CHALMERS

A Performance Profiler for Aiding in Threading Legacy C/C++ Code

Master of Science Thesis

CHRISTIAN KINDAHL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

A Performance Profiler for Aiding in Threading Legacy C/C++ Code

Christian Kindahl

Examiner: Sven-Arne Andréason

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, June 2009

Abstract

There are software tools for aiding and automating the process multi-threading sequential programs that rely on complex static analysis. On large programs the analysis can be too heavy to be practically performed on the entire program. As a result a method is needed for effectively selecting which parts of a program to focus the analysis on. A suitable tool for this purpose is a performance profiler.

This thesis aims at describing the design and implementation of a low-overhead performance profiler for aiding the analysis. It addresses the task with a requirement on operating system and processor architecture portability. The profiler uses a sample based strategy in combination with call stack analysis for collecting information on hot spots and hot paths. It allows profiling based on execution time as well as other metrics such as cache misses, specific arithmetic operations and more.

Benchmarking the profiler with programs from the SPEC CPU2000 suite shows an overhead between 1% and 4% while still retaining a high degree of accuracy.

Sammanfattning

Det finns mjukvaruverktyg för att underlätta och automatisera parallellisering av sekventiella program som använder sig av komplex statisk analys. På stora program kan analysen bli för tung för att vara praktiskt genomförbar på hela programmet. På grund av detta behövs en metod för att välja ut vilka delar av programmet att fokusera analysen på. Ett lämpligt verktyg för detta är en prestandaprofilerare.

Den här rapporten syftar till att beskriva designen och implementationen av en prestandaprofilerare vars syfte är att underlätta analysprocessen. Uppgiften attackeras med kravet att profileraren skall vara portabel över såväl operativsystem som processorarkitekturer. Profileraren använder sig av en samplingsbaserad strategi i kombination med analys av anropsstacken för att samla information om programmets beteende under körning. Profileraren tillåter profilering med avseende på exekveringstid och andra enheter som till exempel cache-missar, specifika aritmetiska operationer med mera.

Prestandaevaluering av profileraren på program från SPEC CPU2000-sviten visar en overhead mellan 1% och 4% samtidigt som hög precision erhålls.

Contents

1 Introduction

1.1 Background

The current trend in computer architecture is to add more computing cores to one chip instead of the previous trend to increase the clock frequency of a single core. The number of cores per chip will soon increase rapidly and in order to take advantage of the extra computing power software needs to be rewritten. Rewriting software designed for single core systems to utilize multiple cores is a difficult task; it's desirable to automate this process using software.

Software development tools have had a difficult time adapting to this trend. One reason is the high complexity of software. While there are theoretical algorithms capable of analyzing entire software systems in order to make them parallel, none of them are practical. The analysis is a very time consuming process.

One approach to this problem is to selectively choose which parts of the program to analyze. A good tool for that is a performance profiler. A performance profiler (from here on referred to as profiler) is software capable of analyzing another program, creating a profile of its runtime characteristics. A profiler can for example obtain information about procedure execution times, call frequencies and advanced processor details such as the number of cache misses, memory accesses and more. This information can be very valuable when selecting which parts of a program to analyze for making it multi-threaded.

1.2 Problem Definition

Performance profiling is not a new concept; it has been around for centuries [2]. Different profiling techniques with varying accuracy have been suggested in both scientific papers and practice, but most of these techniques have not been developed with operating system and processor architecture portability in mind.

This thesis aims at finding a solution for and implementing a platform portable performance profiler that is capable of generating profiles with the information and accuracy that is needed when multi-threading originally sequential C and C++ applications. From within that context this thesis tries to address the following problems:

- *Data Gathering* - Run-time information must be collected from the profiled program. What kind of data can be collected and how can it be obtained?
- *Data Management* - The run-time data must be managed in a space and performance efficient manner in order to minimize the profiler overhead. What strategies and algorithms are suitable for this purpose?
- *Data Analysis* - The last step is to analyze and post process the data, presenting it in a way that makes it useful.

1.3 Purpose

This thesis project aims at designing and implementing an operating system and processor architecture independent performance profiler for aiding the process of multi-threading sequential software. This report serves the purpose of documenting the entire process from analysis to evaluation.

1.4 Scope

For UNIX flavored operating systems there are free open source profilers that not surprisingly claim a certain degree of architecture independence. These profilers are tightly bound to UNIX flavored operating systems, excluding support for Microsoft Windows and other closed source operating systems [9][10][16]. There are also other proprietary profilers that claim support for Microsoft Windows and GNU/Linux specifically but these can only be used on the Intel x86 architecture [8][15]. There is a gap for profilers satisfying both operating system and processor architecture portability.

Even though this project aims at developing a platform independent performance profiler, special sub components will have to be specifically developed for each operating system and processor architecture. Because of limited time it's not possible to include support for all platforms in one step. Instead focus will be put towards a portable design and architecture that relatively easily can be extended with support for additional operating systems and processor architectures. The primary target of this profiler is the GNU/Linux operating system and the x86 and x86-64 architecture.

1.5 Assumptions

Apart from being portable across platforms the purpose of the profiler is to aid the analysis of multi-threading originally sequential C and C++ programs. This implies a few requirements that need to be known from the beginning:

- The profiler does not necessarily have to support profiling parallel programs although it can be desirable to evaluate the performance improvements once a sequential program has been parallelized.
- The applications to be profiled are written in either C or C++ which means that they will be compiled to machine code and not any intermediate byte code.
- Although not required, the parallelization analysis algorithm appreciates performance statistics on source line level.
- Program procedures can be executed in different contexts, it's useful to know the context during the analysis process. The profiler must be able to distinguish between different execution paths leading up to an individual statement or procedure.

1.6 Nema Labs

The profiler developed in this thesis is intended for integration in a software product called FASThread developed by Nema Labs. Nema Labs is a privately held company specialized in providing software development tools for taking advantage of the extra computing power in multi-core and multi-processor systems.

Nema Labs currently employs a team of software engineers and other key individuals, all lead by Professor Per Stenström who is a key scientific contributor in the field of multi-core processors. Nema Labs was founded in 2006 and continues to grow strong. At the time of writing the Nema Labs staff includes about ten employees.

FASThread is a product aimed at introducing parallelism in existing C and C++ software systems. The key idea is to make this process as easy and automated as possible for the software developers using FASThread. FASThread addresses the key challenges of this task and introduces a workflow for solving this problem. In this process performance profiling is a key component.

2 Method

The thesis work was divided into four steps. These steps overlap as new information was discovered along the way.

2.1 Analysis

The first step was to explore different profiling techniques and profilers. A literature study on runtime performance analysis was performed and existing profilers was studied. Most scientific articles were found in the ACM and IEEE Xplore databases, all other articles were found in local university databases.

When enough knowledge had been gathered the software requirements were redefined and a profiling technique was selected.

2.2 Design

The second step was to create a software design. In order to do this, features were grouped into different building blocks in order to categorize them as operating system or processor architecture specific. The goal was to find the largest common divisor among all platforms. Finding the blocks required in-depth studies of different architectures and operating systems. These studies involved reading manuals, low level Application Programming Interfaces (APIs) and reverse engineering. Small proof of concept applications was developed on both GNU/Linux and Microsoft Windows in order to establish the availability of certain features.

When all blocks had been discovered a software architecture was developed. Lastly a class level design was created.

2.3 Implementation

The implementation was carried out in a test-driven fashion. Unit tests were used when possible. Unfortunately, the complex nature of the project prevented some parts of the kernel code from being unit tested. These parts were tested at a higher level using automated shell scripts.

The implementation was regularly compiled and tested on different computer systems including 32- and 64-bit AMD and Intel x86 systems. Small test programs were profiled in order to validate the correctness of the profiler implementation.

2.4 Evaluation

When evaluating the profiler it was compared to existing profilers. Programs from the SPEC CPU2000 benchmark was profiled and timed in order to compare both correctness and performance.

3 Theory

This section provides background theory on software profiling, starting with different types of profile summaries followed by key topics in profiler implementation.

3.1 Profile Summaries

There exist different types of profiles that summarize the data collected by a profiler. The most common are "flat", "call graph" or "call tree" profiles [1], these are briefly presented in this section.

3.1.1 Flat Profiles

Flat profiles explain how much time that is spent in individual procedures. Depending on the type of profiler this information is sometimes complemented with procedure call frequencies. Flat profiles are helpful in determining in which routines a program spend most of its execution time. They do not provide any contextual information on the program flow leading to a procedure call. Figure 1 shows what a flat profile can look like.

%	time (s)	calls	function name
55	5.3	3	function_1
25	2.4	2	function_2
20	2.0	1	function_3
0	0.1	1	main

Figure 1: Example of a flat profile.

3.1.2 Call Graph Profiles

Call graph profiles break down flat profiles by separating between procedure callers and callees. In the call graph, nodes represent the procedures and the arcs the procedure calls. The arcs are often noted with call frequencies and the nodes with procedure execution times [2]. Call graph profiles help answering why a certain procedure is executed. Figure 2 shows an example of a simplified call graph profile.

3.1.3 Call Tree Profiles

In a call tree each node represents a procedure call. A child node represents a call to that procedure from the procedure that the parent node represents. Call trees contain more information than call graphs because they use separate nodes for calls that occur in different contexts [11]. Call graphs on the other hand use the same node for all calls to that procedure. Figure 3 shows an example of a simplified call tree profile.

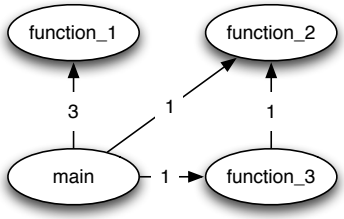

Figure 2: Example of a call graph profile.

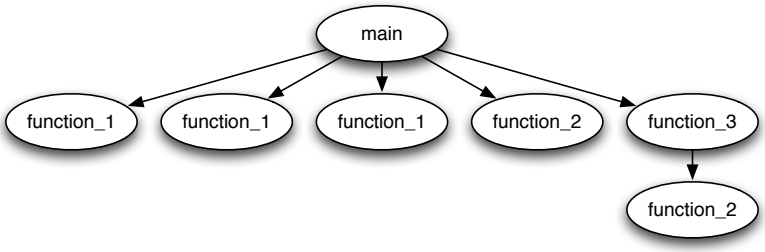

Figure 3: Example of a call tree profile.

A well known variation of the call tree is the call context tree which sometimes is better suited for implementation since it is more compact and requires less memory [14]. Call context trees reuses the same nodes for identical contexts. In the example showed in Figure 3 it would mean that all calls from main to function_1 would be represented as a single node with the arch from main to function_3 attributed with the value three.

3.2 Data Gathering

The first step of software profiling is to collect data during execution of the profiled program. Most software profilers collect information about execution time and call frequencies of procedures and/or statements [13]. Profiling literature is dominated by two strategies for collecting this type of data, sampling and instrumentation [1]. Both have very different characteristics and trade-offs. Because of this they are sometimes used in combination with each other, for example in the most widely used call graph profiler gprof [2][6]. While gprof primarily is instrumentation based it uses sampling to count the time spent in individual procedures.

A third profiling strategy to be mentioned just briefly is simulation. Using this technique the profiler simulates every instruction in the profiled program, without letting any instruction execute directly on the host system. This technique is not an option for the profiler in this thesis, simulating an entire processor instruction set is a huge step backwards in terms of processor architecture portability.

3.2.1 Instrumentation

Instrumentation is the process of extending a program with additional instructions that will collect data that is required for performing a program analysis. In the context of this report instrumentation is used to generate a performance profile, other uses are memory debugging and memory leak detection [27]. Instrumentation is applied on either program source code or directly on a program binary. Source code instrumentation can be done manually by a software developer [3] or automatically by for example a compiler. The GNU Compiler Collection (GCC) is an example of a compiler system that supports automatic source code instrumentation [4].

Compared to source code instrumentation, binary instrumentation is more complex to implement but it has the advantage that the source code is not needed for the programs to profile. This allows a profile to expand over system libraries where the source code may not be available. Inserting additional instructions into a binary executable is not trivial. The address of all instructions following the inserted instructions will be shifted, causing jump and branch addresses to be invalid unless corrected [5]. This can be circumvented by replacing the instructions where the instrumentation should be done with jumps to new routines where the replaced instructions and instrumentation is executed. This avoids the address shifting problem but unfortunately introduces extra overhead because of the extra jumps and possibly also cache misses.

Because of the nature of instrumentation it's important to keep the number of instrumentations to a minimum. The inserted instructions run within the profiled program's process, if not efficiently implemented or inserted at bad places the instrumentations themselves may affect the programs performance yielding inaccurate profile results. This is especially apparent in programs using many small procedures in which the instrumentation account for a larger share of the procedure's total execution time [6]. If small instrumented procedures have high execution frequency the resulting profile may be seriously misleading.

In order to minimize the impact of the above problem, a majority of the existing instrumenting profilers allows only procedure blocks to be instrumented, not individual statements [1]. Analyzing what is causing a certain behavior inside a procedure must be done manually by a software developer.

3.2.2 Sampling

Sampling is the process of taking samples from the context of the program being profiled. A sample contains the value of the instruction pointer register and usually also additional information necessary for tracing which procedure calls that lead up to the sampled instruction. On many processor architectures this information can be obtained by analyzing the profiled program's call stack.

An attractive property of sampling is that it can be used in an unobtrusive process. It allows the profiler to operate completely independently from the process being profiled. One advantage with this property is that unlike instrumentation it does not introduce

biased performance slowdowns in the profiled process. It has also been shown that the overhead of profilers using sampling can be as low as 2%-7% which is an order of magnitude lower than instrumenting profilers [6]. Sampling is considered to be a modern approach to profiling, thus being used by many modern profilers [8][9][10][12].

The low overhead of sampling comes at the cost of accuracy. Sampling is a statistical method yielding approximate results. Instructions executed between samples will pass unnoticed, but on the other hand they are likely to be noticed in other samples if having a significant impact on the performance. Even though the accuracy increases with a smaller sampling period, smaller sampling periods increase the profiler's performance overhead. For finding hot spots or hot paths perfect accuracy is not critical [1].

In the context of profiling, approximation is not only a bad thing. A common profiling problem is how to store all collected data while profiling. The data collected by a profiler can grow very rapidly; especially if the profiled program contains many small procedures (which are common in object oriented software [11]). Approximation by sampling reduces the memory space requirements because not every procedure call will be registered [11].

Another interesting advantage of sampling is the possibility of calculating instruction level cost. This is possible because each sample contains the exact instruction in execution. Pinpointing exactly where most resources are being consumed inside a procedure is advantageous; [1] greatly empathizes the importance of this aspect and argues that it can be very difficult to locate a bottleneck by just knowing what procedure it's located in.

It is often desirable for a profiler to generate either a call graph or call context tree profile in order to present the context of procedure calls. This is very straight forward when using instrumentation, but difficult when using sampling. The problem is that it's difficult to know what have happened between samples. Two samples may evaluate to the same execution path, but there is no easy way of telling which procedures in the second path that has been revisited since the first sample. Knowing this is essential in order to build a call graph or call context tree. To solve this problem each procedure invocation needs to be flagged upon visitation in order to know which procedure calls that has been visited and which have not. [11] suggest an efficient way of doing this, without inserting additional instrumentations. Their solution involves using the least significant bit of the procedure's return addresses. This bit is ignored on some processor architectures.

