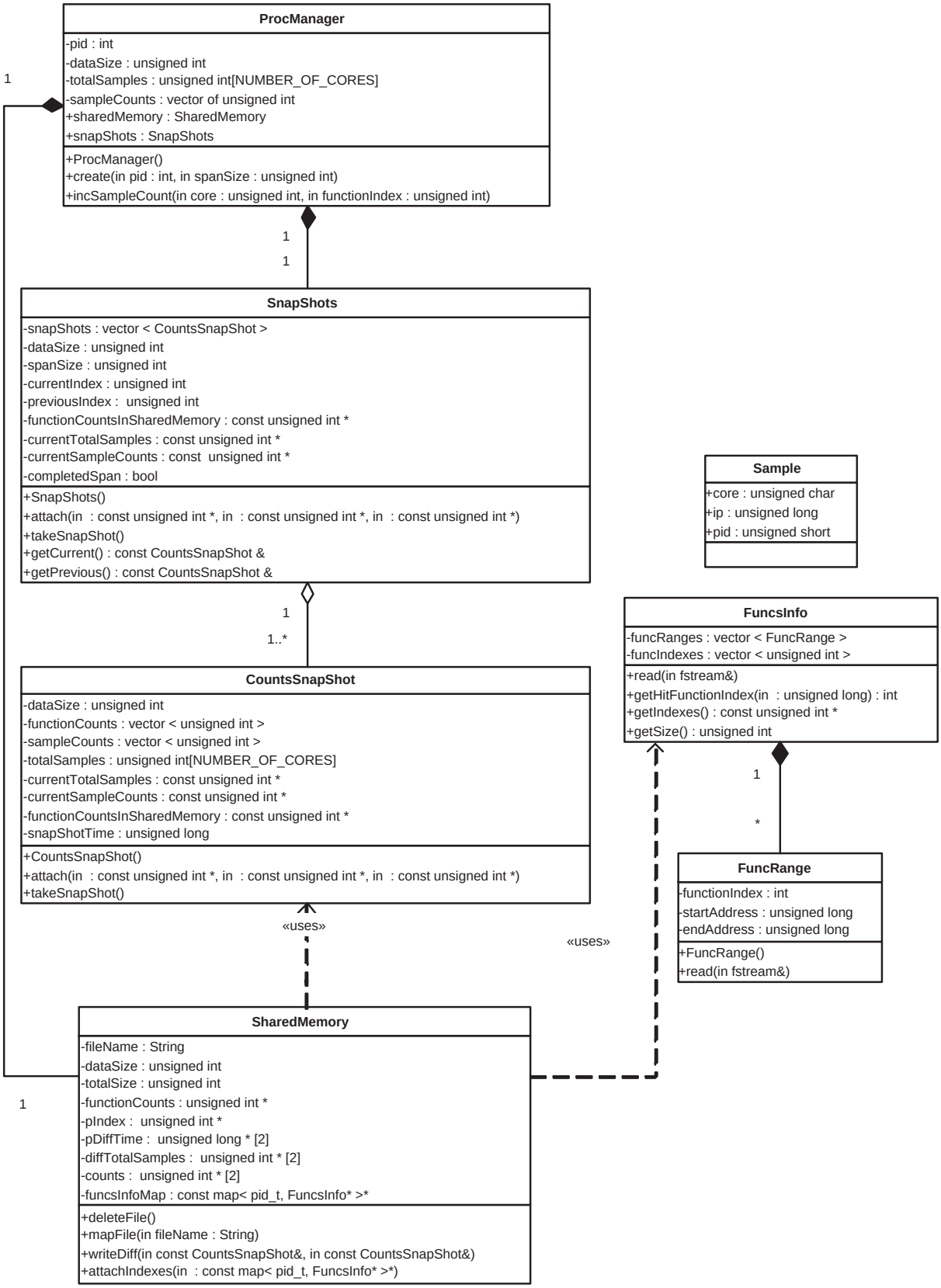

Figure 4.4: Class diagram of Data Types used in *Daemon*

index. Those indexes are also used to index *function samples counts* for the profiled functions. Those *function samples counts* are maintained by the *Daemon*. The *FuncsInfo* class contains a *vector* of *FuncRange* objects, which is used to represent ranges of all to be profiled functions of a process. Besides, it also maintains a vector of indexes for the profiled functions of a process. This vector of indexes is used only for *global profiling* and its purpose will be discussed when we will describe the *SharedMemory* class. The function ranges are read from a file created by the *Extractor*, as was discussed in section 4.2.1. That file contains function ranges in sorted form, so that the *getHitFunctionIndex* method in *FuncsInfo* can locate a function from the program counter value by using binary search. Since to locate a function from a given program counter value, we also need to know the the process ID, our program contain a hash map *FuncsInfoMap* whose values are objects of type *FuncsInfo* and which is keyed by process IDs.

ProcManager Class The *ProcManager* class represents a profiled process and like in case of *FuncsInfo*, we have a hash map *procManagerMap*, whose values are objects of type *ProcManager* and which is keyed by process IDs. Member variable *dataSize* is used to hold the number of profiled functions. The *spanSize* is not directly a member of ProcManager, but is used by the member variable of type *SnapShots* and denotes the *span time*. By *span time*, we actually mean the number of seconds for which profile data has to be maintained. So for example, if *span time* is five seconds, then that means that the *Daemon* will present the *Scheduler* the counts for the last five seconds. The *ProcManager* class also contains a member of type *SnapShots*, which is described in next section. Besides that, it contains a member of *SharedMemory* type which represents the shared memory for a process.

The *totalSamples* and *sampleCounts* member variable are used to represent the samples read from kernel. *totalSamples* is total number of such samples, while *sampleCounts* contains *function sample counts* for the profiled functions. By *function sample counts* for a function, we mean the number of samples that have program counter value pointing to some instruction in that function. The NUMBER_OF_CORES constant is known at compile time, since in the makefile we have used a script that reads the number of cores of the general purpose processor. Such script is also used in the makefiles of the *Injector* and *Extractor*. Therefore all one need to do when porting our profiler to a system which has a general purpose processor with different number of cores is to recompile the *Daemon*, *Extractor* and *Injector* programs.

Note that for *global profiling*, we do not use any object of *ProcManager* type, but use global variables of type *SnapShots* and *SharedMemory* besides global *totalSamples*, *sampleCounts* and *dataSize*.

SnapShots and CountsSnapShot Classes The *CountsSnapShot* class represents snap-shot of *function calls counts* and *function samples counts* at a particular instance. Since, as discussed in the previous section, we want to present to the *Scheduler* the counts for a span of certain time, we have a *SnapShots* class which contains a vector of *CountsSnapShot* objects. The *ProcManager* class uses the member of type *SnapShots* to take the snapshots. This is done at each second in our implementation. The *getCurrent* method returns the reference to the most recently updated element of the *CountsSnapShot* vector, while *getPrevious* return the reference to the one which was updated *spanSize* seconds before. The length of the *CountsSnapShot* vector in *SnapShots* is fixed

and therefore that vector is used as a circular buffer. When a process has been running for less than *spanSize* seconds, *getPrevious* returns the first snap shot as the previous snap shot.

To take the snap-shots, *SnapShots* needs to have access to the *function calls counts* and *function samples counts*. For that purpose, *ProcManager* calls *attach* method of its member of type *SnapShots*. The *attach* function takes three arguments, one is the pointer to the area in shared memory where *function calls counts* are kept, second is the pointer to *totalSamples* in *ProcManager* and third is the address of *sampleCounts* in *ProcManager*. All these pointers are passed as constants, so the information for *SnapShots* is read only. These addresses are further relayed to the elements of *CountsSnapShot* vector in *SnapShots*. *CountsSnapShot* also has a member to store the time at which the snap-shot was taken.

SharedMemory Class The *SharedMemory* class is used to deal with the shared memory files. If *per process profiling* is being performed, than each profiled process will have its own *Shared Memory* file while in case of *global profiling*, only one *Shared Memory* file is used. The *Shared Memory* file is named according to process ID, as was discussed in section 4.2.2. In case of *per process profiling*, the *Shared Memory* file is created by the *Injector* and the *Daemon* only requires to map that file. For mapping the *Shared Memory file*, we have *mapFile* method. The *Shared Memory* file is deleted as soon as the process using it terminates, and to do that there is a *deleteFile* method.

The pointer *functionCounts* points to the start of the *function calls counts* in *Shared Memory*, so that *function calls counts* can directly be read from it by using it like an array. *pIndex* is pointer to the portion of *Shared Memory* that is used to index the data buffers that are used for logging the data read by the *Scheduler*. The value of the index can be either 0 or 1, since there are only two such buffers, as discussed in section 4.2.3. *pDiffTime*, *diffTotalSamples* and *counts* contain pointers to the areas of shared memory containing difference of time, difference of total samples and difference of function calls and samples counts, for a span. The reason all these arrays are of size two is because as discussed previously we have two buffers and therefore each element of the array points to a location in one of those two buffers.

For *global profiling*, we also have a pointer to the *FuncsInfoMap*. The reason *Daemon* uses this hash map for *global profiling* is to allow it to only update information for those functions in the shared memory which are part of the current profiled processes. For this purpose, it uses the vector of indexes in *FuncsInfo*.

4.2.5.2 Code Structure

The *Daemon* consists of two threads and a signal handler. The purpose of the signal handler is to receive the signal which contains process ID of a newly loaded program, as was discussed previously. Besides that signal handler, there are two threads. First one is the main thread and the second one is the data processing thread, whose job is to periodically read samples from the *samples buffer* and *function calls counts* from shared memories and update data for the *Scheduler*. Besides that, it also deletes unused *Instrumentation Code* and *Shared Memory* files. The two threads are described in more detail below.

Main Thread The pseudocode for the *Main* thread is shown as algorithm 1. At the beginning, it reads the configuration file, sets up the signal handler, initializes variables for *global profiling* if required, and creates the *Data Processing* thread. After that it enters an infinite while loop. At the beginning of the while loop, it waits for a sempahore that is posted by the signal handler. The signal handler posts that sempahore whenever it receives the signal from the kernel on loading of a new program for execution. That signal contains the newly loaded program's process ID as its information. The signal handler adds that process ID into a circular queue before posting the semaphore. The reason for using a queue here is to ensure that no signals are lost in case *Daemon* was busy doing some other work and more than one signals were received at that time. After acquiring the semaphore, it extracts all unread Process IDs from the queue and for each of them, it checks whether the process in question has corresponding *profiling information files*. If it does and is not already being run as child process of the *Injector*, the *Daemon* calls the *Injector* utility to create a *Shared Memory* file if required and a copy of the *Instrumentation Code* file, and map them into the address space of that process. The *Shared Memory* file for a process needs to be created only for *per process filing* because for *global profiling*, a global shared memory is used. The *main thread* also creates and initializes objects of type *funcsInfo* and *ProcManager* for a process that has corresponding *profiling information files*. Those objects are added to *funcsInfoMap* and *ProcManagerMap* respectively.

Data Processing Thread The *Data Processing* thread is used to read the samples from the *samples buffer*, read *function calls counts* from *Shared Memory*, process data and write data to the *Shared Memory*, which is read by the *Scheduler*. The pseudocode for *per process profiling* is shown as algorithm 2. Since pseudocode for *global profiling* only differs a little, it is not shown here. For *global profiling*, we do not use the *procManagerMap* Hash Map or any object of type *ProcManager* but instead use global variables of type *SharedMemory*, *SnapShots* and those responsible for maintaining *function samples counts*.

