Eirik Smithsen

Fast Call Graph Profiling

Master's thesis in Computer Science

Supervisor: Magnus Jahre

June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Eirik Smithsen

Fast Call Graph Profiling

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Fast Call Graph Profiling

Eirik Smithsen

June 9, 2020

Assignment text

The end of Dennard scaling and the imminent end of Moore's law is causing disruptive changes to the way computers are designed. An attractive option is to create specialized hardware units — called accelerators — that are able to execute performance-critical code regions (much) more efficiently than a general-purpose processor. Unfortunately, designing accelerators is costly which limits their applicability to high-volume domains such as graphics or machine-learning. Another option is to add tightly coupled reconfigurable fabrics to general-purpose processors and combine this with tools that automatically generate application-specific accelerators.

A key challenge for automatic accelerator generation tools is to determine which part of the application should be accelerated. This is typically determined based on a performance profile of the application which can be collected using instrumentation or sampling. Instrumentation can gather precise information but can interfere with program behavior. Sampling, on the other hand, minorly affects application performance but cannot gather key information such as dynamic call graphs. For these reasons, researchers run the application twice to (i) determine per-function performance with sampling, and (ii) retrieve the dynamic call graph using instrumentation. The performance overhead of this approach can be substantial for long-running applications.

In this thesis, the student should develop an LLVM-compatible approach for gathering application performance profiles that uses both sampling and instrumentation. The objective of the thesis is to investigate if it is possible to design a profiler that combines intelligent instrumentation — to gather key information such as dynamic call graphs without extensive interference — and sampling — to determine the relative performance impact of each function.

Abstract

CPUs are general purpose chips that are able to perform all computations, but they are not optimized to perform any particular computation significantly faster than others. It is possible to create chip that are optimized to perform a limited set of computations that are much faster than the CPU at those specific computations. Such chips are called accelerators. Since CPU performance has stagnated in recent years researchers have suggested using reconfigurable chips such as Field Programmable Gate Arrays and Coarse Grained Reconfigurable arrays as accelerators that can be programmed to accelerate any program. To be able to know which part of a program that should be accelerated the programmer can profile the program to gain information on how the program behaves. If it is possible to profile multiple kinds of information in a single run of the program without the profilers interfering with each other it is possible to save time by not having to run the program multiple times.

In this thesis I have investigated if it is possible to do profile both how much time is spent in each function of the program and profile the dynamic call graph of the program in a single run. To this I have created the Fast Call Graph profiler which can profiles the dynamic call graph of programs. I also evaluated how much overhead the profiler causes and how it compares to similar tools that profile the dynamic call graph.

I found that the Fast Call Graph profiler does not change the time distribution information in any significant way. This means that it is possible to do both time distribution profiling and dynamic call graph profiling in a singe program run and get useful profile results. The overhead of the Fast Call Graph profiler is also significantly lower than the tools I compared it with which were Gprof and the instrumentation built into Clang.

Sammendrag

CPUer er ikke-spesialiserte chiper som kan utføre alle beregninger, og de er ikke optimalisert til å utføre noen beregninger mye raskere enn andre. Det er mulig å lage chiper som er optimaliserte til å utføre et begrenset sett med beregninger mye raskere enn en CPU kan gjøre tilsvarende beregninger. Slike chiper kalles akseleratorer. Siden CPU-ytelse has stagnert de siste årene har forskere foreslått å bruke re-konfigurerbare chiper, slik som Field Programmable Gate Arrays og Coarse Grained Reconfigurable Arrays, som akseleratorer som kan akselerere hvilket som helst program. For at programmerer skal vite hvilken del av programmet som bør aksellereres så han han kjøre kode som obsererer hvordan hans program oppfører seg. Dersom det er mulig å kjøre programmet en gang med kode som observerer flere forskjellige aspekter ved programmet, uten at koden for de forskjellige aspektene påvirker hverandres observasjoner, så er det mulig å spare tid fordi man trenger kun å kjøre programmet en gang.

I denne oppgaven har jeg undersøkt om det er mulig å observere både hvor mye tid som blir brukt i hver funksjon i koden og observere den dynamiske funksjonskall-grafen i en enkelt kjøring av programmet. For å gjøre dette så har jeg laget Fast Call Graph profiler som observerer funksjonskall-grafen til programmer. Jeg har også evaluert hvor mye dette verktøyet sakker ned koden som det observerer og sammenlignet med hvor mye liknende vertøy sakker ned koden.

Det jeg har funnet er at Fast Call Graph profiler påvirker ikke i noe stor grad hvor mye tid programmet bruker i hver funksjon. Det vil si at det er mulig å både observere hvor mye tid som brukes i hver funksjon og observere den dynamiske funksjonskall-grafen samtidig og få brukbare resultater. Fast Call Grap profiler sakker programmet ned betydelig mindre enn Gprof og innebygd Clang observering, som var de andre verktøyene jeg sammenlignet med.

Acknowledgements

I wish to thank Associate Professor Magnus Jahre for his supervision, the discussions we have had and all the suggestions he has provided. I also wish to thank Joseph Rogers and Björn Gottschall for their feedback and insights during my work on this thesis.

Contents

Assignment text . . . iii
Abstract . . . v
Sammendrag . . . vii
Acknowledgements . . . ix
Contents . . . xi
Figures . . . xiii
Tables . . . xv
1 Introduction . . . **1**
1.1 Accelerators . . . 1
1.2 Program acceleration . . . 3
1.3 Transparent acceleration . . . 4
1.4 Fast Call Graph (FCG) Profiler . . . 4
1.5 Tasks . . . 5
1.6 Contributions . . . 6
1.7 Outline . . . 6
2 Background . . . **9**
2.1 Sampling . . . 9
2.1.1 Perf . . . 10
2.2 Static binary instrumentation . . . 10
2.2.1 Clang instrumentation . . . 10
2.3 Dynamic binary instrumentation . . . 11
2.4 Combined approaches . . . 11
2.4.1 Gprof . . . 11
2.4.2 Causal profiling . . . 11
2.4.3 DYPER . . . 12
3 Fast Call Graph (FCG) profiler . . . **13**
3.1 Profiling strategy . . . 13
3.2 Implementing the Fast Call Graph Profiler . . . 14
3.3 Performance overhead . . . 16
4 Experimental setup . . . **17**
4.1 Environment . . . 17
4.2 Benchmarks . . . 18
5 Results . . . **21**
5.1 Profiling overhead . . . 21

5.2 Time distribution impact 25
5.3 Accuracy of function call counts 28
5.4 Indirect calls 29
6 Conclusion **31**
6.1 Conclusion 31
6.2 Further work 31
Bibliography **33**

Figures

1.1 Classification of accelerators 2
1.2 Generic development process 3
1.3 FCG profiler output 5

3.1 Fast Call Graph instrumentation 14

4.1 Evaluation process 18

5.1 GCC and Gprof profiler overhead 22
5.2 LLVM based configurations normalized execution time 23
5.3 Number of function calls per second 24
5.4 Clang instrumentation normalized execution time and function calls per time scatter plot 24
5.5 Fast Call Graph profiler normalized execution time and function calls per time scatter plot 25
5.6 GCC and Gprof stacked top functions 26
5.7 GCC, Gprof (perf data, removed Gprof functions) and Gprof reported data stacked top functions 26
5.8 LLVM based configurations stacked top functions 27
5.9 LLVM based configurations stacked top functions, removed benchmarks with differing inlining 28
5.10 Percentage indirect calls 29

Tables

4.1 Computer hardware and software used 18

Chapter 1

Introduction

From the 1990's to around 2010 the performance of central processing units (CPU) grew exponentially. But the end of Dennard scaling and the rise of dark silicon has in recent years lead to a stagnation in single core CPU performance. To further increase the performance of CPUs the microprocessor makers has increased the number of cores per CPU. The problem with this strategy is that the CPUs are still affected by dark silicon. [1] [2]

1.1 Accelerators

Accelerators has been proposed as a response to this performance stagnation. Accelerators are digital chips that are created to do a specific type of computation either faster than a CPU or to be more energy efficient than a CPU performing the same computations. They are integrated into the computer and can offload some computation from the CPU. Modern computers are already using accelerators, both integrated in and external to the CPU.

One example of a CPU integrated accelerator is the CPU hardware needed to support the AES-NI instruction set for the x86 architectures [3]. Using this hardware for AES computations can, according to Intel, be performed 3 to 10 times faster than performing the computations by using non-AES-NI CPU instructions.

An example of an external accelerator is the graphics processing unit (GPU). GPUs are used to compute and draw graphics onto the computer screen. In most graphics computations the new value of each pixel is independent of the other pixels, which means that the new value for every pixel can be computed in parallel. GPUs makes use of this fact and are made up of a large number of simple arithmetic units that perform computations on pixels in parallel, and there can be from 100s to 1000s of these units in a single GPU. By doing computations on all these units in parallel the GPU is able to perform more graphics computations per second than a CPU. The arithmetic units are also more energy efficient per graphics computation than normal CPU cores since they are running at a lower clock frequency, in the range 1 to 2GHz, and they are also simpler circuits since they only support graphics related computations. [4] [5]

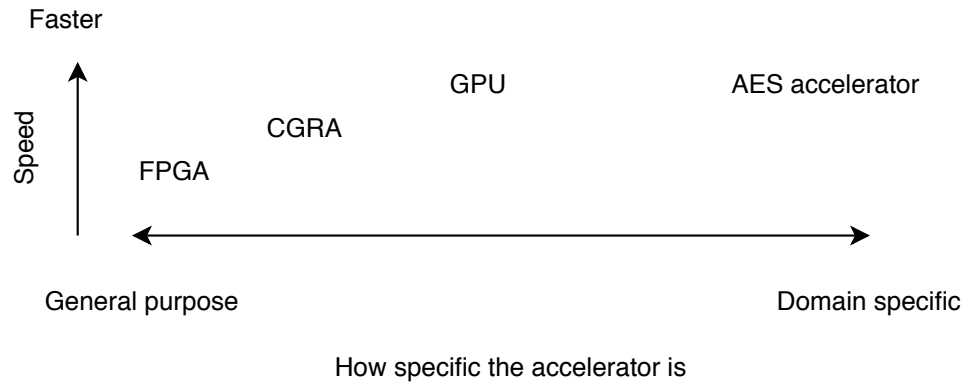


Figure 1.1: Simple and coarse grained classification of accelerators

Accelerators can be classified according to how specific they are and how fast they are. Figure 1.1 shows a simple and coarse grained classifications of the two accelerators I have discussed and two types of accelerators that I will discuss. The AES accelerator is an example of a domain specific accelerator. The circuit is only able to perform AES computations, but it performs them fast and efficient. The GPU is a more general purpose accelerator, since it can be programmed to perform arithmetic computations on a set of data. It is fast because of the large degree of parallelism.