4 Analysis

This section discusses the topics of data gathering, data management and data analysis with respect to the requirements on the profiler in this thesis. This section assumes familiarity with the performance profiling concepts described in section 3.

4.1 Data Gathering

While in the context of multi-threading sequential software perfect call frequencies can be useful (to calculate the time spent in each procedure invocation), instruction level profiling is a much more attractive property because it can identify loops and other heavy operations within a procedure. This information is highly valued because it allows the multi-threading analysis to be directed at more specific targets.

A major drawback with sampling is that compared to source code instrumentation, sampling requires a larger portion platform specific source code. Binary instrumentation is even worse than sampling in this aspect since it involves manipulating machine code directly. The advantages of sampling: low overhead, unobtrusive, system-wide and the possibility to obtain other characteristics such as cache misses makes it the choice for the profiler in this thesis.

This section explains how to collect data using sampling and which data that is needed in order to generate a useful profile.

4.1.1 Sampling Method

As in any other context using statistical sampling a sampling method must be established. Two different methods were considered for the profiler in this thesis, simple random sampling and systematic sampling.

Using simple random sampling the profiled application would be sampled at random intervals. While running multiple profiling sessions on the same application would yield very accurate results on average, a single profile would likely be biased due to the randomness of selecting each sample. This sampling error makes simple random sampling an unsuitable method unless profiling is performed multiple times on the same application to create an average profile. An advantage with this method is that it's relatively easy to implement compared to systematic sampling.

Systematic sampling involves sampling the profiled application at regular intervals. These intervals must be measured in units relative to the profiled application. For example if systematic sampling is carried out with respect to execution time, the intervals must be measured in local execution time of the profiled program, not in real time. The reason for this is that many programs may be running, sharing execution time on the system. This gives raise to some technical implementation problems that will be discussed later.

A major advantage with systematic sampling is that the accuracy can be changed by changing the sampling period. By using an adequately small sampling period it is possible

to obtain a representative performance profile by profiling only once. A drawback is that systematic sampling is vulnerable to periodicities. If the profiled program performs some activity only between samples the activity will pass unnoticed.

The advantage of only having to profile once makes systematic sampling better suited for profiling. It's likely that some programs that will be profiled will have very long execution time thus making simple random sampling impractical. Systematic sampling is therefore the choice for this profiler.

4.1.2 Sampling Mechanism

Up to this point systematic sampling has been selected as the sampling method to use, but some mechanism is required in order to determine the sample intervals. The first most obvious that comes to mind is to use time, but modern processors also includes Hardware Performance Counters (HPCs) enabling sampling based on other metrics. Both mechanisms are of interest for the profiler in this thesis.

Time-based Sampling

There are a few different ways of implementing systematic time-based sampling. Three different strategies have been found and explored for use in this profiler.

The first strategy is to have a timer that decrements only when the profiled application runs or when a system library runs on behalf of the profiled program (note the difference to standard real-time timers). Sampling would then be triggered when the timer expires. This method is somewhat problematic because it requires special support from the underlying operating system. Operating systems compatible with the Portable Operating System Interface (POSIX) do provide such a timer, however it's only accessible from the application itself. It's in other words not possible for an external process (the profiler) to use such a timer on another process. One could make use of these timers by instrumenting the profiled application, but since other operating systems including Microsoft Windows does not provide this type of timer it's not a good solution.

Another strategy would be to sample in real-time (using a standard software timer), independently from the profiled application. In order to do this some method is required in order to relate real-time to local execution time of the profiled application. This can be done by weighing each sample according to how much time the profiled application spent executing since the previous sample. It may be the case that the profiled application did not execute at all between two samples. For example, it may have stopped waiting for user input. If the profiler does not weigh the samples it will put as much value to instructions not it execution as those in execution.

In order to calculate the weight of a sample the process must be timed. Timing a process; that is obtaining the exact accumulated execution time at any given time can be exploited to break cryptographic systems [17]. In a timing attack the attacker tries to break a cryptographic system by measuring the execution time of various parts of a cryptographic algorithm. Because of this obtaining a process's accumulated execution time is prohibited

on many operating systems (Microsoft Windows is an exception). This restriction makes it difficult to attribute weights to the samples making this sampling strategy unsuitable.

A third strategy is to use system-wide sampling. System-wide sampling is a technique often used by kernel profilers [8][28]. It involves using real-time sampling, but instead of sampling a single process every process is sampled. This makes it possible to detect if a process of interest is not executing since the profiler in that case would hit another process when sampling. This solves the timing problem of the previous strategy.

While system-wide profiling does not require the same type of operating system timing facilities as the previous strategies it's more dependent on the hardware. The reason is that standard software timers usually cannot be used for this type of profiling. When a software timer interrupt occurs there is no way of telling which process that was interrupted. Software timers are synchronous, the process executing before the software interrupt will be the operating system kernel. The solution is to use hardware interrupts. When a hardware interrupt occur the hardware context of the interrupted process will be saved to the stack before executing the interrupt handler. In the interrupt handler the context can then be examined to see which process that was interrupted and which instruction that the processor was executing.

Hardware interrupts can be generated by for example the Real-Time Clock (RTC) circuit. A RTC circuit is present in almost every computer system that needs to keep time. The Microsoft Windows NT kernel has an API for accessing a somewhat limited profiling timer based on the RTC. This API is not available to the public and no official API documentation exists. The API only allows the instruction pointer to be sampled so it's only usable for generating flat profiles. Because of this the RTC will have to be accessed without using the mentioned API. Unfortunately the RTC is reserved for internal use through the API making RTC support more difficult to implement on Microsoft Windows than for example on GNU/Linux.

Due to the requirement on platform portability the only applicable strategy for the profiler in this thesis is system-wide sampling. There is no need to profile all processes, only one so the sampling can be optimized by ignoring all samples that are not in the process of interest.

HPC-based Sampling

For at least ten years processors have included built in hardware performance counters that can be used for measuring more advanced properties of software being executed on it [13]. In computer hardware HPCs are implemented as set of specialized registers that can be configured to increment on certain events. For example the registers can be incremented on cache-misses, bus accesses, execution of specific arithmetic operations and much more [19].

One thing that makes them very suitable for sample-based performance profiling is that they can be configured to raise interrupts on overflows. For example, if the profiler presets a HPC register to -1000 the processor core would raise an interrupt when 1000 events have occurred. When an interrupt occurs the profiler attributes one unit to the process

that was interrupted. The HPCs could of course still be useful without using interrupts but in that case they would have to be used in combination with time-based sampling. Reading and storing away the value of the HPC registers when each time-based sample is taken. This would not be efficient, not from a space nor performance perspective.

It should be noted that because of out-of-order execution the values produced by the counters should be considered approximate. A counter increment cannot be 100% tied to one specific instruction [18]. However in a new technique called Instruction Based Sampling (IBS) individual instructions are tagged and monitored throughout the processor pipeline in order to attribute exact statistics to each individual instruction. IBS is currently only supported in the recent generations of AMD's processors [18].

Despite not being entirely accurate HPCs are still very useful, being used in modern performance profilers [8][9]. A problem with HPCs is that they're difficult to use. The implementation of the HPC registers varies greatly between processor architectures and models. They are also typically not accessible from userland.

4.1.3 Sample Data

This section tries to establish what information a sample should contain in order to fulfill the requirements of analysis and also how to obtain it.

Sample Contents

In its most basic form sampling involves reading the value of the instruction pointer register. This register contains the memory address of the instruction that was executed in the program of which the sample was taken. In addition to this it's also of interest to know in which context the instruction was executed. Only knowing the instruction is rarely satisfactory since there is no way of telling why that instruction was executed. One instruction may have high execution frequency in one context, but not in another.

For example, assume that the profiler has found an instruction with a very high execution frequency. Then the high execution count is the result of either repeated calls from within the procedure hosting the instruction or by any parent procedure making repeated calls to its children. To better understand which, the profiler needs to analyze the chain of procedure calls leading up to the sampled instruction.

To conclude, each sample should in addition to the instruction pointer be complemented with information describing the call chain leading up to the sampled instruction. The call chain can be described through a list of memory addresses that can be tied to specific procedures. The memory addresses do not have to point to the beginning of a procedure but can point to any instruction inside them as that is sufficient in order to locate the procedure later on.

Tracing the Procedure Call Chain

For instrumenting profilers tracing procedure calls is easy due to the nature of their design. By instrumenting each procedure entry and exit the profiler can obtain all information that is necessary to keep track of procedure calls. Sampling profilers have on the other hand no natural way of tracking procedure calls; partially due to the fact that certain procedure calls will pass completely unnoticed causing gaps if tracking the call chain.

In order to determine the call chain leading up to a specific instruction it's necessary to walk the program's call stack. The call stack stores all active stack frames. Each stack frame contains information about a procedure call. When a new procedure is called a new stack frame is added, when a procedure returns the stack frame is removed. By traversing the stack frames on the call stack when an instruction is sampled it's possible to retrieve the call chain. The stack frames must be located immediately, before the profiled program continues its execution since the execution may alter the call stack.

Unfortunately stack frame layouts are both processor architecture and compiler calling convention dependent. There is no universal way of traversing the call stack in order to find stack frames. In some cases it's not possible to locate stack frames without using additional compiler generated or operating system specific information [25]. As a result of this the stack walking cannot be made generic but needs a unique implementation depending on both processor architecture and operating system. Because x86 is a very common processor architecture and also the primary development platform of this profiler. The technique for locating stack frames on this architecture will be described in detail.

The content on the call stack depends on the calling convention used by the compiler. Different calling conventions exists for x86 but CDECL (C declaration) or small variations of it is the most common because it supports the semantics required by the C programming language (for example variable argument lists) [20]. Since this profiler targets C and C++ applications, CDECL is assumed to be the calling convention used in all x86 programs to be profiled. CDECL is the default calling convention used by both GCC and Microsoft Visual C++ [23].

It's important to note that although CDECL is default a different calling convention can manually be selected, either by using compiler optimization flags [22] or by using a compiler specific intrinsic [21]. If CDECL is not used the profiler may fail to traverse the call stack. The profiler only depends on a very few features of the CDECL standard so it's possible that other calling conventions will work, as long as they do not violate the stack frame relationship as discussed later.

A typical (small variations are allowed with CDECL) x86 stack frame contains the function parameters; local function variables and also the value of the stack base pointer register (EBP). The value of the EBP register refers to the stack base address in the active procedure. When a new procedure is called, its prologue will update the EBP register to contain the address to use in the new procedure.

Before updating the EBP register the current value must be pushed to the stack so that it can be restored later when the procedure returns. This is done by the procedure's prologue as well. Interestingly the EBP value is always the first value pushed to the

stack. Since the EBP value points to the beginning of a local stack it means that it points to the very location of the previous EBP value. This forms a chain of EBP values on the program call stack.

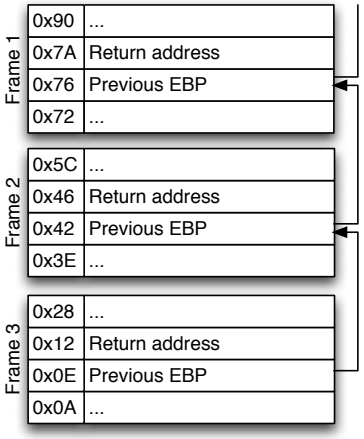

Figure 4: Stack frame organization on call stack.

Figure 4 shows how the call stack frames are organized. The precise contents and process of adding and removing stack frames is omitted from this report (see [20] for details). It's only essential to notice the relationship between the frames. As hinted by figure 4, the chain of stack frames can be traversed like a linked list using the EBP values. For example the following C structure can be used:

```
typedef struct stack_frame_
{
    struct stack_frame_ *next;
    void *return_addr;
} tstack_frame;
```

As shown in the above structure, the return address is located right after (or before depending on how you see it) the EBP value and can thus easily be accessed. The return address provides crucial information. The EBP values themselves are only usable for traversing the call stack; they do not tell anything about which program procedure that the stack frames instantiates. To associate a stack frame with a procedure an address inside an executable binary image is needed. The return address provides this very essential information. It is not an address to the call stack but to an instruction in a binary image (like the value in the instruction pointer register).

Figure 5 shows in a simplified way how program memory may be organized [24]. The program and library binary images are the executable files loaded into memory. One can think of these as EXE and DLL files on Microsoft Windows. Notice the difference between the EBP value and the return address as shown in figure 5.

A difficulty with walking the call stack is to know when to stop. A certain frame structure is assumed but it's not possible to tell when it ends. In order to avoid problems and collect

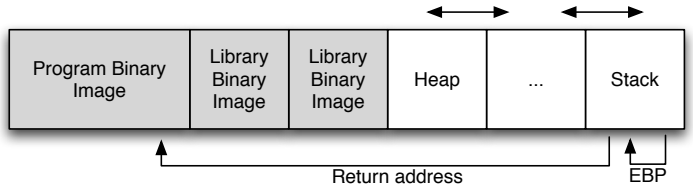

Figure 5: Simplified illustration of program memory organization.

as little garbage data as possible the following checks are proposed for testing the validity of a stack frame:

- Check that the frame points to a higher address than the address of the previous frame. This assumes that the stack grows towards lower addresses. If not, the opposite check should be performed. If the next frame is located backwards in the traversal order the chain is broken.
- Check that the frame address is word aligned, that is 4-byte alignment on 32-bit x86 processors. For example if the address of the next frame points to an odd address it can be discarded as being outside the call stack since the stack frames are word aligned.
- Use any means provided by the operating system kernel in order to validate the memory region that the frame address points to. For example this may include a check to see that the memory region is within user space range.
- Use a fixed limit to the maximum number of stack frames to traverse. This is needed since invalid stack traces may be inconveniently long if some invalid new stack frames satisfy the above conditions.

The above checks will not eliminate the problem of including invalid addresses in the list of trace addresses but it will at least make them fewer. Invalid addresses will likely not translate into a procedure symbol thus being ignored when analyzing the sample data. Nevertheless, it's still desirable to keep the number down in order to reduce the data throughput.

By walking the call stack the profiler can obtain a list of return addresses; addresses that are associated with exactly one procedure in the profiled program or any of the libraries that it use. How the return addresses are translated into procedure names is explained later in this report.

A procedure call trace provides valuable information on why a certain procedure is executed, but it does not alone provide enough information in order to produce a call graph or call context tree. When producing call graph or call context tree profiles it's necessary to know which nodes that has been visited between samples. Consider the following scenario: The profiler have found two samples with identical procedure traces as shown in figure 6.

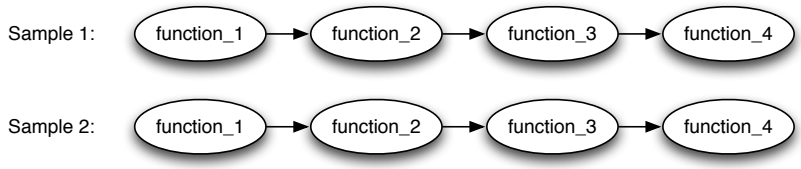

Figure 6: Two identical call traces.

Given only the two samples it's impossible to know what have happened between them. For example, have function_3 called function_4 again or have function_2 perhaps called function_3 again? Maybe the control flow never left function_4 because of some heavy operation inside it. Knowing what have happened between samples is essential for constructing a call graph or call context tree profile because their arcs must be individually weighted.