Samples from the kernel are read at intervals of 200 ms, while *function calls counts* from *Shared Memory* are read every second. The reason we are reading the samples at a relatively higher rate is to keep the size of the *samples buffer* small, because a large buffer size can incur a significant overhead in the kernel as it can cause more cache misses. The read data is processed and written to the *Shared Memory* by using objects of type *SnapShots* and *SharedMemory*. The *getCurrent* method of *SnapShots* returns the most recent counts, while the *getPrevious* method return the counts *spanSize* seconds before. The value of *spanSize* is read from the configuration file and represents the span time for the counts, as was discussed previously. Note that since the instruction to update counts in shared memory by the profiled application is done through a single atomic instruction, as was discussed in section 4.2.3, there is no chance of reading a wrong count value from the *Shared Memory*. However, it might happen that when the counts are being read at an instance, a context switch happens in between and therefore the corresponding time for reading all counts at that instance is not the same. But even if this happens, it would only make a very minor difference, as the *Scheduler* reads the difference of counts between several seconds and if the counts are misplaced by a few milliseconds, it would have a very little impact on the accuracy.

Algorithm 1 Pseudocode for Main Thread of the *Daemon*

```
ReadConfigurationFile();
Setup the Signal Handler to handle signals from the kernel whenever a new program
is loaded to execute;
Write the Process ID value of this Daemon to a file in /proc directory, so that the
kernel can read it and start sending signals to the Daemon;
Create the Data Processing thread;
if global profiling needs to be done then
  Create and Map global shared memory as well as initialize other variables for global
  profiling;
end if
while 1 do
  Wait on the semaphore to be signaled by the Signal Handler.
  while unread Process IDs exist in the PID_queue do
    pid ← getNextPID(PID_queue)
    path_name ← extract_path(pid)
    if program path_name has corresponding profiling information files then
      if process with PID pid is not running as a child process of the Injector then
        Use the Injector utility to create the Shared Memory file if required and map
        the Instrumentation Code and Shared Memory files for the process with PID
        pid;
      end if
      funcsInfoMap[pid] ← new FuncsInfo
      funcsInfoMap[pid].read(getFuncsRangesFileName(path_name))
      if per process profiling is being done then
        procManagerMap[pid] ← new ProcManager;
        procManagerMap[pid].create(pid, spanSize);
      end if
    end if
  end while
end while
```

Here it has to be noted that since the *Daemon* is required to access *Shared Memory* files created by any process and also map files to any process, it is run with root privileges. Also the *Daemon* gives read and write access to the files that it creates, so that a process running in a user account can access them. Secondly, in current implementation, we do not profile processes that are forked as child processes of profiled processes, with the exception of those child processes that use *exec* family of functions to load a program, given that the loaded program has corresponding *profiling information files*. In future, we intend to add functionality to profile any kind of child processes. This can be achieved by having the kernel send a signal to the *Daemon* whenever a process forks a child process and having code in the *Daemon* to take appropriate steps.

Algorithm 2 Pseudocode for *Data Processing* thread of the daemon (for *per process profiling*)

```
numberOfSeconds ← 1;
time0 ← getCurrentTime();
while 1 do
  Sleep for 200 milliseconds;
  Read samples from the samples buffer;
  for all s in samples do
    if funcsInfoMap[s.pid] exists then
      fi ← funcsInfoMap[s.pid].getHitFunctionIndex(s.ip);
      procManagerMap[s.pid].incSampleCount(s.cpu, fi);
    end if
  end for
  time1 ← getCurrentTime();
  diffTimeInSecs ← toSecs(time1 − time0);
  if diffTimeInSecs is greater than or equal to numberOfSeconds then
    for all [pid, funcsInfo] in funcsInfoMap do
      if process with process ID pid no more exists then
        Remove the Instrumentation Code and Shared Memory files for that process;
        Remove funcsInfoMap[pid];
        Remove procManagerMap[pid];
      else
        procMan = procManagerMap[pid];
        procMan.snapShots.takeSnapShot();
        curSS ← procMan.snapShots.getCurrent();
        prevSS ← procMan.snapShots.getPrevious();
        procMan.sharedMemory.writeDiff(curSS, prevSS);
        numberOfSeconds ← numberOfSeconds + 1;
      end if
    end for
  end if
end while
```

4.3 Conclusion

In this chapter, we discussed the overall design and implementation of our profiler. We saw how we used the *Shared Memory* to reduce the overhead of keeping function calls counts in the memory. Furthermore, we discussed the use of *Extractor* and *Injector* utilities that we created. Through the *Extractor* utility we can generate necessary information offline, so that it can be used by the *Injector* to inject necessary files into the address space of a process that has to be profiled. Extracting necessary information offline in this way, greatly reduces code injection time as the program does not need to parse executable files at load time. At the end, we discussed the design of the *Daemon*, which is responsible for collecting data from different places, process it and present it to

the *Scheduler*. The data produced by the *Daemon* is also placed in the *Shared Memory*, so that the *Scheduler* can have a quick access to it.

5 Empirical Evaluation

To test the performance of our profiler, we have used benchmarks from different areas. One of this is PC version of *tcf*, which is a Stationary Noise Filter used in hArtes [4] demonstration. Then we have *minisat2* [13], which is an industrial scale SAT solver. Note that we have removed the randomness part in *minisat2*, so that our results do not vary from run to run. Then we have *H264/AVC encoder*, from MediaBench II benchmarks [5], which is an H264 encoder application. Next, we have *coremark* [6], which is a free synthetic benchmark from *EEMBC* [7]. Finally, we have a benchmark that we created ourselves, known as *multiply*. That program repeatedly calls a multiply function which just returns product of two numbers. Normally, such tiny functions are inlined by the compiler and therefore profiling would not be required for them. However, testing the profiler with such tiny functions gives us an idea of the worst case performance of the Instrumentation Profiling. To avoid inlining of the multiply function, the *multiply* application is compiled with optimizations turned off.

We used a single-core x86 processor to run all of these benchmarks. Moreover, for all timing purposes, we used the *times* function in Linux.

In section 5.1, we present the overhead of the Instrumentation part of our profiler and compare it with that of *Pin*. In section 5.2, we discuss the time it takes to inject instrumentation code into profiled applications. In section 5.3, we discuss the overhead imposed by sampling and the *Daemon*, and compare it with that of *OProfile*, while in section 5.4, we compare the accuracy of our sampler with that of *gprof*. In section 5.5, we discuss the overall overhead of our profiler. Finally, in section 5.6, we discuss the amount of functions which are profilable. This is because some functions, usually of very small size, do not have enough prologue instructions to replace with a jump instruction.

5.1 Instrumentation Overhead

Table 5.1 shows the comparison of instrumentation overhead of our profiler, for both *global profiling* and *per process profiling*, with that of *Pin*. For this purpose, we made a *Pin* tool (See its code in appendix A) that just counts the number of functions called, so that it is equivalent to our profiler. From the table we can see that overhead with *Pin* is always above 20%, while for our profiler, except for the *multiply* application, the overhead is always less than 1.5%. The low overhead for applications other than *multiply* was expected because our profiler only adds three instructions to original functions for profiling. The overhead for the *multiply* application here is relatively large because it repeatedly calls a very tiny function that just returns product of two numbers.

The readings given in table 5.1 were calculated by running each of the benchmark ten times and then taking the mean values.

Table 5.1: Instrumentation Overhead

	Normal	**Pin**	**Per Proc Profiling**	**Global Profiling**
multiply				
Time (secs)	9.681	11.646	10.105	10.092
Overhead	-	20.30%	4.38%	4.25%
coremark				
Time (secs)	12.496	15.550	12.664	12.654
Overhead	-	24.44%	1.34%	1.26%
tcf				
Time (secs)	7.083	8.758	7.107	7.089
Overhead (%)	-	23.65%	$< 1\%$	$< 1\%$
h264enc				
Time (secs)	40.774	125.566	41.037	40.86
Overhead	-	207.96%	$< 1\%$	$< 1\%$
minisat2				
Time (secs)	29.004	37.064	29.142	28.95
Overhead	-	27.79%	$< 1\%$	$< 1\%$

5.2 Instrumentation Code Injection Time

In table 5.2, we have listed the injection time for different applications. The injection time varies slightly from run to run, that is why we took means of 10 runs. The *Map Time* is the time to map the *instrumentation code* and *shared memory* files into the address space of the program. The *Data References Insertion Time*(DRIT) is the time taken to insert the resolved addresses into the instrumentation code and the *Prologues Modification Time*(PMT) is the time taken to replace the prologues of the functions with jump to the instrumentation code.

From table 5.2, we can see that *Data References Insertion Time*(DRIT) and *Prologue Modification Time*(PMT) are less than 10 milliseconds for all benchmarks other than *h264enc*. This is because *h264enc* contain larger number of profilable functions and due to that more time is taken to insert data references and modify function prologues. The *Map Time* does not vary much from application to application. This was expected, since it is just the time to map *Instrumentation Code* and *Shared Memory* files into profiled process's address space and therefore should not be application dependant. Nevertheless, from table 5.2, we can see that the overall Instrumentation Code Injection Time is quite low even for programs with large number of profilable functions.

5.3 Sampling and Daemon Overhead

In table 5.3, we have shown the results achieved without performing instrumentation. The purpose of not performing instrumentation in this case is to quantiy the overhead

Table 5.2: Instrumentation Code Injection Time

Program	**Prog Size (KB)**	**Functions Profiled**	**Map Time (ms)**	**DRIT (ms)**	**PMT (ms)**
multiply	9.4	5	40	< 10	< 10
coremark	19	29	30	< 10	< 10
minisat2	173.6	52	30	< 10	< 10
tcf	91.8	59	40	< 10	< 10
h264enc	663.7	560	60	< 10	10

Table 5.3: Sampling and Daemon Overhead

	Normal	**OProfile**	**Per Proc Profiling**	**Global Profiling**
multiply				
Time (secs)	9.681	9.677	9.695	9.688
Overhead	-	$< 1\%$	$< 1\%$	$< 1\%$
coremark				
Time (secs)	12.496	12.510	12.497	12.495
Overhead	-	$< 1\%$	$< 1\%$	$< 1\%$
tcf				
Time (secs)	7.083	7.121	7.124	7.126
Overhead (%)	-	$< 1\%$	$< 1\%$	$< 1\%$
h264enc				
Time (secs)	40.774	40.838	41.051	41.034
Overhead	-	$< 1\%$	$< 1\%$	$< 1\%$
minisat2				
Time (secs)	29.004	29.037	29.075	29.058
Overhead	-	$< 1\%$	$< 1\%$	$< 1\%$

imposed by sampling and the *Daemon*. The results are compared with those achieved from *OProfile*. The readings given in table 5.3 are means of 10 readings. From that table, we can see that the overhead for both *OProfile* and our profiler is negligible. It has to be noted here that we only used the *timer interrupt event* for *OProfile*, so as to make it functionally equal to our profiler.