FPGA and CGRA, which are abbreviations for Field Programmable Gate Arrays and Coarse Grained Reconfigurable Arrays respectively, are two types of reconfigurable accelerators. Reconfigurable means that it is possible to program the behaviour of the accelerator, i.e. which computations it performs.

A FPGA is a chip consisting of small, programmable logic blocks that are connected via programmable interconnects [6]. The advantage of FPGAs is that the logic blocks can be programmed to simulate logic gates such as AND and OR and therefore it is possible to program a digital circuit, i.e. digital computation, into the FPGA. The only limit on which computations can be programmed is the number of logic blocks and interconnects that the FPGA has. The main downside of FPGAs is that they are normally slow since they are only able to achieve low clock speeds, typically less than 1 GHz.

A CGRA is similar to a FPGA in that it can be programmed to perform a computation. The difference is that a CGRA consists of larger functional units compared to the FPGAs small and simple logic blocks. The functional units performs a somewhat complex computation, such as addition, subtraction, multiplication and/or division, and they can be connected via interconnects the same way as as the logic blocks in FPGAs. [7]. Each functional unit is implemented in hardware and a CGRA can therefore be faster or more energy efficient than a FPGA. The cost of this increased speed over FPGAs is that the CGRA is less flexible than a FPGA, it can only perform computations that consists of the operations that the functional units perform.

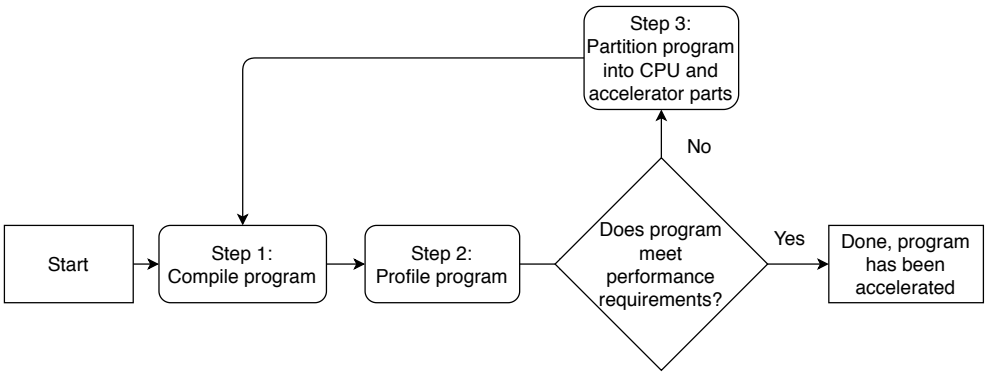


Figure 1.2: Illustration of the generic development process

1.2 Program acceleration

To make programs use the available accelerators both the programmers and their tooling have to be aware of the accelerators and know how to utilize them. They also need some information on how the program behaves so that they can reason about which accelerator to use and how to use it. The act of running a program to gather program behaviour information is called profiling. Interesting program behaviour can be the number of calls to each function in the program, how the program utilizes the cache, how much of the execution time is spent in different parts of the program, etc.

The process of accelerating a program can be described by the generic development process from STHEM [8], illustrated in Figure 1.2. Following that process the programmer compiles and profiles the program on the CPU. If the profile information indicates that the program meets the programmers performance requirements, such as execution time, energy usage, etc. they use the program as it is. If it does not meet their requirements the programmer uses the profile information to choose an accelerator to accelerate the program with and then partition the program into a CPU part and an accelerator part. With this new, partitioned program the programmer then restarts the process from step 1.

One challenge that comes with profiling a representative "version" of the program is that it might take some time, since programs that a programmer wants to accelerate are often long running programs, as if they were short programs there is little to be gained from accelerating them. Profilers will also introduce some execution time overhead, i.e. slow down the program, which can make the profiling step take a lot of time. In addition, in some cases the programmer might have to profile the program several times with different profilers if each profiler gathers different information and the profilers can not run at the same time.

1.3 Transparent acceleration

Transparent acceleration is the process where either a program automatically performs the process described in Figure 1.2 to produce an accelerated binary, or accelerates a program dynamically at runtime as suggested in [9].

NEEDLE [10] is an example of a transparent acceleration tool. Its main goal is to reduce energy consumption, but in the paper they also report slight speedups. The process that NEEDLE follows is similar to the process in Figure 1.2, but it skips the "meets requirements" part and the process is complete after partitioning the program. This process is fully automatic. But before the automatic process can start the programmer has to manually select which function to accelerate. To select the most suitable function the programmer needs profiling information about the program. The reason that NEEDLE has to be told which function to accelerate is that it profiles execution paths starting at the selected function, by enumerating all possible paths from the selected function and also instruments those paths. In some programs the total number of paths from the "`main`" function can become large, needing more than 64 bits to represent the number of paths. And instrumenting every single path in the program will lead to a greater slowdown than if only instrumenting paths in a single function.

To select the function to accelerate the programmer can use a profiler that gathers information on how the execution time is distributed between the functions and select the function with the largest time distribution, since that function has the greatest theoretical absolute potential of speedup. The programmer can combine the time distribution profile with a call graph and function call counts profile to select a function that is both responsible for a relatively large part of the execution time and is called a low number of times, for example if the accelerator they use takes a long time to invoke.

1.4 Fast Call Graph (FCG) Profiler

As part of this thesis I have created the Fast Call Graph Profiler. It can be used to do both time distribution profiling, and call graph and function call count profiling during a single program run. It instruments each function call and keeps a counter for each caller-callee function pair. A caller is the function that the call was performed in, and a callee is the function that is being called. These counters can be used to make an accurate call graph after running the program. An example of the output from the FCG profiler is shown in Figure 1.3. The output lists each caller-callee pair from the source program along with the counter of how many times each call were performed. The FCG profiler works by instrumenting LLVM bitcode, which is the Intermediate Representation (IR) of the LLVM compiler framework [11]. The Clang C/C++ compiler is a part of LLVM that can produce bitcode for C and C++ programs.

The FCG profiler can be combined with the perf profiler, which is a time distribution profiler, to obtain both function call counts and time distribution informa-

Program:

```
function1:
  call function3
  instruction 1
  call function2

function2:
  instruction1
  call function3
  call function3

function3:
  instruction1
```

Profile output (if running function1):

Caller:	Callee:	Number of times:
function1	function3	1
function1	function2	1
function2	function3	2

Figure 1.3: Illustration of the FCG profiler output

tion from a single program run.

1.5 Tasks

From the assignment text I have formulated the following tasks that I am to do:

- T1: create an instrumentation profiler that is compatible with the LLVM compiler framework,
- T2: the profiler should gather dynamic call graphs, which is key information about the program,
- T3: investigate if the profiler from T1 can be used with a sampling profiler to obtain both dynamic call graphs and function time distribution information from a single program run, without the instrumentation causing extensive interference for the sampling profiler,
- T4: in addition I will compare the performance of the T1 profiler against existing tools.

One example of extensive interference, as mentioned in T3, is when profiling a function where a large fraction of the instructions are function calls. If this function is instrumented to count function call the result information from a time sample profiler might attribute a artificially large fraction of the execution time to this function compared to the functions it calls, since the instrumentation adds extra instructions to count each function call and the function consists of a larger fraction of function calls than normal.

I will use Linux's perf [12] as the sampling profiler in task T3. perf is a widely used sampling profiler and since it runs "on top of" the program binary it is compatible is compatible with all compilers.

To investigate and measure task T3 I will compare the output of perf when running with and without the instrumentation profiler and see if the time distribution of functions is different. I will also measure the execution time of the programs with and without instrumentation to quantify the overhead introduced by the instrumentation.

And to answer T4 I will compare the performance of the profiler with two other profilers that both record function call counts and are easy to use, namely gprof [13] which is built into the GCC compiler [14] and the built in instrumentation in the Clang compiler [11] that is used for profile guided optimization (PGO).

For both T3 and T4 I will run the profilers on the benchmarks in the SPEC [15] and PARSEC [16] [17] benchmark suites.

1.6 Contributions

My contributions in this thesis is the following:

- C1: the Fast Call Graph profiler, as a response to task T1 and T2,
- C2: demonstrated that the Linux perf profiler can be used with the Fast Call Graph profiler to obtain caller-calle execution counts and function time distribution as a response to task T3,
- C3: found that the Fast Call Graph profiler does not cause extensive interference in the perf profile data as a response to task T3,
- C4: found that Fast Call Graph profiler causes significantly lower overhead than Clang instrumentation and gprof as a response to task T4.

My solution to T1 and T2 is the Fast Call Graph profiler described in Section 1.4. The profiler records dynamic call graphs for the program and will be described in greater detail in Chapter 3. This is the first contribution, C1.

My second contribution, C2, is that I have evaluated how the Fast Call Graph profiler can be used in conjunction with perf, as a response to task T3. The result of C2 is C3, I found that the time distribution of the benchmarks does not change significantly with profiling, which means that it can be feasible to do both instrumentation and sampling in a single program run as a step of automatic or manual program acceleration.

The last contribution, C4, is that I have show that the overhead of Gprof is very high and that the Fast Call Graph profiler almost always has lower overhead than the Clang instrumentation, which is most likely since the Fast Call Graph profiler records less information than the Clang instrumentation.

1.7 Outline

In the next chapter I introduce additional background information that has not been covered in this introduction. It will also cover related work and state of the art research. Chapter 3 introduces and describes the profiler that I have created

in detail, the Fast Call Graph profiler. Chapter 4 describes how the Fast Call Graph profiler were evaluated.Chapter 5 presents, compares and discusses the results of the Fast Call Graph profiler against gprof and the built in instrumentation in Clang. Chapter 6 sums up my work and concludes the thesis.