To solve this problem each procedure invocation needs to be flagged upon visitation in order to know which procedure calls that has been visited and which have not. [11] suggests an efficient way of doing this, without inserting additional instrumentations. Their solution involves using the least significant bit of the procedure's return addresses; a bit that is ignored on some processor architectures.

For the profiler in this thesis, given its intended usage it has been decided that simply knowing the traces is enough, building a correct call graph or call context tree is not necessary.

4.2 Data Management

When a sample has been taken it needs to be processed and stored away in an efficient manner. This section discusses how to manage the collected sample data.

4.2.1 Data Flow

There are certain restrictions, preventing sample data from being processed and analyzed at the time and place of collection. Virtual memory presents a barrier that will enforce a certain data flow through the profiler.

Many modern operating systems separate virtual memory into kernel space and user space. The kernel space is strictly reserved for use within the operating system kernel and its extensions whereas the user space is reserved for all user mode applications [32].

The profiler in this thesis will perform sampling in kernel mode in order to access the interrupt system, but at some point the sampled data will need to be transferred from kernel space into user space. As a result of this the profiler can be seen as two separate components: a kernel mode driver and a user mode client. The client will interact with the driver, processing sample data collected by the driver in order to produce a profile. This is illustrated in figure 7.

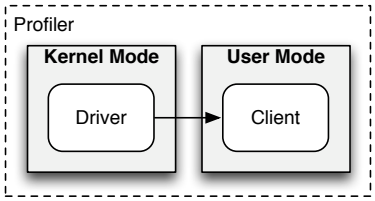

Figure 7: Core profiler components.

There are two possible options for transferring data from kernel space into user space. The first is to make the driver write directly to the user space memory of the client. The second is to make the client read kernel space memory. While it's not possible for a user mode application to read kernel space memory directly, one can create mechanisms (such as virtual file system nodes on Linux) for copying data from kernel space to user space.

The advantage with the first approach is better performance. The sample data only needs to be written to memory once. With the second approach data needs to be written twice. The driver first writes to kernel memory and then the client copies it to user space memory. The advantage with the second approach is a weaker dependency between the driver and client. With the second approach the client is dependent on the driver, but the driver is not dependent on the client. With the first approach both the driver and client are dependent on each other.

The second approach is favored as the choice for this profiler for several reasons. First of all, weaker dependency means that the driver will be easier to use and implement. Furthermore, making a kernel mode driver dependent on a user mode program is bad from a security perspective. The driver would need to include protection mechanisms to ensure that it will not write to places in memory where it is not supposed to. Since the writing operation is asynchronous from the client perspective one can think of several dangerous scenarios, for example if the client program suddenly crashes or if a malicious program tries to trick the driver into writing to another process in order to change its behavior.

4.2.2 Data Storage

This section discusses the issues and proposes a solution for storing sample data during the profiling operation.

Overview

Both performance and memory usage must be taken into account when deciding on how to store away sample data. The performance requirement comes from the fact that sample data is processed in an interrupt routine. The interrupt routine may block other code from executing, including other interrupts. Because of this the code executed in the interrupt routine must be kept to a minimum, otherwise the original system behavior might not be preserved.

The size requirement is implied by the performance requirement. If there wasn't a performance requirement all sample data could be written to a hard drive, but since there is a high performance requirement the sample data must be kept in the computer memory (at least initially as we will see later). Today's hard disks are too slow and also dependent on interrupts [33] which makes them unusable in interrupt routines.

To get an idea of the amount of data that can be collected, consider the following bad but still reasonable scenario. Assume the profiler is running on a 2 GHz 64-bit processor where the profiled program utilizes nearly all processor capacity for one minute before it terminates. If the profiler takes one sample every 50 000th clock cycle the profiler will accumulate 40 000 samples every second. Given an average stack trace of ten addresses for each sample, the size of one sample will be $(10 + 1) \cdot 8 = 88$ bytes large. This will result in 3.35 MiB ($3.35 \cdot 1024^2$ bytes) of sample data every second.

For a program that's being profiled for minutes or possibly hours the data will quickly reach very high levels. Storing the sample data in memory using a dynamic structure (for example array, call graph or context call tree) that grows when needed is a bad idea since the profiler shares memory with the profiled program. The dynamic aspect is especially bad since it will cause the profiler to acquire more memory (that could potentially be used by the program being profiled) the longer the profiling session runs. In other words, the accuracy of the produced profile will decrease with the length of the profiling session.

A better approach is to use some sort of fixed size buffer. The obvious problem with fixed size buffers is that they will likely not be able to store all sample data. Two different approaches have been explored in order to solve this problem.

The first approach is to use granularity buffers. A granularity buffer can be seen as a fixed size hash map where keys producing the same hash value are assumed to be equal. In this context the keys would be memory addresses and the value the number of hits paired with the procedure trace for that address. The key property of a granularity buffer is how the keys are hashed. They are hashed in such a way that keys close to each other will be hashed to the same value, hence the name granularity. This is accomplished by shifting all keys to a lower value according to a preconfigured shift factor.

For example, consider a shift factor of two on the memory addresses: 0x04, 0x06 and 0x10. Shifting the addresses two steps towards a lower value would produce the values: 0x01, 0x01 and 0x04. As the example shows the first two addresses will be seen as the same address (getting two hits). In fact, all values within $2^{shift\ factor}$ addresses will be grouped together. The hashed (shifted) addresses will correspond to a slot in the granularity buffer. Restoring the address from a slot number is simply a matter of shifting the slot value in the opposite direction. The advantage of the granularity buffer is its size which can be expressed through the following equation:

$$buffer\ size = \frac{image\ size}{2^{shift\ factor}} \cdot (trace\ depth + 1)$$

Notice the use of image size in the above equation. The image size represents the size of a binary image for example an executable program or a system library. The reason for using the image size instead of the total address space is obvious. Mapping the entire

address space into a granularity buffer would require a huge buffer unless a very large shift factor is used, and in that case the precision would be very low. Instead multiple buffers could be used, one for each binary image. The result is better precision and lower memory footprint.

The multiplication with trace depth plus one comes from the size of the buffer values. Each address is associated with trace depth number of addresses plus one element for storing the address hit count. The trace depth will not necessarily be for any of the addresses that are mapped into the buffer but for the address corresponding to that specific slot. At first glance it might seem wasteful to allocate space for 16 trace addresses for every slot. One could argue for storing the traces in a separate compact structure instead; for example in a hash map. Unfortunately this type of data structure relies on dynamic memory allocation which is unsuitable. This will be discussed further in the buffer implementation section.

To better understand how much memory granularity buffers require in practice consider the following example: A fairly small application have mapped about 10 MiB of memory (including libraries). If using a zero shift factor and a trace depth of 16 addresses about 160 MiB of memory is needed. If increasing the shift factor to two only 40 MiB is needed. This is all good, but what if the application has a larger memory foot print, for example 100 MiB. In that case a shift factor of five or six may be needed to keep the buffers small enough. A shift factor of six corresponds to 64 addresses. This level of precision is too low, multiple loops or procedures may be grouped together. Granularity buffers are better suited for flat profiling, reducing the buffer size by a factor of 16 in this case.

A completely different approach is to use a standard fixed size buffer that can overflow. Because it's not reasonable to assume that all data will fit in buffer there are two options: either to discard data when the buffer is full or to move the data to the hard drive. As mentioned earlier data cannot be written to the hard drive directly, but one can let another separate process continuously move data from the memory buffer to the hard drive. Since this process is independent from the sampling process it does not have the same performance requirements. This approach fits perfectly with the driver – client model suggested in section 4.2.1. The driver will temporarily store sample data in a fixed size intermediate memory buffer while the client pulls data from the driver's buffer and stores it on the hard drive. The sample data will not be further analyzed until the profiled application is done executing.

Comparing the two buffering techniques it can be concluded that the first approach provides better performance while the second approach provides lower memory usage and better precision. The second approach is favored for its advantages and will be used in this profiler.

Intermediate Buffers

A suitable fixed size buffer data structure is the ring buffer (also known as circular buffer or FIFO queue). A ring buffer can be implemented using a static memory footprint and provides a fast way to both insert and remove data (constant asymptotic time). Another

essential property is the fact that ring buffers can be implemented to support a concurrent environment without using locks. Standard locking mechanisms cannot be used in this case because of the interrupts.

Modern processors along with supporting operating systems use an interrupt priority system [32]. This system allows higher priority interrupts to interrupt lower priority interrupts that are currently being processed. If an interrupt occurs that is of a lower or equal level than the one currently being served it will be queued or ignored. Both interrupt producers suggested in this thesis will raise high priority interrupts. The RTC circuit will generate interrupts on the same level as the clock mechanism controlling the operating system kernel and the HPC registers will generate interrupts of the highest possible level.

To understand the complication of standard locking mechanisms, consider the following scenario: The client process wants to read data from the shared buffer, and in order to do so it acquires a lock on the buffer. If an interrupt occurs before the client releases the lock, the driver will wait for the lock to be released when trying to write to the buffer. The lock can never be released since nothing is able to execute on the processor until the interrupt routine finishes, which it never will do. As a result the system can become completely unusable.

The problem is that conventional locking mechanisms assume that there is a scheduling operation running in the background that is switching between tasks. If for example one thread is sleeping, waiting for a lock to be released, the scheduler will schedule another thread for execution. Eventually the thread holding the lock will be scheduled and the lock will be released. When dealing with high priority interrupts (as in the profiler in this thesis) there will be no scheduling operation in the background. There is no possibility for other tasks to execute and release any locks if the interrupt routine is waiting.

Ring buffers can fairly easily be implemented to be lock-free assuming there will be only one producer (pushing data) and one consumer (pulling data) [29]. There are also more complex lock-free implementations for use in other situations when there are multiple consumers and/or producers [30].

When using time-based sampling, there will be only one producer and one consumer. However, if using HPC-based sampling there can be multiple producers because in a multi-core system each core has its own HPCs configured to raise interrupts. It's not possible to only enable HPC-based sampling on one core and expect the profiled program to execute on it. A process may spawn on any processor core and the operating system kernel may choose to move the execution of a process from one core to another. The producer – ring buffer relationship is illustrated in figure 8. Having multiple producers requires a different solution than just the simple lock-free ring buffer.

One solution is to use a ring buffer implementation that is both lock-free and supports multiple producers, but these tend to be difficult to implement and have noticeably slower performance than ordinary ring buffer implementations. Another solution is to have individual ring buffers for each core, connecting them to a common ring buffer through a synchronization mechanism. This ensures a single producer and consumer for each buffer. The buffer synchronizer will act as the consumer of all core/processor buffers

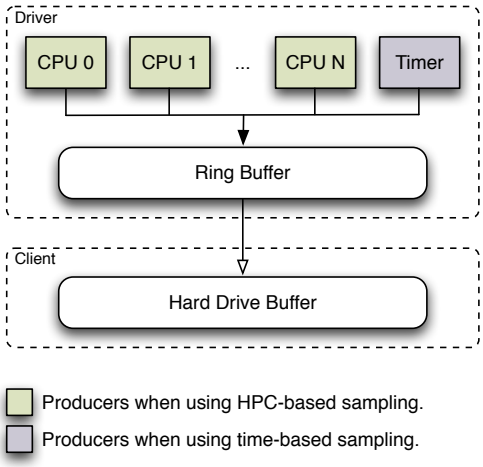

Figure 8: Organization using multiple producers.

and the only producer of the common primary ring buffer. The latter solution is favored in this case due to easier implementation and better performance. Figure 9 illustrates the proposed solution using multiple ring buffers.

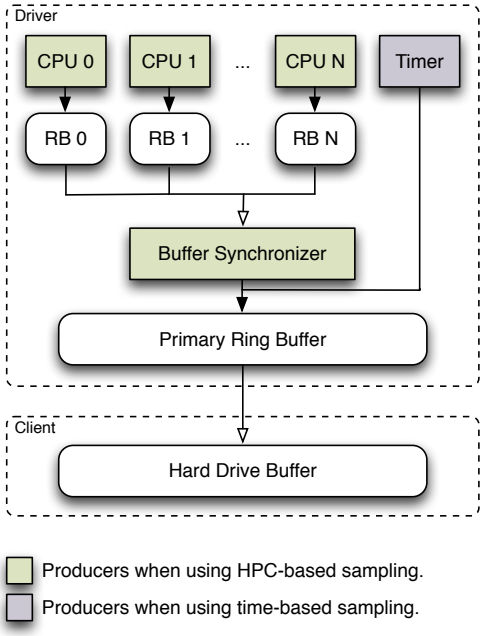

Figure 9: Organization using single producers and consumers.

It should be noted that although the individual performance of the simple lock-free ring buffer may be better than that of the complex one, the total performance including the synchronization mechanism may actually be slower in total compared to using a complex lock-free ring buffer. However, the simple lock-free ring buffer will yield better performance when synchronization is not necessary for example when using time-based sampling.

Buffer Implementation

To this point ring buffers has been suggested as the data structures to use for storing sample data. The next step is to define how samples should be stored inside the buffers. What makes this tricky is the fact that samples can vary in size. For example, all procedure traces will likely not be of equal length.

A common way of implementing ring buffers is to use linked lists. If using a linked list each node in the list can be of different size since separate memory is allocated for each node. This would seem fitting in the context of this profiler, but there are some drawbacks. Linked lists do not have a suitable memory representation, primarily due to the fact that the nodes will not necessarily be stored in sequence in memory. Storing all nodes in sequence is desirable because the data will be moved between the driver and client on a byte for byte basis.

For example, the client will request a certain number of bytes to read from the drivers primary ring buffer. Since the client does not know the size of the next sample beforehand it will likely request too little or too much data. Having the nodes spread in memory forces the driver to map them into sequence during the read operation. Furthermore repeated heap memory allocation and deallocation that is implied by linked lists is both expensive (compared to static allocation) and may lead to memory allocation failure due to heap fragmentation [26].

A better approach is to implement the ring buffers using arrays. The problem with this approach is that all nodes need to be of the same size. If using one node for each sample all nodes would have to be large enough to hold the largest possible sample. This would waste memory on samples that are small, not utilizing all the reserved space. Instead of using one node for each sample the nodes can represent smaller building blocks, for example memory addresses. In order to apply this solution there needs to be a protocol specifying the contents of the nodes so that multiple nodes can be related as one sample (a sample can contain multiple addresses). If the protocol is carefully designed it can be expected that that solution will utilize available memory more efficiently than the one node per sample solution.

The protocol used in the profiler system in this thesis is explained in the next section.

Buffer Protocol

There are two purposes of the protocol used within the sample buffers. The first is to group address together in samples. One node holds one memory address, multiple addresses form one sample. The second purpose is to transport additional information from the driver to the client; information that cannot be obtained by the client itself. For example the driver may monitor the profiled application to see when it loads/unloads a dynamic library or when it terminates. On some operating systems this information cannot be accessed from userland where the client lives.

Two methods have been explored in order to solve the first purpose of grouping addresses together. The first is to let the first node of each sample contain the number of addresses

in the sample. That way the client knows exactly how large each sample is. This is illustrated in figure 10. The second way is to reserve two addresses for use as special sample start and stop blocks. All addresses between a pair of these blocks would group a single sample. Samples with only one address can be represented without any start or stop blocks. This is illustrated in figure 11.

