Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Filip Nybäck

Improving the support for ARM in the IgProf profiler

Master's Thesis
Espoo, October 10, 2014

Supervisor: Professor Jukka K. Nurminen
Advisor: Zhonghong Ou D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Filip Nybäck

Title:
Improving the support for ARM in the IgProf profiler

Date:	October 10, 2014	**Pages:**	79
Major:	Data Communication Software	**Code:**	T-110
Supervisor:	Professor Jukka K. Nurminen		
Advisor:	Zhonghong Ou D.Sc. (Tech.)		

IgProf is an application profiler that profiles mainly performance and memory usage. The profiler is extended and improved in three ways as part of this thesis: support for 64-bit ARM is implemented, the execution time of stack tracing is reduced on both 64-bit and 32-bit ARM and a simple energy profiling module is added.

The IgProf profiler was available on the Intel x86 and x86-64 architectures, as well as on 32-bit ARM, but support for 64-bit ARM was missing. The port of IgProf to 64-bit ARM enables developers to evaluate how applications execute on the new architecture with regard to performance and memory usage. The port of IgProf is going to be used on 64-bit ARM for examination and optimisation of the CMS software, which is related to the Compact Muon Solenoid (CMS) experiment at CERN.

IgProf uses the libunwind library to perform stack tracing as part of the profiling. An optimised version of stack tracing, previously available only on the x86-64 architecture, is ported to both 64-bit and 32-bit ARM. The optimised stack tracing reduces the execution time of profiling, especially when profiling events occur frequently. When a piece of CMS software is profiled for memory usage on 64-bit ARM and the optimised version of stack tracing is used instead of the old version, the execution time of profiling is reduced by approximately 87 %. The overhead in execution time of profiling may not matter when a small application is profiled, but for a big application the overhead determines if it is practically feasible to profile the application.

A simple energy profiling module extends the functionality of IgProf. The energy profiling module is based on sampling and obtains energy measurements from the Running Average Power Limit (RAPL) interface present on recent Intel processors. The profiling results of a simple single-threaded application seem to show a correlation between the execution time and the energy spent in a function. The energy profiling module is still rather limited, but is the first step for IgProf in the direction of energy profiling.

Keywords: performance profiling, memory profiling, AArch64, stack tracing, libunwind, energy profiling

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Filip Nybäck		
Työn nimi:	IgProf-profilointiohjelman ARM-tuen parantaminen		
Päiväys:	10. lokakuuta 2014	**Sivumäärä:**	79
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	TkT Zhonghong Ou		

IgProf on profilointiohjelma, joka profiloi pääasiassa suorituskykyä ja muistinkäyttöä. Profilointiohjelmaa laajennetaan ja parannetaan kolmella tavalla osana tätä diplomityötä: tuki 64-bittiselle ARMille toteutetaan, pinon jäljityksen suoritusaikaa lyhennetään sekä 64-bittisellä että 32-bittisellä ARMilla ja yksinkertainen energianprofilointimoduuli lisätään.

IgProf toimi Intelin x86- ja x86-64-arkkitehtuureilla, kuten myös 32-bittisellä ARMilla, mutta tuki 64-bittiselle ARMille puuttui. IgProfin sovitus 64-bittiselle ARMille mahdollistaa sovellusten suorituskyvyn ja muistinkäytön arviointia uudella arkkitehtuurilla. IgProfin sovitusta on tarkoitus käyttää CERNin Compact Muon Solenoid -kokeeseen (CMS) liittyvän CMS-ohjelmiston tarkasteluun ja optimointiin 64-bittisellä ARMilla.

Osana profilointia IgProf käyttää libunwind-kirjastoa pinon jäljittämiseen. Pinon jäljityksen optimoitu versio, joka oli saatavilla vain x86-64-arkkitehtuurilla, sovitetaan sekä 64-bittiselle että 32-bittiselle ARMille. Optimoitu pinon jäljitys vähentää profiloinnin suoritusaikaa, varsinkin kun profilointitapahtumat esiintyvät usein. Kun CMS-ohjelmiston erään osan muistinkäyttöä profiloidaan 64-bittisellä ARMilla ja pinon jäljityksen optimoitu versio on käytössä vanhan version sijaan, profiloinnin suoritusaika laskee noin 87 %:lla. Profiloinnin aiheuttama suoritusajan pidennys ei liene merkitsevä pientä sovellusta profiloitaessa, mutta laajan sovelluksen suoritusajan pidennys ratkaisee onko profilointi käytännöllisesti toteutettavissa.

Yksinkertainen energianprofilointimoduuli laajentaa IgProfin toiminnallisuutta. IgProfin energianprofilointimoduuli perustuu näytteenottoon ja saa energiamittauksia Running Average Power Limit -rajapinnasta (RAPL), joka löytyy Intelin uudehkoista suorittimista. Yksinkertaisen yksisäikeisen sovelluksen profilointitulokset viittaavat siihen, että funktion suoritusaika ja energiankulutus korreloivat. Energianprofilointimoduuli on vielä melko rajoittunut, mutta on IgProfin ensimmäinen askel energianprofiloinnin suuntaan.

Asiasanat:	suorituskyvyn profilointi, muistinkäytön profilointi, AArch64, pinon jäljitys, libunwind, energianprofilointi
Kieli:	Englanti

Aalto-universitetet
Högskolan för teknikvetenskaper
Examensprogram för datateknik

SAMMANDRAG AV
DIPLOMARBETET

Utfört av: Filip Nybäck

Arbetets namn:
Förbättring av stödet för ARM i profileringsprogrammet IgProf

Datum:	Den 10 oktober 2014	**Sidantal:**	79
Huvudämne:	Datakommunikationsprogram	**Kod:**	T-110
Övervakare:	Professor Jukka K. Nurminen		
Handledare:	TkD Zhonghong Ou		

IgProf är ett profileringsprogram som huvudsakligen profilerar prestanda och minnesanvändning. Profileringsprogrammet utökas och förbättras på tre sätt som en del av detta diplomarbete: stöd för 64-bitars ARM implementeras, exekveringstiden för stackspårning förkortas både på 64-bitars och 32-bitars ARM och en enkel energiprofileringsmodul läggs till.

IgProf fanns tillgängligt för Intels x86- och x86-64-arkitekturer, så väl som för 32-bitars ARM, men stöd för 64-bitars ARM saknades. Porteringen av IgProf till 64-bitars ARM gör det möjligt för utvecklare att utvärdera prestandan och minnesanvändningen för applikationer på den nya arkitekturen. Det finns planer på att använda porteringen av IgProf på 64-bitars ARM för granskning och optimering av CMS-mjukvaran som är relaterad till Compact Muon Solenoid-experimentet (CMS) vid CERN.

IgProf använder libunwind-biblioteket för att spåra stacken som en del av profileringen. En optimerad version av stackspårningen, som fanns tillgänglig endast för x86-64-arkitekturen, porteras till både 64-bitars och 32-bitars ARM. Den optimerade stackspårningen förkortar exekveringstiden för profileringen, särskilt då profileringshändelser inträffar frekvent. När en del av CMS-mjukvaran profileras för minnesanvändning på 64-bitars ARM och den optimerade versionen av stackspårningen används istället för den gamla versionen, reduceras exekveringstiden för profileringen med cirka 87 %. Förlängningen av exekveringstiden som profileringen orsakar är måhända obetydlig för små applikationer, men för stora applikationer avgör förlängningen om profileringen är praktiskt genomförbar.

En enkel energiprofileringsmodul utökar funktionaliteten i IgProf. Energiprofileringsmodulen baserar sig på sampling och erhåller energimätningar från Running Average Power Limit-gränssnittet (RAPL) som Intels nyare processorer implementerar. Profileringsresultaten av ett enkelt enkeltrådigt program tyder på att exekveringstiden och energikonsumtionen i en funktion korrelerar. Energiprofileringsmodulen är ganska begränsad, men är ett första steg för IgProf i riktning mot energiprofilering.

Nyckelord:	prestandaprofilering, minnesprofilering, AArch64, stackspårning, libunwind, energiprofilering
Språk:	Engelska

Acknowledgements

I would like to thank Zhonghong Ou, Jukka K. Nurminen, Giulio Eulisse, Peter Elmer, David Abdurachmanov, Gonçalo Pestana, Kashif Nizam Khan, Tapio Niemi and Vesa Hirvisalo for useful feedback and comments given more or less frequently throughout the whole thesis process. I would especially like to thank my advisor, Zhonghong Ou, for arranging weekly meetings, where we have discussed the latest progress, problems that have arisen and possible solutions. I would like to offer my special thanks to Giulio Eulisse for not only sharing his excellent technical knowledge about IgProf, but also for encouraging me when the energy profiling module seemed to be at a dead end. I am particularly grateful for the help given by David Abdurachmanov, who has asked the right questions about the energy profiling module, helped me with the cycle counter on ARM and provided very valuable profiling data. I would like to express my great appreciation to Google that has supported me financially in the Google Summer of Code project and to Åke Wennström for financing the hardbound copies of the thesis. I would also like to give my special thanks to my husband for supporting and encouraging me not only during the thesis process, but throughout my studies at Aalto University.

Espoo, October 10, 2014

Filip Nybäck

Abbreviations and Acronyms

ABI	application binary interface
API	application programming interface
CFA	canonical frame address
CPU	central processing unit
ELF	executable and linkable format
FP	frame pointer
GPU	graphics processing unit
HPC	high performance computing
ISA	instruction set architecture
PC	program counter
RAPL	running average power limit
SP	stack pointer
TLB	translation lookaside buffer

Contents

Chapter 1

Introduction

1.1 The IgProf profiler

Profiling is a type of program analysis that is performed at run time, i.e. it is dynamic as opposed to static program analysis, which is performed at compile time. The profiling software, the profiler, collects data about the resources an application uses. The resource of interest can be for example execution time, memory, energy consumption or file descriptors. A performance profiler, for example, finds out how much execution time the application spends in different parts (e.g. functions) of the code. The main idea behind profiling is to find the parts of the application where a resource is used the most, i.e. possible bottlenecks. When the bottlenecks have been located, it makes sense to make optimisations where the bottlenecks are, because resource usage is usually not uniformly distributed over the whole application. It is sometimes said that an application spends 80 % of the execution time in 20 % of the code [31]. This is called the 80–20 rule and is a variant of the Pareto principle. The actual percentages may vary from application to application [43], but it makes sense to find the 20 % of the code that uses 80 % of the resources and optimise those parts of the code. [44]

IgProf [27, 81] is a profiler developed within the Compact Muon Solenoid (CMS) software division at CERN, the European Organization for Nuclear Research [20]. The CMS software is used to analyse data and perform simulations related to CMS experiments [64]. IgProf measures and analyses memory and performance characteristics of applications. Unlike some other profilers, e.g. gprof, IgProf does not generally require the application to be recompiled or relinked before profiling. The profiler operates completely in user mode, does not require root privileges to run and can handle dynamically loaded shared libraries. IgProf presents the results of the profiling as a

flat profile and a call graph. The main principles of operation are statistical sampling and dynamic instrumentation of functions. These principles of operation are described in general in section 2.1 and in more detail in sections 3.1.6 and 3.1.7.

1.2 Problem statement

Performance has traditionally been of higher priority than energy consumption in the field of high performance computing (HPC). ARM processors are common in mobile consumer grade equipment, where low energy consumption is of high priority at the cost of performance. It has been proposed that ARM processors could be used for scientific calculations and processing big data [7] [60], because ARM processors generally consume less energy than high-grade processors [17]. The cost of electricity used by a data centre is significant, including the costs of cooling and power distribution [47]. Even a one per cent saving in electricity costs of a big data centre is a significant sum of money [65].

In order to be able to evaluate the performance of software running on ARM processors, developers need profiling software supporting the ARM architecture. IgProf already supported the 32-bit ARM architecture, but support for the 64-bit ARM architecture was missing. It is, however, more likely that 64-bit ARM processors will be used in data centres rather than 32-bit ARM processors [24]. Hence it makes sense to add the 64-bit ARM architecture to the set of supported architectures in IgProf.

IgProf uses the libunwind library to perform stack tracing as part of the profiling. The profiler performs stack tracing frequently, thus it is necessary for the stack tracing to execute fast. Stack tracing in libunwind had already been optimised for speed on the x86-64 architecture, but the optimised version was not available on other architectures. A port of the optimised stack tracing feature of the libunwind library to both ARM architectures is expected to improve the execution time of profiling on these architectures.

The IgProf profiler can perform several types of profiling, e.g. performance profiling, memory usage profiling and file descriptor profiling. However, it lacked the ability to profile energy consumption. The addition of an energy profiling module to IgProf allows developers to evaluate which parts of an application consume the most energy. This information is valuable if the application is to be optimised for energy consumption.

1.3 Goals and scope

Three goals are set for the work related to this thesis. The first goal is to implement support for profiling 64-bit ARM applications in the IgProf profiler. The code of IgProf is mostly architecture independent, but some parts of IgProf are implemented separately for each supported architecture. The part of IgProf that operates on binary instructions is implemented separately for each supported architecture, because binary instructions are encoded differently on different architectures.

The second goal is to port the fast stack tracing feature of the libunwind library to the 32-bit and 64-bit ARM architectures. Fast stack tracing was only available on the x86-64 architecture. The code implementing the fast stack tracing feature is dependent on calling conventions, register usage and stack layout of each supported architecture.

The third goal is to implement an energy profiling module in IgProf. In contrast with the work related to the first and second goal, the energy profiling module extends the functionality of IgProf. The energy profiling module is based on sampling and uses the Performance API (PAPI) library to obtain energy measurements from the Running Average Power Limit (RAPL) interface present on recent Intel processors.

1.4 Terminology

Four instruction set architectures (ISAs) are mainly discussed in this thesis: x86, x86-64, A32 and A64. The name x86 originally stems from the names of earlier Intel processors: 8086, 80186 (or 186 for short), 80286 (286), 80368 (386) and 80486 (486). The 386 processor generation introduced 32-bit registers and a 32-bit address space. In this thesis the name x86 refers only to the 32-bit ISA. Intel calls the x86 ISA IA-32 (Intel Architecture, 32-bit) and another common name is i386. The x86-64 ISA is backward compatible with the x86 ISA. AMD extended the x86 ISA with 64-bit wide registers and a 64-bit address space. AMD calls this ISA AMD64, Intel has used the names IA-32e, EM64T and Intel 64, and the names x86-64, x86_64 and x64 are in common use. The similarly named IA-64 (Itanium) ISA by Intel, is not related to or compatible with the x86-64 ISA. [55]

A 32-bit ARM processor implements both the A32 ISA, i.e. the original ISA of ARM processors, and the T32 ISA, formerly called Thumb, that was added later. All instructions in the A32 ISA are 32 bits, whereas most instructions in the T32 ISA are 16 bits, which means that binary T32 code is generally smaller than binary A32 code. The T32 instruction set is, however,

more limited than the A32 instruction set and does not take full advantage of the ARM architecture. An ARM processor can switch between A32 and T32 mode. A 64-bit ARM processor implements the A64 instruction set architecture. Instructions are still 32 bits, but registers and the address space are 64 bits. When running instructions of the A64 ISA, the processor is in AArch64 mode. The processor also implements the A32 and T32 ISAs from the 32-bit ARM design for backward compatibility. When running instructions of the A32 or T32 ISA, the processor is in AArch32 mode. [11]

In this thesis the term 'x86 architecture' refers to an architecture that implements the 32-bit x86 instruction set and specifically refers to the 32-bit mode of operation. Similarly, the term 'x86-64 architecture' refers to an architecture that implements the x86-64 instruction set and specifically refers to the 64-bit mode of operation. Unless otherwise stated, the term 'ARM architecture' refers to an architecture implementing the A32 instruction set, i.e. the original ARM instruction set, and specifically refers to the A32 submode of the AArch32 mode of operation. Similarly, the term 'AArch64 architecture' refers to an architecture implementing the A64 instruction set and operating in the 64-bit AArch64 mode of operation. Unless otherwise stated, the software discussed in this thesis, e.g. the IgProf profiler, the libunwind and PAPI libraries, are assumed to run on the Linux/GNU operating system.

1.5 Structure of the thesis

The rest of the thesis is structured as follows. Chapter 2 describes profilers in general and includes a comparison of some common profilers. Chapter 3 describes the IgProf profiler and the fast stack trace feature of the libunwind library as they were before any porting efforts. Chapter 4 describes the porting and implementation efforts. It describes what has been done in order to port IgProf to 64-bit ARM, port the fast stack trace feature of the libunwind library to 32-bit and 64-bit ARM and implement the energy profiling module in IgProf. Chapter 5 describes testing and evaluation of the ported software and the energy profiling module of IgProf. In chapter 6 some results from the testing and evaluation are discussed, as well as limitations and future work. Finally, chapter 7 concludes this thesis.

Chapter 2

Background

This chapter describes application profilers in general. Section 2.1 describes the principles of operation common to many profilers, whereas section 2.2 describes output formats commonly used by profilers. Section 2.3 gives a short summary of some profiling software.

2.1 Common principles of operation

There are a few common principles of operation that profilers rely on: statistical sampling, instrumentation and hardware performance counters. Statistical sampling means that the profiler samples the program state periodically, in order to collect statistics on how much execution time the program spends in each part of the program. The profiler can present the results using different granularity of the execution location: individual instructions, basic blocks, functions or programs. Profilers usually implement statistical sampling using an interval timer and a signal as described in section 3.1.6.

Instrumentation means that the code of a program is modified in some way. The instrumentation can happen at compile time (static instrumentation) or at run time (dynamic instrumentation). The target of instrumentation can range from individual instructions to functions. In the context of profiling, instrumentation is used to insert (or inject) code that measures performance or catches function calls. As an example, the number of clock cycles consumed in a function can be obtained, when code that reads the cycle counter is inserted at the beginning of the function, the function itself is executed and finally, the cycle counter is read again at the end of the function. The number of cycles spent in the function is simply the difference between the two readings.