Chapter 2

Background

There are three main approaches to profiling program behaviour. They are sampling, static binary instrumentation and dynamic binary instrumentation. Each of them have their strengths and weaknesses and are suited for different types of profiling. In the following sections I will describe each of the approaches. I have also included a section called Combined approaches where I discuss research and tools that combine multiple of the four main approaches.

2.1 Sampling

Sampling is an approach where the program is regularly interrupted, and at each interrupt the profiler records information about the program or the hardware, such as hardware counters. By varying the interrupt interval, i.e. the number of samples per second, the overhead of the profiler can be adjusted at the cost of granularity in the samples. Fewer samples per second results in lower overhead, since the work performed by the profiler is done fewer times per second. The trade-off of reducing the number of samples per second is that the samples becomes more coarse-grained and could hide behaviour that only occurs for short intervals of time.

Normally the sampling profilers run on the same host as the program being profiled. There are also sampling profiler which are called non-intrusive, which runs on another host and gets profiler information about the host running the profiled program though protocols such as JTAG or external measurements such as power consumption. Aveksha is an example of non-instrusive profiling, the CPU of the host running the profiled program outputs the program counter (PC) over JTAG (this capability is built into the CPU, so little/no overhead) and another host samples the PC. [18]. This is similar to how the Lynsyn board from STHEM [8] works, it samples energy information and sends it the the profiling host. Non-intrusive sampling profilers are supposed to not affect the host running the profiled program, so that the profiled program is not affected by the running profile and the profile data is the "ground truth". Normal, intrusive sampling profilers which runs on the same host as the program being profiled can be implemented

either in the application binary, such as gperftools [19], as userspace programs, such as [20], or in the kernel, such as Linux perf.

Intrusive sampling profilers has normally low overhead, at least if they are only performing light work such as time distribution profile, which is a profile of how much time is spent in each function. For time distribution profiles the overhead is normally a low single digit percentage, the oprofile profiler reports 1-8% depending on workload and sampling frequency. [21] Intrusive samling profilers can influnce the behaviour of the program being profiled and therefore reduce the accuracy and correctness of the profile, but they are often significantly easier to use than non-intrusive profilers since they do not require setting up multiple hosts with the correct capabilities and connecting them together.

2.1.1 Perf

Perf is a intrusive sampling profiler built into the Linux kernel. It can record a large number of different metrics about the running program, e.g. function time distribution, number of instructions executed, number of cache hits etc. It interrupts the running program at a regular interval and records statistics for the program execution since last interrupt. Perf is a low overhead tool to record time distribution of programs and easily accessible since it is part of the Linux kernel.

2.2 Static binary instrumentation

Static binary instrumentation profiling, where additional machine instructions are inserted into the program binary before the binary is executed, are one way of obtaining accurate information about the program being run, and it can be done with a low overhead if the recorded information is easy to compute and store.. This type of profiling can be used to obtain information such as the number of times each function is called and how often each branch in the program is taken. Since this type of profiling inserts extra machine instructions to be executed the result is almost always a slowdown, and more instrumentation/data recorded results in a greater slowdown. During my work I found that in some special cases the instrumented binary can execute faster than the uninstrumented one. I suspect this is because of alignment and/or cache and/or branch prediction gains.

2.2.1 Clang instrumentation

Clang has built in support for instrumenting the binaries it generates to produce profile data that can be used in profile guided optimization (PGO), which is a process where the profile data is used to make decisions when optimizing the program. Clang instruments the binary to count how many times each function is executed and how many times each basic block is executed. The start and end of basic blocks corresponds to branches and function calls in the program. The instrumentation does not record any caller-callee relations or call stacks. It does

however count indirect function calls. The profiler is intented to be used for PGO in clang, but it can also be used as an easy way to get accurate call counts for all functions.

2.3 Dynamic binary instrumentation

Dynamic binary instrumentation instruments the machine code stream on the fly. Can get detailed information about the program without modifying the binary. Examples are Valgrind [22] and Pin [23]. The overhead of this approach can be high and are closely related to the amount of instrumentation being done. Valgrind states that the slowdown factor can range 5 to 100. Since these tools only operates on the binary they are independent of the source program and its build tools unlike some static instrumentation profilers which are implemented in a specific compiler (such as Clang instrumentation).

2.4 Combined approaches

Combined approaches are approaches that combine multiple of the three main approaches.

2.4.1 Gprof

Gprof is a GCC built in function call counting instrumentation and program time distribution sampling profiler. It combines sampling with static binary instrumentation. It instruments the code to count how many times each function is executed and sets up a sampling function that samples at regular intervals to record how much time is spent in each function.

2.4.2 Causal profiling

Causal profiling [24] is a recently proposed way of doing profiling to provide a profile that fits the programmer intuition of how to optimize a program. It can be seen as a combination of sampling profiling and dynamic binary instrumentation since it runs at regular intervals and instruments the program code on the fly when it runs. The way causal profiling works is by slowing down all threads but one which is approximately equivalent to speeding up the one thread not being slowed. During this slow down they record how much slower the program executes and use that to determine how large speedup it is possible to achieve by optimizing the code that ran in the normal-speed thread. The problems associated with the traditional approach of using a sampling profiler that record function time distribution and then optimize the functions with a large percentage of the execution time that causal profiling seeks to solve is that some functions might be waiting for external resources, such as disk or network, and therefore consume a lot of execution time. In multi threaded applications there might also be several

threads that execute at the same time and then waits for all the threads to complete before proceeding. In such a case speeding up any individual thread will not result in a speedup of the whole program since the execution time depends on the slowest of the threads execution together.

2.4.3 DYPER

The DYPER framework [25] is another approach that combines sampling and dynamic binary instrumentation. It lets the programmer set a target performance overhead and the framework then makes sure that the profiling done does not slow the program down more than the target overhead. Profiling in DYPER is done by proflets, which are pieces of code that each profile a different performance aspect. Each proflet can do two types of analysis, basic and detailed. The basic analysis is done in a sampling fashion, where DYPER samples the stack and then the proflets analyse the stack sample. The detailed analysis is implemented by using dynamic binary instrumentation to instrument the program at runtime to obtain the performance data that the proflet is interested in. The proflets also include information about their detailed analysis, how long it takes to run it, how much it slows the program down and its priority. DYPER uses this information from each proflet and the target overhead to determine how often it should do sampling (basic analysis), how often it should do dynamic binary instrumentation (detailed analysis) and how often each proflet should run. The data that DYPER gathers are a statistical approximation to the actual data since it does not profile all the time. The tradeoff is to reduce the overhead compared to profiling all the time by profiling at regular intervals and then extrapolate numbers to cover the whole program execution.

Chapter 3

Fast Call Graph (FCG) profiler

3.1 Profiling strategy

The goal of the profiler is to record the dynamic call graph, i.e. it should count how many times each distinct edge in the call graph is taken, and be compatible with LLVM. To achieve this I considered three different approaches, mcount instrumentation (will be explained soon) or similar instrumentation, using existing instrumentation frameworks or doing LLVM IR instrumentation.

Both the GCC and Clang compilers can be configured to insert calls to the function mcount at function entry. I can then write an implementation of the mcount function that does the necessary edge counting and link the function against the program binary. A drawback of using this approach is that the mcount function does not directly get any information on which function it was called from or which function called the current function. To get this information it is necessary to traverse the stack and find the information there. Traversing the stack is not a zero-cost operation so this approach will give a larger overhead than to just increment a counter. There is also some overhead associated with calling the mcount function, but Link Time Optimization (LTO) might be used to inline the mcount function body in each function. One advantage of this approach is that indirect function calls require no special treatment, they are supported in the same way as direct function calls. GCC can also insert calls to `__cyg_profile_enter` and `__cyg_profile_exit` at function entry and exit respectively. These functions are called with the address of the current function and the call site. This solves the main problem associated with using mcount instrumentation, but it is still a function call which means that there is some overhead in calling the profiling function. It also seems to be GCC only which means that I can not use it since the profiler needs to be LLVM compatible.

The CSI framwork [26] is an instrumentation framework for LLVM. It functions similarly to the mcount function in that it inserts calls to instrumentation functions that the user needs to write and link the binary against. The framework has an instrumentation function that is inserted before function calls which I could have used. The instrumentation function gets the called function and a call site ID as

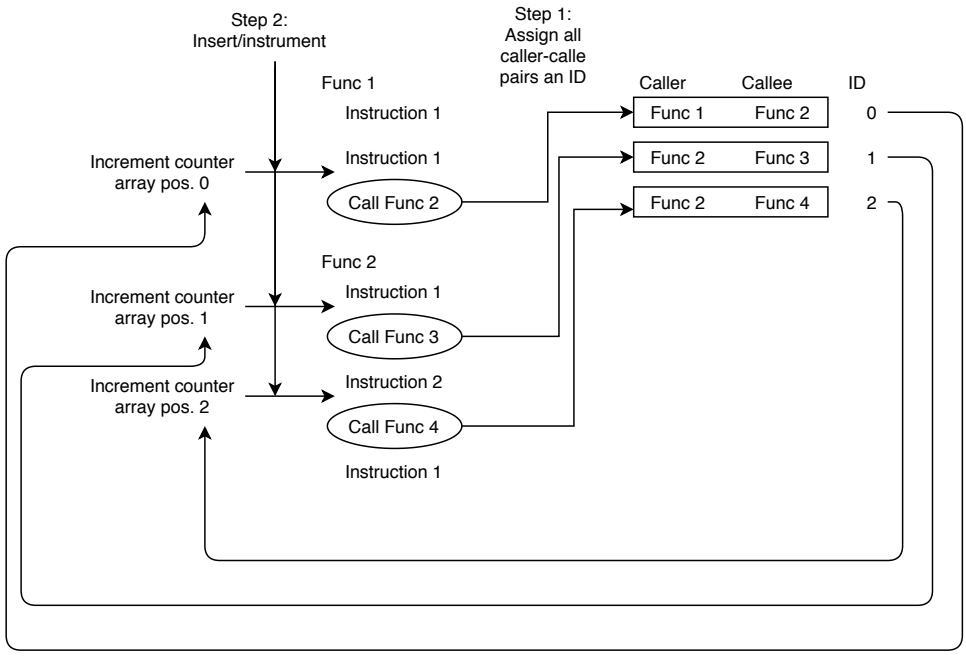


Figure 3.1: Illustration of the Fast Call Graph profiler instruments code

arguments. It might be possible to write the instrumentation function in such a way that it can be optimized into a single increment instruction and inlined during LTO to reduce the overhead. This type of optimization will not work for indirect calls, so they have to be treated separately.