5.4 Sampling Accuracy

In this part, we checked the accuracy of the sampling part of our profiler by comparing it with *gprof*. Here we used only those benchmark applications that take more than 10 seconds to execute. We ran each program five times, both with our profiler and with

Table 5.4: Sampling Accuracy of our profiler

Function	**gprof (%)**	**Our Profiler (%)**	**gprof Std**	**Our Profiler Std**
coremark				
crcu8	31.77	31.07	1.55	0.78
core_state_transition	30.05	31.56	1.39	0.69
core_bench_list	13.85	14.21	1.70	0.57
matrix_mul_matrix_bitextract	5.48	5.69	0.47	0.60
minisat2				
Solver::propagate	74.87	75.02	0.89	0.52
Solver::analyze	13.56	13.42	0.51	0.42
Solver::litRedundant	4.45	4.61	0.32	0.15
Solver::cancelUntil	2.85	2.84	0.32	0.20
h264enc				
SetupFastFullPelSearch	33.588	36.108	2.62	1.31
dct_luma	11.168	10.691	0.43	0.60
biari_encode_symbol	7.326	7.294	0.67	0.48
SetupLargerBlocks	3.828	3.820	0.58	0.15

gprof, so that we could get the mean values and standard deviations. For our profiler, we used *per process profiling*. The results are given in table 5.4. In that table, means of percentages of total time spent and standard deviations for the functions which took the most time according to *gprof* are given. The functions are sorted by percentages of total time spent, given by *gprof*. From the table, it can be seen that the mean values for both our profiler and *gprof* are almost the same, while the standard deviations for our profiler are better than that of *gprof*. The reason for this is that *gprof* by default samples at the rate of 100 samples per second, while in our case the maximum sampling rate is 250 samples per second. Actually we take our samples through the local APIC timer interrupts. For a tickless kernel, which we are using, the frequency of those interrupts vary with workload. The maximum frequency of those interrupts in our case is 250 samples per second.

5.5 Overall Overhead

In the first part of this experiment, we tested our benchmark applications with all parts of the profiler working. The means of ten readings are shown in table 5.5. The results are as expected, that is all applications other than *the multiply* application have overhead of less than 1.5%. Moreover, overall overhead for all application is almost the same as that for instrumentation overhead, thus reinforcing the fact that sampling and the *daemon* have very low overheads.

In the second part of the experiment, we ran all the benchmark applications simulta-

Table 5.5: Overall Overhead of our Profiler for single applications

	Normal	**Per Proc Profiling**	**Global Profiling**
multiply			
Time (secs)	9.681	10.141	10.112
Overhead	-	4.75%	4.45%
coremark			
Time (secs)	12.496	12.653	12.655
Overhead	-	1.26%	1.27%
tcf			
Time (secs)	7.083	7.085	7.088
Overhead	-	$< 1\%$	$< 1\%$
h264enc			
Time (secs)	40.774	41.112	41.158
Overhead	-	$< 1\%$	$< 1\%$
minisat2			
Time (secs)	29.099	29.004	29.040
Overhead	-	$< 1\%$	$< 1\%$

Table 5.6: Overhead of our Profiler with all of the five benchmark applications running

	Normal	**Per Proc Profiling**	**Global Profiling**
Time (secs)	92.32	92.89	93.04
Overhead	-	$< 1\%$	$< 1\%$

neously with all parts of the profiler of working. We repeated the experiment five times for both *per process profiling* and *global profiling* and took the mean values which are shown in table 5.6. The results show that the profiling overhead was less than 1% for both *per process profiling* and *global profiling*. To run all the benchmark applications simultaneously and measure the overall time, we created a special application. The code for that application is given in appendix B.

5.6 Percentage of Profilable functions

Our profiler replaces prologues of to be profiled functions with a jump instruction. For that purpose, a function's prologue must be at least 5 bytes because the jump instruction consumes 5 bytes. Most of the functions do have at least 5 bytes of prologue, but some functions, usually of very small size, do not. By prologue instructions, we mean instructions which prepare the stack and registers for use within a function and not any

Table 5.7: Percentage of Profilable functions

Program	**Opt Level**	**Total Functions**	**Profilable Functions**
tcf	-O0	59	59 (100%)
h264enc	-O2	591	560 (94.8%)
minisat2	-O3	56	52 (92.9%)
coremark	-O2	40	29 (72.5%)

instruction that is used afterwards. In table 5.7, we have shown the number of function which are profilable for different applications. We have also listed the optimization levels used to compile those applications, the purpose of which is to see if optimizations make it any harder to find profilable functions. Except for *coremark*, all applications have more than 90% of profilable functions and both *h264enc* and *minisat2* are using high level of optimizations. The reason *coremark* has only 72.5% of profilable functions is because there are many functions of very small sizes in it.

It should be noted that for comparing Instrumentation overhead with *Pin*, as discussed in section 5.1, to make the comparison with *Pin* fair, we made the *Pin* tool to only instrument those functions which our profiler was also instrumenting.

5.7 Conclusion

In this chapter, we showed the overhead of our profiler from different aspects. This is done by showing the overhead on Instrumentation, Code Injection, Sampling, Daemon and the overall overhead. Besides, we compared the accuracy of our profiler with a popular design time profiler. All the presented results shows that our profiler is very low overhead (less than 1.5%) and very accurate.

6 Conclusion and Future Work

In this chapter, we give the conclusion and discuss further work that needs to be done in future. The conclusion is given in section 6.1, while future work is given in section 6.2.

6.1 Conclusion

In this thesis, we described the design and implementation of a runtime profiler which can be used as part of the runtime environment of MOLEN and similar systems. Use of multitasking operating systems on GPP of reconfigurable systems has necessitated the existence of a runtime system which can properly allocate the reconfigurable hardware to competing threads in the system. For that purpose, the runtime system needs to know the kernels which would bring up the most speed gains if transformed into hardware. For that purpose, it needs to know which kernel is being executed more often and taking more time to execute. Such information can be provided by a runtime profiler. In figure 2.3, we showed the interaction between the *Profiler* and the *Scheduler*. The *Profiler* maintains logs for the *Scheduler* to read. Since a runtime profiler should have minimum overhead possible, the mechanism to read and write to those logs should be as fast as possible. We achieved that low overhead by maintaining the logs in shared memories.

Our profiler is a combination of a Sampling profiler and an Instrumentation profiler. For sampling, we used the technique that is used by almost every other sampling profiler to estimate the time spent by different parts of the code. That technique is to take samples at the occurrence of each timer interrupt. In each such interrupt, we recorded the Program Counter, Process ID of interrupted process and core number at which the interrupt occurred. The samples are put in a circular buffer, which we call the *samples buffer*. Since we tested our profiler on a system with x86 processor, we used the local APIC timers because they are available for each core of the processor. We used a maximum sampling frequency of 250 second, while the *GProf* profiler by default only uses a frequency of 100 samples per second. Due to that reason, our sampling profiling is more precise and almost as accurate as that of *GProf*. The precision and accuracy of the profiler was checked by running benchmark applications multiple times, calculating mean percentage values of time spent by each function in them as given by the profiler, and calculating the standard deviations. The results were compared with those obtained from GProf. See section 5.4 for more details.

Our profiler does Instrumentation Profiling to count the times each function is called. For that purpose, we used a technique similar to the one used by profilers such as *Detours* and *IgProf*, which is to replace the prologue of a function with an unconditional jump to an instrumentation code which updates the count for that function, executes the replaced prologue and jumps back to the remaining part of the instrumented function. To impose minimum overhead, we kept the counts in a shared memory. Experimental results

show that we indeed achieved very low overhead with this technique (see table 5.1). Furthermore, we created an *Extractor* tool to generate instrumentation code file for a program by reading its ELF file. Besides that, the *Extractor* also creates other files to help during injection and modification of code in the profiled application. Since the *Extractor* utility is used offline, the injection of code into a running application and modification of function prologues in it, takes very little time (see table 5.2).

A daemon is used to read the function counts from the shared memories, read the samples from the *samples buffer* and put processed information in the logs for the *Scheduler*. The logs are also kept in the shared memories, so that reading from and writing to them is as fast as possible. To further minimize the overhead, the *Daemon* only reads the samples from the *samples buffer* at intervals of 200 ms and the function counts after each second. The logs are also updated after each second. The empirical evaluation that we did show that the overhead for Sampling profiling and the *Daemon* is negligible. See section 5.3 for more details.

We also discussed the overall overhead of our profiler in section 5.5 and the results show that for most applications, the overhead of our profiler is less than 1.5%.

6.2 Future Work

In future, our profiler will be integrated with the *Scheduler*. Secondly, memory profiling capabilities would be added to it. Moreover, it will be extended to be able profile at finer granularity than just at function level.

6.2.1 Testing With Different Processors

For this thesis, we tested the profiler on a system with x86 processor. In future, we will also test it with other processors, especially those used in embedded and reconfigurable systems, such as ARM and PowerPC. We can expect the overhead to be different for those processors than what we achieved with the x86 processor.

6.2.2 Integration With the Scheduler

In future, our profiler will be integrated with the *Scheduler* to perform its required task. Since our profiler will be required to instrument kernel calls both in case when a kernel is executed in software and when it is implemented as hardware, a *Transformer*, which was shown in figure 2.3, will be required. Just when a kernel would be mapped into hardware, the *Transformer* would modify the software implementation of the kernel in such a way that it only retains the instrumentation part and bypasses other parts. Moreover, when a kernel is removed from the hardware, the *Transformer* will restore the original function, so that the kernel can again start properly executing in software.

Since our profiler is totally software based, it is easy to port it to any runtime system for reconfigurable platforms. Therefore, our profiler can also be integrated into systems like ReconOS [23], besides MOLEN.

6.2.3 Memory Profiling

Our profiler needs to have memory profiling capabilities to be completely useful for MOLEN. As shown in figure 2.2, the reconfigurable hardware has an internal memory. If the kernel which is mapped into the reconfigurable hardware only uses the internal memory, its execution would be much faster than in the case where it also uses the external memory. Normally, functions which only use local variables could easily be mapped into hardware so that they only use the internal memory of the reconfigurable hardware, given the local data is not big enough to not fit into that memory. Therefore for such functions, our profiler can be considered complete, but it cannot be considered complete for any kernel that would use any external memory. In that case, while the kernels would be executing in software, the profiler should be able to give a guess to the *Scheduler* about how much external memory those kernels would use when transformed into hardware. Secondly, memory profiling should also be able to determine data dependencies in the code. This is important because those kernels which have a lot of data dependencies might not be efficiently parallelized.

6.2.4 Further Enhancements

In current implementation, our profiler only profiles at function level, but in future it would be extended so that it is able to profile at finer granularity, such as loops inside functions. Moreover, our profiler only records counts of function calls and average time spent by functions, but it does not give the context of function calls. For example, if *functionA* is frequently called inside *functionB* but is seldom called inside *functionC*, our profiler would not be able to provide this information. Such information might be very useful for the *Scheduler*, as it would be much preferable to map *functionA* into hardware while it is being called from *functionB* then if it is being called from *functionC*.

Bibliography

[1] `http://jikes.sourceforge.net/`, 2005.

[2] `http://perfmon2.sourceforge.net/`, 2009.

[3] `http://oprofile.sourceforge.net/`, 2010.

[4] `http://www.hartes.org/index.php?option=com_content&task=view&id=62&Itemid=87`, 2010.

[5] `http://euler.slu.edu/~fritts/mediabench/mb2/`, 2010.

[6] `http://www.coremark.org/`, 2010.

[7] `http://www.eembc.org/`, 2010.

[8] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Monkia R. Henzinger, S.A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, William E. Weihl, L. M. Berc, Sanjay Ghemawat, M. R. Henzinger, and S.A. Leung, *Continuous profiling: Where have all the cycles gone?*, ACM Transactions on Computer Systems, 1997, pp. 1–14.

