Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Master Degree In Computer Science and Engineering

POLITECNICO
MILANO 1863

A Framework for Estimation and Visualization of Software Energy Consumption

Advisor: Prof. Giovanni Agosta
Co-Advisor: Daniele Cattaneo
Co-Advisor: Andrea Gussoni

Thesis by:
Pietro Ghiglio Matr. 920491

Academic Year 2019–2020

Ringraziamenti

Vorrei ringraziare tutti coloro che mi hanno sostenuto lungo il mio percorso di studi. Un ringraziamento particolare va al Prof. Giovanni Agosta e ai dottorandi Daniele Cattaneo e Andrea Gussoni, per il loro supporto e per l'avermi trasmesso interesse e passione per questa materia.

Ringrazio poi la mia famiglia per la pazienza e l'incondizionato sostegno che mi ha dato, e tutti gli amici e i compagni di corso che mi hanno accompagnato in questi anni.

Un ringraziamento speciale va alla mia ragazza, Claudia, presenza fondamentale durante i mesi di lavoro su questa tesi.

Abstract

This work provides a framework for the estimation of software energy consumption, targeting embedded systems. Energy estimation is a challenging task which many others have attempted in the past. We improve such previous attempts in order to provide the developer with more granular information about the energy hot-spots in the application.

The tool is built using the LLVM framework, and is therefore agnostic to both the high level language and the target architecture. We also provide the possibility to explore and visualize the impact of compiler optimizations on the source code to low level instructions correspondence.

We have designed an energy model that can be constructed with readily available information, and incrementally enriched in order to improve its accuracy. Finally, we evaluate the accuracy of this model on widely used benchmarks.

Sommario

Questo lavoro presenta un framework per la stima del consumo energetico di un programma, con riferimento particolare ai sistemi embedded. Questo problema é stato largamente affrontato in passato. Estendendo metodologie di stima giá note, il framework consente a sviluppatori software di stabilire quali entitá a livello di codice sorgente siano maggiormente responsabili per il consumo energetico totale del programma.

Il tool é stato sviluppato usando il framework LLVM, e di conseguenza è indipendente sia dal linguaggio di programmazione utilizzato, sia dall'architettura hardware sul quale il programma viene eseguito. Esso inoltre fornisce anche la possibilitá di visualizzare e stabilire l'impatto delle ottimizzationi del compilatore sulla corrispondenza tra codice sorgente e istruzioni di basso livello.

Abbiamo poi sviluppato un energy model i cui parametri possono essere determinati con informazioni prontamente disponibili, ed eventualmente arricchito per aumentare la sua accuratezza. Infine, abbiamo svolto una validazione sperimentale della stima del consumo energetico.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Why Measure Software Energy Consumption? Between the metrics that can be used to assess the performances of software, energy consumption is not as employed as others, such as time and memory. This is not surprising, considering that this two resources are often constraining factors, and are also the most intuitive, non functional, ways to quantify how well (or how poorly) the program is running. Nevertheless, the question "how much energy it consumes?" is being asked more and more, along with "how much time does it take?" and "how much memory do I need?".

The energy consumed by the ICT (Information and Communication Technologies) sector is non-negligible: in 2015 it amounted to 800 TWh, or 3.6 % of the global energy consumption. If ICT were a state on his own, it would rank 6^{th} in terms of energy consumption, just after Japan, 5^{th} in the global ranking with 918 TWh [16].

Tied to the energy consumption, also the carbon footprint of software should be taken into account: in [31], Strubell and Ganesh show that training a deep learning based Natural Language Processing model can emit as much carbon dioxide as the one emitted by a car in its average lifetime.

The ICT energy consumption is not only already non-negligible, but it is also projected to grow: in [1], Anders outlines several possible future scenarios, showing that, unless we take an active stance in trying to reduce and optimize energy consumption, the ICT sector may take up to 51 % of the total global energy consumption in 2030. We couldn't find any data regarding the total energy consumption as of 2020, but a report from Cambridge University [7] estimates the energy consumption from bitcoin mining alone to be around the 130 TWh for one year, which is by itself comparable to a country like Argentina.

Besides the global-scale environmental concerns, measuring software energy consumption is becoming increasingly important as the Internet of Things (IoT) infrastructure enlarges, leading to the deployment of billions of deeply embedded devices that need to operate on limited or unreliable sources of energy, such as batteries or energy harvesting [12]. Finally, the fact that the energy consumed by ICT devices may represent up to 50% of energy cost of an organization [26] constitutes a good economical reason for measuring and understanding software energy consumption.

Software Development and Energy Consumption The energy consumption problem in ICT has traditionally been tackled more from the "hardware" point of view. Recent publications, such as [8], advocate also for a "software" perspective on the matter, where developers actively try to assess and optimize the energy consumption of their programs.