The last approach is the one I chose. Instrumenting the LLVM IR directly is more work and harder than writing the instrumentation code in C, as you could with the other two approaches, especially when you are inexperienced with LLVM IR. But I think the approach is the most flexible and the one where it is easiest to obtain a low overhead since I write the actual IR and can be sure that the instrumentation is inlined.

This approach is similar to the built in instrumentation in Clang, but my profiler is going to record a slightly different set of information. As mentioned in the Background Chapter, Clang instrumentation records execution counts for each function and basic blocks. I am not interested in basic block counts and can therefore achieve lower overhead than Clang since I record less data. The other difference is that I will record where each function was called from which is something that Clang does not do.

3.2 Implementing the Fast Call Graph Profiler

I use the NEEDLE source code [27] as a starting point for my profiler since I have worked with it previously and it does LLVM instrumentation. Some parts ofo the

code and design choices are inspired by an demo of LLVM instrumentation by Nick Sumner [28], especially how to store the function names. The profiler is implemented as a stand alone tool that takes a LLVM bitcode file as input and outputs an instrumented binary. To be able to use the profiler on C and C++ programs that consists of more than one compilation element (source file) the Whole Program LLVM (WLLVM) [29] Clang wrapper can be used, which creates a single LLVM IR bitcode file by merging the LLVM IR files of each compilation element. Since the focus of the profiler is low overhead it does not currently support indirect calls, i.e. calls where the target function is computed at runtime, because supporting them would require more instrumentation which leads to larger overhead. This weakness will be further discussed in the Results chapter.

Figure 3.1 shows an illustration of how the profiler is implemented, with call count IDs and instrumentation. To store the call counters I chose to use a C array where each index represents a caller-callee pair. A C array is an efficient and simple way to store the counters since it is a contiguous block of memory and the address for each index can be computed at compile time. The mapping of caller-callee pair to array indexes is done by iterating over the whole LLVM IR and finding all call instructions. This is shown as Step 1 in the figure. Each call instruction is assigned an ID that starts at 0 and increments by 1 for each call instruction. This ID later used as an index into the counters array. During this ID assignment the profiler also gets the name of the caller and calleee functions and stores the pair of function names in an array to be used as labels for each counter value when outputting the profile data at the end of execution. This function name array is also indexed by the ID. If the function call is an indirect call the name of the called function is set to "`indirect_callNNN`" with NNN being a counter that is incremented in the same way as the ID. By handling indirect calls this way it is possible to see how often each indirect function call is executed even though it is not possible to see the target function.

After each call instruction has been assigned an ID the profiler performs the actual instrumentation. This is Step 2 in the figure. Before each function call it inserts instructions to get the counter value at array index ID, increment the value by 1 and then store the value in the array.

When the instrumentation is done the tool links the bitcode against a C function which is set up to run at program end by using the LLVM dtor (destructor) feature. This function outputs the counters from the counter array and the function pair names to a file so that the execution counts can be inspected later. After linking the tool compiles the program into a native binary.

Pre-allocating space in the counter array for each caller-callee pair that appears in the program source is not the most memory efficient way to implement the counters, since some caller-callee pairs might not be called during the course of the program. But by allocating space for each caller-callee pair the profiler knows the counter index for each pair at compile time and there is no need for runtime code to allocate space for each new counter and keeping track of which caller-callee pairs have been executed. Adding more runtime code would have increased the

overhead and since there is normally an abundance of memory available I chose to prioritize speed and low overhead over memory frugality/efficiency.

3.3 Performance overhead

The instrumentation that increments the array counter at runtime consists of four LLVM IR instructions. The instructions are to first compute the counter address, then load the counter value, then increment the counter value by 1 and then lastly store the counter value back to the counter array. The first instruction, the counter address computation, has only constant arguments so during optimization it can be computed and replaced with a constant. On x86 architecture and optimization level -O0 this leads to the four instructions being mapped to three machine code instructions, one MOV to load the counter value, then increment it by one and lastly one MOV to store the value. On higher optimization levels the instrumentation is realized with a single INC instruction.

Execution time overhead will most likely correlate with the total number of function calls since each function call adds either three or one extra instruction depending on optimization level. Since the program has to execute the extra instrumentation for each function call, call intensive functions will most likely get at least a bit larger percentage of execution time with instrumentation. The correlation between function calls and overhead will be further discussed in the Results chapter.

Data listed at [30] indicates that the INC instruction on memory addresses can be executed at one instruction per cycle, with at least 5-6 cycles between multiple INCs of the same address. Since each INC is followed by a function call, which normally takes a at least tens of cycles, it is safe to assume that none of the instrumentation INC instructions will have to wait for each other. This means that the INC instruction instrumentation can be executed at one instruction per cycle and does not cause a large slowdown of the program.

Chapter 4

Experimental setup

4.1 Environment

Table 4.1 lists the main information about the computer used to run all evaluations. GCC is used to compare the Fast Call Graph profiler with gprof, which is the built in profiling in GCC. GCC 7 was used since I encountered problems with GCC 9, which were the most recent version when I started my work, and GCC 7 is the default compiler in Ubuntu 18.04. I used LLVM 3.8 because I used the NEEDLE source code as a starting point for my profiler and NEEDLE uses LLVM 3.8.

My evaluation process is illustrated in Figure 4.1. To record time usage I use the built in SPEC time reporting and /usr/bin/time for PARSEC. Each benchmark is run three times and I use the minimum execution time when comparing configurations since it is an approximation to how fast the program is able to execute under ideal conditions. The overhead is calculated by dividing the execution time by the baseline execution time for each config. The configs and their baseline is listed at the start of the next chapter. The mean overhead for each config is calculated as the arithmetic mean over all the benchmarks. When doing perf profiling I run the benchmarks one time. I only run them once so that I do not have to set up a system that saves the perf data for each SPEC run, since the SPEC tools runs multiple iterations of each benchmark in the same directory, which causes the perf data to be overwritten for every iteration. I also observed that the time measurements were fairly consistent so I found it reasonable to believe that the perf samples would also be consistent. In addition small differences in function time distribution does not matter that much since we are interested in the high level picture, i.e. in which functions is the most time spent. In real life usage, if several functions use about the same amount of time the programmer might choose which function to accelerate based on knowledge of the code and the accelerator. Or they can use another metric as tie breaker, e.g. number of calls to function, number of calls the function makes, size of function, etc.

Table 4.1: Computer hardware and software used

CPU:	Intel i7 6700K, HyperThreading disabled [31]
RAM:	16 GB
OS:	Ubuntu 18.04 [32]
GCC version:	7.5.0
LLVM version:	3.8
WLLVM version:	1.2.7

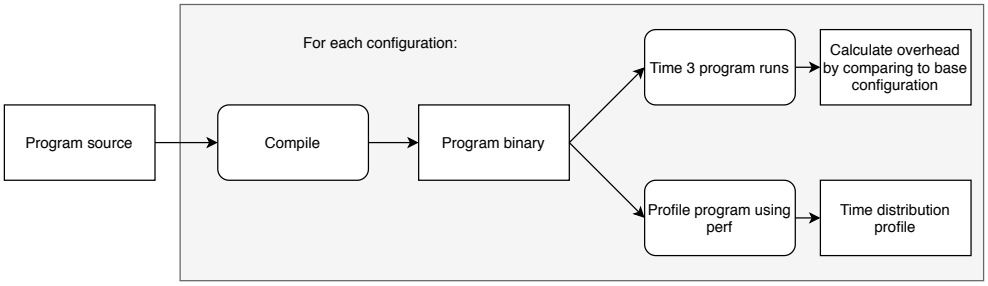


Figure 4.1: Illustration of my evaluation process

4.2 Benchmarks

All benchmarks were compiled with "`-g -O3 -march=native -fno-unsafe-math-optimizations`" compiler options. These compiler options are based on the default options in SPEC CPU. -O3 enables the highest level of compiler optimizations and tries to reduce execution time of the resulting binary. -march=native lets the compiler use all instructions supported by the CPU, e.g. SIMD instructions. -fno-unsage-math-optimizations makes sure that the compiler generates floating poing code that is correct and does not assume anything about arguments.

The SPEC benchmarks are run with one thread for the speed benchmarks and one copy for the rate benchmarks. OpenMP is disabled to make the programs serial. I want serial programs so that the profilers does not have to handle multiple threads potentially writing to the same counters at the same time. Only the benchmarks that execute as a single command were used, so that I avoid having to support running a profiler multiple times in same directory and having the profile information overwrite each other, as discussed previously with running perf a single time. Since I am using the Clang compiler I have only used the benchmarks that consists of C and/or C++ code since Clang does not support Fortran. The benchmarks `510.parest_r` and `525.blender_r` did not compile with the default options in GCC and/or Clang, so I have not used them.

I have use the code at [33] as basis for PARSEC, as the code at the Princeton web site does not build out of the box. The serial version for each benchmark were used, for the same reasons as why I run SPEC serially. All benchmarks were run with the native input. The benchmarks `facesim`, `raytrace` and `dedup` did not compile with Clang so I have not used them. `x264` segfaulted when I ran it after

compiling it with GCC so it was also not used. Lastly I had a problem with the `vips` benchmark, it did not compile/link correctly when using the Fast Call Graph profiler tool. This is a shortcoming of the method used to instrument the binary, and could be overcome by making the profiler into a Clang pass, which is a piece of code that is run as part of the Clang optimizations.

One problem I encountered is that the `ferret` benchmark from PARSEC is not deterministic. When running it multiple times the function-pair call counts reported by the Fast Call Graph profiler are different. Despite not being deterministic it has fairly consistent execution times so I have included it in my testing.