A memory profiler, on the other hand, keeps track of memory blocks

being allocated and released. This information can be obtained from the arguments and return values of function calls to memory allocation and release functions, e.g. *malloc* and *free*. By instrumenting memory allocation and release functions, the profiler can keep track of the allocated memory blocks and collect statistics on memory usage.

A third mechanism that profilers take advantage of are hardware performance counters. Performance counters are registers in the CPU that count hardware events such as cycle ticks, memory references, cache misses or TLB (translation lookaside buffer) misses. Because the performance counters are implemented in hardware, the mechanism is very fast and does not require any modifications to the application. Performance counters can even be used to profile system software, because the mechanism does not rely on software being interrupted or instrumented, which can interfere with system software.

2.2 Common output formats

There are a few common types of output that profilers generate: a flat profile, a call graph profile and an annotated source listing. A flat profile is a list of functions and the time spent in the associated functions. The time is shown in two ways: total time and self time. Total time includes the time spent both in the caller and the callee(s), whereas self time only includes the time spent in the caller, but not in the callee(s). The total time and self time differ when the function (the caller) calls other functions (callees).

The following excerpt from the output of IgProf is an example of a flat profile showing the cumulative time:

```
-----------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total      Total  Function
  100.0      15.34  <spontaneous> [1]
  100.0      15.34  _start [2]
  100.0      15.34  __libc_start_main [3]
  100.0      15.34  main [4]
  100.0      15.34  insertion_sort [5]
   42.8       6.56  swap [6]
-----------------------------------------------------------------------
```

The first column (% total) shows the percentage of total time the program spent executing in the function and its child functions (callees). The second column (total) shows the total time in seconds. Finally, the third column shows the name of the function. Entries 1–3 are implementation details

of the compiler suite (GCC) and the C library (glibc), but the functions are responsible for calling the *main* function. The example above of a flat profile from IgProf shows the cumulative time. A flat profile showing the self time has exactly the same structure in IgProf, but the times presented are obviously self time, i.e. the time spent in the functions themselves excluding the time spent in child functions (callees). [39, 44]

A call graph profile can be presented as text or graphically. The text format of a call graph profile shows a number of entries separated by a line of dashes. The primary line of an entry is the reference point in that particular entry. The lines above the primary line show the callers (the functions that called the primary function) and the lines below the primary line show the callees (the functions that the primary function called). Some profilers (e.g. the Google performance tools [41]) can generate graphical call graph profiles directly, whereas others (e.g. gprof) requires the use of a tool not bundled with the profiler (e.g. Gprof2Dot [30]). [27, 39]

The following excerpts from the output generated by IgProf are examples of call graph profile entries:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Rank    % total       Self       Self / Children   Function
          100.0  .........       1.92 / 1.92         __libc_start_main [3]
[4]       100.0       1.92       0.00 / 1.92       main
           55.3  .........       1.06 / 1.06         fill_array [5]
           44.1  .........       0.85 / 0.85         insertion_sort [6]
            0.6  .........       0.01 / 0.01         munmap [12]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
           44.1  .........       0.85 / 1.92         main [4]
[6]        44.1       0.85       0.66 / 0.19       insertion_sort
            9.7  .........       0.19 / 0.19         swap [9]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

In this example only the fourth and sixth entry are shown, indicated by the numbers in brackets in the first column (rank). Each entry has a primary line, the line with a number in brackets on the left. The primary line shows the primary function for that entry. In entry number four in the example, *main* is the primary function. Entry number six shows that *insertion_sort* is the primary function. The lines above the primary line show the callers. In entry number six, main has called the *insertion_sort* function. The lines below the primary line show the callees. Entry number four shows that *main* has called the *fill_array*, *insertion_sort* and *munmap* functions. The call graph profile is not an exhaustive list of callers and callees. Functions that execute fast may not be sampled at all or even if they are included in the statistics, they may be filtered by the analyser component of the profiler.

The second column (% total) in the example output shows the percentage of the execution time spent in the function including calls to other functions. The interpretation of the fourth (self) and fifth column (children) depends on the role of the line as primary function, caller or callee. On a primary line, the fourth column (self) shows the time spent executing in the function itself and the fifth column (children) shows the time spent executing in the callees. The third column (also called self in IgProf) contains the total time spent in the function, i.e. the sum of the fourth and fifth column. On a caller line, the fourth column (self) shows the time spent in the primary function when the caller called the primary function. The fifth column (children) indicates the time spent in callees of the primary function when the caller called the primary function. On a callee line, the fourth column (self) shows the time spent in the callee itself excluding time spent in callees of the callee when the primary function called the callee. The fifth column (children) indicates the time spent in callees of the callee when the primary function called the callee. Finally, the sixth column shows the name and index of the function. [39]

An annotated source listing shows each line of source code and the number of times the line has been executed according to the profiler. The following excerpt from the output of the *gcov* tool [38], the GNU coverage testing tool, shows the annotated source of a simple insertion sort implementation:

```
        1:   66:void insertion_sort(int a[], int length)
        -:   67:{
        -:   68:    int i, j;
        -:   69:
   100000:   70:    for(i = 1; i < length; ++i) {
2499830266:   71:       for(j = i; j > 0 && a[j - 1] > a[j]; --j) {
2499730267:   72:           swap(a, j - 1, j);
        -:   73:        }
        -:   74:    }
        1:   75:}
```

The first column shows the number of times a line has been executed, whereas the second column is the line number [38]. In this example the lines 66–75 of the source file are shown. Finally, the source code is shown in the third column. Line 66 shows that the function has been entered once. The gcov tool does not consider lines 67–69 executable code. The outer loop, starting at line 70, is executed 100 000 times. The inner loop, starting at line 71, is executed 2 499 830 266 times. Line 72 shows that a swap operation was not performed at every iteration of the inner loop. Lines 73–74 are not executable code. Finally, line 75 shows that the function has returned once, which matches the number of times the function was entered.

2.3 A brief presentation and comparison of some profiling software

There are many application profilers and profiling libraries available. This brief presentation and comparison covers the *IgProf*, *gprof*, *Google Performance Tools*, *OProfile*, *HPCToolkit*, *Intel VTune* and *Instruments* profilers and tools. Profiling, performance and instrumentation libraries are included in this comparison, because they provide functionality that profilers either build upon or implement themselves. The *Valgrind*, *DynamicRIO*, *Intel PIN* and *PAPI* libraries or framworks are presented and compared to the profiling tools mentioned above. Furthermore, the *perfctr*, *perfmon2* and *perf_events* performance monitoring kernel interfaces are included, because they form the basis for many performance monitoring and profiling tools and libraries. In the comparison especially the following criteria are taken into account when they are known:

- if the software is open-source
- if the software requires special compilation of the application
- if the software requires root privileges to run
- if the software can handle dynamically loaded shared libraries
- supported platforms (operating systems and architectures)
- the overhead (memory, execution time)

Table 2.1 at the end of this section summarises the comparison of the profiling tools and libraries.

IgProf is an application profiler that measures and analyses the memory and performance characteristics of applications. It is not generally necessary to recompile or relink the application before profiling. The profiler operates completely in user space and does not require root privileges and can handle dynamically loaded shared libaries. IgProf presents the results as a flat profile and a call graph. The main mechanisms of operation are statistical sampling and dynamic instrumentation of functions. For further details about how the profiler is implemented, see sections 3.1.5–3.1.8. IgProf supports Linux on the x86, x86-64 and ARM architectures. One of the goals of this thesis is to port the profiler to the AArch64 architecture. [24, 28, 81]

The *gprof* profiler is part of the GNU binutils collection of development tools [33]. The profiler requires that a profiling flag (-pg) be enabled when the

application is compiled and linked. The profiler uses statistical sampling and static function instrumentation to perform the profiling. On most platforms the *profil* system call [3] performs the statistical sampling. If the system call is not available, an interval timer is used. The profiler can produce a flat profile, a call graph, a line-by-line profile or an annotated source. In the line-by-line profiling mode gprof maps the sampled execution location to a line of source code instead of mapping it to a function. The gprof profiler is available for Linux on a wide range of architectures, including x86, x86-64, ARM and AArch64. [39]

The *Google Performance Tools (google-perftools)* comprises an implementation of the memory allocation function *malloc* called *tcmalloc*, a heap checker, a heap profiler and a CPU profiler (performance profiler). The heap checker only detects memory leaks, whereas the heap profiler in addition to finding memory leaks collects statistics on memory allocations and releases. The performance profiler generates a flat profile, a call graph or annotated disassembly of the program. The call graph is available in the Graphviz, dot, Postscript, PDF, GIF and Callgrind formats in addition to a text report. KCachegrind can be used to visualize the files in Callgrind format. To use the Google performance tools, it is recommended to link the application to the libraries supplied with the tool set, but it is not strictly necessary to do this. If the application is not linked with the supplied libraries, the LD_PRELOAD mechanism described in section 3.1.5 is used to load the profiler. The performance profiler makes use of statistical sampling and the memory tools instrument memory allocation and release functions (*malloc*, *calloc*, *realloc* and *free*). The statistical sampling is implemented using an interval timer. The tool set supports Linux on the x86 and x86-64 architectures. [41, 42]

The *OProfile* profiler relies on hardware performance counters when measuring the performance of an application. In case hardware performance counters are not available on the processor, OProfile is run in legacy mode and makes use of statistical sampling. What performance counters OProfile can record is specific to the underlying processor architecture. OProfile accesses the counters through the *perf_events* API of the Linux kernel. The profiler comprises a number of command line tools, a kernel driver that collects samples and a daemon that records the samples. OProfile can target a process, a CPU or the whole system. The profiler supports only ELF binaries (not a.out binaries) and Linux 2.6.13 or later on the x86, x86-64, ARM, Alpha, MIPS, Sparc64, PowerPC, AVR32, PA-RISC and s390 architectures. [21, 63]

Intel VTune is a commercial profiler that relies on hardware performance counters and timers. The profiler reports the execution time spent in func-

tions and code segments. The scope of profiling is generally system-wide, but the profiler can also report timing and processor utilisation information when profiling multithreaded applications. VTune has both a command line and a graphical user interface and runs on Linux and Windows. [52, 58]

HPCToolkit is a collection of tools that focus on parallel programs. The toolkit relies on statistical sampling for performance profiling and additionally uses hardware performance counters to collect statistics on operation counts, pipeline stalls and cache misses. Profiling with HPCToolkit does not require any special compilation and adds 1–5 % overhead to the execution time. In addition to serial programs, the toolkit can measure and analyse applications using multithreading through the *pthread* and *OpenMP* interfaces or using Message Passing Interface (MPI). The toolkit includes programs that visualise the results, both in time-centric and code-centric views, in a graphical user interface. HPCToolkit supports Linux on the x86, x86-64 and POWER architectures. [8, 82]

The *Instruments* tool by Apple is a performance and memory profiler. The tool is part of the Xcode development toolset since version 3.0. Instruments has a graphical user interface and presents the results in tables and graphs. The tool relies on sampling and hardware performance counters to collect profiling data. The *iprofiler* command-line tool measures performance without running the graphical Instruments tool concurrently. The collected profiling data is however analysed and presented in the Instruments tool. Instruments targets the (Mac) OS X operating system on x86-64 and the iOS operating system on Apple's ARM-based devices. [10, 62]

Valgrind is a framework and a set of tools for debugging and profiling. Valgrind executes the application in a virtual machine and the tools of Valgrind (called plugins) can examine or modify each instruction of the application. Some of the tools of Valgrind are: *Memcheck*, *Massif*, *Cachegrind* and *Callgrind*. The Memcheck tool examines memory accesses in order to discover memory leaks, uninitialized variables, multiple releases of allocated memory blocks and overlapping source and destination memory blocks in calls to *memcpy* and similar functions. The Massif tool is a heap profiler (memory profiler) that keeps track of the locations in a program, where memory is allocated. Cachegrind is a cache profiler, simulating the interaction between the program and memory caches (L1 and L2) in order to discover how the caches affect the performance. Callgrind generates call-graphs, counts the number of function calls and keeps track of the number of instructions executed in a function. The *KCachegrind* tool [86] visualises output from both Cachegrind and Callgrind. The overhead of running Valgrind is 10 to 50 times the regular execution time. Valgrind supports Linux on x86, x86-64, ARM, PPC32, PPC64, S390X and MIPS, (Mac) OS X on x86 and x86-64

and Android on ARM. [69, 75]

PIN is a framework by Intel performing dynamic instrumentation of functions by recompiling the binary (machine code) just in time. PIN can instrument specific instructions, a certain class of instructions (e.g. branches) or whole procedures or functions. PIN is available on Linux and Windows. The framework could be used as a building block in a profiler, but it is not a complete profiler. [53, 54]

DynamicRIO is a framework designed for code manipulation and instrumentation at run-time. The framework works with basic blocks and enables interception after each basic block. DynamicRIO provides an API for building tools that perform e.g. code modification or profiling. A number of tools have been built on top of DynamicRIO, e.g. Dr. Memory, which is a memory profiling tool. The framwork supports Windows and Linux on the x86 and x86-64 architectures. [18, 19, 58]

PAPI (Performance API) is an open source performance library that provides a platform independent interface to hardware performance counters. The interface is split into two parts: a high level interface and a low level interface. The high level interface is easier to use and targets a limited, predefined set of events that are intended to be similar and comparable on different platforms. The predefined sets of events include for example counters related to memory hierarchy, cache coherence and cycle and instruction counts. The low level interface provides access to all native events (implemented by the processor) as well as to user defined sets of events. This interface is more flexible and gives more control to the programmer compared to the high level interface. The number of events that can be monitored simultaneously is limited by hardware, but PAPI can multiplex access to the performance counters at the cost of accuracy. PAPI uses the *perf_events* interface of the Linux kernel as of version 2.6.32 of Linux. If earlier versions of Linux are used, PAPI uses the *perfctr* or *perfmon* interfaces. PAPI does not require any special compilation of the application being profiled and supports profiling dynamically loaded libraries. The library is available for Linux and other UNIX-like operating systems on several architectures, such as Intel x86 and x86-64, ARM, MIPS and IBM POWER. PAPI is actively developed and support for measuring energy consumption has recently been added [84, 85]. This feature of the PAPI library is used in the energy profiling module added to IgProf as described in section 4.3. [50, 51]

There have been several interfaces and libraries bridging the gap between hardware performance counters and user space tools. Because user space tools usually cannot access all system registers or execute all instructions, one or two layers of middleware are needed between the hardware performance counters and tools executing in user space. *Perfctr* was an early interface

supporting both system-wide and per-thread monitoring [40, 73].

The *Perfmon2* performance monitoring interface is an alternative to the perfctr interface. The Perfmon2 interface needs to be built into the Linux kernel and is accessed by system calls. The *libpfm* library is a complementary user space library that lets tools and applications take advantage of the performance monitoring interface of Perfmon2. Furthermore, a command line tool, *pfmon*, makes use of the libpfm library and indirectly the Perfmon2 kernel interface in order to perform system-wide or per-thread profiling. The tool does not require any special compilation of the application to be profiled. The Perfmon2, libpfm and pfmon software supports Linux on Intel x86, x86-64, Itanium, IBM POWER, Sun SPARC and MIPS. [25, 59, 70]

The *perf_events* interface, also called *Linux Performance Event Subsystem*, is a more recent performance monitoring interface. The standardised performance monitoring interface was added to version 2.6.31 of the main Linux kernel. The *libpfm* library has been updated to take advantage of the perf_events kernel interface and works as the middleware library between the kernel interface and user space tools. There is also a command line tool, *perf*, which is a profiler using the perf_events interface. Recent versions of the perf tool supports the Running Average Power Limit (RAPL) interface that provides energy measurements on recent Intel processors (see section 4.3 for more details). The perf_events interface supports Linux on Intel x86, x86-64, Itanium, ARM, MIPS, IBM POWER, Sun SPARC and SH among others [83]. [23, 40]

Table 2.1: A comparison of some profiling software.

Software	General Properties				Technical Mechanisms			Runs on Linux + Architecture			
	Open source	Special compilation	Root privileges	Dynamically loaded symbols	Sampling	Instrumentation	Performance counters	x86	x86-64	ARM	AArch64
Tools											
IgProf	✓			✓	✓	✓		✓	✓	✓	✓
gprof	✓	✓			✓	✓		✓	✓	✓	✓
Google Performance Tools	✓				✓			✓	✓		
OProfile	✓		✓	✓	✓		✓	✓	✓	✓	
Intel VTune			?	?	✓		✓	✓	✓		
HPCToolkit	✓			✓	✓		✓	✓	✓		
Instruments				?	✓	✓	✓				
Libraries and frameworks											
Valgrind	✓			?		✓		✓	✓	✓	
Intel PIN				?		✓		✓	✓		
DynamicRIO	✓		?	?		✓		✓	✓		
PAPI	✓		?	✓			✓	✓	✓	✓	
Interfaces											
perfctr	✓		?	✓			✓	✓	✓	?	
perfmon2	✓		?	✓	✓		✓	✓	✓	?	?
perf_events	✓		?	✓	✓		✓	✓	✓	✓	✓

Chapter 3

Environment

This chapter describes the IgProf profiler and the fast stack trace feature of the libunwind library as they were before any porting efforts. Section 3.1.1 describes the functionality of IgProf, sections 3.1.2–3.1.4 describe how IgProf is used and sections 3.1.5–3.1.8 describe the technical mechanisms that IgProf relies on. Sections 3.2.1–3.2.2 describe the standard and fast stack tracing functionality of the libunwind library.

3.1 IgProf

3.1.1 Functionality

IgProf is mainly a performance and memory usage profiler. Profiling is a type of program analysis that is performed at run-time, i.e. it is dynamic as opposed to static program analysis, which is performed at compile-time. IgProf is modularly built and contains six profiling modules:

- performance profiling
- memory profiling
- empty memory profiling
- file descriptor profiling
- function instrumentation profiling
- function profiling.

Section 4.3 describes how an energy profiling module was added to the set of available profiling modules.

Performance profiling means that the time spent running the program and in particular the time spent running each function of the program is measured and analysed. IgProf samples the program state periodically in order to collect statistics about how much time the application spends in each function of the application. When it is known where in the application most time is spent, optimisation efforts can be targeted at these locations.