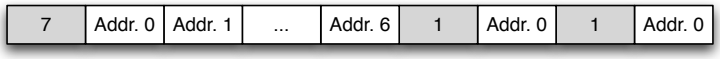

Figure 10: Buffer protocol strategy 1.

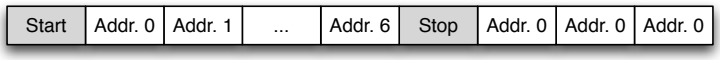

Figure 11: Buffer protocol strategy 2.

The first method will have less memory space overhead for samples larger than one address while the second method will have less overhead for samples consisting of exactly one address. The fact that when generating flat profiles all samples will be exactly one address large favors the second method. For flat profiles the first method would have a constant 100% memory space overhead compared to 0% overhead of the second method.

One could argue for using two different protocols for generating flat and trace-based profiles but that would increase the coupling between the client and the driver. The client must (at the time of reading data) know if the driver performs tracing or not. Furthermore it makes it difficult to mix flat and trace-based samples in the same buffer. This may happen if tracing is only enabled in certain parts when profiling a program. Finally, single address samples may occur even when tracing is performed.

Using the second method requires two addresses to be reserved for indicating sample start and stop. The reserved addresses should be picked in such a way that it's very unlikely that those addresses will occur in a sample naturally. While it's possible to analyze the virtual memory space in order to find unused addresses it's probably not worth the effort. Since the virtual memory space may change the reserved addresses would have to be dynamically updated over time.

Instead it's suggested in this thesis to reserve either the lowest possible or highest possible addresses for this purpose. By investigating the virtual memory region maps on different GNU/Linux systems, no process has been found that maps any of these extreme address regions. On GNU/Linux systems the top address space is reserved for kernel data and the lower for the user process [31]. Since the kernel's behavior is harder to predict than that of a user process the lower memory addresses has been selected for the profiler in this thesis.

In addition to the reserved addresses for indicating the beginning or end of a sample more addresses will be reserved for transferring other notifications as described earlier in this section.

4.3 Data Analysis

In order to generate a meaningful profile the sample data must be analyzed. This section deals with important key steps in the context of post processing the sample data.

4.3.1 Memory Address Translation

Memory addresses are seldom useful as they are but needs to be translated into something meaningful like procedure names or source code files with line numbers. This translation can be described through a series of different steps that will be covered in this section.

Binary Image Lookup

Before a memory address can be tied to a specific symbol, the binary image that the address belongs to must be located. Figure 12 illustrates how sampled addresses may be distributed across the memory space.

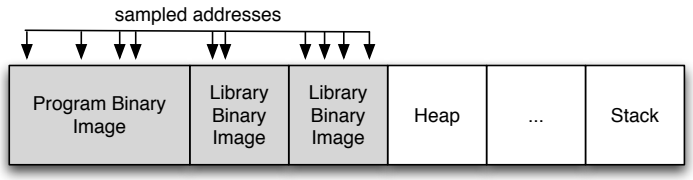

Figure 12: Program sample distribution.

In order to translate an address into a binary image the profiler must maintain a map of all relevant binary images, at which address they are located and how large they are. It's also important to note that new images can be mapped and unmapped to/from the programs memory space during its execution. The profiler must be alert of such changes or it will map an address to the wrong binary image. Luckily it's not necessary to validate the memory mapping every time when taking a sample because the most common operating systems provides means of receiving notifications when an application maps or unmaps a binary image to/from its memory space.

Symbol Lookup

Debug symbol names are names that describe a certain construct in a compiled program. For example procedures and global variables.

Given a binary image and a memory address the profiler must be able to locate the debug symbol name associated with it. In order to do this, symbol information must be included in the binary image itself. On operating systems using the Executable and Linkable Format (ELF) symbol information is often available in binary images by default. In situations where no such information is available it can usually be included by specifying a compiler option.

In ELF and similar formats, procedure symbol names can be accessed as a list of memory address – symbol name pairs. Where the memory addresses refer to the beginning of a symbol. Having this information it's fairly easy to find to which procedure a memory address belong. The address belongs to the last symbol occurring before the memory address.

Unfortunately ELF is not the only format for storing symbol information. While ELF is dominant on UNIX flavored operating systems other operating systems including Microsoft Windows have their own formats. The profiler in this thesis will require unique symbol lookup implementations for each operating system that is to be supported.

For programs written in C the symbol names will match the procedure names exactly. However, in C++ programs the symbol names have been encoded using a technique called name mangling. The reason for this is that in C++ there can be multiple procedures using the same name. To be able to distinguish between the different procedures additional information such as argument types and class names are encoded into the symbol name. As a result the profiler must be able to demangle symbol names before presenting them to the user. As with the formats for storing symbol information the name encoding also differs across operating systems.

Debug Information Lookup

In some cases symbol information is not satisfactory. In order to translate memory addresses into something better than procedure level accuracy (for example source code files with line numbers) special debug information must be present in the binary image. This information is rarely included by default because it adds a significant amount of data to binary images, causing them to sometimes grow by multiple factors in size.

As with symbol information, debug information is also available in different formats. On GNU/Linux the Debug With Attributed Record Format (DWARF) is very common, while Microsoft uses its own format on the Windows platform. When debug information is present finding what source file and line number that corresponds to a memory address is very similar to finding symbol names.

4.3.2 Profile Generation

The final step is to produce a profile and presenting the information for the user. Two different types of profiles have been considered for the profiler in this thesis.

Flat Profiles

Generating a flat profile is very straightforward given a list of symbol names or more detailed information. It's simply a matter of grouping the samples together and summarizing on the occurrence of different symbols and/or other information. Addresses that cannot be translated into symbol names or other debug information is ignored.

Path Profiles

The term path profile is not an established concept but a name given to a small variation of call trees in this report. Path profiles are essentially call tree profiles but instead of organizing the call traces as a tree they're organized as a list of paths. The paths are attributed an execution count and the leaves are attributed detailed information about call frequencies of individual statements inside the leaf procedure.

Constructing path profiles is a heavier operation than constructing flat profiles. First the paths must be constructed. These are constructed as a list of symbol pointers paired with a leaf node. Invalid addresses that don't map to a symbol name are ignored. The paths are then grouped together, merging the leaf nodes of identical paths. The leaf contains a hash map of memory addresses (accessed in the last procedure in the path) mapped to hit counts.

When presenting the profile the symbol pointers are evaluated into procedure symbol names and the memory addresses in the leaves are translated into source file with line number information.

5 Results

This section presents the results from implementing and evaluating the performance profiler developed in this thesis. In the charts displayed in this section the profiler in this thesis goes by the name nprofile.

5.1 Profiler Implementation

This section explains the profiler implementation results, starting with a general architecture overview followed by detailed information about data gathering, management and processing.

5.1.1 Architecture Overview

The profiler system uses the driver – client model as described in section 4.2.1. The client (from here on referred to as the profiling application) is a standalone application written in C++. The driver is implemented in C because C++ is unsuitable for kernel level development due to lack of run-time support.

The profiling application is developed to be completely portable, using a limited set of the C++ language. It connects to the kernel driver through an operating system dependent interface. The interface acts like a bridge, connecting the profiling application and the driver. This interface is needed because different operating systems provide different means of communication between drivers and userland processes. The interface is developed as a separate library that is linked in with the profiling application at compile or run-time.

The profiling driver is to a large extent platform dependent, but certain parts are shared between drivers on different operating systems. For example sample processing and buffer management.

When compiling the driver the build system will select a processor architecture implementation that matches the system that compiles the driver. The processor architecture specific implementations are not dependent on the operating system so the same implementation can be selected on for example both the GNU/Linux and Microsoft Windows. The processor architecture specific implementations include HPC, RTC and tracing support.

Figure 13 shows how the profiler software packages are organized assuming both GNU/Linux and Microsoft Windows support. Of course either a GNU/Linux or Microsoft Windows implementation is used, they're never used together. It should be mentioned that the interface library also contains all other operating system dependent functionality that is needed by the profiling application. For example process, symbol and debug information management.

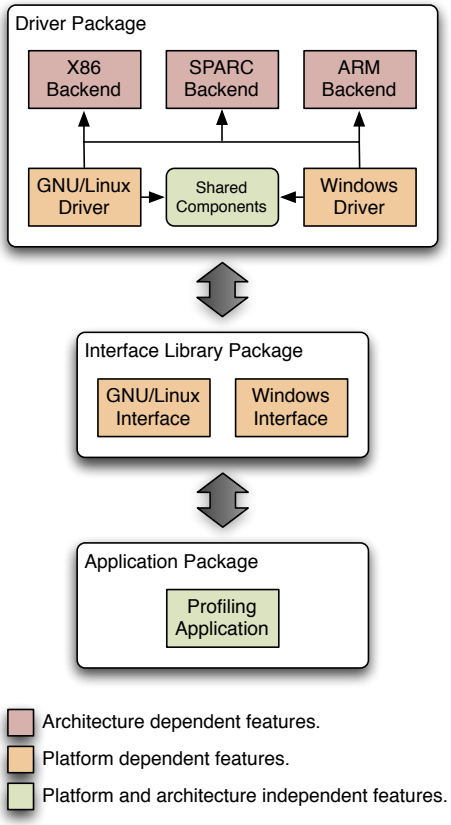

Figure 13: High level architectural overview.

5.1.2 Data Gathering

The data gathering mechanism is defined as a class interface (or rather the C-language counterpart), allowing multiple implementations to be used seamlessly. The GNU/Linux implementation implements two sampling mechanisms. Namely HPC-based sampling and time-based sampling (Linux kernel 2.6 and newer implements a low resolution profiling timer). An RTC-based implementation was omitted partly because the RTC cannot be accessed without recompiling kernel 2.6 and because HPC works well as a RTC replacement.

As each sample is taken, processor architecture specific code will perform the necessary tracing on the call stack (if the profiler is configured to perform tracing). The tracing mechanism is configured to perform maximum 16 trace steps by default. Once done tracing the sample will be written to the appropriate buffer.

The profiler supports configuring different sampling intervals, sampling mechanisms and profile types at run-time before starting the profiling session.

5.1.3 Data Management

As explained in section 4.2.2 the HPC sampling mechanism requires unique processor specific buffers because interrupts prevent standard locking mechanisms from being used. These buffers are implemented as lock-free ring buffers capable of storing 131 072 addresses each. This size includes trace data. Given that all traces will be of maximum depth the buffer will be able to store roughly 8 000 samples. The primary ring buffer is twice the size and thus capable of holding about 16 000 samples.

Depending on the sampling interval, these buffers will be able to hold data from tens of a second to multiple seconds. The size of the buffers have been selected not only for throughput but to be large enough to hold all data from the time that the driver has started the sampling process to the time that the profiling application starts moving samples from the driver to the hard drive. There will be a small delay between these events. If any of the driver's ring buffers overflow, samples will be lost. By selecting large enough buffers this can be avoided.

Once the profiling application has begun moving data from the driver the chance of overflows are less likely. Inside the driver the buffer synchronization mechanism works by moving data from the processing unit buffers to the primary ring buffer in intervals. The lengths of these intervals vary because a fixed interval will not satisfy all cases. For example, the synchronization interval must be shorter when a smaller sampling period is used than when a larger sampling period is used.

It would be possible to use a small synchronization interval in order to cover all sampling intervals, but that is not optimal because a small synchronization interval will prove less efficient for larger sampling periods. If for example using a very large sampling period it may be that each synchronization transfer zero to one sample. In this case the overhead of the synchronization mechanism becomes more apparent.

Instead of using a fixed size interval, the lengths between the synchronization iterations are calculated dynamically. The lengths are calculated in each synchronization iteration so that the next iteration is expected to transfer half of the buffers contents from the processing buffer to the primary ring buffer. If the amount of data suddenly increases or decreases the synchronization mechanism will adapt. Transferring half the contents gives some marginal if the amount of data suddenly increases. It's important to note that even if the sampling periods are constant the amount of data is not because procedure trace depths will vary.

Even though the synchronization intervals are dynamic there are still limitations. The driver's synchronization mechanism works within the limitations of the operating system kernel. On Linux version 2.6 the kernel is configured to run in 250 Hz by default. This means that the buffer synchronization mechanism inside the kernel driver may run at most 250 times per second. It may not be desirable to synchronize each time the kernel runs for the sake of overhead so the synchronization mechanism has been configured to run at most 50 times per second. This yields a maximum throughput of 400 000 ($8000 \cdot 50$) samples per second. This should be enough for the nearest future. Sampling 400 000 samples per second on a 2.7 GHz processor would result in a sampling interval of 6 750 cycles. As the performance evaluation shows in section 5.3 the profiling overhead at this magnitude of sampling intervals is significant.

The lower synchronization limit has been set to five times per second. This cannot be set infinitely low because the synchronization mechanism must be able to quickly respond if the amount of data increases.

The same synchronization algorithm as described above is used when transferring data from the driver's primary ring buffer to the hard drive. There are in other words at two different synchronization mechanisms in use.

5.1.4 Data Analysis

The sample data files stored on the hard drive by the profiling application is stored in a generic format. This means that if an application is profiled in order to produce a path profile, the same sample data file can be used to generate a flat profile.

When the profiled program has finished executing or if a user has signaled the profiler to stop sampling it will produce a readable profile from the sample data. The profiler always tries to locate debug information within an executable but if not present it will fall back on standard symbol information and in worst case memory addresses.

The entire data analysis step is performed by the platform independent profiling application with some aid from the operating system dependent interface.

5.2 Profiler Correctness

This section evaluates the correctness of the profiler, both from a theoretical sampling and implementation perspective.

5.2.1 Sampling Theory

Since sampling is a statistical process, statistical methods can be applied in order to evaluate the correctness of the results. There are two types of errors to expect from the sampling process: Periodic errors due to systematic sampling and random sampling errors. The former type of errors will occur if the profiled program performs some activity in periodic intervals between samples are taken, thus avoiding detection. The only way to counter this type of errors is to increase the sampling frequency and/or performing multiple profiling sessions. This type of error is highly dependent on the population being sampled and is therefore difficult to assess.

Random sampling errors are easier to predict using statistical tools. A good method for establishing confidence in statistical results is to calculate the margin of error. Given a performance profile that specifies in percentages how much execution time that was spent in each individual procedure, the margin of error explains the amount of error that can be expected at one specific procedure estimation. The maximum margin of error defines the biggest error that can be expected for any procedure. This is strongly related to the number of samples taken and can be calculated from the following formula:

$$maximum\ margin\ of\ error = \frac{0.5 z_{\alpha/2}}{\sqrt{n}}$$

n is the number of samples taken and $z_{\alpha/2}$ express the confidence that can be put into the margin of error. The background theory on how $z_{\alpha/2}$ is calculated is outside the scope of this report. For example, a confidence level of 95% yields $z_{\alpha/2} = 1.96$ and a confidence level of 99% yields $z_{\alpha/2} = 2.58$. The confidence level express how reliable the maximum margin of error is, if using a 99% confidence level we can be 99% sure that the maximum margin of error is correct. It's easy to see from the formula that a higher confidence level will yield a larger error margin. In academia a 95% confidence level is generally accepted as a good for most applications and will consequently be used in this evaluation.

$$maximum\ margin\ of\ error\ (95\%) = \frac{0.5 \cdot 1.96}{\sqrt{n}} = \frac{0.98}{\sqrt{n}}$$

For example, if the profiler collects 100 samples the error margin would be 9.8% with 95% certainty. This means that if the profiler has found a routine to be responsible for 30% of the total execution time it can be expected (with 95% certainty) that in reality the profiled program spend between 20.2% and 39.8% of its total execution in that routine.