In `511.povray_r`, `538.imagick_r` and `638.imagick_s` the SPEC benchmark runner runs a verification binary after running the benchmark to validate the output. Since the SPEC compile options applies to all binaries being built for each benchmark, the validation-binaries are also built with Gprof and Clang instrumentation profiling enabled. This means that the validation binaries also produce profile data, which overwrites the profile data from the benchmark binary since the profiling uses a standard name for the profile output file. This is the same type of problem that made me use only single command benchmarks and run perf a single time. Despite this I have included the benchmarks in the overhead evaluation since I discovered the problem after obtaining time measurements for them. And the Fast Call Graph profiler does not have this problem with, since for that profiler I have to manually instrument the main binary.

Chapter 5

Results

The five compiler and profiling configurations I have evaluated are:

- GCC,
- Gprof,
- Clang,
- Clang instrumentation,
- Fast Call Graph profiler.

The GCC configuration is used as a baseline for the Gprof configuration. The Gprof configuration is compiled using GCC with the `-pg` command line argument to enable Gprof. Gprof is an easily accessible profiler for gathering function execution counts. Clang is used as a baseline for the Clang instrumentation and the Fast Call Graph profiler configurations. Clang instrumentation is enabled by using the `-fprofile-generate` command line argument and is another easily accessible profiler for gathering function execution counts. The last configuration is the Fast Call Graph profiler. For this configuration the benchmarks are compiled using WLLVM, which used Clang under the hood, and then instrumented with the Fast Call Graph profiler.

5.1 Profiling overhead

To compare the execution time/profiling overhead of the profilers I have normalize the execution time to the execution time of the baseline for each configuration. Since GCC and Gprof uses another baseline than Clang, Clang instrumentation and Fast Call Graph profiler I have present them in different figures.

Figure 5.1 shows normalized execution times for the GCC and Gprof configurations. The overhead for Gprof varies a lot, from approximately zero for lbm to almost 250% for povray. The mean overhead is close to 45%. Because of the variability it can be hard for programmers to develop an intuition of how much Gprof will slow down the program being profiled. Not knowing how much the program will be slowed down can be discouraging if the program normally takes a long time to run, e.g. if it normally 4 hours to run a program it might take more

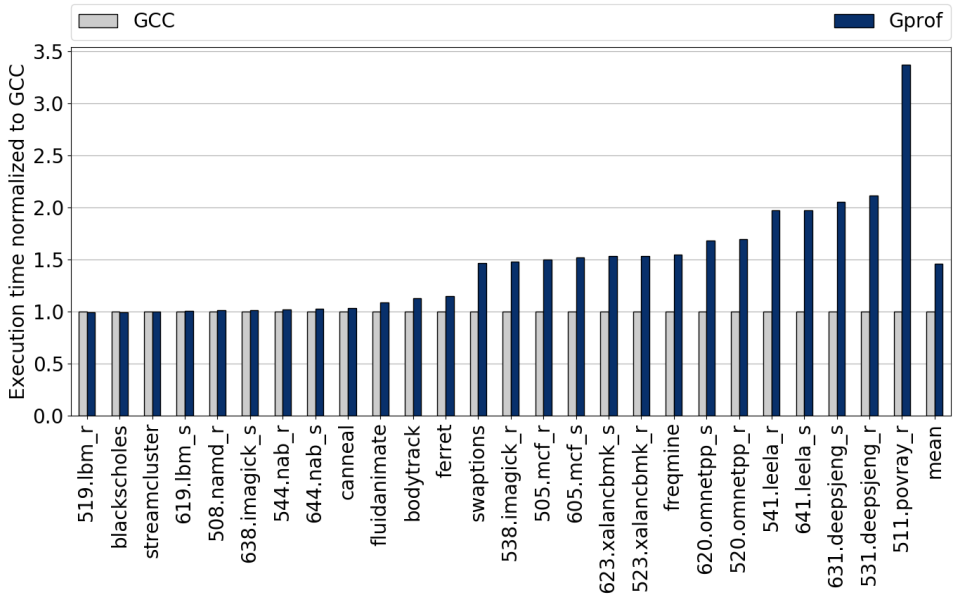


Figure 5.1: Normalized execution time for Gprof profiling. The key takeaway is that the Gprof overhead varies a lot and in multiple cases causes a slowdown of more than 2x.

than 8 hours to run it with Gprof.

In Figure 5.2 the normalized execution time of the LLVM based configurations are plotted. The overhead caused by the two LLVM based profilers are well below Gprof, both the average and maximum overhead. The figure shows that the instrumentation done by Clang is much more costly than only counting function calls as the Fast Call Graph profiler does. The difference is most likely caused by the fact that Clang instrumentation counts how many times each basic block is executed and a program consists of more basic blocks than function calls. Fast Call Graph profiler has a mean overhead of approximately 2% and Clang instrumentation mean overhead is close to 20%.

The maximum overhead for the Fast Call Graph profiler is on `511.povray_r`, with 11% overhead. Gprof also had the highest overhead for this benchmark. This could indicate that the benchmark has many calls, since the overhead of the Fast Call Graph profiler is related to the number of function calls. This will be discussed in a later section.

There are several interesting benchmarks in the figure. Both `519.lbm_r` and `ferret` is sped up when instrumented, `519.lbm_r` by the Clang instrumentation and `ferret` by the Fast Call Graph profiler.

The `519.lbm_r` Clang instrumented binary runs about 3% faster than the baseline Clang binary. To investigate this I ran the tool `perf stat` on both binaries and saw that the instrumented binary executes a lot fewer branches than the baseline, about 20% fewer. The baseline does also execute more instructions than the in-

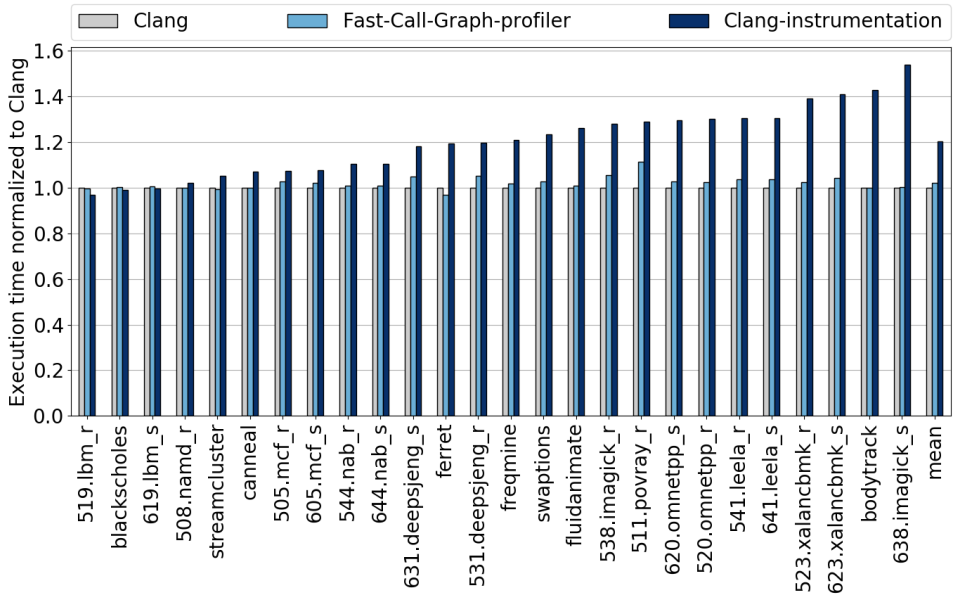


Figure 5.2: Normalized execution time for the Fast Call Graph profiler and the Clang instrumentation. The key takeaway is that the Fast Call Graph profiler has lower overhead than Clang instrumentation in almost all benchmarks and a significantly lower mean overhead.

strumented binary, but the difference in number of instructions executed is much smaller than the difference in branches, the difference in instructions executed is ca. 1.5%. I do not know the reason for this difference, but I looked at the assembly of each of the binaries and saw that the hot loop in the benchmark seemed to be structured differently in the two binaries, which could explain the fewer branches. Why Clang/LLVM chooses to structure or optimize the hot loop differently when profiling is enabled is not clear to me, but it could be that the extra instrumentation instructions makes the code cross some optimization thresholds which applies optimizations that proves to improve performance.

In the `ferret` benchmark the binary instrumented by the Fast Call Graph profiler executes about 3% faster than the Clang baseline. This speedup happens despite the fact that `perf stat` reports that the instrumented binary executes on the order of 10^9 more instructions than the baseline. The instrumented binary also executes more branches than the baseline but at the same time the number of branch misses are lower than the baseline. Less branch misses could explain the speedup since a branch miss is normally costly because the instruction pipeline must be flushed.

The graph in Figure 5.3 shows the number of function calls per second. When comparing it to the Fast Call Graph profiler overheads there seems to be a correlation between the number of calls and the overhead. This is what I expected to be the case, and the graph reinforces that expectation.

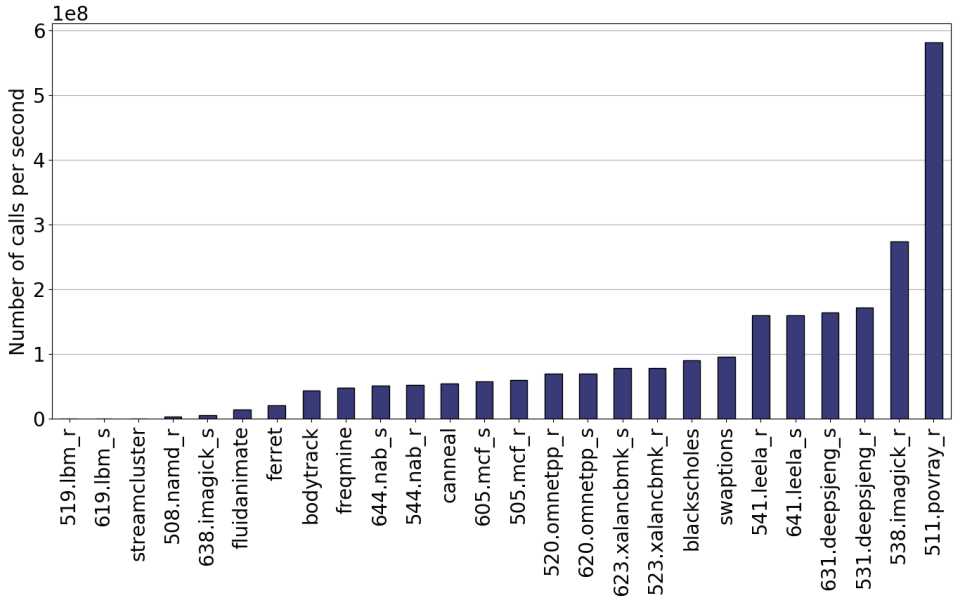


Figure 5.3: Number of function calls per second for each benchmark. The key takeaway is that the benchmarks differ widely and `511.povray_r` is an outlier.