Memory profiling means that dynamic memory management is tracked during the execution of the application. Dynamic memory can be allocated using the *malloc* function of the C library and released using the *free* function. The *calloc* function is similar to *malloc*, but fills the allocated memory block with zeros. A memory block previously allocated using the *malloc* and *calloc* functions can be resized using the *realloc* function. IgProf catches calls to these and a few other memory management functions by instrumenting them (see section 3.1.7). IgProf collects statistics about how often (the frequency) and how much (the amount) memory is allocated and released. Active memory allocations, i.e. memory blocks that have been allocated but not released, at the termination of the application are memory leaks.

Another type of memory profiling is also available in IgProf: empty memory profiling. There are three modes of operation available when profiling empty memory. In the default mode of operation IgProf scans for pages containing only zeros at release. When a memory allocation function (e.g. *malloc*) requests a new page of memory from the operating system, the operating system usually returns a page filled with zeros. This is different from the *calloc* function that fills the memory block with zeros every time the function is called, whereas a new page is filled with zeros only once before the operating system makes the new page available. If the page is part of a memory block that is first released and then allocated again using *malloc*, the page is not filled with zeros again, because the page is not new anymore. If the page contains only zeros at the time of release, it is likely that the application has not used the page at all during the lifetime of the *application execution* or the application has set the memory to zero on purpose, e.g. using the *calloc* or *memset* functions or in initialisation functions. In the second mode of operation a memory block is filled with a bit pattern at allocation, but the profiler still scans for pages containing only zeros. If the page contains only zeros at the time of release, the application has set the memory to zero on purpose. In the third mode of operation the profiler fills a memory block with a bit pattern at allocation and scans for pages containing the same bit pattern at release. If a page still contains the bit pattern at the time of release, it is likely that the application has not used the page during the lifetime of the *most recent allocation*.

IgProf can not only keep track of active memory resources, but also file descriptors. A file descriptor is allocated when a file is opened using the *open* function or indirectly using the *fopen* library function (calls the *open* function). The file descriptor duplication functions *dup* and *dup2* as well as the socket functions *socket* and *accept* also allocate file descriptors. File descriptors are released using the *close* function. IgProf catches calls to these functions and keeps track of active, i.e. allocated but not released, file descriptors during file descriptor profiling.

The function instrumentation profiler is similar to the performance profiler in the sense that it measures the time the application spends in each function. However, in this mode of operation IgProf does not use statistical sampling to measure the time, but instead each function is instrumented at compile-time. IgProf measures the time the function spends executing and counts the number of times the function is called. This is the only type of profiling in IgProf that needs recompilation of the application with a profiling flag enabled.

The function profiler, like the function instrumentation profiler, uses function instrumentation to measure the time spent in a specific function and to count the number of times the function is called. Unlike the function instrumentation profiler, the function profiler uses the dynamic instrumentation mechanism of IgProf (see section 3.1.7) and does not require any special compilation.

The difference between performance profiling, function instrumentation and function profiling is the scope of the profiling. Performance profiling targets all functions of the application and all libraries loaded for the application. Function instrumentation targets the functions of one or more translation units. A translation unit comprises a C/C++ source file and all included header files. Functions in libraries that are not recompiled with the profiling flag enabled are not instrumented. Function profiling targets a specific function.

3.1.2 Counters

IgProf uses the notion of counters when collecting data. A counter represents a variable in a data set, i.e. something that is being measured or counted. The counters can be thought of as columns in a table of collected data. Each datum in the data set has an associated stack trace, including the current location of execution. This allows observations, or measurements, to be linked to a specific location in an application. The performance profiling module, having a single counter, makes explicit use of the stack trace associated with each datum. Multiple counters can be in use during a single profiling run, e.g.

the memory profiling module has three counters: one for the total amount of allocated memory, one for the maximum amount of allocated memory and one for the live memory allocations, i.e. the total amount of allocated memory less the amount of released memory. Table 3.1 shows the counters available in the different profiling modules of IgProf.

Table 3.1: Counters available in the different profiling modules of IgProf.

Profiling Module	Counters
Performance	PERF_TICKS
Memory	MEM_TOTAL, MEM_MAX, MEM_LIVE
Empty memory	MEM_LIVE
File descriptor	FD_USED, FD_LIVE
Function instrumentation	CALL_TIME, CALL_COUNT
Function	CALLS_TOTAL
Energy	NRG_PKG, NRG_PP0, NRG_PP1, NRG_DRAM

The energy profiling module was implemented as part of this thesis. The module and the related counters are described in section 4.3.

3.1.3 Using IgProf

Profiling using IgProf is done in two steps. In the first step, profiling data is collected while the application to be profiled is running. In the second step, the collected profiling data is analysed and presented. This section describes the first step. The following section (section 3.1.4) describes the second step.

IgProf is a command line profiling tool. Unlike some other profiling tools, e.g. gprof, IgProf does not generally require recompilation of the application with any profiling flags enabled. The application is compiled normally and IgProf profiles when the application is running. The command line of the application, including any arguments to the application, is prepended with *igprof* and the arguments to IgProf:

```
igprof arguments to igprof application arguments to the application
```

The arguments to the application naturally depend entirely on the application, whereas the arguments to IgProf determines how IgProf runs. Some arguments to IgProf do not depend on the type of profiling that is being

performed, but are more general in nature. The *-o* or *--output* argument and the following argument specify the output file. Unless an output file is specified, the default name of the output file conforms to the pattern *igprof.pid.gz*, where *pid* is the process id of the application running. The *-z* or *--compress* argument will compress, or more specifically gzip, the output file. When profiling large applications, IgProf may generate a large amount of output and it can be a good idea to compress the output.

The *-pp* or *--performance-profiler* argument enables the performance profiler. The default way of measuring the time of execution in the performance profiler is to include both the time spent in user mode and the time spent in kernel mode on behalf of the application. When the -pu or --user-time argument is given to IgProf, the time spent only in user space is measured. When the *-pr* or *--real-time* argument is given, real (wall clock) time is measured. See section 3.1.6 for implementation details.

The *-mp* or *--memory-profiler* argument enables the memory profiler in IgProf. The *-mo* or *--memory-overhead* argument and the following specifies how memory overhead, imposed by memory alignment restrictions and minimum allocation size, is taken into account. There as three alternatives for the following argument: *none*, *include* and *delta*. The *none* alternative does not include the overhead in memory statistics, the *include* alternative does include the memory overhead and the *delta* alternative records the overhead itself.

The *-ep* or *--empty-profiler* argument enables the empty memory profiler. When a memory block is released, the empty memory profiler scans for pages containing only zeros. When the *-ei* or *--empty-init-memory* argument is given, memory blocks are filled with a bit pattern at allocation, but the profiler still scans for pages containing only zeros at release. When the *-eu* or *--empty-track-unused* argument is given, memory blocks are filled with a bit pattern at allocation and the profiler scans for pages containing only the bit pattern.

The *-fd* or *--file-descriptor* argument enables the file descriptor profiler. The file descriptor profiler does not make use of any further arguments.

The *-finst* argument enables the function instrumentation profiler. However, the source files targeted for instrumentation need to be compiled by gcc with the *-finstrument-functions* flag enabled. The compiler inserts an extra function call at the beginning and at the end of each function. Not all source files need to be compiled with the flag.

An argument beginning with the *-fp* substring enables the function profiler. IgProf profiles memory allocation functions when the *-fp:malloc* argument is given. When a function to be profiled returns an integer or a pointer, the argument follows the pattern *-fpi:funcname*, where *funcname* is

the name of the function to be profiled. When a function returns a floating point result, the argument follows the pattern *-fpf:funcname*. Any of the *-fp* arguments can be appended by a colon and the name of a library in order to specify the library the function is part of.

3.1.4 Analysing and presenting profiling data

When the data collection step of the profiling has finished, the result is a file containing profiling data. The *igprof-analyse* program analyses and presents the results of the profiling:

```
igprof-analyse -d -v -g profiling-data-file
```

The *-d* argument demangles function names describing locations of execution. When a C++ function is compiled, its name is mangled in order to differentiate multiple functions sharing the same name but having different numbers of arguments or arguments of different type [61]. Exactly how the compiler mangles, or transforms, the names is specific to each compiler [34]. For example, when version 4.8.2 of the GNU C++ compiler, g++, compiles the functions *int func(int)*, *int func(double, char)* and *int func(void)* on x86-64, the compiler assigns the names *_Z4funci*, *_Z4funcdc* and *_Z4funcv* respectively to these functions. The mangled names are used when an application is linked and in the resulting linked application. If the *-d* argument is not used when running *igprof-analyse*, the mangled name is shown. However, the mangled names are not particularly easy or intuitive to read, so *igprof-analyse* offers demangling of names.

The *-v* or *--verbose* argument enables more verbose output from the *igprof-analyse* program, whereas the *-g* or *--gdb* argument makes the analyser use a combination of the *gdb* [32], *nm* [35] and *objdump* [36] tools for resolving addresses to function names. *gdb* is the GNU debugger, *nm* is a tool that lists symbols from object and executable files and *objdump* is a tool that provides more general information about object and executable files.

The *-r* or *--report* argument followed by the name of a counter selects which counter to show in the output. By default the first counter is presented. Table 3.1 shows the counters available in the different profiling modules of IgProf.

3.1.5 Loading IgProf

The IgProf profiler is implemented as a shared (dynamically loaded) library. The reason for this is that a shared library can be injected into the process of

an application at load time without any need to relink or otherwise modify the application. This mechanism is a feature of the GNU loader, *ld* [1], that loads libraries referred to in the *LD_PRELOAD* environment variable before any other libraries are loaded. Any initialisation code in the preload libraries is run before the actual application is run. The modules of IgProf run their initialisation code when the shared library is loaded. This initialisation code sets up interval timers in case of performance profiling and instruments functions in all modes of profiling.

The preloaded and other shared libraries used by the application are part of the same process as the application itself. This means that they share the resources that are available to the process, e.g. memory space, memory mapping, file descriptors, child processes, signals and signal handlers [76]. The application can access data and code in the libraries, but the libraries can also access the data and code in the application, because the application and the libraries are loaded into the same process.

The interface between the user and the shared library of IgProf is a shell script called *igprof*. The shell script parses and validates command line arguments for IgProf itself. The rest of the command line is preserved for the application to be profiled. The shell script assigns the parameters of igprof to environment variables that the shared library can access. Because IgProf is loaded as a shared library, it is not possible to provide parameters as command line arguments, but environment variables are accessible also to shared libraries. In addition to the parameters, the script also sets the environment variable LD_PRELOAD to the path of the shared library of IgProf. Finally the script runs the command line of the application as provided by the user, which causes the IgProf shared library to be loaded before the application starts executing, because the script earlier set the LD_PRELOAD environment variable.

3.1.6 Statistical sampling and interval timers

The performance profiling in IgProf is based on statistical sampling. Statistical sampling means that the profiler samples the program state periodically in order to collect statistics on how much time an application spends executing in each part of the program. A distribution of the execution location is obtained, when the execution location is sampled and stored periodically. When function entries are present in the symbol table of a program, it is possible for the profiler to know which function each sampled execution location is part of, i.e. which function was called at each sampling event.

IgProf uses an interval timer to interrupt the running program and sample the execution state. The interval timer is started using the *setitimer* function

[4] and the interval between sampling events is specified. When the timer has expired, a signal is generated and the timer is restarted. There are three different interval timers available as shown in table 3.2.

Table 3.2: Interval timers.

Interval timer	Signal	Argument to IgProf
ITIMER_PROF	SIGPROF	-pp, --performance-profiler
ITIMER_VIRTUAL	SIGVTALRM	-pu, --user-time
ITIMER_REAL	SIGALRM	-pr, --real-time

ITIMER_PROF is the default interval timer used in IgProf. The timer runs both when the process executes in user mode and in kernel mode on the behalf of the process. When the timer expires, the signal SIGPROF is sent to the process. In order to be able to catch the signal, IgProf needs to set up a signal handler for this signal using the *sigaction* function before starting the interval timer. Alternatively, if the *-pu* or *--user-time* argument is given to the *igprof* script, IgProf uses the ITIMER_VIRTUAL interval timer that runs only when the process executes in user space, but not in kernel space. In this case, when the timer expires, the signal SIGVTALRM is sent to the process. A third alternative is to give *-pr* or *--real-time* as an argument to the igprof script. IgProf will then use the ITIMER_REAL interval timer that runs in real, or wall clock, time. When the timer expires, the signal SIGALRM is sent to the process. [4]

3.1.7 Function instrumentation

The profiler needs to catch some functions calls in order to count the number of times a function has been called, to inspect the arguments of the function call or to detect function calls that possibly would stop IgProf from working. In order to catch the function calls, a small block of memory, a trampoline, is allocated. The memory block needs to be writable and executable. The trampoline comprises four parts as shown in figure 3.1: a jump to a wrapper function, a copy of the first few instructions of the original function, a jump to the instruction immediately following the first few instructions that were copied and finally, a patch area. The use of the patch area is described in section 4.1.3. Even if the trampoline comprises mostly executable code, the trampoline is not a complete function. It lacks a function prologue, a function epilogue and does not return to the caller.

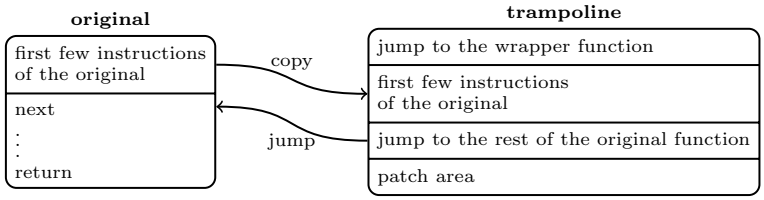

Figure 3.1: The trampoline memory block.

The first few instructions of the original function to be instrumented are overwritten with a jump instruction. The target of the jump is the beginning of the trampoline. A data structure, called a hook, is associated with the trampoline. The hook contains a function pointer to the location in the trampoline where the original first few instructions are stored. This function pointer is called the chain. IgProf uses a wrapper function in order to provide a replacement function with the hook structure as a function argument in addition to the original arguments. The wrapper function then calls the replacement function.

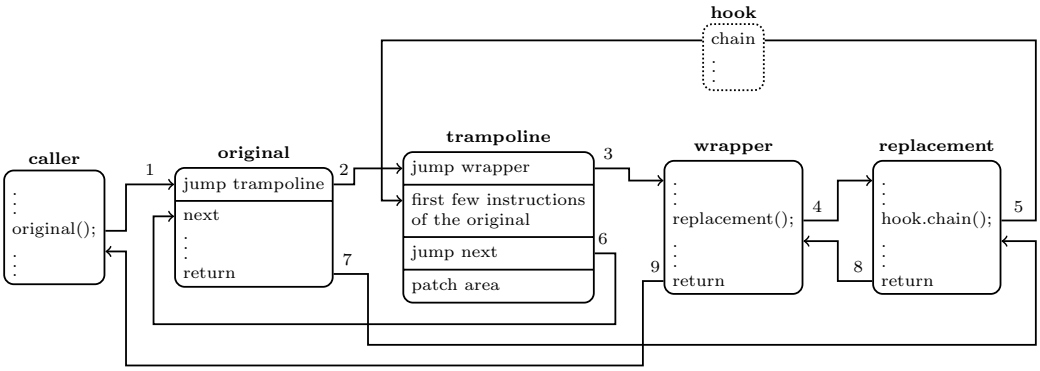

Figure 3.2: Function call to a an instrumented function.

Figure 3.2 shows the transfer of control when an instrumented function is called. When the caller function calls the original function, the control is transferred to the beginning of the instrumented original function (step 1) as expected. Because the first instruction of the instrumented function is now a jump, the execution transfers to the beginning of the trampoline (step 2). The trampoline also starts with a jump instruction; this time the target is the wrapper function (step 3). The wrapper function calls the replacement function (step 4), adding the hook structure to the arguments. The replacement function can inspect or modify the arguments, keep track of resource usage or just count the number the function has been called. At some point the replacement function usually calls the original function by using the chain function pointer of the hook structure (step 5). The chain

pointer points to the copy of the first instructions of the original function. When these instructions have been executed, the execution will continue with the jump (step 6) to the instructions following the replaced instructions in the beginning of the original function. When the original function returns, control is transferred back to the real replacement function (step 7). In case the replacement function does not call the original function through the chain pointer of the hook structure, steps five to seven are omitted. When the replacement function returns, control is transferred back to the wrapper function (step 8) as expected. However, when the wrapper function returns (step 9), the control is transferred to the caller function, because neither step 1 nor step 2 changed the call stack.

3.1.8 Stack tracing in IgProf

It is common that functions call other functions that in turn call other functions, et cetera. These nested function calls form a chain of function calls and the list of functions in the chain of function calls is called a stack trace. If a function f calls function g that in turn calls function h, the stack trace of the nested function calls is the list [h, g, f], where the most recent function call comes first. The name stack trace refers to the call stack, because at least some data is pushed on the stack when doing nested function calls. Exactly what data is pushed on the stack depends on the architecture and the calling convention, but can be e.g. the return address, arguments and the values of registers to be preserved during a function call.

When IgProf makes an observation during profiling, the current stack trace is attached to the observation (see section 3.1.2). IgProf itself implements stack tracing on x86, whereas it relies on the stack tracing feature of the libunwind library to perform stack tracing on the x86-64, ARM and AArch64 architectures. This section describes the most commonly used calling convention on the x86 architecture, cdecl, and how stack tracing works with this calling convention [57]. The calling conventions on UNIX-like systems on the x86-64 [66], ARM [12] and AArch64 [14] architectures are slightly more complex than the cdecl calling convention and are described briefly in sections 3.2.1 and 4.2.

The call stack consists of the call frames of nested function calls. The frame pointer points to the most current frame. On the Intel x86 architecture the base pointer register, *ebp*, usually contains the frame pointer. The stack pointer is called *esp*. The stack grows downwards on most platforms, including x86. [57]

How exactly a call frame looks like depends on the architecture and the calling convention used. The calling convention defines how arguments and

return values are passed between functions. Arguments and return values can be placed on the stack, in registers or a combination of both. The calling convention defines in what order arguments are passed and the division of the responsibilities between the calling function, i.e. the caller, and the called function, i.e. the callee.