Increasing the number of samples from 100 to 1000 would give a maximum error margin of approximately 3.1%. Further increasing the number of samples to 10 000 will lower the maximum error margin to slightly below 1%. In the previous example where 30% of the total execution time was found to occur in one specific routine, a 1% maximum error margin would expect 29% to 31% of the total execution time to be spend in that function with 95% certainty.

100 or 1000 samples are very few in the context of pure sample-based performance profiling. For a program running for one second (which can be considered short) this would

result in only collecting 100 or 1000 samples per second. In the evaluation that follows later the sample-based profilers was configured to sample every 100 000th clock cycle. The tests was performed on a 2.7 GHz processor system resulting in the profilers collecting about 27 000 samples every second. The smallest of the test programs executed for approximately 30 seconds resulting in about 810 000 samples in total. This yields a maximum margin of error slightly above 0.1%.

5.2.2 Profiler Implementation

While the sampling method can be evaluated in theory, the profiler implementation must also be evaluated. Evaluating the correctness of the implementation is difficult when there is no fully known reference case to compare to. In order to precisely assess the profiler's correctness it's necessary to manually analyze a program in order to calculate its precise performance given a known set of input data. This is impractical so instead two different programs have been profiled using different performance profilers. The results have then been compared to each other with the intention of finding that all sampling-based profilers produce similar output. If that is the case a certain confidence can be established in the sampling implementation.

Two programs from the SPEC CPU2000 benchmark suite have been selected for evaluating the profiler implementation. The first program is 256.bzip2 (using input.graphic) and the second is 197.parser (using input.ref), both from the CINT2000 set of programs. The programs have not been selected for their individual characteristics but only because they can be expected to stress the system (as any other SPEC CPU2000 program) which is useful when analyzing the profiler overhead later on. The bzip2 program performs repeated data compression and decompression. The parser program performs syntactical parsing of English texts into an internal memory structure.

All profilers except gprof are configured to sample every 100 000th clock cycle and to generate flat profiles. gprof uses a special software profiling timer that has a limited resolution (same as the kernel). gprof samples at 100 Hz by default, changing this value requires modifying the kernel. Because of this the gprof sampling interval has been left to the default. All profilers performed five profiling sessions on each program in order to establish a mean profile. All evaluations have been performed on a GNU/Linux system running the 2.6 kernel. The top ten procedures where the profiled programs spent most of their time have been specifically evaluated.

Figure 14 show the flat profile evaluation of bzip2. Please note that the profiler in this thesis goes by the name nprofile in all diagrams in this report. All profilers show a trend of producing very similar results. The ordering of the top the procedures is almost identical among the profilers with the exception from gprof that ranks spec_getc above qsort3. This can be explained by the lower sampling frequency used by gprof, and the fact that both functions account for only a small part of bzip2's total execution time.

Figure 15 shows the flat profile evaluation of the parser program. In this case there is also a small difference in the procedure ordering. gprof ranks the execution time of match higher than that of xalloc. This can be dismissed with the same arguments as for bzip2.

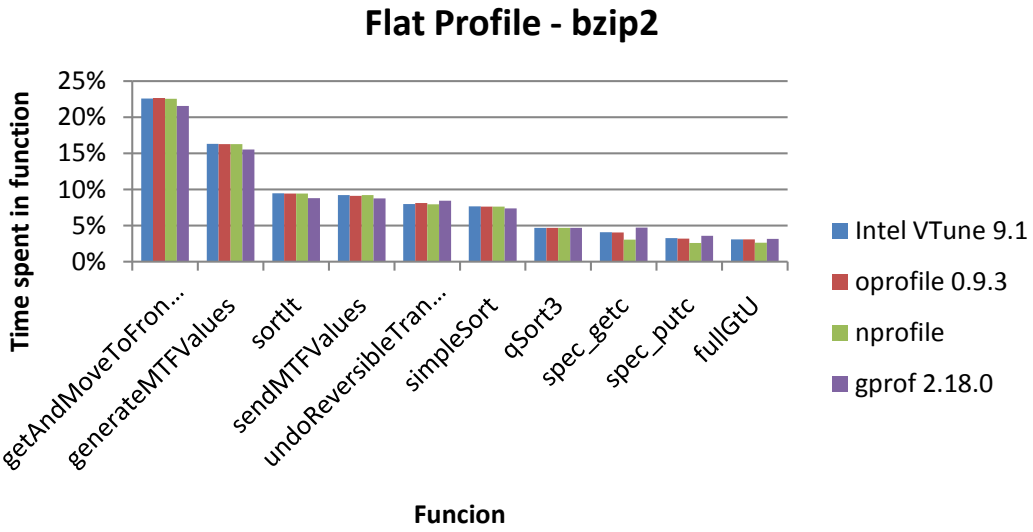

Figure 14: Flat profile of the bzip2 program.

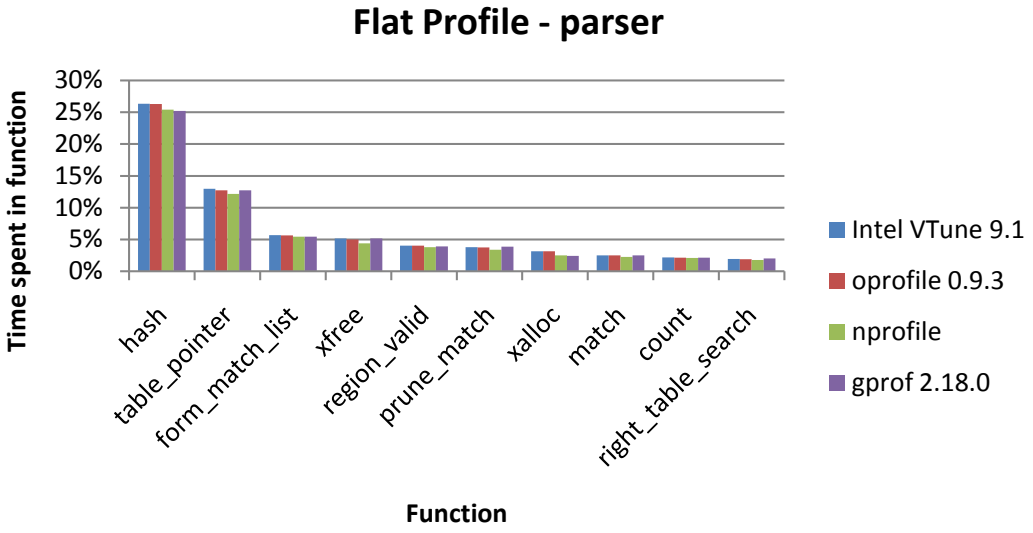

Figure 15: Flat profile of the parser program.

Since each profiler performed five profiling sessions on both bzip2 and parser it's of interest to analyze how much difference there was between each set of results. The standard deviation is a good tool for this. Standard deviation is a measure on how much variation that occurs within the different results.

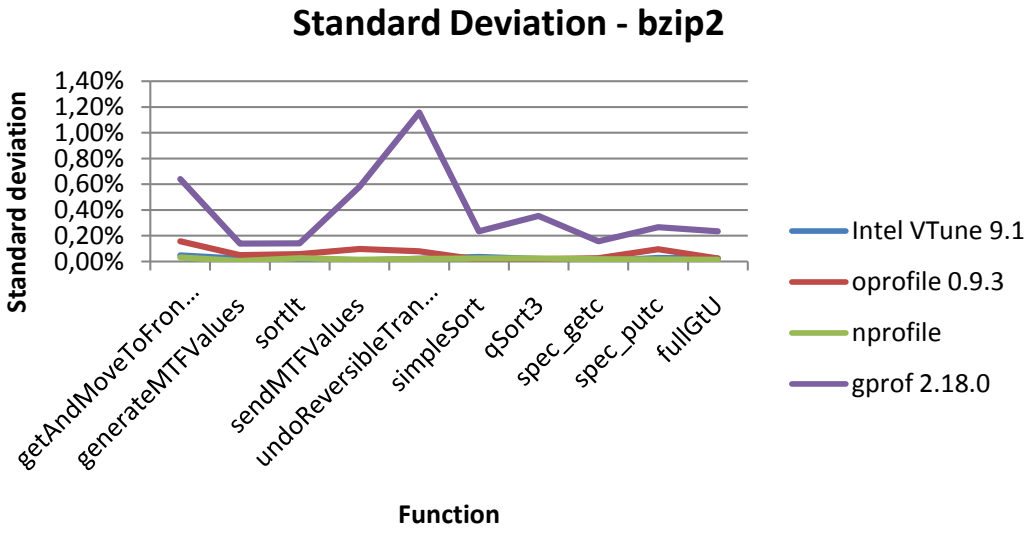

Figure 16: Standard deviation of the bzip2 flat profile.

Figure 16 shows the standard deviation of the top ten procedures in bzip2. Given that approximately 810 000 samples were taken by oprofile, nprofile and Intel VTune, the maximum error of margin is expected to be 0.11%. It's hardly readable from figure 16 but nprofile and Intel VTune keeps well below the margin and oprofile in all cases but for the 1st procedure where it reaches 0.16%. gprof collected about 3000 samples which yields a maximum margin of error of approximately 1.8%. This explains the higher standard deviation of gprof shown in figure 16.

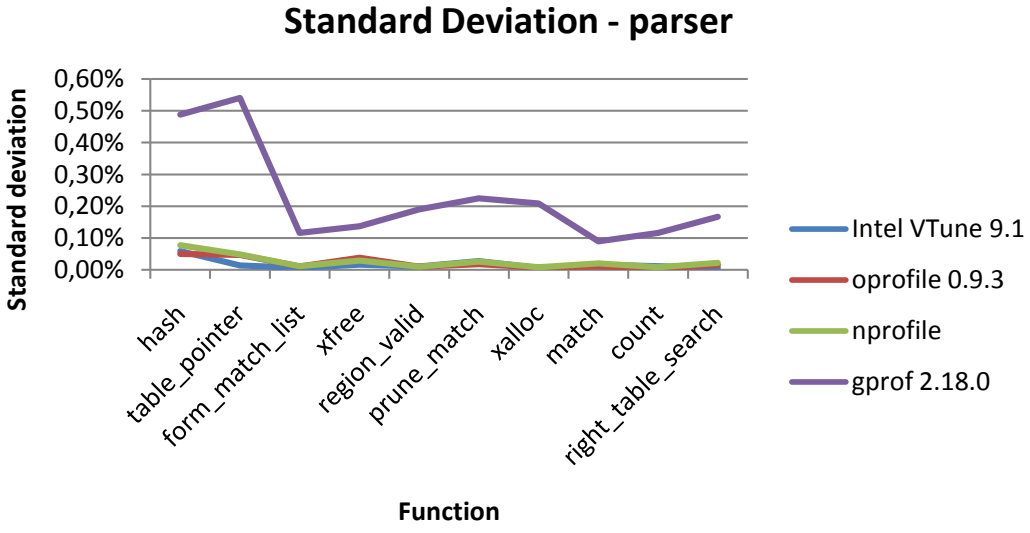

Figure 17: Standard deviation of the parser flat profile.

Figure 17 show the standard deviation for the parser program. The expected maximum margin of error, given approximately 3 672 000 samples is 0.05% (with 95% certainty).

Intel VTune and nprofile slightly crosses this limit for the hash procedure. Except for that Intel VTune, nprofile and oprofile keeps below the margin. As previously stated gprof collects fewer samples which explain its higher standard deviation.

5.3 Profiler Performance

In context of performance profilers, the term performance refers to the execution speed overhead. Another performance aspect could be memory usage but execution speed overhead is what is what is evaluated in this report. Overhead means how much execution time the profiling process adds to the original execution time of the profiled program. The performance evaluation in this thesis uses the same bzip2 and parser programs as in the profiler correctness evaluation.

5.3.1 Measuring Execution Time

Measuring execution time of a program can be difficult, especially if a program competes with other programs on the same system. The programs used for evaluation in this thesis are expected to drain most of the computer resources. Other programs running on the system was kept to a minimum and the test system was not used for other tasks while running the test programs.

The execution time was calculated in real-time in order to account for the profiler's execution time. To measure time, both test programs were extended to include a call to a a function called init_exec_time in the beginning of their main routine and a call to a function called print_exec_time in the end of their main routine. These functions are defined as follows:

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

static unsigned long start_ticks = 0;

static unsigned long ticks(void)
{
    struct timeval tt;
    gettimeofday(&tt,(struct timezone *)0);