[9] Matthew Arnold and Barbara G. Ryder, *A framework for reducing the cost of instrumented code*, ACM SIGPLAN Notices, Vol 36, No. 5 (2001), 168–179.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, *Dynamo: A transparent dynamic optimization system*, ACM SIGPLAN Notices, 2000, pp. 1–12.

[11] Bryan Buck and Jeffrey K. Hollingsworth, *An api for runtime code patching*, Int. J. High Perform. Comput. Appl. **14** (2000), no. 4, 317–329.

[12] Intel Corp., *Intel itanium 2 processor reference manual for software development and optimization. intel corporation*, June, 2002.

[13] N. Een and N. Sorensson, *An extensible sat-solver*, Theory and Applications of Satisfiability Testing, 2004, pp. 333–336.

[14] G. Eulisse and L. A. Tuura, *Igprof profiling tool*, Computing in High Energy Physics and Nuclear Physics (2004), 665.

[15] Alan Eustace and Amitabh Srivastava, *Atom: a flexible interface for building high performance program analysis tools*, Proceedings of the USENIX 1995 Technical Conference (Berkeley, CA, USA), USENIX Association, 1995, pp. 25–25.

[16] Fabrizio Ferrandi, Luca Fossati, Marco Lattuada, Gianluca Palermo, Donatella Sciuto, and Antonino Tumeo, *Automatic parallelization of sequential specifications for symmetric mpsocs*, IESS, 2007, pp. 179–192.

[17] Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, and Antonino Tumeo, *Performance estimation for task graphs combining sequential path profiling and control dependence regions*, MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign (Piscataway, NJ, USA), IEEE Press, 2009, pp. 131–140.

[18] Wenyin Fu and Katherine Compton, *An execution environment for reconfigurable computing*, FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 2005, pp. 149–158.

[19] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, *gprof: a call graph execution profiler*, ACM SIGPLAN Notices, Vol 39, No. 4 (2004), 49–57.

[20] Galen Hunt, , Galen Hunt, and Doug Brubacher, *Detours: Binary interception of win32 functions*, In Proceedings of the 3rd USENIX Windows NT Symposium, 1998, pp. 135–143.

[21] John H. Kelm and Steven S. Lumetta, *Hybridos: runtime support for reconfigurable accelerators*, FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays (New York, NY, USA), ACM, 2008, pp. 212–221.

[22] J. Lu, H. Chen, P. Yew, and W. Hsu, *Design and implementation of a lightweight dynamic optimization system*, Journal of Instruction-Level Parallelism, Vol 6 (2004), 2004.

[23] Enno Lübbers and Marco Platzner, *Reconos: Multithreaded programming for reconfigurable computers*, ACM Trans. Embed. Comput. Syst. **9** (2009), no. 1, 1–33.

[24] ChiKeung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), ACM, 2005, pp. 190–200.

[25] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, *A high-level compilation toolchain for heterogeneous systems*, In Proceedings of IEEE International SoC Conference (SOCC), September 2009.

[26] Ajay Nair and Roman Lysecky, *Non-intrusive dynamic application profiler for detailed loop execution characterization*, Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems (New York, NY, USA), ACM, 2008, pp. 23–30.

[27] A.G. Ross and F. Vahid, *Frequent loop detection using efficient nonintrusive on-chip hardware*, IEEE Transactions on Computers, Vol 54, No. 10 (2005), 1203–1215.

[28] M. Sabeghi and K.L.M. Bertels, *Toward a runtime system for reconfigurable computers: A virtualization approach*, Design, Automation and Test in Europe (DATE09), April 2009.

[29] ———, *Interfacing operating systems and polymorphic computing platforms based on the molen programming paradigm*, Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture, June 2010.

[30] M. Sabeghi, V.M. Sima, and K.L.M. Bertels, *Compiler assisted runtime task scheduling on a reconfigurable computer*, 19th International Conference on Field Programmable Logic and Applications (FPL09), August 2009.

[31] A. Srivastava and D.W. Wall, *A practical system for intermodule code optimization at link-time*.

[32] G. Stitt, R. Lysecky, and F. Vahid, *Dynamic hardware/software partitioning: a first approach*, Proceedings of the 40th annual Design Automation Conference (New York, NY, USA), ACM, 2003, pp. 250–255.

[33] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, *Design and evaluation of dynamic optimizations for a java just-in-time compiler*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol 27, No. 4 (2005), 732–785.

[34] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte, *The molen polymorphic processor*, IEEE Transactions on Computers, Vol 53, No. 11 (2004), 1363–1375.

[35] Y. D. Yankova, K. Bertels G. Kuzmanov, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, *Dwarv: Delftworkbench automated reconfigurable vhdl generator*, In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07), 2007, pp. 697–701.

Pin Tool for Instrumenting Function Calls Counts

A

Listing A.1 shows the pin tool that we created to compare the Instrumentation Profiling performance of our profiler with that of Pin. The tool is equivalent to the Instrumentation Profiling part of our profiler because it keeps counts of functions calls. Moreover, to make the tool only profile the function which our profiler profiles, at the beginning, it reads a file that is produced by our *Extractor* utility and which contains the names of the functions to be profiled. This is done by calling function *add_functions* at line 131. This function add names of all functions that have to be profiled in a hash table. Then at execution of each function for the first time, the tool checks if the function has to be profiled by looking for it in the hash table. If the function is found in the hash table, profiling of it is enabled. Note that the file to read functions to be profiled is read before the program is executed by the tool and therefore does not add add time to execution, since in each of our benchmark programs, we have put the clocks at start and end of the code inside the benchmark programs. Moreover, the tool checks if a function has to be profiled only when a function is called for the first time. For example, if a function gets called 10,000 during program execution, the tool will look at the hash table only once and will not look at it the other 9,999 times. So there is negligible overhead of doing this.

Listing A.1: Code of Pin Tool for Instrumenting Function Calls Counts

```
1  //
2  // This tool counts the number of times a routine is executed
3
4  #include <fstream>
5  #include <iomanip>
6  #include <iostream>
7  #include <string.h>
8  #include "pin.H"
9  #include <map>
10
11 //#define MULTI_THREADED
12
13 #ifdef MULTI_THREADED
14 PIN_LOCK lock;
15 #endif
16
17 map<string, int> funMap;
18
19 // Holds Procedure's call count
20 typedef struct RtnCount
21 {
22     string _name;
23     RTN _rtn;
24     UINT64 _rtnCount;
25     struct RtnCount * _next;
26 } RTN_COUNT;
27
28 // Linked list of instruction counts for each routine
29 RTN_COUNT * RtnList = 0;
```

```
30
31 // This function is called before every procedure is executed
32 VOID docount(UINT64 * counter)
33 {
34 #ifdef MULTI_THREADED
35     GetLock(&lock, 1);
36     (*counter)++;
37     ReleaseLock(&lock);
38 #else
39     (*counter)++;
40 #endif
41 }
42
43 const char * StripPath(const char * path)
44 {
45     const char * file = strrchr(path,'/');
46     if (file)
47         return file+1;
48     else
49         return path;
50 }
51
52 // Pin calls this function every time a new rtn is executed
53 VOID Routine(RTN rtn, VOID *v)
54 {
55
56     // Allocate a counter for this routine
57     RTN_COUNT * rc = new RTN_COUNT;
58
59     // The RTN goes away when the image is unloaded, so save it now
60     // because we need it in the fini
61     rc->_name = RTN_Name(rtn);
62     rc->_rtnCount = 0;
63
64     if( funMap.find(rc->_name) != funMap.end() )
65     {
66             // Add to list of routines
67             rc->_next = RtnList;
68             RtnList = rc;
69
70             RTN_Open(rtn);
71
72             // Insert a call at the entry point of a routine to increment the call count
73             RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)docount,
74             IARG_PTR, &(rc->_rtnCount), IARG_END);
75
76             RTN_Close(rtn);
77     }
78 }
79
80 // This function is called when the application exits
81 // It prints the name and count for each procedure
82 VOID Fini(INT32 code, VOID *v)
83 {
84     ofstream count("proccount.out");
85
86     count << setw(5) << left << "PNum" << " "
87           << setw(65) << " Procedure"
88           << setw(12) << " Calls" << endl;
89
90     for (RTN_COUNT * rc = RtnList; rc; rc = rc->_next)
91     {
92           count << left << setw(5) << funMap[rc->_name] << " "
93                 << setw(65) << rc->_name << " "
94                 << setw(12) << rc->_rtnCount << endl;
95     }
96 }
97
```

```
98  /* ================================================================== */
99  /* Print Help Message                                                 */
100 /* ================================================================== */
101 INT32 Usage()
102 {
103     cerr << "This Pintool counts the number of times a routine is executed\n";
104     cerr << "and the number of instructions executed in a routine\n";
105     cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
106     return -1;
107 }
108
109 void add_functions()
110 {
111         ifstream fin( "funcs.txt" );
112
113         while ( fin )
114         {
115                 int fn;
116                 string str;
117
118                 fin >> fn;
119                 fin >> str;
120                 if ( fin )
121                         funMap.insert( pair<string, int>(str, fn) );
122         }
123 }
124
125 /* ================================================================== */
126 /* Main                                                               */
127 /* ================================================================== */
128
129 int main(int argc, char * argv[])
130 {
131     add_functions();
132
133 #ifdef MULTLTHREADED
134     // Initialize the pin lock
135     InitLock(&lock);
136 #endif
137
138     // Initialize symbol table code, needed for rtn instrumentation
139     PIN_InitSymbols();
140
141     // Initialize pin
142     if (PIN_Init(argc, argv)) return Usage();
143
144     // Register Routine to be called to instrument rtn
145     RTN_AddInstrumentFunction(Routine, 0);
146
147     // Register Fini to be called when the application exits
148     PIN_AddFiniFunction(Fini, 0);
149
150     // Start the program, never returns
151     PIN_StartProgram();
152
153     return 0;
154 }
```

Program to Run the benchmarks concurrently

B

Listing B.1: Code of Program to run all the benchmarks concurrently

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <sys/times.h>
#include <pthread.h>
#include <unistd.h>

#define TICK_FREQ 100

void start_clock(void);
void end_clock(char *msg);

static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;

pid_t pids[5];

void start_clock()
{
    st_time = times(&st_cpu);
}

void end_clock(char *msg)
{
    en_time = times(&en_cpu);

    fputs(msg,stdout);
    printf("%g\n", (double)(en_time - st_time) / TICK_FREQ );
}

int main()
{
        int status, i, n = 5, ret;
        pid_t pid;

        start_clock();
        /* Start children. */
        for (i = 0; i < n; ++i)
        {
                if ((pids[i] = fork()) < 0)
                {
                        perror("fork");
                        abort();
                }
                else if (pids[i] == 0)
                {
                        switch( i )
                        {
                                case 0:
                                        ret = execl ("./minisat", "minisat", "bm.cnf", (char *)0);
                                        break;
                                case 1:
                                        ret = execl ("./coremark.exe", "coremark.exe", (char *)0);
```