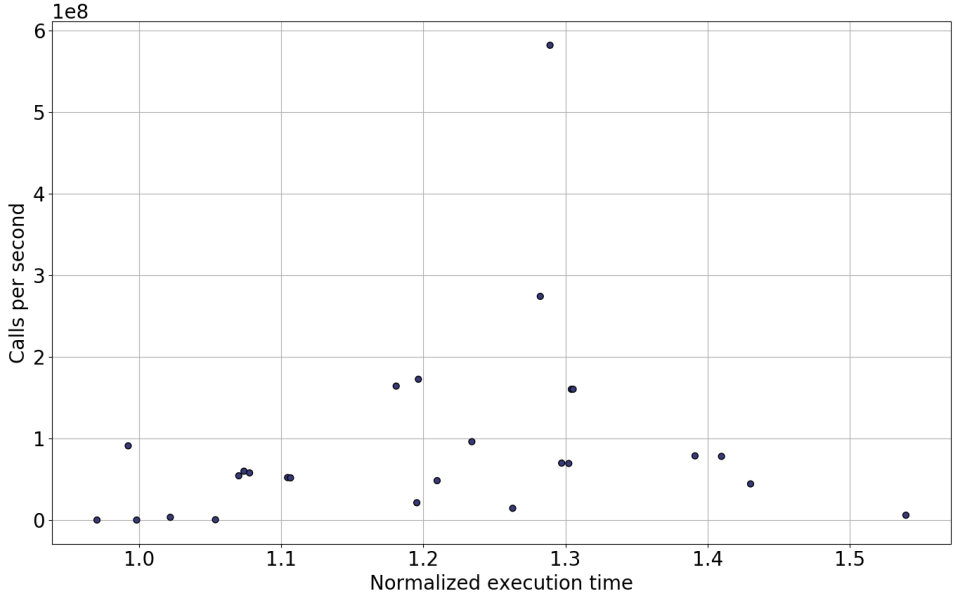


Figure 5.4: Scatter plot to show correlation of normalized execution time and number of function calls per second for the Clang instrumentation. The key takeaway is that there does not seem to be any clear correlation between the variables.

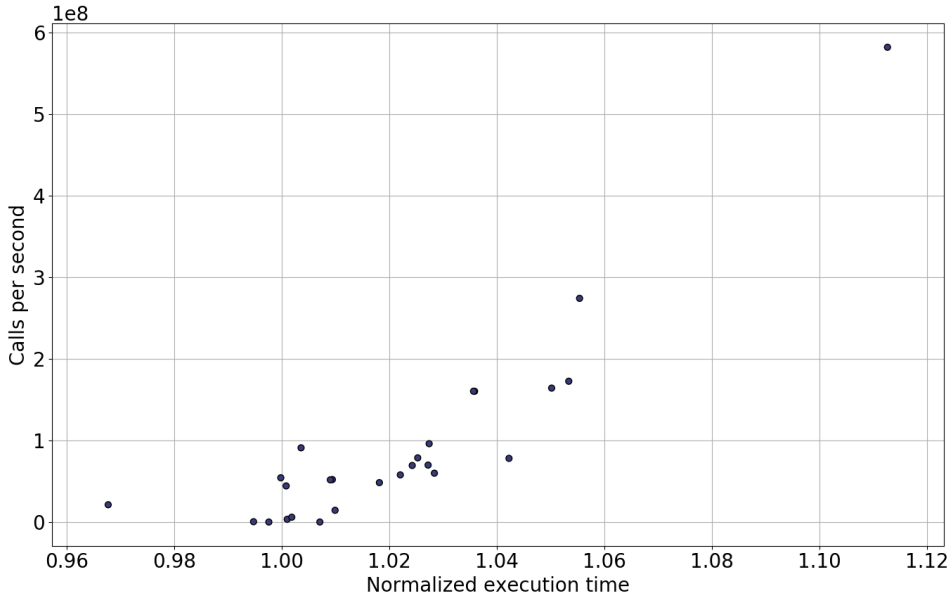


Figure 5.5: Scatter plot to show correlation of normalized execution time and number of function calls per second for the Fast Call Graph profiler. The key takeaway is that there seems to be a positive correlation between the variables.

To further investigate if there is a correlation I have created Figure 5.4 and Figure 5.5 which plots the normalized execution time against the number of calls per second for Clang instrumentation and the Fast Call Graph profiler. For the Clang instrumentation it is hard to see any correlation between the variables, so there must be additional variables that explains the overhead. But the Fast Call Graph profiler shows a positive correlation between the overhead and the number of calls per second and suggests that the number of calls per second can be used to predict the overhead of using the Fast Call Graph profiler on a program.

5.2 Time distribution impact

Figure 5.6 shows the amount of time used in each of the top 5 GCC functions for GCC and Gprof when profiled with perf. The main thing to take note of is that the percentages differ widely between the configurations. The reason that the Gprof functions makes up a smaller share of the execution time is that the profiling functions takes part of the execution time.

To try to compensate for the Gprof profiling functions I have created Figure 5.7. The GCC data is the same as in the previous figure. But the Gprof data is the perf data from running Gprof with the Gprof functions removed and the resulting percentages normalized to 100%. By doing this GCC and Gprof agree well on the time percentages. I also included Gprof-Gprof which is the time percentages

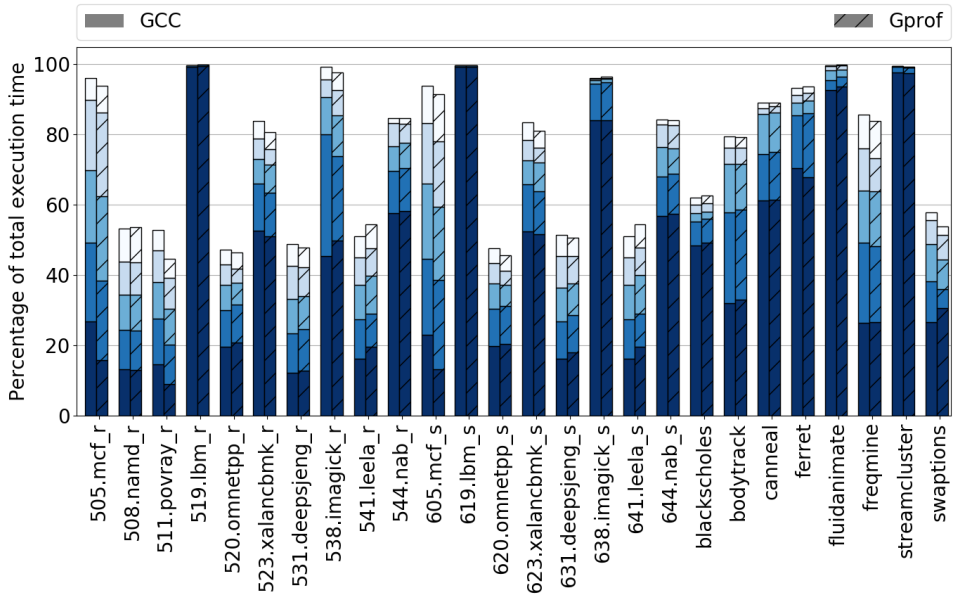


Figure 5.6: The percentage of execution time for the top 5 functions as reported by perf. The key takeaway is that the `mcf`, `deepsjeng` and `leela` benchmarks has a large difference in total execution time for the plotted functions.

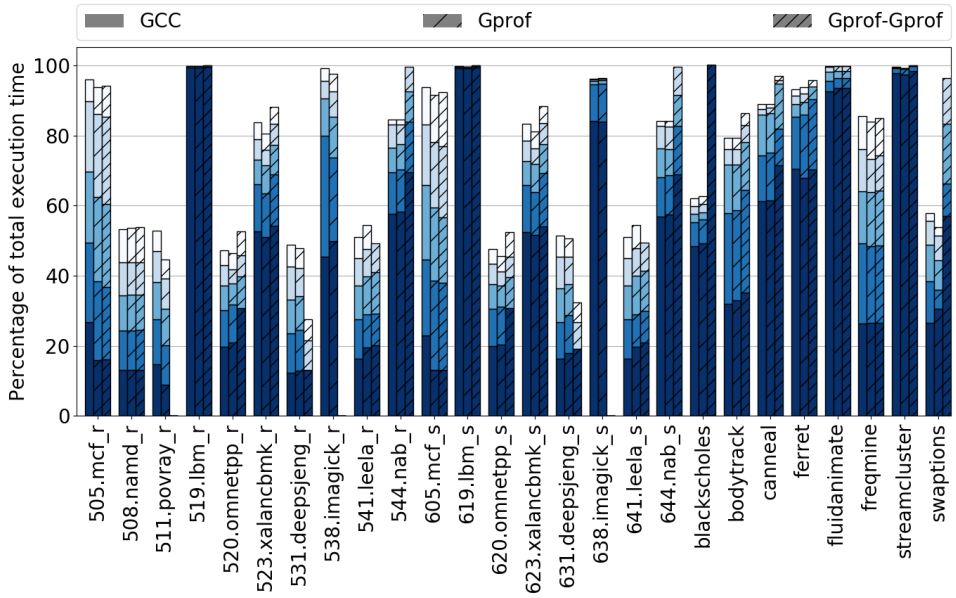


Figure 5.7: The percentage of execution time for the top 5 functions as reported by perf (for GCC and Gprof) and by Gprof (for Gprof-Gprof). The key takeaway is that GCC and Gprof are similar while Gprof-Gprof has some outliers.

reported by Gprof and not perf. Because of the overwriting problem mentioned in the last Chapter I do not have Gprof data for `511.povray_r` and the two `imagick` benchmarks. Gprof-Gprof does not agree that well with GCC and I think the reason for that is that Gprof does not count the time spent in library functions. The key takeaway is that using perf and removing the profiling functions from the Gprof samples gives a fairly accurate representation of the true time distribution.