On the Intel x86 architecture, the GCC compiler uses the *cdecl* (C declaration) calling convention for C programs [29]. The *cdecl* calling convention states that arguments are passed right to left on the stack and integer return values are stored in the *eax* register. The caller has the responsibility to clean up the stack after the function call. [57]

If there is a function called *func*, with the prototype *int func(int arg1, int arg2, int arg3)* and the function is called : *func(1, 2, 3)*, the function call would be translated into the following assembly (written in the AT&T assembly syntax used by the GNU assembler):

```
pushl $3       ; push the third argument (the value 3)
pushl $2       ; push the second argument (the value 2)
pushl $1       ; push the first argument (the value 1)
call func      ; call the function
addl $12, %esp ; restore the stack
```

The arguments are pushed onto the stack in reverse order, i.e. from right to left. The call instruction first pushes the address of the next instruction to be executed after the call, i.e. the address of the *addl* instruction in the example, on the stack and then jumps to the address of the function specified (func in the example). When the function has returned, the stack needs to be restored. Instead of the stack being popped three times and the results being discarded, the stack pointer is directly increased by the number of bytes allocated for the arguments, i.e. 12 bytes (three integers four bytes each).

A function begins with a function prologue and ends with a function epilogue. On the x86 architecture, a typical function prologue looks like this:

```
pushl %ebp      ; push the previous base pointer (frame pointer)
                ; on the stack
movl %esp, %ebp ; copy the stack pointer to the base pointer
subl $20, %esp  ; decrement the stack pointer by 20 for 20 bytes of
                ; space for automatic local variables
```

The previous base pointer is pushed on the stack, so that it can be restored in the function epilogue. The stack pointer is copied to the base pointer, i.e. the base pointer now contains a pointer to the call frame of the called function. The base pointer is generally not modified in the function between the function prologue and the function epilogue, i.e. in the body of

the function. This means that the base pointer contains a copy of the stack pointer before automatic local variables are allocated on the stack. Finally, the stack pointer is decremented by the number of bytes used by the automatic local variables in the function. In the example 20 bytes are allocated, which could hold for example five integers four bytes (32 bits) each.

Figure 3.3 shows the call frame after the function prologue has been executed. Because the stack usually grows downwards on Intel x86, previous call frames are located at higher addresses than the current call frame. The arguments are pushed on the stack in reverse order. The call instruction causes the return address of the caller to be pushed on the stack. The function prologue pushes the previous frame pointer on the stack and allocates memory for automatic local variables on the stack.

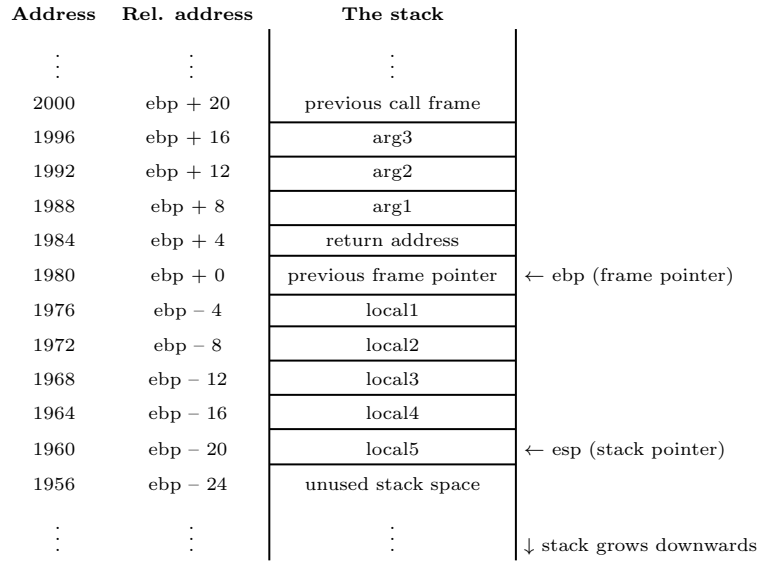

Figure 3.3: The call frame after the function prologue has executed.

The function epilogue follows the body of the function. On the x86 architecture, a typical function epilogue looks like this:

```
movl %ebp, %esp ; copy the base pointer back to the stack pointer
popl %ebp       ; pop the previous base pointer (frame pointer)
                ; off the stack
ret             ; return to the caller
```

The function epilogue effectively undoes what the function prologue has done. The epilogue restores the stack pointer to the value it had before the automatic local variables were allocated by copying the base pointer to the stack pointer. Note that there is no need to add the number of bytes

allocated for local variables to the stack pointer. The previous base pointer (frame pointer) is popped off the stack and now points to the call frame of the caller. The first two instructions of the function epilogue are usually replaced by a single *leave* instruction that has the same effect and is shorter. Finally, the *ret* instruction returns to the caller by first popping the return address off the stack and then jumping to this return address.

The previous frame pointer points to the previous call frame. This is also true for the previous call frame et cetera. The frame pointers on the stack create a chain of call frames. When this chain is followed and the return address of each call frame is stored, the whole chain of nested function calls, i.e. the stack trace, is revealed.

3.2 Stack tracing in libunwind

The main purpose of the libunwind library is to provide functionality to unwind stack frames. A use case for stack unwinding is e.g. exception handling. When an exception triggers, the stack needs to be unwound, one or several frames. The libunwind library also provides the functionality to obtain the current stack trace. IgProf uses only the stack trace feature of libunwind.

3.2.1 Standard stack tracing

The calling conventions differ slightly on x86 and x86-64. Like on the x86, the return address is pushed on the stack when a function call is done and popped off the stack when the function is returning. However, the first six integer arguments of a function are treated differently. They are stored in the registers *rdi*, *rsi*, *rdx*, *rcx*, *r8* and *r9*. Subsequent arguments are stored on the stack, still from right to left, like on the x86 architecture. Unlike on x86, the calling convention on x86-64 does not require that the base pointer, called *rbp* on x86-64, takes the role of a frame pointer. If the base pointer is not used as the frame pointer, the stack pointer, called *rsp* on x86-64, takes the function of the frame pointer instead. In fact the base pointer register is allowed to be used as a general-purpose register. The compiler can keep track of which register to use and the resulting program works correctly, but when a stack trace is performed the same way as on x86, it may fail. [49, 66]

Some additional information is needed when the stack is traced on x86-64 [49]. This information may be found in the executable in the form of unwind information [87]. The unwind information is stored in a *.eh_frame* [77] or *.debug_frame* [37, 71] section in the executable. The information is necessary for programs coded in languages that implement exceptions, e.g.

C++, but it is also useful for debugging purposes, e.g. when *gdb* (the GNU debugger) performs a backtrace (stack trace). The unwind information keeps track of whether registers change during the execution of a function and in case they do change, how the previous value of the register is saved. Usually the previous value of a register that is reused or might be overwritten by a function call is saved on the stack at some offset from a reference register, i.e. either the stack pointer or the frame pointer. The unwind information specifies the offset and the reference register.

The libunwind library has the functionality to parse and use the unwind information found in executables [68]. The library also implements stack tracing that uses the unwind information. The standard (and slow) way of obtaining a stack trace in libunwind is to first save the state of the registers (the context) to a data structure called the *cursor* and then unwind the stack one frame at a time as many times as possible.

Each frame has an associated value of the program counter and the canonical frame address (CFA). The values of the program counter of each frame make up the stack trace. The CFA is defined as the value of the stack pointer when the caller calls the callee [37]. The stack pointer may change before execution reaches the start of the called function, e.g. on the x86-64 architecture the return address is pushed on the stack, which modifies the value of the stack pointer. In this case, the CFA and the value of the stack pointer are not the same when the function prologue is entered. [71]

3.2.2 Fast stack tracing on x86-64

Each time the stack is being traced, the context is first saved and then the stack is unwound one frame at a time based on the unwind information. When an application is profiled for performance or memory usage using IgProf, the stack is traced many times a second. When the stack is being traced, it is not unlikely that some location of execution, i.e. the value of the program counter associated with a frame, has already been encountered during previous stack unwinding. If an application spends 80 % of the execution time in 20 % of the code [31], i.e. it follows the 80–20 rule or a variant of the Pareto principle, there might be opportunities for optimisations.

One such optimisation is to cache the data needed to calculate the previous frame based on the current frame. The first time a location of execution is encountered, the standard unwind step is performed and the information necessary to calculate the previous location of execution and the previous frame address is cached. The cache of the frame descriptors is implemented as a hash table with the location of execution as the key. When the same location of execution is encountered again, the cache already has the infor-

mation necessary to find the previous frame and there is no need to parse and interpret the unwind information.

A frame descriptor contains the following fields:

- frame type: standard, signal, guessed or unknown
- which register is the reference register (the stack pointer or the base register)
- the offset to add to the reference register to calculate the CFA
- the offset to add to the CFA to calculate the address of the previous base pointer
- the offset to add to the CFA to calculate the address of the previous stack pointer

The most common frame type is the standard frame. When a frame descriptor has been obtained, either from the cache or as the result of an unwind step, the following steps are carried out to calculate the previous frame based on the current frame and the frame descriptor:

1. the previous CFA is the value of the current reference register, i.e. the stack pointer or the base pointer, ± an offset
2. the previous location of execution is obtained from the stack at the address CFA – 8
3. the previous base pointer is either the current unmodified base pointer or the CFA ± an offset
4. the previous stack pointer is the CFA itself.

A signal frame is pushed on the stack when a signal occurs and the operating system saves the context, i.e. a snapshot of the whole register file, and some signal related information on the stack before the signal handler executes. For this frame type the fast stack tracing stores only the size of the signal frame in the frame descriptor. The previous CFA is calculated as the current CFA + the size of the signal frame. The previous location of execution, the previous stack pointer and the previous base pointer are obtained directly from the context structure on the stack.

The guessed frame type is similar to the standard frame type, except that memory accesses are validated more strictly. If a frame is not recognised as standard, stack or guessed frame type, frames are unwound the standard (and slower) way, using the unwind information.

Another opportunity for optimisation is the very first step of stack tracing, saving the context. In practice stack unwinding is only dependent on a few registers for the purpose of building the stack trace. The GNU C library implementation of the *getcontext* function saves 20 registers, 14 general purpose registers and the regular and SSE (SIMD) floating point state registers in the x86-64 implementation. In addition to this, the signal mask is obtained through a system call, which is costly because of the context switch between user and kernel mode. The libunwind library implementation of *getcontext* excludes the system call. Furthermore, there is an implementation of *getcontext* optimised for the fast stack feature. It saves only the *rbx*, *rbp*, *rsp* and *rip* registers, i.e. four out of 20 registers, and excludes the system call.

Chapter 4

Implementation

This chapter describes the port of IgProf to the AArch64 architecture (section 4.1) and the port of the fast stack trace feature in the libunwind library to both the AArch64 and the ARM architectures (section 4.2). More specificically, this chapter describes what has been ported and how it has been ported. Furthermore, section 4.3 describes how an energy profiling module was implemented in IgProf.

4.1 The port of IgProf to AArch64

The code of IgProf is mostly architecture independent, which means that most of the code works on any architecture without modifications. There are, however, some parts of the code that are implemented separately for each supported architecture. The separate implementations are due to differences in instruction encoding, the lack of atomicity in C/C++ and differences in how system registers are accessed. The main tasks that are architecture dependent in IgProf are:

- generating jumps
- identifying PC-relative instructions
- patching PC-relative instructions
- atomic increment and decrement operations
- reading the cycle counter register.

Sections 4.1.1–4.1.5 describe the details of porting these five tasks to the AArch64 architecture.

4.1.1 Generating jumps

When functions are instrumented (section 3.1.7), jumps to and from the trampoline are generated (see figure 3.1). Three alternatives are presented in this section for generating jumps.

The first alternative comprises a single relative jump instruction, *b*:

```
b relative_address ; jump to the relative address
```

The relative address is the difference between the destination and source addresses. The advantages of this solution are that the generated instruction sequence is short, only one instruction, and it does not overwrite any register. The disadvantage is that the jump can reach only ±128 MB from the location of the *b* instruction.

The second alternative is a sequence of three instructions:

```
adrp x16, abs_addr              ; load bits 12..63 of address
add x16, x16, #low_address_bits ; load bits 0..11 of address
br x16                          ; jump to the address in register x16
```

The *adrp* instruction loads the higher 52 bits of the absolute target address, based on the relative address of the target, into the *x16* register. The calling convention for AArch64 allows the use of registers *x16* and *x17* as scratch registers between a function call and the execution of the called function [14]. The *add* instruction loads the lower 12 bits of the absolute address. The address is loaded in two steps, because every instruction is 32 bits on AArch64 and some of the bits are used to encode the instruction itself. Finally, the *br* instruction jumps to the address in register *x16*. This alternative can jump at most ±4 GB from the location of the *adrp* instruction. The weaknesses of this solution is that the *x16* register is overwritten and the jump sequence is three instructions long, while the range of the jump does not cover the whole address space.

The third alternative is similar to the second alternative in the sense that first the absolute address is loaded into the *x16* register and then the jump is carried out:

```
ldr x16, addr_ptr ; load the address at the address pointer
                  ; into register x16
br x16            ; jump to the address in register x16
addr_ptr:
.dword abs_addr   ; the absolute address itself
```

Instead of loading the absolute address in two instructions, the *ldr* instruction loads the address in one instruction. The address encoded as part of the

ldr instruction, i.e. *addr_ptr*, is merely a pointer to the full 64-bit address, because every instruction is 32 bits and some of the bits are used to encode the instruction itself. Like in the second alternative, the *br* instruction jumps to the address in the register. Finally the absolute address (the literal) is appended to the jump sequence. The 64-bit absolute address (the *.dword* pseudoinstruction) corresponds to two instructions in terms of space needed. This solution has the advantage that the jump target can be at any address in the 64-bit address space. The disadvantages are that the *x16* register is overwritten and the jump sequence is four instructions long.

The first alternative, the one-instruction jump, is used at the beginning of an instrumented function, in order to transfer control to the beginning of the trampoline (step 1 in figure 3.2). Because the *b* instruction can jump at most ±128 MB, the trampoline needs to be allocated within ±128 MB from the start of the instrumented function. When the trampoline is allocated, the file /proc/self/maps, containing a memory map of the current process, is examined to find a memory area close enough to the instrumented function. The same mechanism is used on the x86-64 architecture to find memory areas for trampolines. A benefit of using the one-instruction jump is that only one instruction needs to be relocated to the trampoline. The jump back from the trampoline to the instrumented function (step 6 in figure 3.2) is also a one-instruction jump, because the distance between the start of the instrumented function and the trampoline is the same, even if the direction is the opposite.

The third alternative, equivalent to four instructions in length, is used to transfer control from the trampoline to the wrapper function (step 3 in figure 3.2). This solution allows the wrapper function to be located anywhere in the address space. At this point in execution no instruction of the original uninstrumented function has been executed and the *x16* register can be used safely to load the address of the jump target.

4.1.2 Identifying PC-relative instructions

When the first few instructions are copied from the beginning of the original function to the second section of the trampoline as part of the dynamic instrumentation, most of the instructions will execute equally well at their new location in the trampoline. There are, however, eleven instructions that are sensitive to their location in memory at execution. These instructions use addressing relative to the program counter, i.e. the address of the instruction being executed. The eleven instructions can be divided into three groups:

- the load instructions *ldr* and *ldrsw*
- the address calculation instructions *adr* and *adrp*

- the relative jump instructions *b*, *bl*, *b.cond*, *cbz*, *cbnz*, *tbz* and *tbnz*.

These eleven instructions need to be modified, or patched, in order to work correctly. Before the PC-relative instructions are patched, they need to be identified. The tasks of identifying and patching instructions could be combined into a single task, but the structure of IgProf suggests that these tasks are performed in two steps.

Figure 4.1 shows the encoding of the instructions using PC-relative addressing. All instructions are represented as 32-bit words on the AArch64 architecture. The fields consisting of zeros and ones determine which instruction is to be executed, e.g. *adr* or *cbz*, and determines the layout of the other fields of the instruction word. Identifying a specific instruction is a matter of matching a bit pattern of the instruction word being examined and the bit pattern of the instruction prototype. Only the bits positions containing zeros and ones in figure 4.1 are considered; the other bit positions belong to fields having variable content.

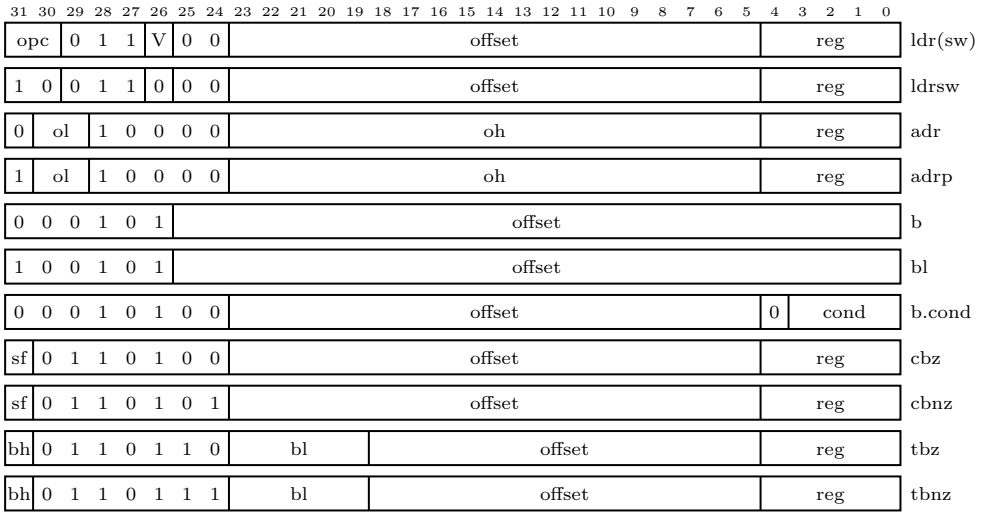

Figure 4.1: Encoding of the instructions using PC-relative addressing.