```
                                        break;
                                case 2:
                                        ret = execl ("./obj_app", "obj_app", "tester.snd", "out", (char *)0);
                                        break;
                                case 3:
                                        ret = execl ("./lencod.exe", "lencod.exe", (char *)0);
                                        break;
                                case 4:
                                        ret = execl ("./mult", "mult", (char *)0);
                                        break;
                        }
                        exit(0);
                }
        }

        while (n > 0)
        {
                pid = wait(&status);
                printf("Child with PID %ld exited with status 0x%x.\n", (long)pid, status);
                --n;  // TODO(pts): Remove pid from the pids array.
        }

        end_clock("Total Time for all applications:");
}
```

Paper Submitted to HiPEAC 2011

C

A Runtime Profiling Tool for Multicore, Polymorphic Platforms

BLIND

ABSTRACT

Runtime multitasking support on Reconfigurable Computers requires complicated resource management techniques in which the FPGA area has to be shared between multiple concurrent tasks dynamically. Such a resource allocation mechanism needs to know the current configuration and load of the system in order to decide about the allocation of the resources. A runtime profiler is an important tool which can give vital information about the running applications on the system. In this paper, we present the design and implementation of a runtime profiler which is responsible to produce statistics about the code running on the system. We have performed a set of experiments in order to show the overhead of our proposed profiler. The evaluation results show that the overhead imposed by the profiler is less than 1.5% of the total execution time and the information generated by the profiler is almost as accurate as a design time profiler such as *gprof*.

1. INTRODUCTION

Limited commercial success of Reconfigurable systems has been due to difficulty in programming them. Usually separate tools are used for writing software and designing hardware. Compilers which can partition software and hardware are rare and cumbersome to use. Also, synthesis tools are required to program the reconfigurable logic. Therefore the overall design and implementation cycle is difficult and demands care from the designers and implementors of the system.

Furthermore, when moving towards multi applications, multi tasking scenarios, it is even more difficult for the designers to deal with such systems as the exact configuration of the system is not known at design time. These requirements necessitate the existence of a runtime system which is responsible for operating the system and performs the resource management. The resource management by itself is a very complicated task and is dependent on the available information for decision making [23].

One important tool required to achieve this is a runtime profiler which can give vital statistics about code running on the reconfigurable computer. Those statistics can then be used by the runtime system to decide which parts of the code need to be translated into hardware.

In this paper, we present a runtime profiler, which is intended to be running concurrently with the applications in the actual application execution time. Therefore, one key difference between such a profiler and traditional design time profilers such as *gprof* is that it has to be very low overhead. Furthermore, the information collected by such a runtime profiler needs to be stored in special data structures in such a way that storing and retrieving them can be performed very fast. The major contribution of this paper is therefore proposing a very light weighted runtime profiler with a well defined interface which enables it to be integrated in any runtime system.

The rest of the paper is organized as follows. In Section 2, we give a brief overview over the background and related work. Section 3 presents the design and implementation of the proposed runtime profiler followed by the section 4, which shows the results of the empirical evaluation. The paper is wrapped up by section 5 in which conclusion of the paper is given.

2. BACKGROUND AND RELATED WORK

In our background study, we first discuss the previous work on runtime environments and how they incorporated runtime profilers. Then, we focus on the profilers and discuss different types of profilers. Furthermore, we give an overview of the profilers for Embedded and Reconfigurable systems. We also discuss the merits and demerits of those different kind of profilers and discuss to what extent they meet our needs.

2.1 Runtime Environments

In this section, we discuss runtime environment of MOLEN [27] and similar systems which use an online profiler. The block diagram of MOLEN's runtime system [21] is shown in figure 1. The *scheduler* reads statistics from the profiler about hotness of profiled kernels and decides how to allocate the hardware. The *scheduler* use the services of the *Transformer* to replace the software implementation of a kernel to its hardware implementation. The *Kernel Library* contains precompiled set of tasks, saved in form of metadata and containing information such as execution time, power

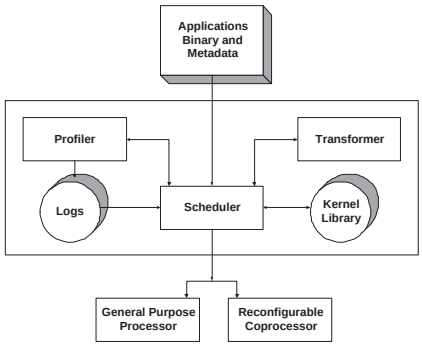

Figure 1: MOLEN's Runtime Environment

consumption and configuration latency among others.

Like MOLEN, Warp processing [25] also transform computing intensive kernels running on a general purpose processor to hardware. The Warp profiler is implemented as hardware and is used to detect kernels at runtime. Within the Warp Processors, the Dynamic CAD tool is used to map kernels to hardware, while the binary updater is used to change the binary of the program running on the general purpose processors, so that it uses the FPGA. The difference between Warp Processors and MOLEN is that while MOLEN can only map those kernels to FPGA for which the hardware implementation is available, Warp Processors can generate code for the FPGA on the fly by using a Just in Time (JIT) compiler. However, this JIT compiler imposes a lot of restrictions on the program code such as the loops, etc.

Authors in [9] discuss techniques to dynamically optimize performance of Multicore processors by using hardware performance counters. The similarity with our approach lies in the fact that a monitor or profiler is used to give feedback to the operating system for dynamic optimization. The monitor is used primarily to reduce the communication and contention among threads in the system. The system records events like cache misses and TLB misses to make decisions. Modern processors also have a *data address register* which is used for sampling data addresses. This register is updated by the performance monitoring unit of the processor on an event. The system uses information from this register to figure out the extent of sharing of data between the cores and reduce cache contention, cache pollution and improve locality.

2.2 Profiling Techniques

In this section we give an overview of different types of profilers. Besides, we review the existing profilers for Embedded and Reconfigurable systems. Based on the methods of gathering information, profilers can be categorized into two major groups, Instrumentation-based and Sampling-based. These two types of profilers are discussed below.

2.2.1 Instrumentation-based Profilers

Instrumentation-based Profilers work by inserting instrumentation code into the application to be profiled. The instrumentation code can be injected either at compile time, link time or run time. The advantage of such profilers is that they are very accurate and more portable than other kinds of profilers, but on the downside they have relatively larger overhead.

An example of an Instrumentation-based Profiler that modifies code at link time is ATOM[13]. It uses OM[24], a link time modification system. To perform profiling using ATOM, one needs to write an instrumenting code file and an analysis code file. The instrumenting code file is used to specify places in the original code where instrumentation needs to be performed and to specify profiling routines, whereas the analysis code file contains the implementation of those profiling routines. The OM generic object modification library is linked with the afore mentioned instrumentation code file to produce a tool which reads the user application and modifies it by adding calls to the routines specified in the analysis file.

Pin[17] follows ATOM's model, but unlike ATOM it modifies applications at runtime. It uses a Just-in-time compiler to insert instrumentation code and optimize code. Through the Pin API it is possible to observe all architectural states of a process, such as memory, control flow or contents of registers. Just like ATOM[13], one needs to write analysis and instrumentation routines to create a profiling tool. Pin can attach to a process for instrumentation and then detach from it, therefore saving instrumentation costs when not needed. Pin intercepts first instruction and compiles straight-line sequence from that instruction onwards. The control is then transferred to the generated sequence, but Pin makes sure that it regains control in case a branch exits the sequence. In that case and after regaining the control, Pin generates more code from the branch target onwards. At each fetch of code by JIT, Pin has the ability to instrument it before translation. To speed up execution, translated and instrumentation code is kept in a code cache.

Detours [15] is a library which is used for instrumenting *Win32* functions on x86 machines. Let us consider a source function that calls a target function. Detours replaces prologue of the target function with a jump to a user-provided detour function, at runtime. The replaced prologue is saved in a trampoline function. That trampoline also contains an unconditional jump to the rest of the target function. Therefore, when the source function calls the target function, the call is redirected to the user-supplied detour function. The detour function performs interception preprocessing and then either returns to the source function or calls the target function through the trampoline. The target function, in the end, returns control to the detour function. At that point, the detour function does interception postprocessing and returns control to the source function. Igprof [12] uses a very similar technique to instrument programs.

Dyninst [10] is a library for program instrumentation that allows insertion of code into running programs. Dyninst provides an API (Application Program Interface) to achieve that. By using this API, one can write a program that attaches to a running program and inserts code into it. After insertion of the new code, the modified program also executes the injected code. In this way, code can be added to a running program for profiling, debugging or any other pur-

pose. Dyninst internally uses *ptrace* to attach to and inject code into running applications.

GProf[14] is a profiler for Linux, that uses both instrumentation and sampling techniques. To profile a program with GProf, one has to compile and link it with profiling enabled, so basically instrumentation code is injected at compile and link time. At entrance, the profiled functions call a runtime routine for profiling. The profiling routine counts the number of time a function is called. It also records the caller of a function by storing the return address of the call. In this way, it can built a Call Graph. Besides that, GProf also reads the Program Counter value at each timer interrupt. In this way it maintains a Program Counter Histogram. A postprocessing program combines the Program Counter Histogram data and Call Graph arcs to show the time spent in each function as well as time spent by functions called from other functions.

2.2.2 Sampling-based Profilers

Sampling-based profilers periodically take samples of Program Counter's value. Therefore, the gathered data is a statistical approximation. They are not as accurate as Instrumentation based Profilers but have much lower overhead because no extra instruction is inserted in the original code. As a result, they are relatively non-intrusive and allow the profiled program to run almost at its normal speed. Sampling profilers normally also exploit the hardware performance counters of modern processors, to look for instructions causing more instruction cache misses, data cache misses or pipeline stalls etc.

GProf[14], also uses sampling besides instrumentation for profiling. It regularly takes samples of the Program Counter, by using kernel's timer interrupt. Therefore, very little overhead is added to the system because the kernel has to service the timer interrupt anyway. Since Linux kernel's timer interrupt typically occurs after each 10 ms, this sampling technique will not give proper figures for a code which consumes a lot of execution time but is seldom hit at the occurrence of the timer interrupt.