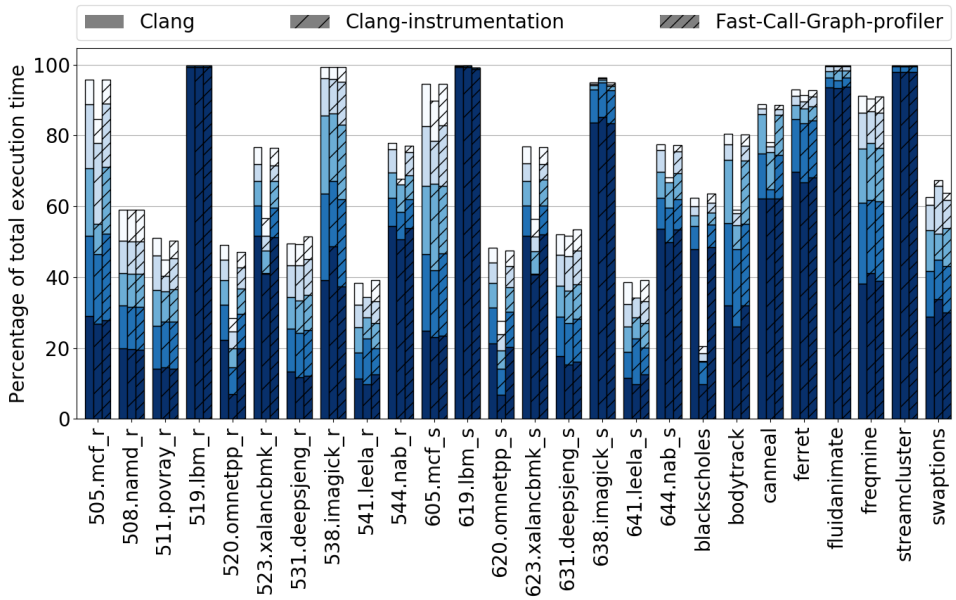


Figure 5.8: Comparison of the percentage of execution time for the top 5 functions as reported by perf for Clang-instrumenter and the Fast Call Graph profiler. The key takeaway is that the Fast Call Graph profiler agrees with the Clang distribution.

In Figure 5.8 I have plotted the top 5 functions for the Clang based configurations. Fast Call Graph profile is similar to the Clang baseline but Clang-instrumentation has several benchmarks where there are differences. I investigated the differences and found that they seemed to be caused by different inlining strategies in Clang and Clang-instrument.

To make the comparison more fair for Clang-instrument I have created Figure 5.9. In it I have removed all benchmarks where there are significant differences in how functions are inlined. I did this by finding the functions in each configuration that accounted for more than 5% of the execution time. Then I compared the set of functions from each configuration and removed the benchmarks where the sets did not contain the same functions. The key takeaway is that the function time distribution is fairly similar to Clang for both profilers and they can therefore be used together with perf to profile function time distribution and function call counts.

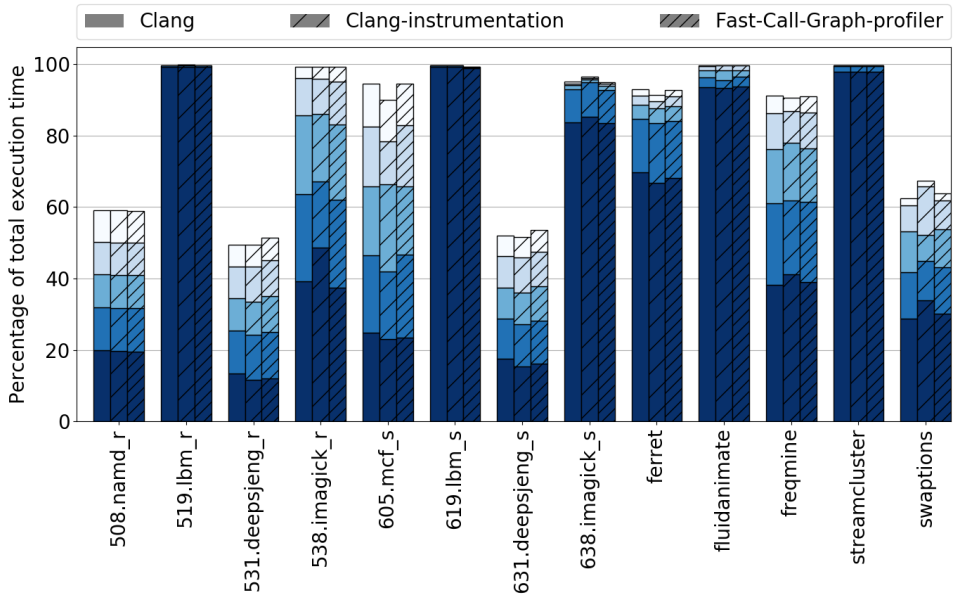


Figure 5.9: Comparison of the percentage of execution time for the top 5 functions as reported by perf for Clang-instrument and the Fast Call Graph profiler. Benchmarks where the configurations differ in inlined functions (of the functions accounting for > 5% execution time) have been removed. The key takeaway is that both Clang-instrument and the Fast Call Graph profiler changes the distribution little enough to make it usable for selecting functions to accelerate.

5.3 Accuracy of function call counts

In the `blackscholes` benchmark I discovered something about the function call counting done by the Fast Call Graph profiler and the Clang instrumentation. In the hot function, the function where the program spends the most time, there is a call to the `sqrt` function in the standard C library. This function call is optimized to use the built in x86 sqrt instruction by both Clang/LLVM and GCC when lowering/translating the program to machine code. Since this optimization happens at the machine code generation stage, and not as part of inlining, both the Fast Call Graph profiler and the Clang instrumentation counts the number of times the `sqrt` function is executed, even though it is only an instruction and there is no function call. The `sqrt` instruction were always used by the compilers when I tested, regardless of which compiler flags I used. This observations suggests that there could be some cases the function called is not really a function call. I assume that this function call to instruction mapping is only done for basic arithmetic functions and that it will not pose a problem for normal function calls.

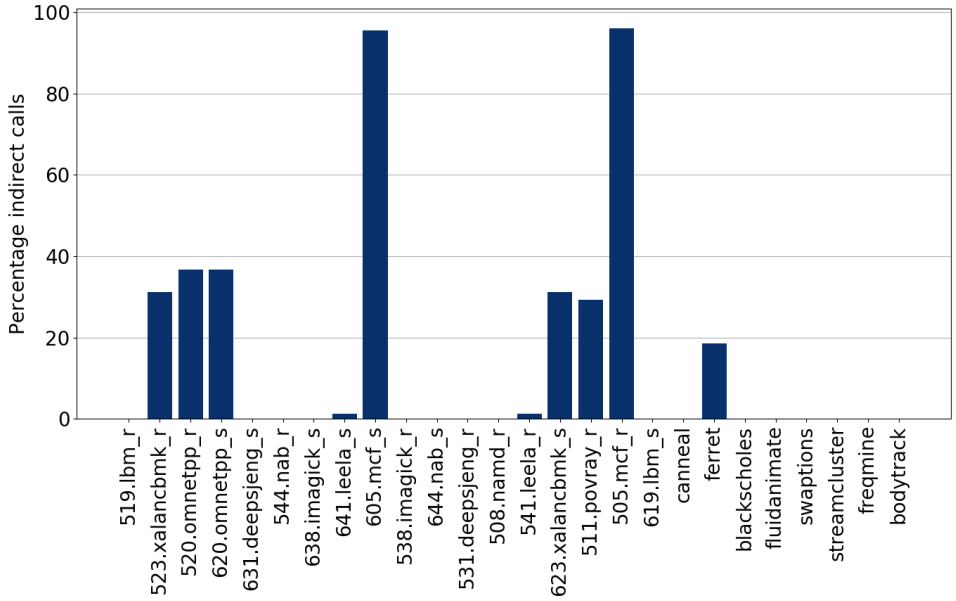


Figure 5.10: Percentage of indirect function calls in each benchmark. The key takeaway is that the `mcf` benchmarks has almost only indirect calls and that the majority of the benchmarks has little or no indirect function calls.

5.4 Indirect calls

Since the profiler does not store the target of an indirect call, it is not possible to attribute indirect calls to the correct caller-callee function pair. Figure 5.10 shows that this is mostly a problem for the mcf benchmarks where more than 90% of the function calls are indirect calls. For the majority of the benchmarks it is not a problem since they do not use indirect calls.

As explained in the Profiler chapter it is possible to add support for correctly counting indirect calls in my profiler at the cost of additional overhead when profiling indirect calls. In this thesis low overhead were the top priority and indirect calls were not prioritized. And as the figure shows indirect calls is not widely used in most of the benchmarks so it is not a major problem that the profiler does not support indirect calls.

Chapter 6

Conclusion

6.1 Conclusion

The key takeaway is that small amounts of call recording instrumentation does not change the function time distribution in any significant way, so it is possible to run both instrumentation and sampling at the same time and obtain data that can be used when selecting which function to accelerate when using an accelerator such as an FPGA.

My main contribution is C1, the Fast Call Grap profiler. This profiler instruments LLVM bitcode to count the number of times each caller-callee pair in the program is executed to generate a dynamic call graph. This contribution fulfills task T1 and T2.

I have also demonstrated that the Fast Call Graph profiler can be used together with the perf profiler to get both function time distribtion information and dynamic call graph information from a single program run. This demonstration is contribution C2 and it partly fulfills task T3. Contribution C3 fulfills the rest of task T3, it is that I have evaluated how the Fast Call Graph profiler interacts with the perf profile and found that there is little difference between running with and without Fast Call Graph instrumentation, so it is reasonable to run both profilers at the same time.

The last contribution, C4 as a fulfillment of task T4, is that I have shown that the Fast Call Graph profiler has a much lower average overhead compared to the Gprof and Clang instrumentation profilers.

6.2 Further work

I have identified multiple things that can be done as further work following this thesis.