Figure 4.2 shows an example of how instructions are identified in practice. Instructions are identified by first calculating the bitwise AND of the instruction word and a mask. The mask contains ones in the bit positions having fixed bit values, i.e. the field(s) that encode the instruction itself, and zeros in the bit positions having variable bit values. The result of the bitwise AND is then compared to an instruction prototype and if they are equal, the instruction is correctly identified. In other words, if the expression (*instruction* AND *mask*) == *prototype* is true, the instruction being examined matches the prototype. The instruction prototype contains zeros in the bit

positions having variable bit values and the unmodified bit values in the bit positions having fixed bit values. The mask and the instruction prototype are specific to the instruction to be tested for a match. Figure 4.2 shows how the mask and the prototype are formed based on the encoding of the *adr* instruction.

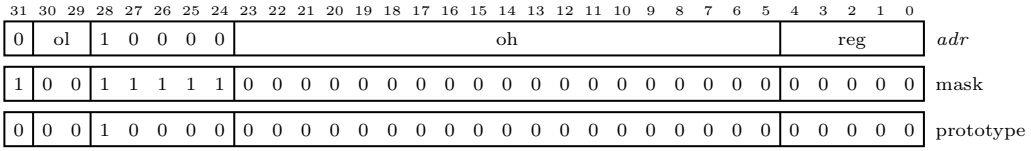

Figure 4.2: Identification of the *adr* instruction.

4.1.3 Patching PC-relative instructions

The instructions that use relative addressing need to be patched in order to work correctly. Depending on the functionality of the instruction, there are different solutions to achieve the same effect as if the instruction were executed in the original location. It is not necessary to execute the exact same instruction; it can be replaced by another instruction or a sequence of instructions that carries out the same task as the original instruction. Ideally the solution should be short and not modify registers that the original instruction does not modify.

The first group of instructions using relative addressing comprises the load instructions *ldr* and *ldrsw*. The *ldr* (load register) instruction loads a value into a register. There are a few variants of the *ldr* instruction, but only one variant loads a (constant) literal relative to the program counter. This variant of the *ldr* instruction loads a word at an address relative to the program counter into a register. Figure 4.1 shows the encoding of the *ldr* instruction. The *offset* field contains a PC-relative address that is first scaled (multiplied) by four and then sign extended. The effective address is obtained by adding the scaled and sign extended offset to the value of the program counter. The relative address has a range of ±1 MB and a granularity of 4 bytes.

Table 4.1 shows how the V and *opc* fields determine the bit width of the literal and the type and width of the destination register. The AArch64 architecture has two sets of integer registers, w and x, and three sets of SIMD (single instruction, multiple data) and floating point registers, s, d and q. The w and s registers are 32-bit wide, the x and d registers are 64-bit wide and the q registers are 128-bit wide. The *reg* field of the *ldr* instruction

further specifies the number (0–31) of the register inside the register set, e.g. if $V = 0$, $opc = 00$ and $reg = 2$, the destination register is *w2*.

Table 4.1: Characteristics of the literal and the destination register addressed by the *ldr* and *ldrsw* instructions.

V	opc	width of literal	width of register	register set	instruction
0	00	32	32	w	ldr w*reg*, *offset*
0	01	64	64	x	ldr x*reg*, *offset*
0	10	32	64	x	ldrsw x*reg*, *offset*
1	00	32	32	s	ldr s*reg*, *offset*
1	01	64	64	d	ldr d*reg*, *offset*
1	10	128	128	q	ldr q*reg*, *offset*

The *ldrsw* (load register signed word) instruction is very similar to the *ldr* instruction, but has an addition step. The 32-bit literal is sign extended to 64 bits to match the bit width of the destination register.

When an *ldr* or *ldrsw* instruction is copied from the beginning of the original function to the trampoline, the relative address of the instruction does not point to the location where the literal is stored anymore. One solution is to recalculate the relative address of the literal, but this is possible only when the literal and the trampoline are close enough, i.e. at most 1 MB apart. A more general solution is to copy the literal itself to the patch area of the trampoline and change the relative address in the *ldr* or *ldrsw* instruction to the relative address of the literal in the patch area. There is no need to change the *reg* field of the *ldr* or *ldrsw* instruction.

The second group of instructions using relative addressing comprises the address calculation instructions *adr* and *adrp*. The *adr* and *adrp* instructions calculate the effective (absolute) address of an address relative to the program counter and store the effective address in a register. Figure 4.3 shows the encoding of the *adr* instruction and the calculation of the effective address. The offset is encoded as a PC-relative address with a range of ±1 MB and a granularity of one byte. The offset is split into two parts: the two lowest bits are stored in the *ol* (offset low) field and the higher bits are stored in the *oh* (offset high) field. The offset is first sign extended to 64 bits and then added to the value of the program counter to form the effective address. Finally the effective address is loaded into the x register specified by the *reg* field of the instruction, e.g. *x2* if $reg = 2$.

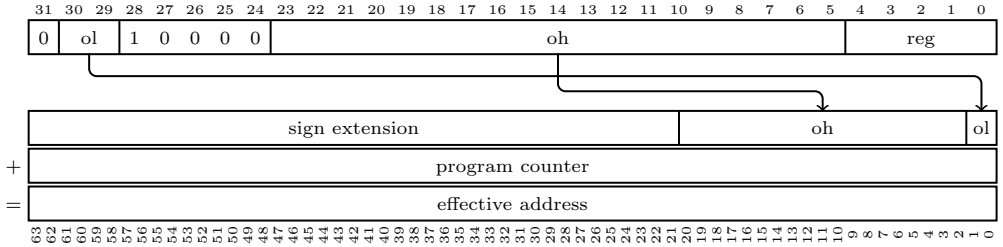

Figure 4.3: The encoding of the *adr* instruction and the calculation of the effective address.

The *adrp* instruction works similarly to the *adr* instruction, but instead of calculating the address of an individual byte, *adrp* calculates the address of a 4096-byte page. Figure 4.4 shows the encoding of the *adrp* instruction and the calculation of the effective address. The offset is split into two fields, like for the *adr* instruction, but is scaled (multiplied) by 4096. This gives a range of ±4 GB for the relative address. The lowest twelve bits of the value of the program counter used for the address calculation are cleared to zero in order for the calculation to result in an effective address that aligns to 4096-byte pages. The *reg* field specifies which x register is the destination register.

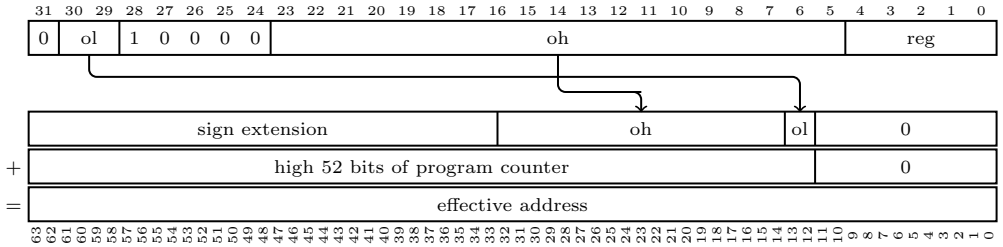

Figure 4.4: The encoding of the *adrp* instruction and the calculation of the effective address.

Because the *adr* and *adrp* instructions use relative addressing, the calculation of the effective address will yield an incorrect result if the instruction is copied to a different location. As with the *ldr* and *ldrsw* instructions, the relative address could be corrected if the trampoline were close enough to the intended target of the address calculation. A more general approach is to precalculate the effective address and store it as a 64-bit literal in the patch area of the trampoline. The *adr* or *adrp* instruction in the second section of the trampoline is replaced by an *ldr* instruction that loads the literal into the same destination register as the *adr* or *adrp* instruction specified.

The third group of instructions using relative addressing comprises relative jump instructions. The jump instructions can further be divided into

conditional and unconditional jump instructions. The *b* and *bl* instructions are unconditional jump instructions, whereas the *b.cond*, *cbz*, *cbnz*, *tbz* and *tbnz* instructions are conditional. All the relative jump instructions have an offset field, ranging from 14 to 26 bits in width, that is first scaled by four and then sign extended to form the relative address, like the *ldr* and *ldrsw* instructions. Because all instructions on AArch64 are 32 bits (four bytes) wide and are aligned to four bytes, it is not necessary to include the two lowest bits, that are always zero, in the encoding of the instructions.

The *b* (branch) instruction simply jumps to a relative address. The *bl* (branch with link) instruction first loads the return address into the link register (*x30*) and then jumps to the relative address. The return address is the address of the following instruction, i.e. PC + 4, and is used as the location to jump to when a function returns. Figure 4.1 shows the encoding of the *b* and *bl* instructions. The relative address, specified in the *offset* field, has a range of ±128 MB.

The *b.cond* (branch conditionally) instruction jumps to a relative address only when the specified condition holds. Figure 4.1 shows the enconding of the *b.cond* instruction. The relative address, specified by the *offset* field, has a range of ±1 MB. The *cond* field specifies the condition on which the jump is carried out. The condition depends on one or two of four condition bits in a status register. Typically the *b.cond* instruction is preceded by a compare instruction that sets the value of the condition bits in the status register.

The *cbz* (compare and branch if zero) and *cbnz* (compare and branch if not zero) instructions jump conditionally when the value a register is zero or not zero respectively. Figure 4.1 shows the encoding of the *cbz* and *cbnz* instructions. The *sf* field determines the register set: when $sf = 0$, a w register is compared to zero and when $sf = 1$, an x register is compared to zero. The *reg* fields specifies the number of the register inside the register set, e.g. if $sf = 1$ and $reg = 3$, the register to compare is *x3*. The relative address, specified in the *offset* field, has a range of ±1 MB.

The *tbz* (test bit and branch if zero) and *tbnz* (test bit and branch if not zero) instructions jump conditionally when a specified bit in a register is zero or not zero respectively. Figure 4.1 shows the encoding of the *tbz* and *tbnz* instructions. The fields *bh* and *bl*, when concatenated, form a six-bit bit position. If the bit at the specified bit position in the register is zero, in case of *tbz*, or one, in case of *tbnz*, the jump is carried out. The *reg* field specifies which x register is examined. The relative address, specified by the *offset* field, has a range of ±32 kB.

All the relative jump instructions use relative addressing, which means that when the instructions are copied to a different location, the jump target is incorrect. One solution is to calculate a new relative address and update

the offset field, but it only works when the trampoline and the jump target are close enough. A more general solution is to make a two-step jump. The relative address of the relative jump instruction is redirected to the patch area of the trampoline. A jump sequence that jumps to the original target is generated in the patch area as described in section 4.1.1. This solution has the advantage that even if a relative jump instruction cannot jump very far by itself, e.g. the *tbz* and *tbnz* instructions, the two-step solution enables a jump with a greater range than the original relative jump instruction. If the third jump sequence presented in section 4.1.1 is generated, the two-step jump can target any address in the 64-bit memory space.

4.1.4 Atomic increment and decrement operations

IgProf uses atomic increment and decrement operations as a simple synchronisation mechanism between multiple threads. The profiler uses some global variables as flags to indicate that profiling or tracing (inspection of IgProf itself) is enabled or as a counter to limit the number of concurrent threads outputting statistics about memory mapping when tracing. The atomic operations implemented in IgProf are the increment and decrement operations. The atomic operations are needed, because normal increment and decrement operations, e.g. ++*counter* or --*counter* in C/C++, are not guaranteed to be atomic, even if they look like single operations in C code. A normal increment operation could be compiled into three instructions:

```
ldr x3, counter ; load counter from memory into register x3
add x3, x3, #1  ; add one to x3
str x3, counter ; store x3 back to counter in memory
```

Assume that there are two threads, t_1 and t_2, that are about to increment the same variable, *counter*, at the same time and that *counter* has the initial value of 1. Both threads have their own copies of register *x3*, may they be called $x3_1$ and $x3_2$ respectively. Thread t_1 loads the value stored in *counter*, i.e. 1, into $x3_1$. Thread t_2 also loads the value stored in *counter*, still 1, into $x3_2$. Both threads increment their own copies of the *x3* register, which results in $x3_1 = 2$ and $x3_2 = 2$. Finally, both threads store their own copies of *x3* back to *counter* in memory. Regardless which thread managed to store to *counter* last, the value of counter is 2. If the two threads had incremented the counter sequentially, the value of the counter would have been 3, which is most likely the intended result.

The problem with the sequence of instructions above is that several threads have concurrent access to the counter variable. In order to be able to increment the variable atomically, exclusive access to the variable is needed

instead. The AArch64 architecture follows the load-linked/store-conditional (LL/SC) paradigm [72] characteristic of RISC processors to provide exclusive access. A special load instruction, generally called load-linked, first loads the variable to be modified into a register from memory. Then the register is modified, e.g. incremented, and finally a special store instruction, generally called store-conditional, stores the value of the register back into memory if the variable in memory has not changed between the load linked and store conditional instructions. The store conditional also sets a flag in a register indicating whether the store operation was successful or not. [11, 13]

The following assembly code shows how the atomic increment is implemented on the AArch64 architecture:

```
loop:
ldaxr x0, [x1]     ; load-linked word at x1 into register r0
add x0, x0, #1     ; add one to x0
stlxr w2, x0, [x1] ; store-conditional register x0 to memory at x1
cbnz w2, loop      ; jump back to the beginning of the
                   ; loop if store-conditional failed
```

The address of the variable to be incremented is stored in register *x1* before the code is executed. First the load-linked instruction loads the variable stored at address *x1* in memory into register *x0*. The *add* instruction increments the value of *x0* by one. The store-conditional instruction tries to store the value of *x0* into the memory location at *x1* and stores the success flag into register *w2*. Finally, the jump instruction jumps back to the beginning of the code block if the store-conditional instruction failed.

4.1.5 Reading the cycle counter register

If the cycle counter register is available to user processes, IgProf reads the cycle counter before and after an event in order to measure the duration of the event in clock cycles. The cycle counter increases with every clock cycle and there are $1/f$ clock cycles per second, where f is the frequency of the processor. On the AArch64 architecture the cycle counter is read using the *mrs* instruction that copies the value of a system register to a general-purpose register. The cycle counter register, called PMCCNTR_EL0, is one of the performance monitor registers that, in turn, are a group of system registers on the AArch64 architecture [13]. By default the cycle counter register is not accessible to user processes. The control register PMUSERENR_EL0, which is accessible only in kernel mode, controls access to the cycle counter register. For example a kernel module, which executes in kernel mode, can set the control register and hence enable reading the cycle counter register.

4.2 Fast stack tracing on AArch64 and ARM

The calling conventions on UNIX-like systems on the ARM and AArch64 architectures are similar to the calling convention on the x86-64 architecture. Like the x86-64 calling convention, the first few integer arguments of a function are passed in registers. The first four integer arguments are passed in the registers *r0–r3* on the ARM architecture [12], whereas the first eight integer arguments are passed in the registers *r0–r7* on the AArch64 architecture [14]. However, unlike on the x86 and the x86-64 architectures, the most recent return address is not stored on the stack, but in a register, the link register. In order for nested function calls to be allowed, the previous return address needs to be saved for later restoration. The function prologue usually saves the link register on the stack together with a frame pointer register in case a frame pointer register is used at all.

On the AArch64 architecture, the *r29* register, alias the *fp* register, is regularly used as the frame pointer in non-leaf functions, i.e. in functions that make function calls. The reference register of a frame descriptor (see section 3.2.2) can be either the stack pointer or the *r29* register on the AArch64 architecture.

ARM processors can execute code of both the A32 and the T32 instruction sets. Functions compiled for the A32 instruction set tend to use only the stack pointer and no frame pointer. Functions compiled for the T32 instruction set, on the other hand, cannot easily access the stack pointer and instead tend to use the *r7* register, also called the *work register*, as the frame pointer. Hence the reference register of a frame descriptor (see section 3.2.1) can be either the stack pointer register or the *r7* register on ARM.

In addition to the fields of a frame descriptor on x86-64, a frame descriptor in the ARM and AArch64 ports contains a CFA-relative offset to the location on the stack where the link register is saved in the function prologue. This is not needed on the x86-64 arhictecture, because the return address is always stored at a constant offset from the value of the CFA. In non-leaf functions, i.e. functions that do make function calls, the link register is stored on the stack. In leaf functions, i.e. functions that do not call any functions, the link register does not need to be saved on the stack, because the link register is not overwritten.

Standard frames are handled similarly on the ARM and AArch64 architectures as on the x86-64 architecture. The following steps are carried out to calculate the previous frame based on the current frame and a frame descriptor:

1. the previous CFA is the value of the current reference register, i.e. the

stack pointer or the *r29* register on AArch64 or the stack pointer or the *r7* register on ARM, ± an offset

2. the previous location of execution is obtained from
   (a) the stack at the CFA ± an offset, if the link register is saved on the stack
   (b) the saved value of the link register, if the link register is not saved on the stack and if the stack frame two positions more recent than the previous frame is a signal frame
3. the previous frame pointer, i.e. the *r7* register on ARM and the *r29* register on AArch64, is either unmodified or stored at the CFA ± an offset on ARM
4. the previous stack pointer is the CFA itself.

Signal frames are pushed on the stack in the same way as on the x86-64 architecture. The signal frame contains a snapshot of the whole register file. The previous CFA is calculated as the current CFA + the size of the signal frame. The previous location of execution, the previous stack pointer and the previous frame pointer are obtained directly from the context structure on the stack. In addition, the value of the link register is saved separately. The location of execution of the previous stack frame is the saved value of the program counter register, whereas the saved value of the link register is used as the location of execution of the stack frame two positions earlier, if the corresponding function is a leaf function.

The guessed frame type is not used on the ARM and AArch64 architectures, but frames can be of unknown type, if they are not recognised as standard or signal frames. If frames of unknown type are encountered, the fast stack tracing fails and the standard (and slower) stack trace mechanism, using unwind information, is used to obtain the stack trace.