OProfile[2] is a sampling-based profiler available for Linux Operating systems. It also allows optional use of Hardware Performance Counters to aid profiling. Events corresponding to different types of Performance Counters can be used to profile different types of events like cache misses or dynamic stalls. If a processor does not have any Hardware Performance Counter, only the kernel timer interrupt is utilized. At the occurrence of the interrupt, the program Counter's value and Process ID of the process which was running at that time is read. The gathered profiled data is used offline for analysis. The profiler gives information about proportion of total time spent in each functions. If appropriate hardware performance counters are found on the processor, then it is also possible to see instructions consuming more cycles, causing more data caches misses, instruction cache misses, dynamic stalls and branch mispredictions etc.

The authors in [7] describe a low overhead (1-3%) sampling-based profiling system designed to run continuously on multiprocessor production systems. The sampling is done at a high rate to increase the accuracy of the profiler. Samples of Program Counter are periodically taken on each processor. Like OProfile, it uses Processor's Hardware Performance Counters for counting different events like cache misses and cycles. The Profiler resets Performance Counter with a random value at each interrupt to lessen biasedness towards hitting certain instructions at occurrence of an interrupt. The profiler's data is used offline to analyze time spent in each function, cycles per instruction and variations of data from different profile runs.

ADORE(Adaptive Object code RE-optimization)[16] is a trace based user-mode dynamic optimization system which uses a sampling profiler based on hardware performance counters of IA64. The system is able to apply runtime optimizations such as data cache prefetching. It is implemented as a shared library on Linux, which is linked to the application at startup. Two threads are created, one contains the original program while the other is the dynamic optimization thread. At start up, Perform[1], which is a performance monitoring interface for Linux developed by HP, is initiated. Perfmon is used to read, write and reset the Hardware Performance Counter in the Performance Monitoring Unit of the Processor. It also determines the sampling rate and creates a kernel buffer known as System Sampling Buffer (SSB). A call back function is provided by the system to copy events logged by SSB to a user buffer when the SSB overflows. The overall overhead of the system is only 1-2%.

2.2.3 Comparison of Instrumentation based and Sampling based Profilers

We have seen that Instrumentation-based profilers are more accurate than Sampling-based profilers, but have higher overheads. Sampling-based profiling can be effectively used to profile an application continuously, like in case of [7]. Another great benefit of Sampling-based profiling is that it is non-intrusive and one does not have to modify any code at either compile time or load time. The downside of Sampling-based profilers however is that the information they give is a statistical approximation. Also, we can not use it to count the number of function calls. Therefore the best way is to combine combine good aspects of Sampling and Instrumentation profiling to achieve better results. This is done by several profilers, like [8], [14] and [26]. For this reason, our profiler also employs both sampling and instrumentation profiling.

2.3 Runtime Profilers for Reconfigurable Systems

All the profilers that we have been discussing in the previous section had one thing in common. They are all meant to speed up programs running on general purpose processors. In this section, we will describe research and implementation of profilers, which are used to dynamically partition applications among software running on general purpose processor(s) and a reconfigurable fabric.

2.3.1 Frequent Loop Detection Profiler

The Warp processor discussed in section 2.1 uses a frequent loop detection profiler to help perform dynamic hardware/-software partitioning. The profiler works by monitoring instructions on the memory bus. Whenever it reads a backward branch, it updates a cache entry which stores the

branch frequency. The cache can hold maximum 16 entries, and each entry is of 16 bit. The profiler only considers loops who have branch offsets of less than 256, or 64 instructions[20]. The purpose of considering only branches with small offsets, is to limit the size of configurable logic and thus save power and area. Due to this and the small cache size, the overhead of the profiler is low at less than 1% of power and area on a MIPS processor. The information from the profiler is read by a Partitioning Co-Processor, found in the Dynamic Partitioning Module, to select most hot branches and transform them into hardware through the configurable Logic.

2.3.2 *Dynamic Application Profiler*

DAProf[18] improves upon the frequent loop detection profiler. Like the frequent loop detection profiler, it also looks for backward branches on microprocessor's instruction bus and is also implemented in hardware. While the frequent loop detection profiler only consider loops with branch offset of less than 256, or 64 instructions, DAProf considers loops whose branch offsets are less than 1024, or 256 instructions. Secondly, while the frequency loop detection profiler only counts iterations of loops, DAProf counts both number of executions and average number of iterations per execution for a loop. The reason of doing so is because it is preferable to map those loops into hardware, which have fewer executions and higher iterations per execution. For example a loop that executes only twice and have 500 iterations per execution would be preferred over the one which executes twenty times and have 50 iterations per execution. This is because the communication requirements for the later loop would be greater.

We have seen that although great amount of work has been done on runtime profilers, still very little effort has been done on runtime profilers for dynamic hardware/software partitioning. The frequent loop detection profiler and DAProf are totally hardware based and therefore not very portable. On the other hand, our profiler is totally software based and hence can be easily ported to different systems.

3. DESIGN AND IMPLEMENTATION

Our profiler runs on the Linux operating system. Its task is to keep statistics of hotness of kernels running on a general purpose processor and present that information to the *scheduler*. Our profiler performs both Instrumentation and Sampling profiling. The current implementation has been done and tested successfully on machines with x86 processors, both single-core and multi-core ones.

Through Instrumentation profiling, we want to record the number of times different functions have been called. To instrument programs, we need some mechanism to inject code into programs. Our code injection technique is inspired by *Detours*, *IgProf* and *DynInst*. Like *Detours* and *IgProf*, our code injector replaces the prologue of each function that has to be profiled with an unconditional jump to the instrumentation code and the instrumentation code contains a jump back to the rest of the profiled function besides the prologue of the profiled function. Like *DynInst*, our profiler uses *ptrace* to attach to running programs and inject code into them.

We are not only interested in the number of times different functions are called, but also approximate time spent per function. For that purpose, we perform sampling, which is discussed later.

Our profiler consists of several different parts. The interaction between those parts is shown in figure 2. The *extractor* utility is used to create instrumentation code file besides others, from an executable *elf* file. The *injector* utility is used to inject instrumentation code and map *shared memory* into the address space of the profiled program. The local APIC Timer interrupts update the *samples buffer*. Finally, the *daemon* combines information from different places in a form that can be quickly and easily read by the *scheduler*. The purpose of the *shared memory* is to keep the *function calls counts* of profiled functions as well as information to be read by the *scheduler*. The reason we have employed the *shared memory* technique is because it is the fastest possible method to communicate data. We discuss each part of the profiler in more details below.

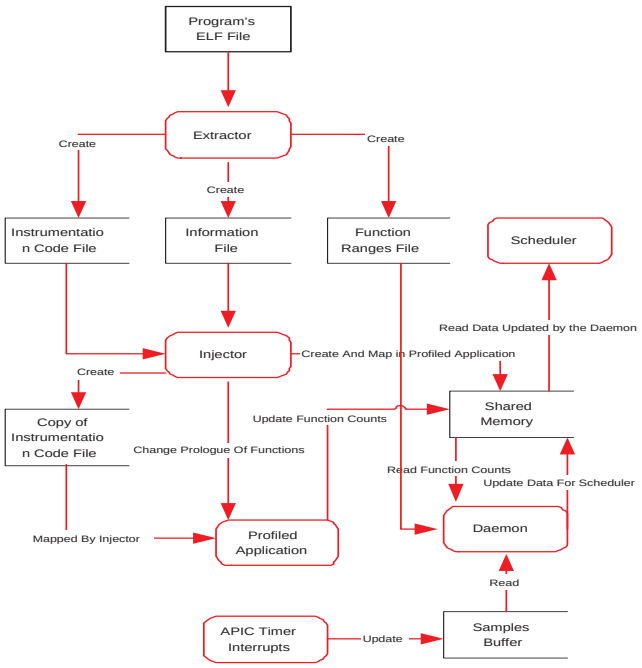

Figure 2: Interaction of different parts of our profiler with the *profiled application* and the *scheduler*

3.1 Extractor

The purpose of the instrumentation code is to record the *function calls counts*. The instrumentation code is generated offline but is injected into the profiled program at runtime. The *extractor* utility reads an executable file and generates instrumentation code from it. Besides that, it also creates two other files. One of those contain information about the to be profiled program and its instrumentation code, while the other is used to hold the address ranges of the profiled functions. We collectively refer to all the three files as *profiling information files*. The rationale behind creating these files offline is to reduce the overhead of injecting instrumentation code and replacing function prologues of the to be

profiled program at runtime. The instrumentation code and information files are used by the *injector*, while the file which contains the address ranges of the to be profiled functions is used by the *daemon*.

A copy of the instrumentation code file generated by the *extractor* utility is mapped by the *injector* into the address space of the profiled program. The *injector* also replaces prologues of to be profiled functions with jumps to locations in the instrumentation code that update calls count for those functions. That is done by using information from the *information file* generated by the *extractor*. Besides that, information such as code segment offset of the instrumentation code, starting addresses and indexes of to be profiled functions, addresses of labels in the instrumentation code, and the hash table used in the *shared memory* are also included in the *information file*. The index of functions are used to index their calls counts in the *shared memory*. The same indexes are used to index *function samples counts* by the *daemon*, as discussed in section 3.5. In current implementation, we only extract information from a program's ELF file and therefore are only able to profile static functions.

3.2 Injector

The *injector* utility is used to create the *shared memory* file and to map *instrumentation code* and *shared memory* files into a process that has to be profiled. It attaches to a process by using *ptrace*. To map a file into the address space of the to be profiled process, it backs up some data from it as well as processor registers and injects the code for mapping the file in the place from where data was backed up. It then executes the mapping code and after executing it, it puts back the backed up data and saved processor registers. After execution of the mapping code, the to be profiled process is able to access the mapped file, as by that time it has been mapped into the address space of that process. We use the *mmap* system call to map files.

After mapping *instrumentation code* and *shared memory* files into a process, the *injector* also modifies the prologues of to be profiled functions by replacing them with a jump to a location in the *instrumentation code*. Besides that, it calculates and inserts addresses of the *shared memory* locations referred in the *instrumentation code* as well as parameters to jump instructions. For that purpose, it uses the addresses returned by the *mmap* system call and the relative offsets defined in the *information file* produced by the *extractor*.

Let us take an example function shown in listing 1, that has to be profiled. The corresponding *instrumentation code* is shown in listing 3. For multicore processors, the *incl* instruction is preceded by *lock* (not shown here), so that only one core can update the count at a time.

After modification by the *injector*, the original function becomes like the one shown in listing 2. Here the jump instruction is followed by a *nop* instruction because the jump instruction consists of 5 bytes while the replaced prologue consisted of 6 bytes. Also note that the address of the *shared memory* location in the *instrumentation code* is not known until runtime and therefore its address is calculated and inserted by the *injector* by using the starting address of the mapped *shared memory* and the index of the function defined in the *information file*. For this example, we have assumed the index to be 0. Similarly, the jump parameter in x86's machine code takes relative addresses and since the starting address of the mapped *instrumentation code* is not known until runtime, that parameter is also calculated and inserted by the *injector*. Moreover, the address of reference to the shared memory location and address of the jump instruction in the *instrumentation code* are also calculated by the *injector* by using information from the *information file* as well as starting address of mapped *instrumentation code*.

Listing 1: Sample function in original form