    return tt.tv_sec * 1000 + tt.tv_usec / 1000;
}

void init_exec_time(void)
{
    start_ticks = ticks();
}
```

```
void print_exec_time(void)
{
    printf("Executed in: %.3f seconds.\n",
           (float)(ticks() - start_ticks)/1000);
}
```

The function prints the time used since the program was started.

5.3.2 Results

To establish a reference execution time both test programs have been executed five times each in order to calculate a mean execution time without any profiler interference. All other tests using performance profilers have been carried out in the same fashion by performing five tests and calculating a mean. In order to evaluate scalability different sampling intervals have been used. Please recall that the profiler in this thesis goes by the name nprofile in all diagrams in this report.

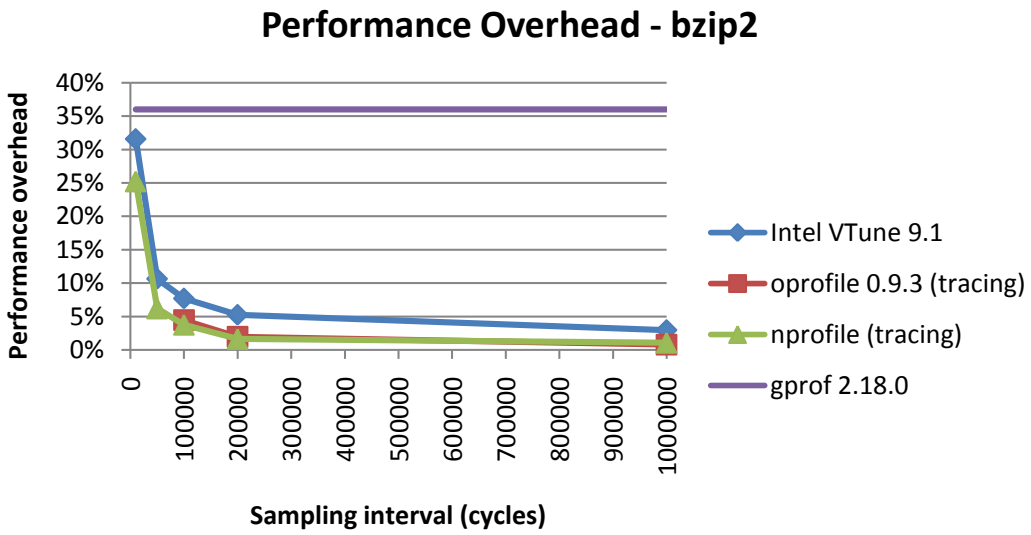

Figure 18: Performance overhead in bzip2.

Figure 18 show the performance overhead when profiling the bzip2 application. A trend can clearly be seen among the sampling-based profilers, having a significantly lower overhead than gprof. An interesting note is that both oprofile and nprofile performs better than Intel VTune although VTune does not perform any tracing.

Figure 19 shows the performance overhead when profiling the parser application. A similar trend can be seen in this graph. The sampling-based profilers perform similarly. As in the bzip2 case, nprofile performs slightly better than oprofile. Apart from architectural and implementation differences a small part of the performance difference to oprofile can probably be explained by oprofile performing system wide sampling, monitoring all processes while nprofile focuses on one application. Intel VTune also performs system wide sampling.

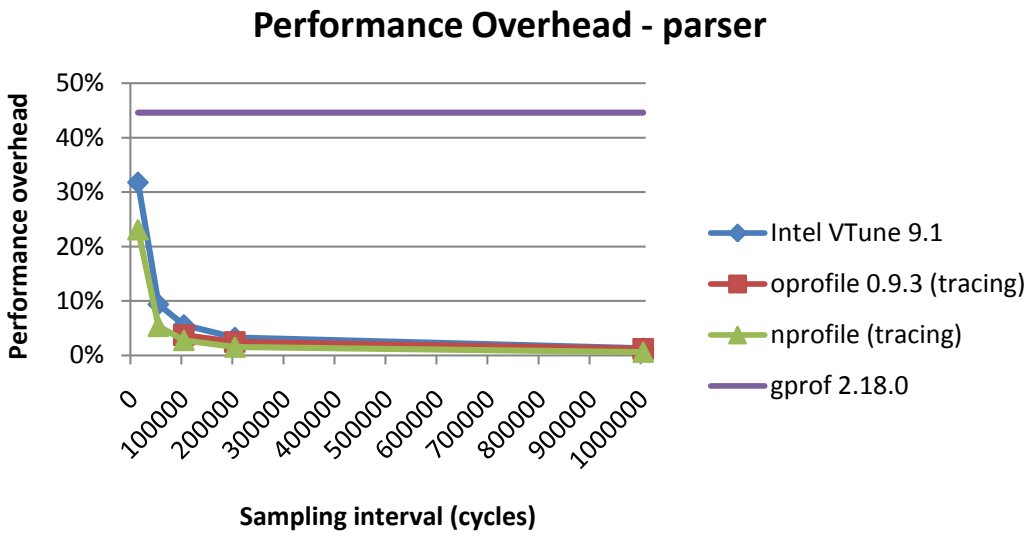

Figure 19: Performance overhead in parser.

It can also be of interest to analyze the impact of tracing the procedure call chain for each sample. In order to do this both nprofile and oprofile have been evaluated with tracing enabled and disabled. Intel VTune and gprof has been omitted from this part since VTune does not support tracing at all (in sampling mode) and gprof always performs tracing but using instrumentation. Figure 20 shows the performance overhead of using tracing.

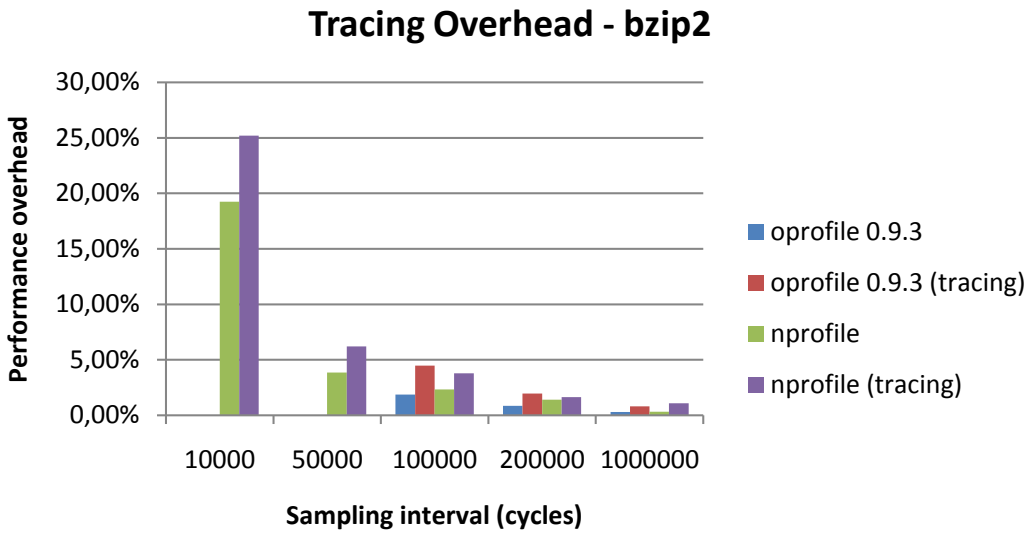

Figure 20: Tracing overhead in bzip2.

As figure 20 shows the profiler overhead increases more dramatically when the sampling period decreases. When using a smaller sampling period the tracing overhead becomes more apparent. Compared to oprofile, nprofile seems to have a higher overhead when tracing is not enabled or when a large sampling period is used. This and the lower overhead when tracing suggests that nprofile's tracing mechanism is more efficient than that of oprofile.

As figure 21 shows the tracing overhead is not as apparent in the parser application and

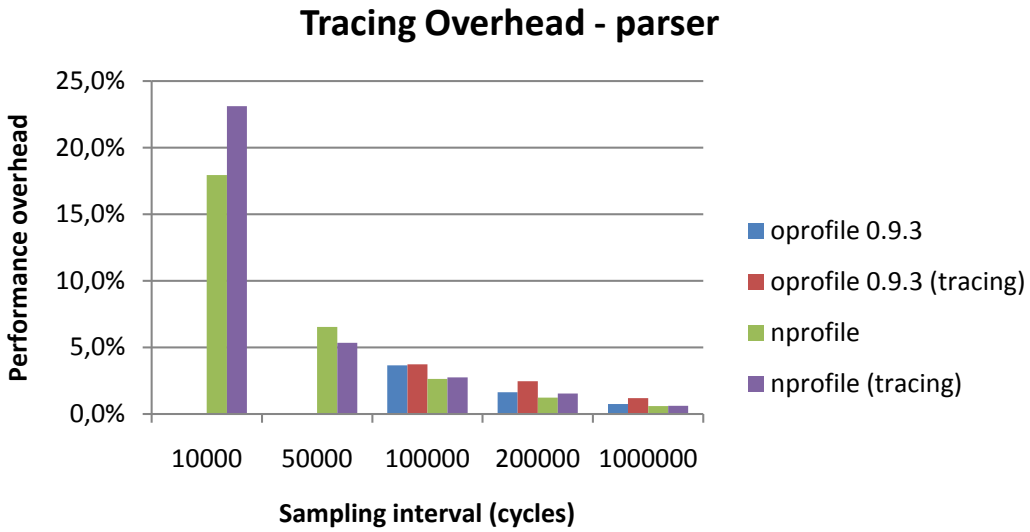

Figure 21: Tracing overhead in parser.

that is likely because the parser application does not produce as long traces as the bzip2 application. Compared to the trace overhead in bzip2, nprofile performs noticeably better than oprofile in all configurations. This is surprising given that nprofile performed worse than oprofile in some situations in the bzip2 case. One possible explanation for this is that the parser program has a different memory profile than bzip2, colliding with that of oprofile, introducing system stalls through cache misses for example. Another possible explanation is that oprofile does not scale as well as nprofile. The parser program has about four and a half times longer execution time than bzip2, thus producing significantly more sample data.

6 Discussion

6.1 Summary

One of the biggest challenges of developing this profiler was not the fact that it should profile sequential C and C++ programs but the fact that it should be portable across operating systems and processor architectures. Of course, a profiler targeted towards Java or .NET programs would be much less sophisticated and require less effort (from my part) since these programs run in a very controlled environment. Both the official Java and .NET virtual machines provide well defined interfaces for profiling [34][35].

The difficulty of developing a portable C and C++ software profiler greatly depends on the method for data gathering. Source code instrumenting profilers can be implemented as completely processor architecture independent. In contrast, the other techniques such as binary instrumentation, sampling and simulation in varying degrees depend on the underlying processor architecture.

The simulation strategy was ruled out almost immediately when evaluating the different methods. Simulating an entire processor instruction set will introduce too much work when extending such a profiler with support for new processor architectures. Instrumentation was also ruled out, primarily due to its problem with biased overhead on small procedures compared to larger ones. Source code instrumentation was briefly considered due to its attractive property of being processor architecture independent.

Compared to sampling, instrumenting profilers are expected to introduce a different magnitude of overhead in the profiled application. This is very much supported by the performance comparison in this report. One should also keep in mind that the instrumenting profiler gprof that was compared is not purely instrumentation based but actually uses sampling to measure execution time. A purely instrumentation based profiler can be expected to introduce even more overhead than gprof.

Despite being an approximate process the theoretical evaluation shows that sampling is very accurate as long as a significant number of samples are taken. Only 10 000 samples are required to achieve a maximum margin of error slightly below 1%. In this context 10 000 samples are a few. The performance evaluation shows that a sampling rate of over 20 000 samples per second (100 000 sample interval on 2 GHz processor) is no match for current systems. This means that under normal conditions the profiled program can be so short that it finishes in half a second and the resulting profile is still very accurate. Considering that the programs that this profiler targets (sequential, heavy on computing) will have an execution time of at least multiple seconds, a very high confidence can be put into the resulting profiles.

A major drawback with sampling is that introduces a strong dependency on the underlying operating system, probably a stronger dependency than most of the other profiling techniques. On open source operating systems such as GNU/Linux this is generally not a problem, but on closed operating systems such as Microsoft Windows it can be. The hardest part with the project analysis was to understand how sampling could be implemented in Microsoft Windows. At times it was even a question on if it could be (with

reasonable effort).

On Microsoft Windows the hardware resources that are required for sampling seems to be allocated and reserved for internal use. This is however not 100% confirmed since practically no documentation exists on the subject. There are a few serious sampling based profilers available for Windows and it's possible that their developers (major companies) have signed agreements with Microsoft in order to access some hidden API.

For the profiler in this thesis a somewhat obtrusive method was developed for Windows in order to setup an interrupt handler. A small proof of concept application was developed in order to validate the technique. It may or may not be the best solution but it is good enough to establish confidence that a sampling based profiler can be implemented using this method.

The sampling mechanism aside there are still many dependencies such as operating system dependencies for process, symbol, debug information management as well as processor architecture and compiler dependencies for procedure call tracing. From a portability standpoint this is unfortunate but they presented no major issue in terms of implementation. As a result of these dependencies the profiler was constructed in a modular fashion in order to be extendible with support for new platforms. Adding support for a new operating system or processor architecture is not easy per se, but it doesn't require in depth knowledge of all the profiler's internals.

Extending the profiler is a matter of implementing and connecting the different interfaces defined in the profiler architecture. To aid the development the profiler system also provides a set of unit tests that can be used to validate the correctness of certain parts of the implementation. Despite the profiler's helpful architecture it can still be a difficult task extending the profiler. For example, knowledge of driver development and interrupt handling is necessary when adding support for a new operating system.

6.2 Conclusions

There is no silver bullet for designing and implementing a portable performance profiler of the kind in this thesis. The only way to write a purely portable profiler would be to limit the profiler to use only standard C or C++ library functions (assuming the profiler is written in C or C++). Neither of these libraries provides enough utility to perform any kind of profiling. Of all profiling techniques, source code instrumentation comes closest of achieving this. Nevertheless, not even source code instrumentation can be implemented using standard C or C++ functions.

There are only two viable methods for accurate profiling for hot spots or hot paths and that is simulation and sampling. Instrumentation has one advantage over sampling and that is calculating exact call frequencies. The profiler in this thesis focus is on finding hot spots and hot paths hence instrumentation is not a good option. If exact call frequencies is the most important result instrumentation is a good alternative. If both call frequencies and hot spots/paths are of interest it may be a better idea to look into simulation or developing two separate profilers, one for sampling and another for instrumentation. The latter approach is actually used in Intel's VTune profiler suite.

Implementing a portable sampling based profiler is far from straight forward. It carries more platform dependencies than desirable. It's tempting to compromise the accuracy and go with source code instrumentation just for the ease of it. This thesis suggests a sampled based profiler design that tries to minimize the amount of platform specific code under a modular extensible architecture. It shows that it is possible to build a profiler system that supports portability across platforms while retaining high accuracy.

Performance evaluation of the profiler implementation demonstrates that not only does it provide high accuracy, but it does that at a very low cost. Depending on the length of the sampling period, the overhead can be as low as 1% to 4%. This while retaining a maximum margin of error of under 1% (with 95% certainty).

6.3 Future Work

There are several areas in which the profiler can be improved, both in terms of design and implementation.

One thing is to further improve the buffer synchronization mechanism. The performance evaluation shows that the performance overhead increases with the amount of data processed by the profiler. The current buffer synchronization implementation focuses towards avoiding buffer overflows, but it may not be the most efficient one in terms of performance. For example, the size of the data chunks that are copied between the buffers is not optimized with respect to the processor and hard disk caches.

Furthermore, the buffer synchronization mechanism in the driver could potentially be replaced by a single lock free ring buffer supporting multiple producers and consumers. Even though the individual performance of such a ring buffer implementation can be expected to be slower than the current ring buffer implementation, it would be interesting to evaluate the performance difference compared to the entire buffer synchronization system. That is to replace the buffer hierarchy and synchronization mechanism inside the kernel driver with a single lock free ring buffer.

Another potential issue is that the profiler only performs sampling on one process. In some situations it might be of interest to profile child processes spawned by the profiled application. The profiled program may for example spawn a new process that will account for a large part of the processing time. In its current state the profiler will not detect this activity.

Even if not essential for the profiler's intended usage it could be of interest to extend the profiler to support generation of call graph or call context tree profiles. This could be done using the method suggested by [11] but a study would have to be done in order to establish which processor architectures that support this method.

Another interesting feature that has not been mentioned in this report is the ability of being able to profile thread stalls; where a program spend most of its time waiting. The profiler in this thesis focuses towards profiling applications that has not yet been parallelized, but one could imagine using it for the purpose of evaluating the parallelization itself. In this case profiling thread stalls could be very useful.

The current profiler implementation only supports profiling using one HPC register at a time. Most modern processors have at least two registers available per processor core. A nice feature could be to enable profiling using multiple registers at once, for profiling different aspects of the program. This would increase the profiler overhead but one would not have to profile a program multiple times as when using only one HPC register.

Finally, when using the profiler in HPC mode the sample intervals are specified as a number of events. For example 1000 cache misses or 100 000 clock cycles. Specifying sampling intervals in this way makes assumptions on the processor configuration of the current system. For example 100 000 clock cycles will take longer time to execute on a 1 GHz processor than a 3 GHz processor. It would be desirable to make the profiler at least calculate a default interval depending on the current processor configuration, perhaps based on the processor's clock frequency.

References

[1] M. Dunlavey, *Performance Tuning with Instruction-Level Cost Derived from Call-Stack Sampling*, ACM SIGPLAN Notices, Vol. 42, 8, 4–8, 2007

[2] S. Graham, P. Kessler, M. McKusick, *gprof: a Call Graph Execution Profiler*, ACM SIGPLAN Notice, Vol. 17, 6, 120–126, 1982

[3] S. Ghemawat, *Google CPU Profiler*, `http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`, January 12 2009

[4] Free Software Foundation, Inc., *Debugging Options - Using the GNU Compiler Collection (GCC)*, `http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Debugging-Options.html#Debugging-Options`, January 12 2009

[5] M. Dagenais, K. Yaghmour, C. Levert, M. Pourzandi, *Software Performance Analysis*, Cornell University Library, arXiv:cs/0507073v1, 2005

[6] N. Froyd, J. Mellor-Crummey, R. Fowler, *Low-Overhead Call Path Profiling of Unmodified, Optimized Code*, International Conference on Supercomputing, Session 3, 81–90, 2005

[7] M. Zagha, B. Larson, S. Turner, M. Itzkowitz, *Performance Analysis Using the MIPS R10000 Performance Counters*, Supercomputing, 1996, Vol. 2005-05-02, 16–16, 1996

[8] Intel Corporation, *Intel VTune*, `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`, January 15 2009

[9] J. Levon, *About OProfile*, `http://oprofile.sourceforge.net/about/`, January 15 2009

[10] Hewlett-Packard, *Prospect: Easy-to-use, non-intrusive profiling for Linux*, `http://prospect.sourceforge.net/`, January 15 2009

[11] M. Arnold, P. Sweeney, *Approximating the Calling Context Tree via Sampling*, IBM Research Report, 2000

[12] Sun Microsystems, Inc., *Sun Studio 12: Performance Analyzer*, 819-5264, 2007

[13] G. Ammons, T. Ball, J. Larus, *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling*, Conference on Programming Language Design and Implementation, 85–96, 1997

[14] X. Zhuang, M. Serrano, H. Cain, JD. Choi, *Accurate, Efficient, and Adaptive Calling Context Profiling*, Conference on Programming Language Design and Implementation, Session: Runtime optimization and profiling, 263–271, 2006

[15] Advanced Micro Devices, Inc., *AMD Developer Central - AMD CodeAnalyst*, `http://developer.amd.com/CPU/CODEANALYST/Pages/default.aspx`, February 2 2009

[16] Valgrind Developers, *Valgrind Home*, `http://valgrind.org/`, February 2 2009

[17] P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Lecture Notes In Computer Science, Vol. 1109, 104–113, 1996

[18] P. Drongowski, *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*, Advanced Micro Devices, Inc., 2007

[19] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 3A, 2008

[20] S. Friedl, *Intel x86 Function-call Conventions - Assembly View*, `http://www.unixwiz.net/techtips/win32-callconv-asm.html`, February 16 2009

[21] Microsoft Corporation, *__cdecl (C++)*, `http://msdn.microsoft.com/en-us/library/zkwh89ks(VS.71).aspx`, February 16 2009

[22] K. Johnson, *Frame pointer omission (FPO) optimization and consequences when debugging, part 1*, `http://www.nynaeve.net/?p=91`, February 16 2009

[23] A. Jönsson, *Calling conventions on the x86 platform*, `http://www.angelcode.com/dev/callconv/callconv.html`, February 16 2009

[24] University of Alberta, *Understanding Memory*, `http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html`, February 17 2009

[25] K. Frei, *X86 Unwind Information*, `http://blogs.msdn.com/freik/archive/2006/01/04/509372.aspx`, February 17 2009

[26] Microsoft Corporation, *Low-fragmentation Heap (Windows)*, `http://msdn.microsoft.com/en-us/library/aa366750(VS.85).aspx`, February 24 2009

[27] N. Nethercote, J. Fitzhardinge, *Bounds-Checking Entire Programs Without Recompiling*, Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004), 2004

[28] K. Johnson, *An introduction to kernrate (the Windows kernel profiler)*, `http://www.nynaeve.net/?p=45`, February 16 2009

[29] J. Corbet, G. Kroah-Hartman, A. Rubini, *Linux Device Drivers, 3rd Edition*, O'Reilly, 2005

[30] E. Ladan-Mozes, N. Shavit, *An Optimistic Approach to Lock-Free FIFO Queues*, In proceedings of the 18th International Conference on Distributed Computing (DISC), 117–131, 2004

[31] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Prentice Hall, 2004

[32] U. Vahalia, *UNIX Internals: The New Frontiers*, Prentice Hall, 1996

[33] R. Hyde, *The Art of Assembly Language Programming*, `http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/toc.html`, 1996

[34] B. Long, *.NET Internals: The Profiling API*, http://www.blong.com/Conferences/DCon2003/Internals/Profiling.htm, 2009

[35] C. Austin, *J2SE 5.0 in a Nutshell*, http://java.sun.com/developer/technicalArticles/releases/j2se15/, 2004

Glossary

Application Programming Interface (API) Programmers interface to a piece of software. 9, 17, 46

Call stack Dynamic stack structure for storing information about subroutine calls in a program. 3, 13, 19–21, 35, 54, 59

Debug symbol The name of a construct in an already compiled executable binary, for example a procedure. 30

GNU Compiler Collection (GCC) A compiler system for various programming languages. 13

Hardware Performance Counter (HPC) Special purpose hardware registers for monitoring. 16–18, 26, 33, 35, 48

Hot path An execution path of a program that is consuming a significant amount of resources. 3, 14, 46

Hot spot A small section of a program that is consuming a significant amount of resources. 3, 14, 46

Instruction Based Sampling (IBS) Precise sampling method developed by AMD. 18

Portable Operating System Interface (POSIX) IEEE standard defining an API for operating system libraries. 16

Procedure prologue Piece of code appearing in the beginning of a function that initializes the stack and registers for later use. 19

Real-Time Clock (RTC) Computer clock circuit, keeping track of the current time. 17, 26, 33, 35

Stack frame Sometimes called activation record, it contains state information about a subroutine call. 19–21

Userland Application space separated from the kernel with restrictive permissions. 18, 28, 33

A User Documentation

A.1 Introduction

This appendix regards the profiler usage. It begins with a small tutorial, followed by a more detailed user guide and user referenced. Finally there is a section on how to compile and install the profiler system.

For someone who should install the profile the installation guide is a good start. For someone who wants to start using the profiler immediately the tutorial is a good start.

A.2 Tutorial

First, make sure that the nprofile module is loaded into the kernel:

```
lsmod | grep nprofile
```

If not, load it:

```
insmod <path>/nprofile.ko
```

List the available events and select one that you want to profile on:

```
nprofile -l
```

For this tutorial the event: nprof.clk_unhalted is recommended and assumed to be used. This is a wrapper event incrementing on unhalted clock cycles. The original events doing this are named differently on different processor models, hence the need for a wrapper.

Now it's time to start profiling, for example you can use the following command:

```
nprofile --event=nprof.clk_unhalted:100000 -k --profile=path <program>
```

This will launch and start profiling the specified program, taking samples every 100 000th unhalted clock cycle. When the program is done executing the profile summary will be printed to the screen.

Since the -k option was used the raw sample data file is kept on the hard drive. It's named <PID>.nprof. The very same data file can now be used to re-generate a profile, for example of a different kind.

```
nprofile -r --profile=flat <PID>
```

The above command line will generate a flat profile instead of a path profile from the data collected by the last profiling session.

A.3 User Guide

Starting and Stopping a Profiling Session

Profiling can be stated by either lunching a new program through the profiler, or by attaching to a running process. The only difference in usage is whether a program (with optional arguments) or a process identifier is specified as the last argument to the profile. The profiler will know if it should launch a new program or attach to an existing one.

By default, the profiler will continue to profile until the profiled program exits, but it's possible to stop profiling at any time while letting the previously profiled program continue its execution. An ongoing profiling session can be aborted by sending an interrupt signal (Ctrl + C) to the profiling application. The profiler will stop sampling and produce a profile of the data gathered up to the point of abortion.

Selecting a Profile Type

A profile type is selected using the --profile option. Currently two types are supported, flat and path. Path profiling involves analyzing the call stack in order to determine the procedure call trace leading up to the sampled instruction. Generating this type of profile introduces a higher execution time overhead.

It should be noted that on large programs running for a long time it may happen that there will be lots of addresses getting very few hits. This can clutter the generated profile with unimportant results. In these cases it may be a good idea to use the --limit option that will strip all addresses that receives a hit count lower or equal to the specified limit. These addresses will never be ignored during the sampling process but only when generating the final profile. It can thus be a good idea to use the option -k to keep the raw sample data file on the hard drive and then tweak the output profile any number of times using the --limit option until the desired output profile has been generated.

Specifying a Profiling Event

Different processor models (yes, not architectures) provides very different profiling events. For this reason a wrapper event have been created to work across models and processor architectures. This event is called nprof.clk_unhalted and triggers on unhalted clock cycles.

A certain degree of carefulness should be put into selecting the sampling interval. If making a mistake using 1000 instead of for example 10 000 the computer may freeze entirely since all computing power is spent in the interrupt routine. This is a limitation of the profiler implementation. This of course depends on the profiling event. If using unhalted clock cycles 1000 is way too small, but not necessarily when using cache misses.

Regenerating a Profile

If sampling has been performed with the -k option the raw sample data file will be kept on the hard drive after the profiling session is completed. This can be useful for sampling in path profile mode and then generating a flat profile if the same data.

The raw sample data files will be stored using the name <PID>.nprof in the current directory. To generate a new profile of the same data use:

```
nprofile -r <OPTIONS> <PID>
```

Please note that the extension .nprof should not be appended after the process identifier.

A.4 Reference Manual

Usage: nprofile [OPTIONS] [PID or FILE...]

Options	Description
-k	If specified the raw sample data file will not be removed after the profiling session is complete. It will be stored under the file name: <PID>.nprof in the current directory.
-r	Indicates report only mode. This option is used in order to generate a report from an existing raw sample data file. This flag should be used in combination with a PID number. For example, if there is a file named 42.nprof nprofile should be called like: nprofile -r 42
-l	Lists all available profiling events.
-h	Prints a help message.
--event=<NAME>:<INTERVAL>	Specifies the profiling event and interval to use. The name should be the name reported when listing events. The interval should be selected with care, only a few sanity checks are performed on the interval value.
--limit=<LIMIT>	Specifies a sample hit limit. For example a limit of 1000 will ignore all entries (source lines, procedures, memory addresses) with 1000 and fewer hits.
--profile=<TYPE>	Specifies the profile type. If not in report generation mode the will affect how the sampling is performed. Supported profile types are: [flat|path].

A.5 Installation Guide

This installation guide is currently only for GNU/Linux systems and assumes a source code distribution model.

Compiling and Installing the Kernel Module

The first step towards installing the profiler system is to compile and then install the kernel module. In order to compile the module the system must have the Linux kernel headers installed. To check the availability you can do the following:

```
ls /lib/modules/‘uname -r‘/build/
```

An error will be printed if the path doesn't exist. In that case the Linux kernel headers must be installed.

When the kernel headers are installed navigate to the nprofile/driver/linux directory and type make. This will build the kernel module and link it into a file named nprofile.ko. To load the driver type:

```
insmod nprofile.ko
```

The driver can also be installed system-wide being superuser and typing:

```
make install
```

Compiling and Installing the libnprofile Library

This library is the interface between the profiling application and the kernel module. To compile libnprofile the following libraries are required: libelf, libdward, libiberty. In order to recompile the build scripts GNU autotools is needed. To recompile libnprofile, navigate to the nprofile/libnprofile directory and do the following:

```
./reconfigure
```

Then run the configure script with your preferred options and finally call make and make install as usual.

Compiling and Installing the Utility Library libnutil

This library lacks non-standard dependencies exept for GNU autotools. Simply navigate to nprofile/libnutil and do the following:

```
./reconfigure
```

Then run the configure script with your preferred options and finally call make and make install as usual.

Compiling and Installing the nprofile Application

This is the main application that will be called directly when profiling. nprofile depends on libnprofile and libnutil. If not installed system-wide they must be specified to the configuration script using the following argument:

```
./configure --with-nlibs=<path>
```

This application builds the same way as the libraries.

B System Documentation

B.1 System Specification

Overview

A general architectural overview is shown in figure 22. The dashed objects are not fully implemented in reality but shown in order to demonstrate the organization. The colors show to which source code package the functionality belongs.

The profiler application links directly to the interface so there is no message driver communication at that layer. The driver's communication with the interface towards the profiling application is specific for all operating systems. There is no standard protocol for this as each operating system provides very different means of communication with drivers. Please recall that the interface runs in a user mode process while the driver obviously doesn't.

GNU/Linux Driver Communication

The GNU/Linux driver creates virtual file system nodes in order to establish communication between the interface and the driver. The driver creates the following nodes by default:

- /proc/nprofile-buffer (read only)
- /proc/nprofile-control (read and write)
- /proc/nprofile-events (read only)

The names of these nodes can be configured in the nprofile/driver/linux/const.h file which also contains other easy configurable options.

The nprofile-buffer node is used for reading the contents of the drivers primary ring buffer. Data should be read in multiples of the pointer size. For example on a 32-bit system data should be read in multiples of four.

The nprofile-control device is used for controlling the operation of the driver. By reading from the device, information about its current status is obtained. The output data is defined as a list of entries where each entry has the following format:

```
<name>: <value>
```

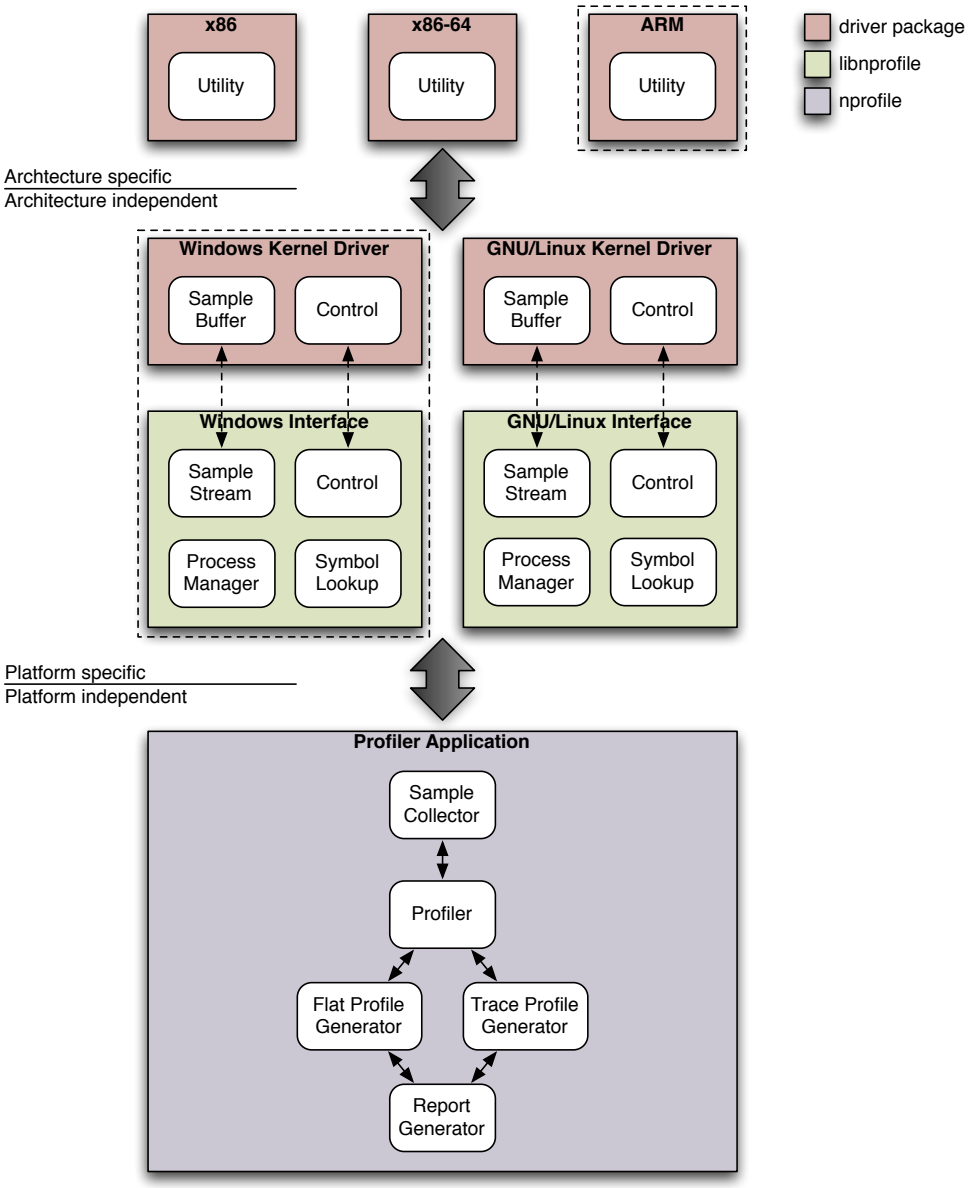

Figure 22: Architectural overview.

All entries are separated by a single new-line character. By default the following entries are defined:

- "sampling: "[0|1]
- "tracing: "[0|1]
- "num_pointers: "[0-9]+
- "pointer_size: "[0-9]+
- "profiler_hits: "[0-9]+

The sampling option tells if the driver is currently performing sampling or not. The tracing option is similar, telling if the driver performs call stack tracing. The num_pointers option tells how many memory addresses that are currently stored in the driver's primary ring buffer. The pointer_size tells the size of a single memory address in bytes. The profiler_hits option is only present in debug mode and tells the number of samples that hit the profiling application. This may be useful when analyzing the overhead of the profiling application alone.

To print the current status to standard output, execute the following command in a shell:

```
cat /proc/nprof-control
```

By writing to the nprofile-control device instead it's possible to control the backend. This is done by writing a command to the node. The default commands are defined as follows.

- "sampling: "(("start "[^:]+:[0-9]+,[0-9]+)|("stop"))
- "tracing: "["enable"|"disable"]
- "reset"

The sampling command starts a new or stops an ongoing sampling process. In order to better understand the above regular expressions consider the following example for starting a sampling process from a shell:

```
echo "sampling: start nprof.clk_unhalted:100000,0" > /proc/nprof-control
```

The above command will start profiling the init process with process identifier zero using the nprof.clk_unhalted event in intervals of 100 000 events. Stopping the sampling process is simply a matter of writing the following from a shell:

```
echo "sampling: stop" > /proc/nprof-control
```

The tracing command works in a similar manner, simply enabling and disabling tracing within the driver.

It should be noted that issuing the start sampling command will purge all buffers within the driver. The tracing command will not alter the buffer contents but can be switched on and off freely during the profiling operation.

Buffer Protocol

The contents of the primary ring buffer that is filled by the driver and eventually read by the profiling application uses a custom protocol. The protocol operates under the following assumptions:

- The buffer is organized into blocks. These blocks are always of the size of a pointer on the current system. That is 4 bytes on a 32-bit system.
- There are special reserved block values that are used for special purposes.
- The value in all blocks that are not reserved contains a memory address.

The special reserved blocks are defined below:

Block Value	Description
0	Start of a trace. All addresses following these blocks belong to the same sample. The first address is the actual sampled address. All subsequent addresses are the procedure trace from callee to caller.
1	End of a trace. All addresses preceding these blocks belong to the same sample.
2	Process exit. The profiled process has exited.

B.2 Detailed System Specification

GNU/Linux Back-End Class Diagram

Figure 23 shows the class diagram for the profiler back-end. The class diagram looks somewhat unconventional with many singleton classes. This is due to the fact that the back-end is actually implemented in the language C. Grey boxes indicate operating system specific code, in this case for GNU/Linux. Dark grey boxes indicate operating system and processor architecture specific code. The light grey indicate architecture specific code. There is only one light grey box and that is the Tracer class.

GNU/Linux Front-End Class Diagram

Figure 24 shows the class diagram for the profiler front-end. This includes the nprofile application and the libnprofile library. Grey boxes are classes within the driver interface (libnprofile).

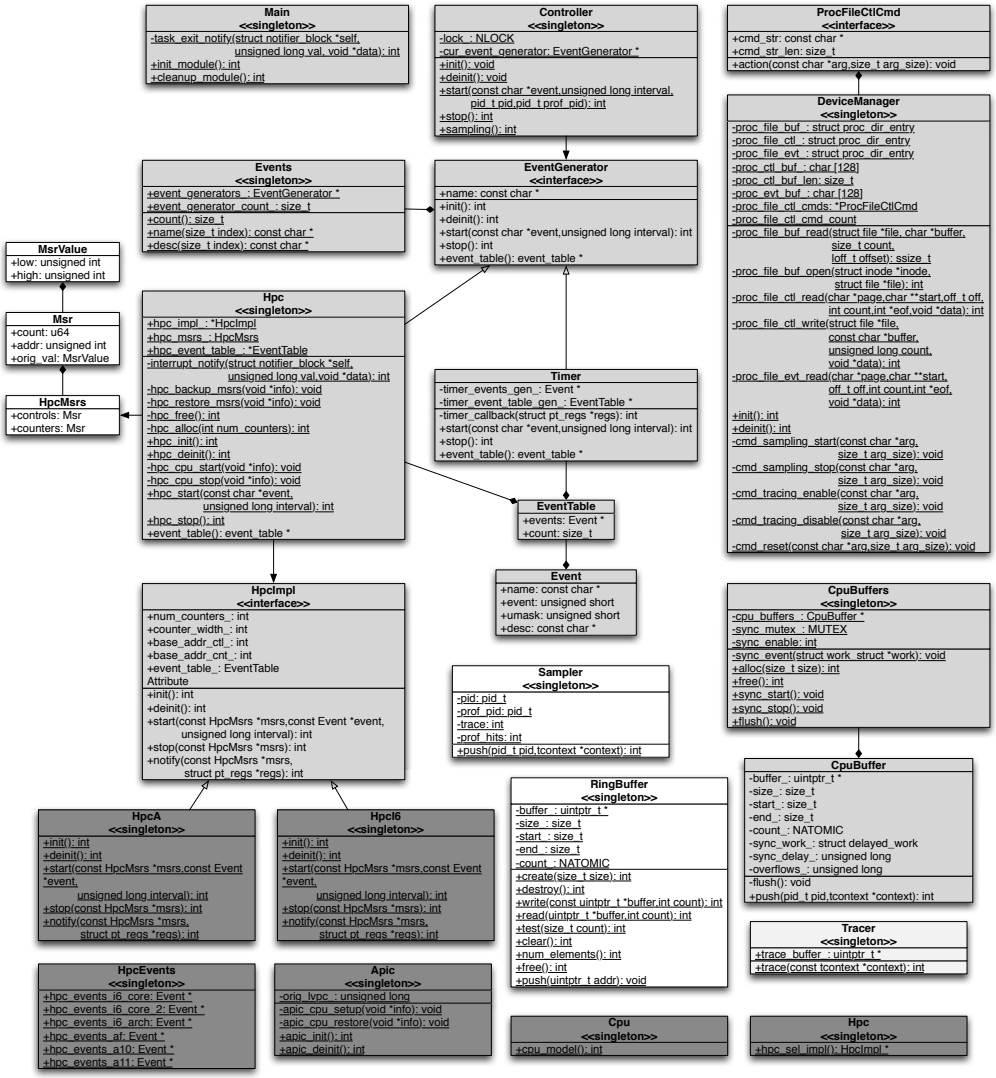

Figure 23: Class diagram of profiler back-end.

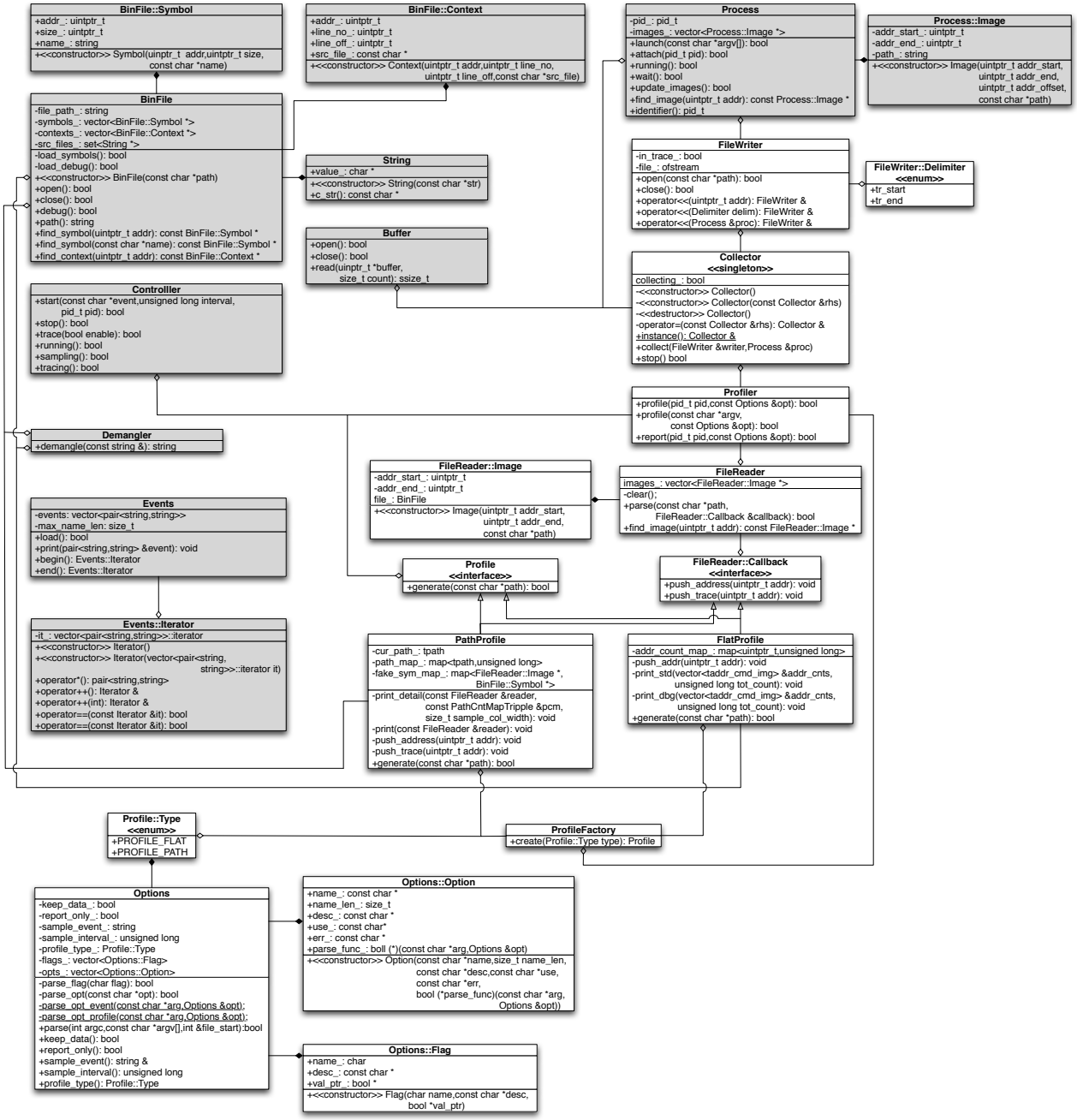

Figure 24: Class diagram of profiler front-end.

B.3 Testing Guide

The unit tests are located in the nprofile/tests directory. The test system shares dependencies with the profiling system and adds one further dependency, namely the Boost Unit Testing Framework. To compile the test suite on GNU/Linux systems, type make to use the included Makefile-based build system. To run the tests, execute the runtests.sh script:

```
./runtests.sh
```

B.4 Testing Protocol

The following functionality is tested using the unit test system:

- Driver control interface: Unit tests check that the driver responds to correctly commands and that it maintains a correct state.
- Driver buffer interface: Unit tests check that reading data from the driver through the driver interface in libnprofile works properly.
- Process management: Unit tests check that processes can be launched and that memory maps can be retrieved from them.
- Symbol lookup: Unit tests check that symbol information can be obtained from binary executables.
- Procedure call tracing: Unit tests check that procedure tracing works on the testing platform.
- Driver's primary ring buffer: All available operations on the primary ring buffer implementation are carefully tested.

B.5 Remaining Work

This is a list of what remains to be implemented:

- The Windows implementation.
- Monitoring of dynamic library loading and unloading in the driver. The current profiler implementation is not aware if the profiled application loads or unloads a dynamic library. This must be implemented if profiling programs that perform these actions. This applied to libraries loaded with the dlopen command, not dynamic libraries that are automatically loaded at startup.

C Profiler Comparison Matrix

Below is a profiler feature comparison matrix as asked for by Nema Labs.

Profiler	Method	Output	OS	Compiler	Architecture
nprofile (current)	Ss,Sh	F,C	L		x86,amd64
gprof	Ic	F,C	L	GCC	
oprofile	Ss[†],Sh	F,C	L		x86,amd64, alpha,arm
valgrind	Ir	F,C	L		x86,amd64, ppc32,ppc64, arm,mips
prospect	Ss,Sh	F,C	L		x86,amd64, alpha,arm
Google PerfTools	Ic	F,C	L		
Luke Stackwalker	Ss	F,C	W	MSVC	x86,amd64
Sleepy	Ss	F	W	MSVC	x86,amd64
Intel VTune (S)	Ss,Sh[‡]	F	L,W		x86,amd64
Intel VTune (I)	Ir	C	L,W		x86,amd64
AMD CodeAnalyst	Ss,Sh*	F,C*	L,W		x86,amd64

Profiler	Accessible API/Back-end	Parallel Profiling	Notes
nprofile (current)	Yes	Yes	
gprof	Yes	No	
oprofile	Yes	Yes	Needs superuser access.
valgrind	Yes	Yes	
prospect	Yes	Yes	Uses oprofile kernel module.
Google PerfTools	Yes	Yes	
Luke Stackwalker	No	Yes	Poor sampling mechanism implementation
Sleepy	No	No	Poor sampling mechanism implementation
Intel VTune (S)	Yes	Yes	
Intel VTune (I)	Yes	Yes	
AMD CodeAnalyst	No[¤]	Yes	Uses oprofile kernel module on GNU/Linux.

Method: Ic = Instrumentation (compile-time), Ir = Instrumentation (run-time), Ss = Sampling (software based), Sh = Sampling (hardware based)
Output: F = Flat, C = Contextual (call graph or call tree)
OS: L = GNU/Linux, W = Microsoft Windows

[†]=Not encouraged, tricky to use.
[‡]=Works on Intel processors only.

*=Works on AMD processors only.
II=Different APIs exist for different operating systems.