Like on the x86-64 architecture, some optimisations were considered for the variants of the *getcontext* function used by the fast stack trace feature on both the ARM and AArch64 architectures. In the AArch64 port of libunwind, the C library variant of the *getcontext* function is used when performing slow stack tracing. The C library version saves all registers to a context structure. A more lightweight and faster alternative variant was developed with fast stack tracing in mind. This variant saves only three registers to the context structure: the frame pointer register, the stack pointer register and the program counter.

In the ARM port of libunwind, the library variant of the *getcontext* function is not used, but instead a single instruction is used to store the contents of the registers to the context structure. This relies on a unique feature of the ARM instruction set, an instruction that stores any number of the 16 registers to memory. An experimental alternative variant was developed for fast stack tracing, but measurements show that the experimental variant did not improve performance significantly over the single-instruction variant. The experimental variant stored three registers to the context structure: the *r7* register, the stack pointer and the program counter.

4.3 Energy profiling in IgProf

In contrast to the porting efforts being done as part of this thesis, the energy profiling module introduces some new functionality to IgProf. The new energy profiling module obtains energy measurements from the RAPL interface through the PAPI library at a constant interval and attributes the current energy measurements to the current location of execution.

4.3.1 The RAPL interface

Recent Intel processors (SandyBridge processors and later) implement the Running Average Power Limit (RAPL) interface [56] that allows energy usage to be limited and monitored. The interface is implemented as model-specific registers (MSR) in the processor. The registers are updated approximately once every millisecond and the energy measurements are reported in units of 15.3 µJ. The unit of measurements is specified in a separate MSR, but currently Intel only provides measurements in units of 15.3 µJ. The RAPL interface provides energy measurements for four energy domains: processor package, power plane 0, power plane 1 and DRAM.

A processor package describes a processor die that can contain multiple cores, on-chip devices and other uncore components (components other than the cores) [56]. The processor package energy domain includes all the energy that the processor die consumes including all core and uncore components. A computer can have several processor packages and each processor package provides its own energy measurements. Energy measurements for processor packages are available on desktop and server grade processors.

Power plane 0 (PP0) describes the CPU cores. In recent desktop and server grade processors, there are usually multiple cores in one processor package. Each processor package provides a single energy measurement that is the total energy consumption of all cores in the processor package. PP0

is available on both desktop and server grade processors from Intel. Power plane 1 (PP1), on the other hand, describes some 'specific device in the uncore'. The components of a processor package that are not part of the cores are referred to as the uncore collectively. In practice an on-chip graphics processing unit (GPU) is usually a PP1 component on desktop grade processors [22]. PP1 is available only on desktop grade processors from Intel and not on server grade processors. Instead, the DRAM plane is available on server grade processors from Intel. The DRAM plane describes directly attached DRAM. To summarise, desktop grade processors provide measurements for the processor package, PP0 and PP1 energy domains, whereas server grade processors provide energy measurements for the processor package, PP0 and DRAM energy domains. The processor package domain includes the energy consumption of both the cores and the uncore components. The energy consumption of the components of the uncore not covered by PP1 are calculated as $energy(uncore) = energy(package) - energy(PP0) - energy(PP1)$ [22]. [56, 84]

Only the kernel and kernel modules that run at the ring 0 privilege level, i.e. the highest privilege level in Intel terminology, can access the RAPL registers [84]. The Linux kernel module *msr* [2] provides access to the model-specific registers as files (one file per core) in the file system at /dev/cpu/*/msr. The first step to access a model-specific register is to perform a *seek* operation using the number of the MSR as the offset. The *seek* operation selects which MSR to operate on. The second step is to actually access the MSR by reading or writing eight bytes, i.e. 64 bits, to the file.

4.3.2 Usage of the PAPI library

IgProf does not read the MSR files directly, but uses the PAPI (Performance API) library [84] as an interface, through which the MSRs are read. This has the advantage of decoupling IgProf from the *msr* kernel module. The PAPI library is used as a high-level interface to read energy measurements. If the RAPL interface changes or if PAPI manages to gain access to the RAPL interface through some other means than the *msr* kernel module, the source code of IgProf would not need to be modified, but an update to the PAPI shared library would suffice. When new energy measurement components become available through PAPI, little work is needed to make use of these PAPI components in the energy profiling module of IgProf. Recent processors of the AMD family 15h provide power measurements in watts through a model-specific register [9]. There are plans to support also this interface in future versions of PAPI.

The energy profiling module performs the following five steps to prepare

PAPI for energy measurements:

1. the PAPI library is initialised
2. the RAPL component of PAPI is located
3. an event set containing RAPL events is created
4. memory is allocated for the energy readings
5. the event set is activated.

In more detail, the first step, initialising the PAPI library, is performed to ensure that the version of the PAPI shared library available at the time of execution is compatible with the version of PAPI that was used when the application, in this case IgProf, was compiled. This step is necessary, because the library and the application need to share the same view of data structures passed across the interface between the library and the application.

The second step is locating the RAPL component. It is possible to build the PAPI library with or without the RAPL component. The default is to exclude it, but when the parameter *--with-components=rapl* is passed to the configuration script, the RAPL component is built into PAPI. PAPI provides an interface to iterate through all built-in components at runtime. Components are identified by name and finding the RAPL component is simply a matter of string matching. In addition to the name, each component has a flag that indicates whether the component is enabled or disabled. The RAPL component is disabled if the processor does not support RAPL, e.g. when IgProf is running on an AMD processor.

The third step is to create an event set. A PAPI event usually corresponds to a hardware event [51], e.g. an instruction being executed or a cache miss, but a RAPL event represents a certain amount of energy being used by a component in one of the four energy domains of RAPL. An event set specifies a collection of related PAPI events. Each event in an event set corresponds to a measurement that is read and stored as an entry in an array. IgProf iterates through all available events in the RAPL component of PAPI and adds the integer type events that correspond to the four energy domains of RAPL. PAPI also provides floating point measurements, but IgProf is designed to collect integer data.

The fourth step is to allocate space for two arrays. One array is used to store the most recent energy readings, whereas the other array contains the previous readings. When both the current and the previous readings are available, it is easy to calculate the difference between them. Finally, as the fifth step, the event set created in step three is activated.

4.3.3 The energy profiling module

When the initialisation is completed successfully and IgProf has set up the signal handler and the interval timer (see section 3.1.6), the energy profiling module is running. Figure 4.5 shows how the energy profiling module operates. At a certain interval the current energy consumption and location of execution are sampled. The difference between the current and the previous energy readings is attributed to the current location of execution. When the energy profiling is ready and analysed, the results may show that some locations in the executable have a higher amount of energy associated with them than others. These bottlenecks may be worth optimising for energy consumption.

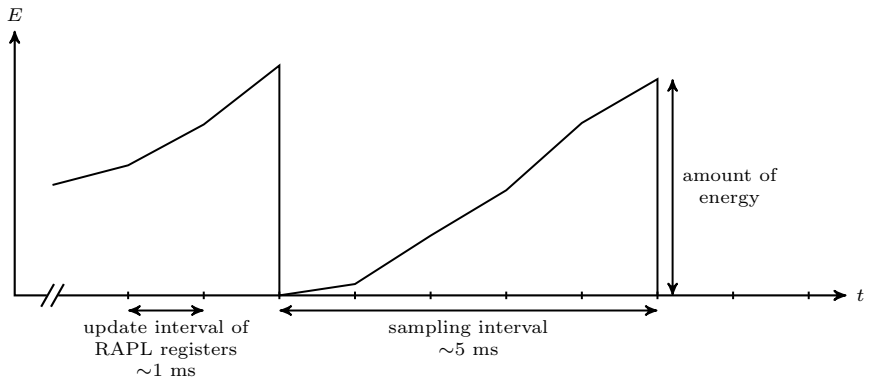

Figure 4.5: Principle of operation of the energy profiling module.

The energy profiling module uses four counters (see section 3.1.2) to represent the four RAPL energy domains: the NRG_PKG counter represents the total energy consumption of all processor packages, the NRG_PP0 counter represents the total energy of all cores, the NRG_PP1 counter represents the total energy of all PP1 components and the NRG_DRAM counter represents the total energy consumption of all DRAM components. Because only three of the four energy domains usually are available on any processor implementing the RAPL interface, one of the four counters is left unused.

The energy profiling module is based on the performance profiling module, because they both have in common that they make observations at a regular interval. Both modules utilise an interval timer and a signal handler to implement the sampling. Section 3.1.6 describes more in detail how this mechanism is used in IgProf. Inside the signal handler the RAPL measurements are queried through the PAPI library. As described above, the energy profiling module keeps the previous set of measurements in order to be able calculate the delta values. Each delta value together with the current stack

trace is added to the counter that corresponds to the energy domain of the delta value. To summarise the functionality of the energy profiling module, the module reads the energy consumption at a constant interval, calculates the difference in energy consumption since the previous reading and adds the difference to the counters of the energy profiling module.

To profile energy consumption using IgProf, the *-np* or *--energy-profiler* argument is given to IgProf. The obvious choice for the abbreviated variant of the argument, *-ep*, was already taken by the empty memory profiler. The *igprof* script was extended to recognise and validate these arguments. Section 3.1.5 describes further how the script works and why it is used. The following command line starts the energy profiling module of IgProf:

```
igprof -z -np application arguments to the application
```

The *-z* argument compresses the output as described in section 3.1.3, but is not strictly necessary, whereas the *-np* argument enables the energy profiler. When the profiling has finished successfully, the result is a file containing the profiling data. The default name of the output file conforms to the pattern *igprof.pid.gz*, where *pid* is the process id of the application running. The *-o* argument is used to name the output file differently (see section 3.1.3). To analyse and present the results of the profiling, the *igprof-analyse* program is used:

```
igprof-analyse -d -v -g profiling-data-file
```

Section 3.1.4 describes the *-d*, *-v* and *-g* arguments. The output of igprof-analyse comprises a flat cumulative profile, a flat self profile and a call graph profile. The flat cumulative profile tells how much energy was spent in each function, including the energy spent in child functions, whereas the flat self profile does not include the energy spent in child functions for each function, but only the energy spent in the function itself. The call graph shows the relation between functions, i.e. the parent and child functions of each function. Section 2.2 describes the output formats in greater detail.

Chapter 5

Testing and evaluation

This chapter describes how the port of IgProf, the port of the fast stack trace feature in the libunwind library and the energy profiling module in IgProf were tested and evaluated.

5.1 The port of IgProf to AArch64

The port of IgProf to the AArch64 architecture does not bring any new functionality, but enables the profiler to be run on the new architecture. The parts of IgProf that were ported to the AArch64 architecture mainly affect the function instrumentation mechanism in IgProf. In order to ensure that the function instrumentation works correctly, a test program was written. IgProf instruments 29 functions, but only some functions are instrumented simultaneously. When the *-d* argument is given on the command line to IgProf, the profiler shows which functions are instrumented. The test program calls 27 of the functions that IgProf instruments. The remaining two functions are *__cyg_profile_func_enter* and *__cyg_profile_func_enter*, both specific to the GCC compiler. The instrumentation of these two functions are most easily tested with the function profiling mode of IgProf.

The test program is run with profiling enabled:

```
igprof -d -z profiling-mode ./insttest
```

The set of instrumented functions depends on which profiling module (see section 3.1.1) is enabled. Instrumented functions common to all profiling modules are: *exit*, *_exit*, *kill* and *pthread_create*. When performance profiling is enabled (started with *-pp* on the command line), the performance profiling module instruments the following functions: *close*, *fclose*, *fork*, *system*, *pthread_sigmask* and *sigaction*. It is not even necessary that a sampling event occurs, the test program calls the instrumented functions directly.

The memory profiling module (enabled by the *-mp* argument) and the empty memory profiling module (enabled by the *-ep* argument) instrument the following memory management functions: *calloc*, *free*, *malloc*, *memalign*, *posix_memalign*, *realloc* and *valloc*. The file descriptor profiling module (enabled by the *-fd* argument) instruments functions allocating or releasing file descriptors: *close*, *dup*, *dup2*, *open*, *__open64*, *socket*, *accept*.

IgProf furthermore has a few modes of operation that profile the profiler itself. These modes, known as tracing modes, are not available through the *igprof* script, but are enabled using the *igprof-trace* script and are usually of less interest than the main profiling modes (see section 3.1.1). These tracing modes keep track of memory mapping and unmapping and exception handling. The tracing modes instrument the memory mapping and unmapping functions *mmap*, *mmap64* and *munmap* and the exception handling function *__cxa_throw*.

Runs of the test program shows that the function instrumentation works correctly and the return value of every function is checked for errors. If an error occurs, the test program will terminate with an error message. However, a flaw in the implementation of the test program was discovered. The test program verifies that the socket functions work correctly by implementing a server in one thread and a client in another. A race condition sometimes occurs between the calls to the functions *accept* and *connect*.

The port of IgProf to AArch64 was also tested in a different way. A piece of CMS software [64] was profiled for performance and memory usage on both AArch64 (a development board) and x86-64 hardware. Table 5.1 shows the ten functions spending the most execution time on both architectures. Among the top ten functions on both architectures there are floating point mathematical functions of the GNU C library (*__ieee754_log*, *__log_finite* and *__ieee754_atan2*), a library initialisation function (*_init*), memory management functions (*malloc* and *free*) and functions specific to the CMS software (*fastjet::ClusterSequence::_minheap_faster_tiled_N2_cluster* and *CoreSimTrack::chargeValue*). However, the relative order of the functions in the top ten lists is not the same between the two architectures. The function *MuonAssociatorByHits::getShared*, at place 5 on the x86-64 list, is just outside the top ten list of the AArch64 architecture, at place 11. The *memcpy* function, on the other hand, is at place 8 on the AArch64 list and at place 25 on the x86-64 list. Functions names starting with an underscore are usually reserved for functions specific to the implementation of libraries and complementary header files. A specific function can be present on one architecture and be absent on another architecture, if the functionality is implemented differently on the two architectures. This seems to be the case with the *std::__adjust_heap<...>* function. The function at place 8 on the x86-64 list

could not be resolved to a name.

Table 5.1: The top ten functions spending the most execution time in a piece of CMS software on AArch64 and x86-64.

	AArch64		x86-64	
#	Time	Function	Time	Function
1	313.76	_init	75.19	__ieee754_log
2	125.14	__log_finite	67.99	_init
3	98.13	free	59.71	free
4	87.16	malloc	50.01	malloc
5	70.75	fastjet::ClusterSequence ::_minheap_faster_tiled_N2 _cluster	44.78	MuonAssociatorByHits ::getShared
6	49.62	std::basic_string<...> ::_M_rep	41.40	fastjet::ClusterSequence ::_minheap_faster_tiled_N2 _cluster
7	48.73	__ieee754_atan2	38.92	CoreSimTrack::chargeValue
8	41.12	memcpy	37.54	@?0xffffffffff600104
9	35.62	CoreSimTrack::chargeValue	35.80	__ieee754_atan2
10	32.54	_wordcopy_fwd_aligned	30.10	std::__adjust_heap<...>

The same piece of CMS software was also profiled for memory usage on the AArch64 and the x86-64 architectures. Table 5.2 shows the ten functions using the most memory on both architectures. The first position on the top ten list of both architectures is an allocation function, *__gnu_cxx::new_allocator <char>::allocate* on AArch64 and *std::basic_string<char, ...>::_Rep:: _S_create* on x86-64. Even if the function names differ, the functions allocate approximately the same amount of memory. The functions at positions two to seven are the same on both architectures, following the same relative order and allocating more or less the same amount of memory. The function *AnalyticalPropagator::propagatedStateWithPath* is at position eight on x86-64 allocating 10.0 GB of memory and at position thirteen on AArch64 allocating only 4.47 GB of memory. Positions eight to nine on the AArch64 list corresponds to positions nine to ten on x86-64 both with regard to the relative order and amount of memory allocated. The *ClusterTPAssociation-Producer::getSimTrackId<...>* function at position ten on the AArch64 list is just outside the top ten list at position eleven on x86-64. In comparison to the top ten functions of the performance profile on the AArch64 and x86-64 architectures, the top ten functions of the memory profile show less variation in the relative order. The amount of memory allocated by the functions is

approximately the same on both architectures.

Table 5.2: The top ten functions using the most memory in a piece of CMS software on AArch64 and x86-64.

	AArch64		x86-64	
#	Mem (GB)	Function	Mem (GB)	Function
1	43.6	__gnu_cxx::new_allocator<char> ::allocate	43.6	std::basic_string<char, ...> ::_Rep::_S_create
2	20.4	TrackingParticle ::TrackingParticle	20.4	TrackingParticle ::TrackingParticle
3	15.4	std::vector<PSimHit, ...> ::_M_emplace_back_aux<...>	15.4	std::vector<PSimHit, ...> ::_M_emplace_back_aux<...>
4	11.9	std::_Vector_base <TrackingParticle, ...> ::_M_create_storage	11.9	std::_Vector_base <TrackingParticle, ...> ::_M_create_storage
5	11.4	std::vector<TrackingParticle, ...> ::operator=	11.4	std::vector<TrackingParticle, ...> ::operator=
6	10.7	std::vector<PSimHit, ...> ::operator=	10.7	std::vector<PSimHit, ...> ::operator=
7	10.2	edm::MessageSender ::MessageSender	10.4	edm::MessageSender ::MessageSender
8	9.69	std::vector<PSimHit, ...> ::_M_range_insert<...>	10.0	AnalyticalPropagator ::propagatedStateWithPath
9	6.66	QuickTrackAssociatorByHits ::prepareEitherHitAssociator OrClusterToTPMap	9.69	std::vector<PSimHit, ...> ::_M_range_insert<...>
10	6.31	ClusterTPAssociationProducer ::getSimTrackId<...>	6.51	QuickTrackAssociatorByHits ::prepareEitherHitAssociator OrClusterToTPMap

During the profiling runs on AArch64, a minor problem was discovered, which caused the profiler and the application to terminate with a segmentation fault. Memory was accessed in a memory area, that the application and the profiler did not have permission to access. The problem turned out to be related to virtual dynamically linked shared objects (vDSO) [6]. Some functions that are usually part of the kernel are instead available in user space in a vDSO library. On the AArch64 architecture there are four functions available in the vDSO library: *clock_gettime*, *clock_getres*, *gettimeofday* and *rt_sigreturn*. On the system (Fedora 19 Remix) the profiling was performed, the application called the *gettimeofday* function, but there was not enough unwind information available for the functions in the vDSO library, which

caused the stack tracing in libunwind to access an incorrect memory location. A quick solution to the problem was to disable profiling functions in the vDSO library. This solution was not implemented by the author and is not part of the port of IgProf to AArch64, but rather as a separate patch to IgProf.