```
080483b4 <mult>:
 80483b4:        push   %ebp
 80483b5:        mov    %esp,%ebp
 80483b7:        sub    $0x10,%esp
 80483ba:        mov    0x8(%ebp),%eax
 80483bd:        imul   0xc(%ebp),%eax
 80483c1:        mov    %eax,-0x4(%ebp)
 80483c4:        mov    -0x4(%ebp),%eax
 80483c7:        leave
 80483c8:        ret
```

Listing 2: After modification by the *injector*

```
080483b4 <mult>:
 80483b4:        jmp    lbl_start1
 80483b9:        nop
 80483ba:        mov    0x8(%ebp),%eax
 80483bd:        imul   0xc(%ebp),%eax
 80483c1:        mov    %eax,-0x4(%ebp)
 80483c4:        mov    -0x4(%ebp),%eax
 80483c7:        leave
 80483c8:        ret
```

Listing 3: Instrumentation code for sample function

```
        .text
lbl_start1:
        incl shared_mem[0]
        push %ebp
        mov %esp,%ebp
        sub $0x10,%esp
lbl_jmp1:
        jmp 0x80483ba
```

The *injector* utility can be used in two ways. One way is to run the to be profiled process as the child process of the *injector*. In that case the program name and its arguments are passed as arguments of the *injector*. The second way is to give the process ID of the process that has to be profiled as argument of the *injector*, preceded by -i flag. In that case, the *injector* attaches to the running program, maps *instrumentation code* and *shared memory* files into it and then detaches from it. The second option is used by the *daemon*, as discussed in section 3.5. As attaching to a running program can crash the program, there is an option (by using -s flag) in the *extractor* utility to change the original program in such a way that it stops at the beginning. There is also an option (by using -u flag) in *extractor utility* to revert back to the original program. If a to be profiled program is intended to always run as the child process of the *injector*, then there is no need to insert that stop condition at the beginning because when a program is profiled in that man-

ner, the *injector* gets hold of that program from the very beginning.

3.3 Sampler

The sampling is performed by using the local APIC Timer interrupts of each core of the processor. At each such interrupt, we record the instruction pointer, process ID of the interrupted process and the number of core on which the interrupt occurred. We save that information in a circular buffer, which we call *samples buffer*. The *samples buffer* is periodically read by the *daemon* along with the *function calls counts*.

3.4 Shared Memory

For each profiled process, there is one *shared memory* file. It is accessed by several components in the system, as shown in figure 2. First entry in the *shared memory* is the number of functions to be profiled. After that is the area which holds the *function calls counts*. The data in that area is updated by the profiled process. The next portion of the *shared memory* is used to present data to the *scheduler*. The data in that portion is updated by the *daemon*. Finally, the lower portion of the *shared memory* contains a hash table whose purpose is discussed in section 3.6.

3.5 Daemon

The task of the *daemon* is to collect information from different places and present it to the *scheduler* in the most efficient manner. The *daemon* consists of two threads and a signal handler. Actually, we have modified the *do_execve* function in the kernel so that it sends a signal to the *daemon* whenever it starts executing a new program.

3.5.1 Data Structures

The data types used in the *dameon* are discussed below.

FuncRange and FuncRanges *FuncRange* type is used to represent range of a function. It holds starting and ending addresses of a function and the *function index*. As discussed previously, all the functions that have to be profiled are given an index, so that they can be used to index the *shared memory* to update *function calls counts*. Those indexes are also used to index *function samples counts* for the profiled functions. The *FuncRanges* type contains collection of *FuncRange* objects and is used to represent ranges of all profiled functions in a process. The function ranges are read from a file created by the *extractor*. To locate a function from a given instruction pointer value, we also need to know the process ID, therefore the *dameon* also contains a hash map *FuncRangesMap*, whose values are objects of type *FuncRanges* and which is keyed by process IDs. For each sample read from *samples buffer*, the function which was being executed at the time the sample was taken is noted and from it the *samples count* for that function is updated.

ProcManager The *ProcManager* type is used to represent a profiled process and like in case of *FuncRanges*, we have a hash map *procManagerMap*, whose values are objects of type *ProcManager* and which is keyed by process IDs. *ProcManager* holds the *functions samples counts*. It also contains objects of type *SnapShots* and *SharedMemory*, which are used to represent snap-shots of the counts and *shared memory* of the profiled process. Those two types are discussed next.

SnapShots and CountsSnapShot The *CountsSnapShot* type represents snap-shot of the *function calls counts* and *function samples counts* at a particular instance. Besides that, it also holds the time at which the snap-shot was taken. The *SnapShots* type contains a collection of *CountsSnapShot* objects. *ProcManager* uses a *SnapShots* object to take the snapshots and return the differences of counts between a specified length of time.

Shared Memory The *Shared Memory* type is used to represent *shared memory* files. The contents of the *shared memory* were discussed in section 3.4. The *daemon* uses the *shared memory* to read the *function calls counts* and write data for the *scheduler*.

3.5.2 Working

The *daemon* consists of two threads and a signal handler. The purpose of the signal handler is to receive the signal which contains process ID of a newly loaded program, as was discussed previously. Besides that signal handler, there are two threads. First one is the main thread and the second one is the data processing thread, whose job is to periodically read samples from the *samples buffer* and *function calls counts* from shared memories and update data for the *scheduler*. Besides that, it also deletes unused *instrumentation code* and *shared memory* files. The two threads are described in more detail below.

Main Thread The pseudocode for the *Main* thread is shown below.

Algorithm 1 Pseudocode for Main Thread of the *daemon*

```
ReadConfigurationFile();
Setup the Signal Handler to handle signals from the kernel whenever a new program is loaded to execute;
Write the Process ID value of this daemon to a file in /proc directory, so that the kernel can read it and start sending signals to the daemon;
Create the Data Processing thread;
while 1 do
    Wait on the semaphore to be signaled by the Signal Handler.
    while unread Process IDs exist in the PID_queue do
        pid ← getNextPID(PID_queue)
        path_name ← extract_path(pid)
        if program path_name has corresponding profiling information files then
            if process with PID pid is not running as a child process of the injector then
                Use the injector utility to create the shared memory file and map the instrumentation code and shared memory files for the process with PID pid;
            end if
            funcRanges ← new FuncRanges();
            funcRanges.read(getFuncRangesFileName(path_name))
            funcRangesMap[pid] ← funcRanges
            procManagerMap[pid] ← new ProcManager();
            procManagerMap[pid].create(pid, spanSize);
        end if
    end while
end while
```

At the beginning, it reads the configuration file, sets up the

signal handler and creates the *Data Processing* thread. After that it enters an infinite while loop. At the beginning of the while loop, it waits for a sempahore that is posted by the signal handler. The signal handler post that sempahore whenever it receives the signal from the kernel when a new program is loaded. That signal contains the newly loaded program's process ID as its information. The signal handler adds that process ID into a circular queue before posting the semaphore. The reason for using a queue here is to ensure that no signals are lost in case *daemon* was busy doing some other work and more than one signals were received at that time. After acquiring the semaphore, it extracts all unread Process IDs from the queue and for each of them, it checks whether the process in question has corresponding *profiling information files*. If it does and is not already being run as child process of the *injector*, the *daemon* calls the *injector* utility to create a *shared memory* file and a copy of the *instrumentation code* file, and map them into the address space of that process. The *main thread* also creates and initializes objects of type *funcRanges* and *ProcManager*, for a process that has corresponding *profiling information files*. Those objects are then added to *funcRangesMap* and *ProcManagerMap* respectively.

Data Processing Thread The *Data Processing* thread is used to read the samples from the *samples buffer*, read *function calls counts* from *shared memory*, process data and write data to the *shared memory*, which is read by the *scheduler*. The pseudocode is shown as algorithm 2. Samples from the kernel are read at intervals of 200 ms, while *function calls counts* from *shared memory* are read every second. The reason we are reading the samples at a relatively higher rate is to keep the size of the *samples buffer* small, because a large buffer size can incur a significant overhead in the kernel as it can cause more cache misses. The read data is processed and written to the *shared memory* by using objects of type *SnapShots* and *SharedMemory* in *procManager*. The *getCurrent* method of *SnapShots* returns the most recent counts, while the *getPrevious* method return the counts *spanSize* seconds before. The value of *spanSize* is read from the configuration file and represents the span time for the counts. For example, if *spanSize* is declared as 5, the *getPrevious* method returns the counts 5 seconds before and therefore the counts written in *shared memory* are the ones since the last 5 seconds. However, if the profiled process has been running for less than *spanSize* seconds, then the counts since the beginning are returned.

Here it has to be noted that since the *daemon* is required to access *shared memory* files created by any process and also map files to any process, it is run with root privileges. Also the *daemon* gives read and write access to the files that it creates, so that a process running in a user account can access them. Secondly, in current implementation, we do not profile processes that are forked as child processes of profiled processes, with the exception of those child processes that use *exec* family of functions to load a program, given that the loaded program has corresponding *profiling information files*. In future, we intend to add functionality to profile any kind of child processes. This can be achieved by having the kernel send a signal to the *daemon* whenever a process forks a child process and adding code in the *daemon* to take appropriate steps.

Algorithm 2 Pseudocode for *Data Processing* thread of the daemon