The first thing is to add support for indirect calls to the Fast Call Graph profiler and measure it there is any additional overhead. Adding indirect call support will make the profiler more accurate at the cost of a likely increase in overhead. One

way of adding support for indirect calls is to keep the direct call instrumentation as it is and then write new instrumentation for indirect calls that are only inserted at indirect call sites. By implementing it this way the indirect call support will not add any additional overhead to direct calls but only add additional overhead to indirect calls.As Figure 5.10 shows the usage of indirect calls is varied in the benchmarks I ran and are not used in the majority of them.

The next thing that can be done is to extend NEEDLE so that it can read profile information from perf and Fast Call Graph profiler to automatically select which function to accelerate. Doing this can make NEEDLE and automatic FPGA acceleration more approachable and easier to use for developers.

The third thing that can be done is to make the Fast Call Graph profiler into a Clang pass so that WLLVM is not needed. A Clang pass is code that runs as part of the optimization process when compiling programs. Having the profiler as a Clang pass will make it easier to use since it will allow you to compile your programs normally and not use WLLVM. As already mentioned I did encounter a problem with the `vips` benchmark where it would not compile since the compilation and linker options were not correct when the profiler tool tried to generate the binary. That problem would most likely have been solved by having the profiler as a Clang pass.

Bibliography

[1] S. Borkar and A. A. Chien, 'The future of microprocessors', *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, 'Dark silicon and the end of multicore scaling', in *2011 38th Annual international symposium on computer architecture (ISCA)*, IEEE, 2011, pp. 365–376.

[3] Intel. (2012). Intel Advanced Encryption Standard Instructions (AES-NI), [Online]. Available: `https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html`.

[4] (2020). List of Nvidia graphics processing units, [Online]. Available: `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`.

[5] S. Huang, S. Xiao and W.-c. Feng, 'On the energy efficiency of graphics processing units for scientific computing', in *2009 IEEE International Symposium on Parallel & Distributed Processing*, IEEE, 2009, pp. 1–8.

[6] Xilinx. (2020). What is an FPGA?, [Online]. Available: `https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html`.

[7] B. De Sutter, P. Raghavan and A. Lambrechts, 'Coarse-grained reconfigurable array architectures', in *Handbook of signal processing systems*, Springer, 2019, pp. 427–472.

[8] A. Sadek, A. Muddukrishna, L. Kalms, A. Djupdal, A. Podlubne, A. Paolillo, D. Goehringer and M. Jahre, 'Supporting utilities for heterogeneous embedded image processing platforms (STHEM): An overview', in *International Symposium on Applied Reconfigurable Computing*, Springer, 2018, pp. 737–749.

[9] N. Paulino, J. C. Ferreira, J. Bispo and J. M. Cardoso, 'Transparent acceleration of program execution using reconfigurable hardware', in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2015, pp. 1066–1071.

[10] S. Kumar, W. Sumner, V. Srinivasan, S. Margerm and A. Shriraman, 'NEEDLE: Leveraging Program Analysis to extract Accelerators from Whole Programs', in *High Performance Computer Architecture (HPCA2017), 2017 IEEE 23rd International Symposium on*, IEEE, 2017.

[11] C. Lattner and V. Adve, 'LLVM: A compilation framework for lifelong program analysis & transformation', in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 75.

[12] A. C. De Melo, 'The new linux'perf'tools', in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[13] S. L. Graham, P. B. Kessler and M. K. Mckusick, 'Gprof: A call graph execution profiler', *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[14] R. M. Stallman, 'Gnu compiler collection internals', *Free Software Foundation*, 2002.

[15] (). SPEC CPU 2017, [Online]. Available: `http://spec.org/cpu2017/`.

[16] C. Bienia, S. Kumar, J. P. Singh and K. Li, 'The PARSEC benchmark suite: Characterization and architectural implications', in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[17] C. Bienia, 'Benchmarking Modern Multiprocessors', PhD thesis, Princeton University, Jan. 2011.

[18] M. Tancreti, M. S. Hossain, S. Bagchi and V. Raghunathan, 'Aveksha: A Hardware-Software Approach for Non-Intrusive Tracing and Profiling of Wireless Embedded Systems', in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '11, Seattle, Washington: Association for Computing Machinery, 2011, pp. 288–301, ISBN: 9781450307185. DOI: `10.1145/2070942.2070972`. [Online]. Available: `https://doi.org/10.1145/2070942.2070972`.

[19] Google. (2020). Gperftools, [Online]. Available: `https://github.com/gperftools/gperftools`.

[20] (). Ntnu. 2020. pperf. https://github.com/eecs-ntnu/pperf.

[21] J. Levon and P. Elie, *Oprofile: A system profiler for linux*, 2004.

[22] N. Nethercote and J. Seward, 'Valgrind: a framework for heavyweight dynamic binary instrumentation', *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, 'Pin: building customized program analysis tools with dynamic instrumentation', *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[24] C. Curtsinger and E. D. Berger, 'Coz: Finding Code That Counts with Causal Profiling', in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, Monterey, California: Association for Computing Machinery, 2015, pp. 184–197, ISBN: 9781450338349. DOI: 10.1145/2815400.2815409. [Online]. Available: https://doi.org/10.1145/2815400.2815409.

[25] S. P. Reiss, 'Controlled dynamic performance analysis', in *Proceedings of the 7th international workshop on Software and performance*, 2008, pp. 43–54.

[26] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I.-T. A. Lee and C. E. Leiserson, 'The CSI framework for compiler-inserted program instrumentation', *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 2, pp. 1–25, 2017.

[27] S. Kumar. (2017). NEEDLE source code, [Online]. Available: https://github.com/sfu-arch/needle.

[28] N. Sumner. (2020). LLVM demo code, [Online]. Available: https://github.com/nsumner/llvm-demo.

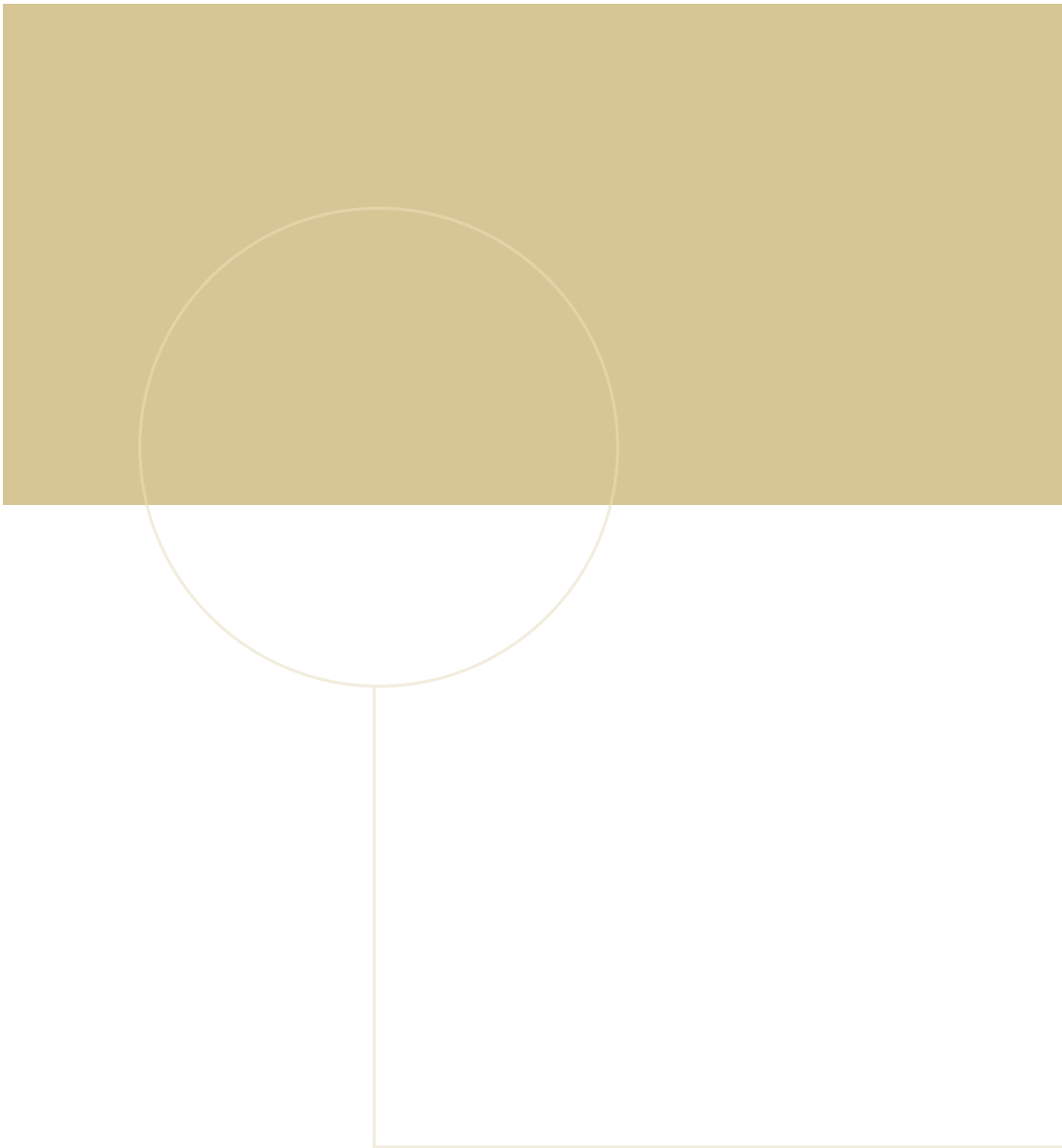
[29] (2020). Whole Program LLVM, [Online]. Available: https://github.com/travitch/whole-program-llvm.

[30] A. Fog, '4. Instruction tables', p. 240, 2019. [Online]. Available: https://agner.org/optimize/instruction_tables.pdf.

[31] Intel. (2020). Intel 6700K, [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/88195/intel-core-i7-6700k-processor-8m-cache-up-to-4-20-ghz.html.

[32] Canonical. (2018). Ubuntu 18.04, [Online]. Available: https://releases.ubuntu.com/18.04/.

[33] C. Santilli. (2020). PARSEC benchmark source code, [Online]. Available: https://github.com/cirosantilli/parsec-benchmark.



NTNU

Norwegian University of
Science and Technology