5.2 Fast stack tracing on AArch64 and ARM

Like the port of IgProf to AArch64, the port of the fast stack trace feature to the AArch64 and ARM architectures does not bring any new functionality to the libunwind library, but enables stack tracing to be performed faster than the standard way of performing stack tracing. Two test programs that come with the libunwind library were used to verify that the stack trace is correct and to measure the execution time of stack tracing.

The first test program, *Gtest-trace*, verifies that the fast stack trace implementation returns the same result as the standard stack trace implementation. The test program compares the result of fast and standard stack tracing in three cases. In the first case stack tracing is called from a regular function. The function that calls stack tracing is by definition not a leaf function, so the first test case does not cover leaf functions. In the second case stack tracing is called from inside a signal handler. This case is different from the first one, because there is a signal frame on the stack. Signal frames are handled differently than regular frames in the fast stack tracing implementation, beacuse the layout of the frames are different. Section 3.2.2 describes stack frames in general and section 4.2 describes the handling of signal frames on the AArch64 and ARM architectures in particular. The third case is similar to the second case, but the signal handler uses an alternate stack. An alternate signal stack is set up by the function *sigaltstack* [5].

Early runs of the *Gtest-trace* test program revealed failures in the second and third cases, which both test that signal frames are handled correctly. The stack trace showed that the fast stack trace implementation was able to calculate the previous frame of the signal frame. However, the frame two positions earlier than the stack frame was not correctly calculated. The cause of the problem turned out to be a leaf function two positions earlier than the stack frame. Because leaf functions not necessarily store the link register on the stack, the stored value of the link register in the signal frame needs to be saved when fast stack tracing handles signal frames. When this problem was solved, the *Gtest-trace* test program confirmed that the fast stack trace implementation handles signal frames and leaf functions correctly.

Leaf functions can be part of the stack trace only if a signal interrupts the execution of the leaf function.

The second test program, *Gperf-trace*, measures the performance of stack tracing. When the measurement was carried out on the Foundation Model emulator [15] that emulates the AArch64 architecture, fast stack tracing executed approximately in 4 % of the execution time of standard (slow) stack tracing. On real AArch64 hardware the percentage was reported to be around 3.3 % [74]. Similarly, when the measurement was carried out on the Qemu emulator [16] that emulates the ARM architecture (among others), fast stack tracing ran in approximately 5 % of the execution time of standard stack tracing.

The impact of stack tracing on the execution time of IgProf was examined on a a development board with an ARMv8 processor, when a piece of CMS software was profiled with both fast and standard stack tracing enabled in libunwind. For comparison the same software was also run without any profiling enabled. The execution time of the software without any profiling enabled is considered the reference time. When memory profiling was performed with fast stack tracing enabled, the execution time was around 3.3 times (n_{fast}) the reference time on AArch64. The execution time was approximately 25.3 times ($n_{standard}$) the reference time when standard stack tracing was enabled. The memory profiling with fast stack tracing enabled ran in

$$\frac{n_{fast}}{n_{standard}} = \frac{3.3}{25.3} \approx 13\ \%$$

of the execution time of profiling with standard stack tracing enabled. In other words, the execution time of profiling was reduced by approximately 87 %.

For comparison the same piece of software was also profiled for performance. The execution time of performance profiling with fast stack trace enabled in libunwind was approximately 1.1 times the reference time. The great difference in impact between performance profiling (1.1) and memory profiling (3.3) is due to the fact that the performance profiler takes a sample (including a stack trace) around 200 times a second, whereas the memory profiler makes an observation at every memory allocation and release. The CMS software typically allocates and releases memory around 1 000 000 times a second.

5.3 The energy profiling module

The energy profiling module was evaluated in two ways. First, the total energy consumption of an application measured by the energy profiling module was compared to the energy consumption measured by a separate energy measurement tool. This comparison evaluates how the energy profiling module distributes the energy over the energy domains as a whole. Second, energy profiles of applications were compared to performance profiles. The results of this comparison may indicate some correlation between energy and time spent in functions. The measurements in this section were conducted on a desktop computer equipped with an Intel Core i7 processor.

A small tool was written that measures the energy consumption over the whole execution of an application. The tool, *measure-rapl*, uses the PAPI library to obtain energy measurements from the RAPL interface. The preexisting *likwid* toolset [79, 80] provides similar functionality, but only measures the energy consumption of the processor package and the DRAM energy domains, lacking measurements of the power plane 0 (CPU cores) and power plane 1 (GPU) energy domains entirely. The energy domains of RAPL are described in further detail in section 4.3.1. Furthermore, the likwid toolset does not account for overflows in the 32-bit RAPL registers, a shortcoming that was addressed in the *measure-rapl* tool.

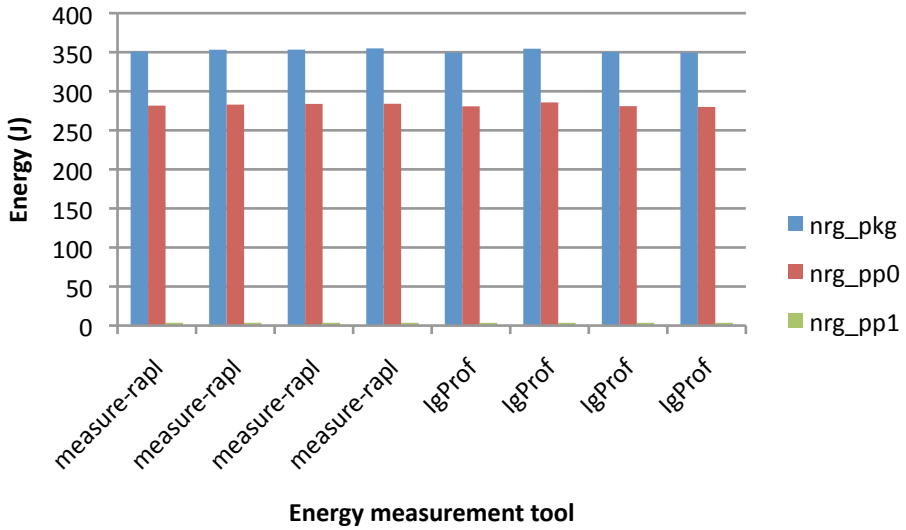

Figure 5.1: Energy consumption as measured by the measure-rapl tool and the energy profiling module of IgProf.

Figure 5.1 shows the total energy consumption as measured by the measure-rapl tool and the energy profiling module of IgProf. The application

being measured and profiled was a simple test program performing insertion sorting. The first four sets of measurements show the energy as measured by the measure-rapl tool. The following four sets of measurements show the energy as measured by the energy profiling module of IgProf. The columns in each set of measurements describe the processor package, power plane 0 (CPU cores) and power plane 1 (GPU). The values in each energy domain are very similar between the different sets of measurements, both between sets of measurements from the same tool and between the two tools. The same finding was observed when the energy consumption of the *stream* benchmarking tool and a piece of CMS software (described below) was measured with both the energy profiling module and the measure-rapl tool.

The *stream* benchmarking tool [67] was profiled for energy consumption and performance using IgProf. The stream benchmarking tool performs four types of vector operations to generate workload. The *Add* operation adds two vectors and stores the result in a third vector ($C[i] = A[i] + B[i]$), the *Copy* operation copies a vector to another ($C[i] = A[i]$), the *Scale* operation scales a vector and stores the result in another ($C[i] = kA[i]$) and the *Triad* operation first scales one vector, then adds another vector and finally stores the result in a third vector ($C[i] = A[i] + kB[i]$). The stream tool was compiled with the *-DTUNED* flag, in order for the four operations to be implemented as separate functions.

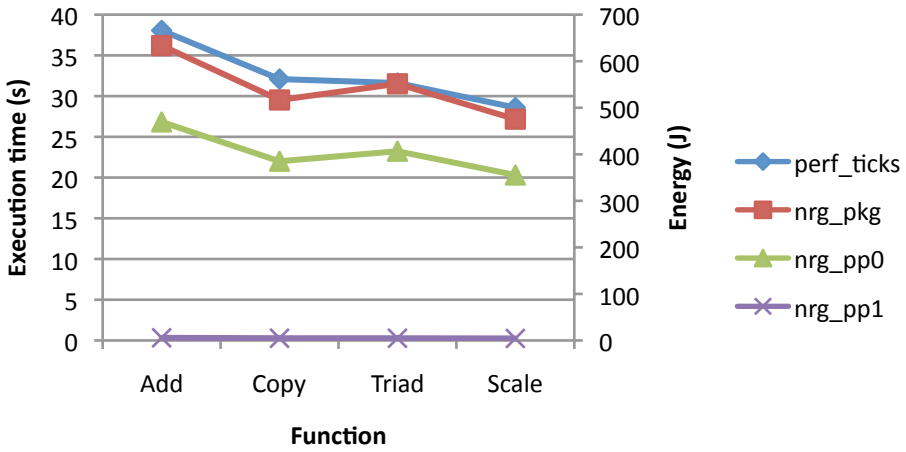

Figure 5.2: The results of performance and energy profiling of the stream tool.

Figure 5.2 shows the results from performance and energy profiling of the stream tool. The X-axis describes the four functions Add, Copy, Triad and Scale. The left scale of the Y-axis and the perf_ticks series describe the execution time spent in each function, whereas the right scale of the Y-axis

and the nrg_pkg, nrg_pp0 and nrg_pp1 series describe the amount of energy spent in each function. The energy consumption of the processor package domain and the power plane 0 (describing the CPU) seem to follow the time spent in the functions, whereas the energy consumption of power plane 1 (describing the GPU) seems to be fairly constant.

In addition to the stream tool, a piece of CMS software was profiled for performance and energy. The software is multithreaded, it is much more complex than the simple stream tool and executed for around an hour and a half. Figure 5.3 shows the results from performance and energy profiling of the piece of CMS software. Like in the previous plot, the X-axis describes functions, in this case numbered instead of named for brevity. The top twenty functions of the performance profile (self time) and top twenty of each energy domain in the energy profile are shown in the plot. The plot shows 37 functions, which is less than the total number of entries (80) in the four top twenty lists (perf_ticks, nrg_pkg, nrg_pp0 and nrg_pp1), because the top twenty lists have many functions in common. Similar to the previous plot, the left scale of the Y-axis and the perf_ticks series describe the execution time spent in each function, whereas the right scale of the Y-axis and the nrg_pkg, nrg_pp0 and nrg_pp1 series describe the amount of energy spent in each function. Unlike the profiling results of the stream tool (see figure 5.2), the profiling results of the CMS software does not show any clear relation between the time and energy spent in a function.

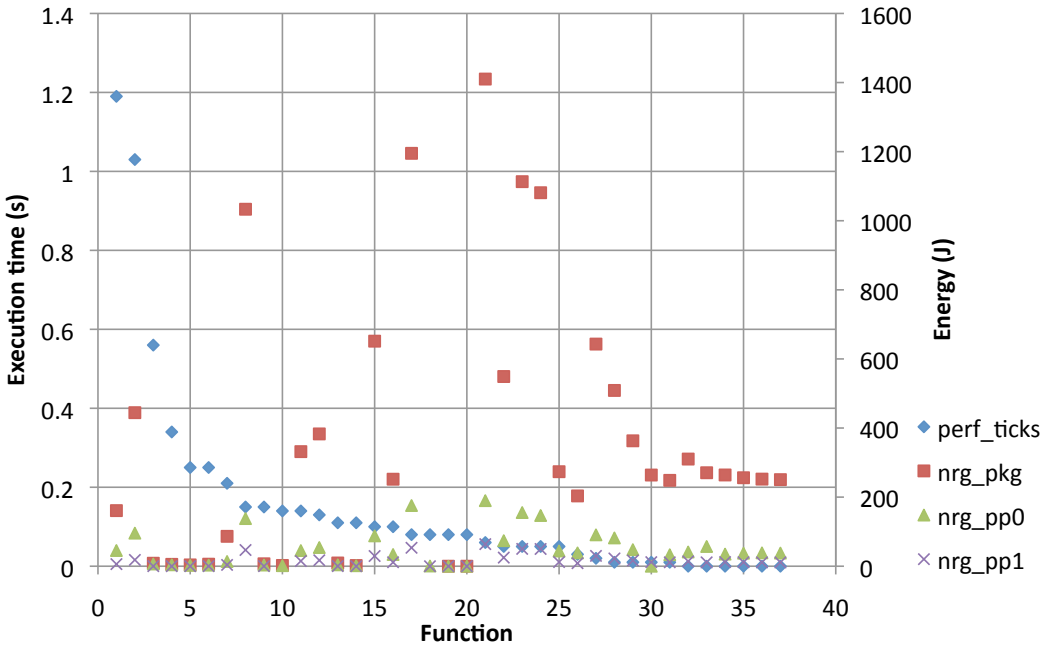

Figure 5.3: The results of performance and energy profiling of a piece of CMS software.

Chapter 6

Discussion

In this chapter the port of IgProf to AArch64, the stack trace feature in the libunwind library and the implementation of the energy profiling module in IgProf are discussed, especially limitations and future work.

6.1 The port of IgProf to AArch64

The parts of IgProf that were ported to the AArch64 architecture mainly affect the function instrumentation mechanism in IgProf. The implementation of function instrumentation (see section 4.1) on ARM identifies a small number of instructions that are common in function prologues. If the compiler generated a prologue containing instructions outside this set of instructions recognised by the function instrumentation part of IgProf, the instrumentation would fail. The port of IgProf to AArch64 takes a more general approach. The function instrumentation part of IgProf identifies and patches instructions that use addressing relative to the program counter (PC), but leaves other instructions untouched. The instructions allowed in the function prologue is not confined to a small set of instructions, but can be any instructions of the AArch64 instruction set. In other words, the function instrumentation is more general in the AArch64 implementation of IgProf than in the ARM implementation.

The implementation of function instrumentation is, however, more limited on AArch64 than on ARM regarding allocation of memory for a trampoline (see sections 3.1.7 and 4.1.1). The trampoline needs to be within 128 MB from the function being instrumented on the AArch64 architecture, because the target of a single jump instruction is limited to ±128 MB. The allocation of memory for a trampoline on the x86-64 architecture has a similar limitation. The trampoline needs to be located within 2 GB from the function

being instrumented, even though the memory space is 2^{64} B on the x86-64 architecture. This restriction does not apply to the placement of a trampoline on the ARM or x86 architectures; a trampoline can be placed anywhere in the 4 GB memory space. In practice, the allocation of memory has succeeded when CMS software has been profiled on AArch64, even if the software loads a large number of libraries into the memory space of the process.

The CMS software [64], which is used to analyse data and perform simulations related to CMS experiments, is in the process of being ported to AArch64. So far, small parts of the CMS software have been profiled on the AArch64 architecture (see section 5.1), mainly to try out the port of IgProf to AArch64. However, when the port of the CMS software to AArch64 becomes more mature, IgProf will be used to examine the execution of the CMS software on both x86-64 and AArch64. A comparsion of the profiling results might reveal differences in the execution of the CMS software on the two architectures. IgProf will also be a helpful tool when the CMS software is being optimised. The profiler helps to find the parts of the software that use the most resources, i.e. the bottlenecks. Optimisations can then be targeted at the bottlenecks in the software.

6.2 Stack tracing using libunwind

When the ports of the fast stack trace feature to the AArch64 and ARM architectures are taken into account, IgProf uses fast stack tracing on three out of four architectures. On the x86 architecture IgProf implements stack tracing itself. Stack tracing is easier to do on x86 than on the other supported architectures, because unwind information is not generally necessary. On the x86-64, ARM and AArch64 architectures, however, stack tracing is more complex due to the dependency on unwind information. Hence, it is reasonable to use a library that performs stack tracing with the help of unwind information instead of reimplementing stack tracing in the application. Because IgProf already is dependent on the libunwind library on three out of four architectures, it could be feasible to investigate if stack tracing could be outsourced to libunwind also on the x86 architecture and what the consequences for performance would be. It is not clear if the fast stack trace feature would make a big difference on x86, because stack tracing on this architecture is not generally dependent on unwind information. However, the interest for the x86 architecture may not be as great anymore since the x86-64 architecture has overtaken the place of x86 in commodity computers. Furthermore, the implementation of stack tracking on x86 in IgProf works well, so the only benefit would be to reduce the amount of code in IgProf.

The fast stack trace feature reduces the execution time of profiling, especially when profiling events happen frequently. When a piece of CMS software was profiled for memory usage and fast stack tracing was used instead of standard stack tracing, the execution time of profiling was reduced by approximately 87 %. The overhead of profiling may not matter when small applications are profiled, but when big applications, that take hours or even days to execute, are profiled, the overhead is significant. For big applications, for example the CMS software, the overhead determines if it is practically feasible to profile the applications. The execution time of stack tracing alone was also measured. The execution time of fast stack tracing was approximately 4 % of the execution time of standard stack tracing.

6.3 The energy profiling module

The total energy consumption measured by the energy profiling module during the whole execution of an application seems to correspond well with the energy consumption measured by a separate energy measurement tool. This was true for both small single-threaded programs and larger multi-threaded software. A small single-threaded benchmarking tool was profiled for both performance and energy consumption. The profiling results seem to show a correlation between the execution time and the energy spent in a function. This was not the case with a larger piece of multi-threaded software. There was not any obvious relation between energy consumption and execution time. It is very likely that the multiple threads of the application cause trouble, because the energy profiling module does not take into account multiple threads or processes. The energy measurements provided by the RAPL interface through the PAPI library are system-wide. A possible solution would be to keep track of the number of threads and split the measured energy evenly between the threads. The greatest flaw in this solution is that it assumes that all threads are running. If some threads are running and some are sleeping, energy consumption should not be attributed to the sleeping threads.