```
numberOfSeconds ← 1;
time0 ← getCurrentTime();
while 1 do
  Sleep for 200 milliseconds;
  Read samples from the samples buffer;
  for all s in samples do
    if funcRangesMap[s.pid] exists then
      fn ← funcRangesMap[s.pid].getHitFuncNumber(s.ip);
      procManagerMap[s.pid].incSampleCount(s.cpu, fn);
    end if
  end for
  time1 ← getCurrentTime();
  diffTimeInSecs ← toSecs(time1 − time0);
  if diffTimeInSecs is greater than or equal to
  numberOfSeconds then
    for all [pid, funcRanges] in funcRangesMap do
      if process with process ID pid no more exists then
        Remove the instrumentation code and shared memory
        files for that process;
        Remove funcRangesMap[pid];
        Remove procManagerMap[pid];
      else
        procMan = procManagerMap[pid];
        procMan.snapShots.takeSnapShot();
        curSS ← procMan.snapShots.getCurrent();
        prevSS ← procMan.snapShots.getPrevious();
        procMan.sharedMemory.writeDiff(curSS, prevSS);
        numberOfSeconds ← numberOfSeconds + 1;
      end if
    end for
  end if
end while
```

3.6 Interface

In section 3.4, we mentioned that the lower portion of the *shared memory* contains a hash table. For MOLEN, the keys to that hash table are profiled functions' names and values are indexes of those functions. Actually MOLEN uses SET and EXECUTE instructions to map a kernel into hardware [22]. Those instructions are placed in the code by the MOLEN compiler [19]. However at runtime, they are just hints to the runtime system and it is the responsibilty of the runtime system to decides which kernel to map. The SET instruction takes the name of the kernel to be mapped as the parameter. Now since in our system, the data written by the *daemon* is contained in an array form, where functions are identified by the indexes of that array, the *scheduler* requires this hash table to get the indexes from function names. It has to be noted here that the form of this hash table is dependant on the current scheduler of MOLEN and therefore will be different for different schedulers. For example, if MOLEN adds capabilities for Just In Time FPGA compilation, the parameter of interest might be a function's address instead of its name, and therefore keys of the hash table would be addresses of functions instead of their names.

The *shared memory* structure presented in Figure 3 is the only part that needs to be known by any runtime system which needs to integrate with our profiler. The rest of the profiler can be seen as a black box. All the generated data by the profiler is accessible from here. We need to make sure that the *scheduler* does not read the data at the same time the *daemon* is writing to it. To achieve that, we have used the concept of double buffering, that is the *daemon* writes

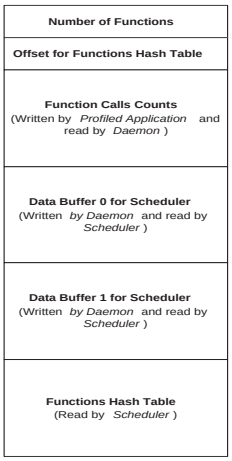

Figure 3: Contents of the *Shared Memory*

data to one buffer and the next time it writes to the other. When it completes writing data to one buffer, it changes the *index* to point to that buffer, so that the *scheduler* can read it from there. This scheme obviously assumes that the *scheduler* can read from the buffer in much less time than the time between two consecutive writes by the *daemon*. This is a fair assumption because the *daemon* does not need to update information very quickly. For example, in our case, we update the data only once in a second.

4. PERFORMANCE EVALUATION

To test the performance of our profiler, we have used benchmarks from different areas. One of this is PC version of *tcf*, which is a subset of hArtes [3]. Then we have *minisat2* [11], which is an industrial scale SAT solver. Note that we have removed the randomness part in *minisat2*, so that our results do not vary from run to run. Then we have a *jpeg decoder* and a *H264/AVC encoder*, from MediaBench II benchmarks [4]. Finally, we have *coremark* [5], which is a free synthetic benchmark from *EEMBC* [6]. We used a single-core x86 processor to run these benchmarks.

In section 4.1, we present the overhead of the Instrumentation part of our profiler and compare it with that of *Pin*. In section 4.2, we discuss the time it takes to inject instrumentation code into profiled applications. In section 4.3, we discuss the overhead imposed by sampling and the *daemon*, and compare it with that of *OProfile*, while in section 4.4, we compare the accuracy of our sampler with that of *gprof*. In section 4.5, we discuss the overall overhead of our profiler. Finally, in section 4.6, we discuss the amount of functions which are profilable. This is because some functions, usually of very small size, do not have enough prologue instructions to replace with a jump instruction.

4.1 Instrumentation Overhead

The table 1 shows the comparison of instrumentation overhead of our profiler with that of *Pin*. For this purpose, we made a *Pin* tool that just counts the number of functions called, so that it is equivalent to our profiler. From the table we can see that overhead with *Pin* is always above 20%, while for our profiler, except for the *multiply* application, the overhead is always less than 1.5%. The *multiply* application repeatedly calls a multiply function which just returns product of two numbers. Normally such tiny functions are inlined by the compiler and therefore would not be profiled. However, testing the profiler with such tiny functions gives us an idea of the worst case performance.

Table 1: Instrumentation Overhead

Program	Normal (secs)	Pin (secs)	Our profiler (secs)	Pin OH	Our Profiler OH
multiply	9.681	11.646	10.105	20.30%	4.38%
coremark	12.496	15.550	12.664	24.44%	1.34%
tcf	7.083	8.758	7.107	23.65%	$< 1\%$
djpeg	2.269	3.180	2.202	40.15%	$< 1\%$
h264enc	40.774	125.566	41.037	207.96%	$< 1\%$
minisat2	29.004	37.064	29.142	27.79%	$< 1\%$

The readings given in table 1 were calculated by running each of the benchmark ten times and then taking the mean values.

4.2 Instrumentation Code Injection Time

In table 2, we have listed the injection time for different applications. The injection time varies slightly from run to run, that is why we took means of 10 runs. The *Map Time* is the time to map the *instrumentation code* and *shared memory* files into the address space of the program. The *Data References Insertion Time*(DRIT) is the time taken to insert the resolved addresses into the instrumentation code and the *Prologues Modification Time*(PMT) is the time taken to replace the prologues of the functions with jump to the instrumentation code.

From table 2, we can see that *Data References Insertion Time*(DRIT) and *Prologue Modification Time*(PMT) are 0 for all benchmarks other than *h264enc* and *djpeg*. This is because *h264enc* and *djpeg* contain larger number of profilable functions and due to that more time is taken to insert data references and modify function prologues. It is difficult to analyze why *Prologue Modification Time*(PMT) for *djpeg* is taking 50 ms, while for *h264* it is only taking 9 ms, considering that there are much more functions to be profiled in case of *h264*. It seems that it is quite much dependant on the operating system, since the modification in code is done by using the *ptrace* system call. The *Map Time* does not vary much from application to application. This was expected, since it is just the time to map the files into profiled process's address space and therefore should not be application dependant. Nevertheless, from table 2, we can see that Instrumentation Code Injection Time is quite low even for programs with large number of profilable functions.

4.3 Sampling and Daemon Overhead

In table 3, we have shown the results achieved without performing instrumentation. The purpose of not performing instrumentation in this case is to quantiy the overhead imposed by sampling and the *daemon*. The results are compared with those achieved from *OProfile*. We only used benchmark applications which take more than 10 seconds to execute. The readings given in table 3 are means of 10 readings. From that table, we can see that the overhead

Table 2: Instrumentation Code Injection Time

Program	Prog Size (KB)	Number Of Funcs Profiled	Map Time (ms)	DRIT (ms)	PMT (ms)
multiply	9.4	5	39	0	0
coremark	19	29	27	0	0
minisat2	173.6	52	27	0	0
tcf	91.8	59	45	0	0
djpeg	103	177	64	7	50
h264enc	663.7	560	63	5	9

Table 3: Sampling and Daemon Overhead

Program	Normal (secs)	OProfile (secs)	Our Profiler (secs)	OProfile OH	Our Profiler OH
coremark	12.496	12.510	12.497	$< 1\%$	$< 1\%$
minisat2	29.004	29.037	29.075	$< 1\%$	$< 1\%$
h264enc	40.774	40.838	41.051	$< 1\%$	$< 1\%$

for both *OProfile* and our profiler is negligible. It has to be noted here that we only used the *timer interrupt event* for *OProfile*, so as to make it functionally equal to our profiler.

4.4 Sampling Accuracy

In this part, we checked the accuracy of the sampling part of our profiler by comparing it with gprof [14]. We used *minisat2* and *coremark* to perform this experiment. We ran each program five times, both with our profiler and with *gprof*, so that we could get the mean values and standard deviations. The results are given in table 4. In that table, means of percentages of total time spent and standard deviations for the functions which took the most time according to *gprof* are given. The functions are sorted by percentages of total time spent, given by *gprof*. From the table, it can be seen that the mean values for both our profiler and *gprof* are almost the same, while the standard deviations for our profiler are better than that of *gprof*. The reason for this is that *gprof* by default samples at the rate of 100 samples per second, while in our case the maximum sampling rate is 250 samples per second. Actually we take our samples through the local APIC timer interrupts. For a tickless kernel, which we are using, the frequency of those interrupts vary with workload. The maximum frequency of those interrupts in our case is 250 samples per second.

4.5 Overall Overhead

Table 4: Sampling Accuracy of our profiler

Function	gprof (%)	Our Profiler (%)	gprof Std	Our Profiler Std
coremark				
crcu8	31.77	31.07	1.55	0.78
core_state_transition	30.05	31.56	1.39	0.69
core_bench_list	13.85	14.21	1.70	0.57
matrix_mul_matrix_bitextract	5.48	5.69	0.47	0.60
minisat2				
Solver::propagate	74.87	75.02	0.89	0.52
Solver::analyze	13.56	13.42	0.51	0.42
Solver::litRedundant	4.45	4.61	0.32	0.15
Solver::cancelUntil	2.85	2.84	0.32	0.20

Table 5: Overall Overhead of our profiler

Program	Normal (secs)	Our Profiler (secs)	Overhead
multiply	9.681	10.141	4.75%
coremark	12.496	12.653	1.26%
tcf	7.083	7.085	$< 1\%$
djpeg	2.269	2.256	$< 1\%$
h264enc	40.774	41.112	$< 1\%$
minisat2	29.099	29.004	$< 1\%$

Table 6: Percentage of Profilable functions

Program	Opt Level	Total Funcs	Profilable Funcs
tcf	-O0	59	59 (100%)
h264enc	-O2	591	560 (94.8%)
minisat2	-O3	56	52 (92.9%)
djpeg	-O2	208	177 (85.1%)
coremark	-O2	40	29 (72.5%)

In this experiment, we tested our benchmark applications with all parts of the profiler working. The means of ten readings are shown in table 5. The results are as expected, that is all applications other than *the multiply* application have overhead of less than 1.5%. Morevover, overall overhead for those application is almost the same as that for instrumentation overhead, thus reinforcing the fact that sampling and the *daemon* have very low overheads.

4.6 Percentage of Profilable functions

Our profiler replaces prologues of to be profiled functions with a jump instruction. For that purpose, a function's prologue must be at least 5 bytes because the jump instruction consumes 5 bytes. Most of the functions do have at least 5 bytes of prologue, but some functions, usually of very small size, do not. By prologue instructions, we mean instructions which prepare the stack and registers for use within a function and not any instruction that is used afterwards. In table 6, we have shown the number of function which are profilable for different applications. We have also listed the optimization levels used to compile those applications, the purpose of which is to see if optimizations make it any harder to find profilable functions. Except for *coremark*, all applications have more than 85% of profilable functions and besides *tcf*, all are using high level of optimizations. The reason *coremark* has only 72.5% of profilable functions is because there are many functions of very small sizes in it.

It should be noted that for comparing Instrumentation overhead with *Pin*, as discussed in section 4.1, to make the comparison with *Pin* fair, we made the *Pin* tool to only instrument those functions which our profiler was also instrumenting.

5. CONCLUSION

In this paper, we described the design and implementation of a runtime profiler which can be used as a part of MOLEN runtime environment. The profiler is a combination of a Sampling profiler and an Instrumentation profiler. We discussed in detail different parts of the profiler namely the *extractor*, *injector*, *sampler*, *shared memory*, *daemon* and data

structures being used by each. Then, we showed the overhead of our profiler from different aspects. This is done by showing the overhead on Instrumentation, Code Injection, Sampling and the overall overhead. Besides, we compared the accuracy of our profiler with a popular design time profiler. All the presented results shows that our profiler is very low overhead (less than 1.5%) and very accurate.

6. REFERENCES

[1] `http://perfmon2.sourceforge.net/`, 2009.

[2] `http://oprofile.sourceforge.net/`, 2010.

[3] `http://www.hartes.org/index.php?option=com_content&task=view&id=62&Itemid=87`, 2010.

[4] `http://euler.slu.edu/~fritts/mediabench/mb2/`, 2010.

[5] `http://www.coremark.org/`, 2010.

[6] `http://www.eembc.org/`, 2010.

[7] J. M. Anderson, L. M. Berc, J. Dean, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, W. E. Weihl, L. M. Berc, S. Ghemawat, M. R. Henzinger, and S. Leung. Continuous profiling: Where have all the cycles gone? In *ACM Transactions on Computer Systems*, pages 1–14, 1997.

[8] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices, Vol 36, No. 5*, pages 168–179, 2001.

[9] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review, Vol 43, No. 2*, pages 56–65, 2009.

[10] B. Buck and J. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications, Vol 14, No. 4*, pages 317–329, 2000.

[11] N. Een and N. Sorensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 333–336, 2004.

[12] G. Eulisse and L. A. Tuura. Igprof profiling tool. *Computing in High Energy Physics and Nuclear Physics*, page 665, 2004.

[13] A. Eustace and A. Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[14] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN Notices, Vol 39, No. 4*, pages 49–57, 2004.

[15] G. Hunt, , G. Hunt, and D. Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[16] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism, Vol 6*, page 2004, 2004.

[17] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[18] A. Nair and R. Lysecky. Non-intrusive dynamic application profiler for detailed loop execution characterization. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 23–30, New York, NY, USA, 2008. ACM.

[19] E. M. Panainte, K. Bertels, and S. Vassiliadis. Compiling for the molen programming paradigm. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pages 900–910, September 2003.

[20] A. Ross and F. Vahid. Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Transactions on Computers, Vol 54, No. 10*, pages 1203–1215, 2005.

[21] M. Sabeghi and K. Bertels. Toward a runtime system for reconfigurable computers: A virtualization approach. In *Design, Automation and Test in Europe (DATE09)*, April 2009.

[22] M. Sabeghi and K. Bertels. Interfacing operating systems and polymorphic computing platforms based on the molen programming paradigm. In *Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2010.

[23] M. Sabeghi, V. Sima, and K. Bertels. Compiler assisted runtime task scheduling on a reconfigurable computer. In *19th International Conference on Field Programmable Logic and Applications (FPL09)*, August 2009.

[24] A. Srivastava and D. Wall. A practical system for intermodule code optimization at link-time.

[25] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: a first approach. In *Proceedings of the 40th annual Design Automation Conference*, pages 250–255, New York, NY, USA, 2003. ACM.

[26] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS), Vol 27, No. 4*, pages 732–785, 2005.

[27] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The molen polymorphic processor. *IEEE Transactions on Computers, Vol 53, No. 11*, pages 1363–1375, 2004.