An improved version of the energy profiling module would have a better strategy for distributing the measured energy over the running threads of the application being profiled and other processes. Ideally PAPI would provide energy measurements per thread, but even per process would be better than the current system-wide measurements. Furthermore, the energy profiler could be extended to support other sources of energy measurements, e.g. the Texas Instruments INA231 power monitor chip [26, 78] available on the ODROID-XU+E board [48] based on ARM.

Another shortcoming of the energy profiling module is that the energy consumption of short pieces of code executing faster than the update interval of the RAPL registers, i.e. approximately 1 ms, cannot reliably be measured using the RAPL interface without synchronisation. A solution to this problem has been developed at Dresden University of Technology [45, 46]. A framework called HAECER performs more fine-grained measurements by synchronising the start of execution of the function to be measured to an update in RAPL registers. After the function has returned, the time before the next update in RAPL registers is measured. This mechanism, however, extends the overall execution time and is more appropriate for measuring the energy consumption of short fragments of code rather than large applications. The update interval of the RAPL registers is not exactly 1 ms, but has a variation of ± 0.02 ms [46]. This may be critical when measuring short fragments of code, but when profiling big applications the differences in update interval should even out.

A drawback relating to the implementation of the energy profiling module is the added dependency on the PAPI library. The *cmake* configuration script that is run as part of the building process of IgProf tries to find a header file and a library file of PAPI. If the PAPI files cannot be found, the configuration script determines that PAPI is not available and IgProf is built without integration with PAPI, but the rest of the functionality of IgProf is still available as before. To summarise, the build-time dependency on PAPI is optional in general, but required if the energy profiling module is to be used. The energy profiling module could quite easily be rewritten to use the *msr* kernel module directly if it was desirable to remove the dependency on the PAPI library. On the other hand, the PAPI library also provides measurements unrelated to energy consumption, such as the number of cache misses or number of instructions executed. IgProf could be extended with further profiling modules that would use other types of PAPI events.

Chapter 7

Conclusions

The goals of the thesis work was to port IgProf to 64-bit ARM, port the fast stack trace feature of the libunwind library to both 64-bit and 32-bit ARM and extend the functionality of the profiler with a simple energy profiling module. The profiler was available on the Intel x86 and x86-64 architectures, as well as on 32-bit ARM, but support for 64-bit ARM was missing. The port of IgProf to 64-bit ARM enables developers to evaluate how applications execute on the new architecture with regard to performance and memory usage. The major part of the porting effort of IgProf to AArch64 (64-bit ARM) concerns the function instrumentation mechanism in IgProf. A test program was written and run to verify that the function instrumentation was carried out correctly. So far, small parts of the CMS software have been profiled with the port of IgProf to AArch64. When the port of the CMS software to AArch64 becomes more mature, IgProf will be used for examination and optimisation of the CMS software on 64-bit ARM.

IgProf uses the libunwind library to perform stack tracing as part of the profiling. The fast stack trace feature in the libunwind library was ported from the x86-64 architecture to both 64-bit and 32-bit ARM. It is necessary for the stack tracing to execute fast, because the profiler performs stack tracing at every sampling event or call to memory management functions. The performance profiler takes a sample (including a stack trace) around 200 times a second, whereas the memory profiler makes an observation at every memory allocation and release. CMS software typically allocates and releases memory around 1 000 000 times a second. When a piece of CMS software was profiled for memory usage and fast stack tracing was used instead of standard stack tracing, the execution time of profiling was reduced by approximately 87 %. The overhead of profiling may not matter when small applications are profiled, but for big applications, the overhead determines if it is practically feasible to profile the applications.

In contrast to the port of IgProf to AArch64 and the port of the fast stack trace feature of the libunwind library to both 32-bit and 64-bit ARM, the simple energy profiling module extends the functionality of IgProf. The energy profiling module obtains energy measurements from the RAPL interface present on recent Intel processors at a certain sampling interval and attributes the accumulated amount of energy since the previous sampling event to the current location of execution. The profiling results of a simple single-threaded application seem to show a correlation between the execution time and the energy spent in a function. However, the implementation of the energy profiling module is still rather naïve and limited, e.g. it does not take into account multiple threads or other processes that also consume energy. An improved version of the energy profiling module would have a better strategy for distributing the measured energy over the running threads of the application being profiled and other processes. Furthermore, the energy profiler could be extended to support other sources of energy measurements.

Bibliography

[1] ld.so(8) - Linux programmer's manual. Manual page. http://man7.org/linux/man-pages/man8/ld.so.8.html. Accessed 2014-05-27.

[2] MSR(4) - Linux programmer's manual. Manual page. http://man7.org/linux/man-pages/man4/msr.4.html. Accessed 2014-07-24.

[3] profil(3) – Linux programmer's manual. Manual page. http://man7.org/linux/man-pages/man3/profil.3.html. Accessed 2014-08-26.

[4] setitimer(2) - Linux man page. Manual page. http://linux.die.net/man/2/setitimer. Accessed 2014-03-12.

[5] sigaltstack(2) – Linux programmer's manual. Manual page. http://man7.org/linux/man-pages/man2/sigaltstack.2.html. Accessed 2014-08-22.

[6] vDSO(7) - Linux programmer's manual. Manual page. http://man7.org/linux/man-pages/man7/vdso.7.html. Accessed 2014-08-21.

[7] Abdurachmanov, D., et al. Explorations of the viability of ARM and Xeon Phi for physics processing.

[8] Adhianto, L., et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22 (2010), 685–701. DOI 10.1002/cpe.1553.

[9] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, 2013. Order number 42301 Rev 3.14. http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf. Accessed 2014-07-25.

[10] Apple. Instruments user guide. https://developer.apple.com/library/mac/documentation/developertools/conceptual/instrumentsuserguide/InstrumentsUserGuide.pdf.

[11] ARM. ARMv8 instruction set overview, 2011.

[12] ARM. Procedure call standard for the ARM architecture, 2012. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf`. Accessed 2014-08-04.

[13] ARM. ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile, 2013.

[14] ARM. Procedure call standard for the ARM 64-bit architecture (AArch64), 2013. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf`. Accessed 2014-05-07.

[15] ARM. ARMv8-A Foundation Model user guide, 2014. Revision D. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0677b/index.html`. Accessed 2014-08-22.

[16] Bellard, F. Manual – QEMU. Web site. `http://wiki.qemu.org/Manual`. Accessed 2014-08-29.

[17] Blem, E., Menon, J., and Sankaralingam, K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *The 19th IEEE International Symposium on High Performance Computer Architecture (HPCA-2013)* (2013), pp. 1–12. DOI 10.1109/HPCA.2013.6522302.

[18] Bruening, D. L. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004. `http://www.burningcutlery.com/derek/docs/phd.pdf`.

[19] Bruening, D. L., et al. DynamoRIO API. Web page. `http://dynamorio.org/docs/`. Accessed 2014-08-29.

[20] CERN. About CERN. Web page. `http://home.web.cern.ch/about`. Accessed 2014-09-30.

[21] Cohen, W. E. Tuning programs with OProfile. *Wide Open Magazine*, Premiere issue (2004), 53–62.

[22] Demmel, J., and Gearhart, A. Instrumenting linear algebra energy consumption via on-chip energy counters. Tech. Rep. UCB/EECS-2012-168, University of California at Berkeley, 2012. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-168.pdf`. Accessed 2014-06-23.

[23] Edge, J. Lots of new perf features, 2014. Web site. https://lwn.net/Articles/593690/. Accessed 2014-09-02.

[24] Elmer, P., Eulisse, G., and Lupton, R. Future development of the IgProf application performance profiler and analysis tool, 2013.

[25] Eranian, S. Perfmon2: a flexible performance monitoring interface for Linux. In *Proceedings of the Linux Symposium* (2006), vol. 1, pp. 269–288.

[26] Eulisse, G., et al. Techniques and tools for measuring energy efficiency of scientific software applications, 2014. Slide set. https://indico.cern.ch/event/258092/session/7/contribution/91/material/slides/0.pdf. Accessed 2014-09-09.

[27] Eulisse, G., and Tuura, L. IgProf, the Ignominous Profiler. Web page. http://igprof.org/. Accessed 2014-03-27.

[28] Eulisse, G., and Tuura, L. A. IgProf profiling tool, 2004. http://indico.cern.ch/event/0/session/6/contribution/63/material/paper/0.pdf. Accessed 2014-03-04.

[29] Fog, A. Calling conventions for different C++ compilers and operating systems, 2014. http://agner.org/optimize/calling_conventions.pdf. Accessed 2014-09-30.

[30] Fonseca, J. Gprof2dot. http://code.google.com/p/jrfonseca/wiki/Gprof2Dot. Accessed 2014-03-27.

[31] Fowler, M. Yet another optimization article. *IEEE Software*, May/June (2002), 20–21. ISSN 0740-7459.

[32] Free Software Foundation. Debugging with GDB. Manual. https://sourceware.org/gdb/current/onlinedocs/gdb/. Accessed 2014-08-01.

[33] Free Software Foundation. GNU Binutils. Web page. http://www.gnu.org/software/binutils/. Accessed 2014-09-30.

[34] Free Software Foundation. Interoperation - using the GNU compiler collection (GCC). Web page. https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Interoperation.html. Accessed 2014-07-31.

[35] Free Software Foundation. Nm(1) - GNU development tools. Manual page. http://man7.org/linux/man-pages/man1/nm.1.html. Accessed 2014-07-31.

[36] Free Software Foundation. Objdump(1) - GNU development tools. Manual page. `http://man7.org/linux/man-pages/man1/objdump.1.html`. Accessed 2014-08-01.

[37] Free Standards Group. DWARF debugging information format, version 3, 2005. `http://www.dwarfstd.org/doc/Dwarf3.pdf`. Accessed 2014-08-05.

[38] GNU project. gcov – a test coverage program. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Accessed 2014-09-10.

[39] GNU project. GNU gprof. `https://sourceware.org/binutils/docs/gprof/index.html`. Accessed 2014-03-26.

[40] Goel, B., McKee, S. A., and Själander, M. *Techniques to Measure, Model, and Manage Power*, vol. 87 (Green and Sustaiable Computing: Part I) of *Advances in Computers*. Elsevier, 2012, ch. 2. ISBN 978-0-12-396528-8.

[41] Google. Google performance tools. `http://code.google.com/p/gperftools/wiki/GooglePerformanceTools`. Accessed 2014-03-26.

[42] Google. Gperftools. `http://gperftools.googlecode.com/svn/trunk/doc/index.html`. Accessed 2014-03-26.

[43] Gregory, J. *Game Engine Architecture*, second ed. CRC Press, Boca Raton, FL, 2015 [sic]. ISBN 978-1-4665-6001-7.

[44] Hager, G., and Wellein, G. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2011. ISBN 978-1-4398-1192-4.

[45] Hähnel, M. Energy/utility for L4Re. Master's thesis, Dresden University of Technology, 2012.

[46] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review 40*, 3 (2012), 13–17. DOI: 10.1145/2425248.2425252.

[47] Hamilton, J. Cooperative expendable micro-slice servers (CEMS): Low cost, low power servers for internet-scale services. In *CIDR 2009 - 4th Biennal Conference on Innovative Data Systems Research* (2009).

[48] Hardkernel. ODROID-XU+E. Web page. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079&tab_idx=2. Accessed 2014-09-09.

[49] Hubička, J. Porting GCC to the AMD64 architecture. In *Proceedings of the GCC Developers Summit* (2003), pp. 79–106.

[50] Innovative computing laboratory, University of Tennessee. PAPI - overview. http://icl.cs.utk.edu/papi/overview/index.html. Accessed 2014-06-12.

[51] Innovative computing laboratory, University of Tennessee. PAPI - user guide. http://icl.cs.utk.edu/projects/papi/wiki/User_Guide. Accessed 2014-06-12.

[52] Intel. Intel VTune Amplifier XE 2013. http://software.intel.com/en-us/sites/default/files/Intel-VTune-Amplifier-XE-2013-PB-082113.pdf Accessed 2014-06-12.

[53] Intel. Pin - a dynamic binary instrumentation tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool Accessed 2014-06-12.

[54] Intel. Pin 2.13 user guide. http://software.intel.com/sites/landingpage/pintool/docs/62732/Pin/html/ Accessed 2014-06-12.

[55] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, 2014. Order number 253665-050US. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf. Accessed 2014-06-23.

[56] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, 2014. Order number 325384-050US. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf. Accessed 2014-06-23.

[57] Irvine, K. R. *Assembly language for x86 processors*, 6th ed. Pearson/Prentice Hall, Upper Saddle River, 2011. ISBN 978-0-13-602212-1.

[58] Janjusic, T., and Kavi, K. *Hardware and Application Profiling Tools*, vol. 92 of *Advances in Computers*. Elsevier, 2014, ch. 3. ISBN 978-0-12-420232-0.

[59] Jarp, S., Jurga, R., and Nowak, A. Perfmon2: A leap forward in performance monitoring. *Journal of Physics: Conference Series 119* (2008).

[60] Kartik, S. V., Couturier, B., Clemencic, M., and Neufeld, N. Measurements of the LHCb software stack on the ARM architecture. *Journal of Physics: Conference Series 513* (2014). DOI 088/1742-6596/513/5/052014.

[61] Kefallonitis, F. Name mangling demystified, 2007. `http://www.int0x80.gr/papers/name_mangling.pdf`. Accessed 2014-07-31.

[62] Kuperberg, M. *Quantifying and Predicting the Influence of Execution Platform on Software Component Performance*, vol. 5 of *The Karlsruhe Series on Software Design and Quality*. KIT Scientific Publishing, 2011. ISBN 978-3-86644-741-7.

[63] Levon, J. OProfile manual, 2004. `http://oprofile.sourceforge.net/doc/index.html`. Accessed 2014-03-26.

[64] Malik, S. The CMSSW documentation suite: The CMS offline workbook. Web site. `https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBook`. Accessed 2014-09-30.

[65] Mars, J., Tang, L., and Hundt, R. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *ISCA '13 Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), pp. 619–630.

[66] Matz, M., Hubička, J., Jaeger, A., and Mitchell, M. System V application binary interface AMD64 architecture processor supplement, 2012. Version 0.99.6. `http://www.x86-64.org/documentation/abi.pdf`. Accessed 2014-08-04.

[67] McCalpin, J. D. STREAM: Sustainable memory bandwidth in high performance computers. Web page. `http://www.cs.virginia.edu/stream/`. Accessed 2014-09-04.

[68] Mosberger-Tang, D. unw_step – advance to next stack frame. Manual page. `http://www.nongnu.org/libunwind/man/unw_step(3).html`. Accessed 2014-08-05.

[69] Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, 2004. `http://valgrind.org/docs/phd2004.pdf`.

[70] Novell. Perfmon2–hardware-based performance monitoring, 2012. Web page. `http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.perfmon2.html`. Accessed 2014-09-01.

[71] Oakley, J., and Bratus, S. Exploiting the hard-working DWARF: Trojans with no native executable code. Tech. Rep. TR-2011-680, Computer Science Dept. Dartmouth College, Hanover, New Hampshire, 2011. `http://ph-neutral.darklab.org/talks/tr2011-680.pdf`. Accessed 2014-08-05.

[72] Patterson, D. A., and Hennessy, J. L. *Computer organization and design*, fifth ed. Morgan Kaufmann, 2014. ISBN 978-0-12-407726-3.

[73] Pettersson, M. Linux performance counters driver. Web page. `http://sourceforge.net/projects/perfctr/`. Accessed 2014-09-02.

[74] Roux, Y. Re: [libunwind-devel] fast stack trace for AArch64 and ARM. Message on mailing list. `http://lists.nongnu.org/archive/html/libunwind-devel/2014-08/msg00020.html`. Accessed 2014-08-22.

[75] Seward, J., et al. Valgrind documentation. `http://valgrind.org/docs/manual/index.html`. Accessed 2014-03-26.

[76] Tanenbaum, A. S., and Woodhull, A. S. *Operating systems: design and implementation*, third ed. Pearson/Prentice Hall, Upper Saddle River, NJ, 2009. ISBN 978-0-13-505376-8.

[77] Taylor, I. L. .eh_frame, 2011. Web page. `http://www.airs.com/blog/archives/460`. Accessed 2014-08-05.

[78] Texas Instruments. Ina231. Web page. `http://www.ti.com/product/INA231/description`. Accessed 2014-09-09.

[79] Treibig, J. likwid – lightweight performance tools. Web site. `https://code.google.com/p/likwid/`. Accessed 2014-09-05.

[80] Treibig, J., Hager, G., and Wellein, G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (2010). DOI 10.1109/ICPPW.2010.38.

[81] Tuura, L., Innocente, V., and Eulisse, G. Analysing CMS software performance using IgProf, OProfile and callgrind. *Journal of Physics: Conference Series 119* (2008). DOI 10.1088/1742-6596/119/4/042030.

[82] University, R. HPC Toolkit. `http://hpctoolkit.org/`. Accessed 2014-08-26.

[83] Weaver, V. The unofficial Linux perf events web-page. Web page. `http://web.eece.maine.edu/~vweaver/projects/perf_events/`. Accessed 2014-09-02.

[84] Weaver, V. M., et al. Measuring energy and power with PAPI. *Proceedings of the International Conference on Parallel Processing Workshops* (2012), 262–268. DOI: 10.1109/ICPPW.2012.39.

[85] Weaver, V. M., et al. PAPI 5: Measuring power, energy, and the cloud. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013), 124–125. DOI: 10.1109/ISPASS.2013.6557155.

[86] Weidendorfer, J. KCachegrind. `http://kcachegrind.sourceforge.net/html/Home.html`. Accessed 2014-03-26.

[87] Wielaard, M. J. Stack unwinding, 2007. Web page. `https://gnu.wildebeest.org/blog/mjw/2007/08/23/stack-unwinding/`. Accessed 2014-08-05.