In a survey from 2016, Manotas et al. [23] interviewed several software engineers from companies such as Microsoft, Google and ABB, asking if they were concerned with energy consumption. They concluded that software engineering practitioners care and think about energy when they build applications; however, they are not as successful as they could be because they lack the necessary information and support infrastructure.

Our Contribution The purpose of this work is, therefore, to provide a prototype of a tool that may be useful to software developers concerned with the energy consumption of their applications.

This documents is structured as follows:

- Chapter 1 provides a technical background, regarding the compilation pipeline and LLVM.
- Chapter 2 provides an overview of the state of the art regarding energy consumption estimation.
- Chapter 3 describes the tool in all its components, and the energy estimation technique employed.
- Chapter 4 focuses on the analysis and visualization of the effects of compiler optimizations.
- Chapter 5 describes the experimental evaluation that we have conducted.

Chapter 1

Background

This chapter will provide an overview of some technical items, useful to understand the rest of the document. Our work can be classified in the general scope of *program analysis*. We will, therefore, introduce LLVM, a widely employed framework for compiler construction and program analysis. In addition, since in this work we rely heavily on its infrastructure, we will present debug information, its general purpose, and the most common format in which it is stored, the DWARF format.

1.1 LLVM

The LLVM Project [33] is a collection of modular and reusable compiler toolchain technologies. It is built around an intermediate representation called LLVM-IR, and provides a set of APIs to interact with it. LLVM provides an optimizer that works on the intermediate representation, and also several code generation helpers that allow to target all the main hardware architectures.

1.1.1 LLVM-IR

The LLVM-IR is a language that resembles a generic assembly language, while also providing some high level features such as unlimited registers, explicit stack memory allocation and pointer deferentation. This allows LLVM-IR to be both the ideal target for high-level language developers, that do not have to worry about architecture specific details, and also the ideal source language for compiler back-end developers, that have to implement only a translator from LLVM-IR to their target architecture's assembly language, without concerning about high-level language features.

The LLVM-IR is accessible in three formats: in-memory representation, that allows manipulation through the LLVM APIs, binary format, used by many LLVM tools, and the human-readable textual format, that can also very conveniently be parsed by means of the APIs.

1.1.2 SSA, Basic Blocks and Phi nodes

The LLVM-IR is by definition in SSA (Static Single Assignment) form. The SSA form requires a variable to be assigned only once, and requires every variable to be defined before it is used.

Basic blocks are the fundamental units of which functions are made of. A basic block is a sequence of instructions with no branch, jump or returns between them. Intuitively, when the first instruction in a basic block is executed, all the others are executed as well. The blocks that are possibly executed after a block are called its *successors*, the blocks that were possibly executed before are called *predecessors*.

Since a basic block may have multiple predecessors, and each predecessor may contain a definition of of a variable, PHI-Nodes are introduced as a mean to disambiguate which of the definitions to use. Consider the following snippet of C code:

```
1  int example(int n){
2    int x;
3    if(n > 0)
4      x = n+1;
5    else
6      x = n-1;
7    return x;
8  }
```

It is translated in the following LLVM-IR segment:

```
1  define dso_local i32 @example(i32 %0) {
2    %2 = icmp sgt i32 %0, 0, !ID !1
3    br i1 %2, label %3, label %5, !ID !2
4
5  3:                                                ; preds = %1
6    %4 = add nsw i32 %0, 1, !ID !3
7    br label %7, !ID !4
8
9  5:                                                ; preds = %1
10   %6 = sub nsw i32 %0, 1, !ID !5
11   br label %7, !ID !6
12
13 7:                                                ; preds = %5, %3
14   %.0 = phi i32 [ %4, %3 ], [ %6, %5 ], !ID !7
15   ret i32 %.0, !ID !8
16 }
```

We see that:

1. The resulting code is in SSA form. For example, this can be seen from the fact that each assignment to variable x in the source code corresponds to a new definition in the LLVM code.

2. A phi-node is added in order to disambiguate between the assignment in the *if* basic block and the one in the *else* basic block.

1.1.3 Class Hierarchy

The class hierarchy defined in the LLVM APIs consists of hundreds of classes, a complete and exhaustive view is given by the LLVM Documentation [1]. The main components of the hierarchy are:

- Module: the entire program/compile unit. Contains the global values of the program (mainly the global variables and the functions) and other information needed for the compilation.
- Function: a function in the compile unit, contains mainly a set of arguments and its control flow graph in the form of a set of basic blocks.
- Basic Block: a set of instructions with no branches between them.
- Instruction: An instruction of the IR.

Another key class in the LLVM class hierarchy is the Value class. It represents anything that has a type and can be used as an operand to an instruction: function arguments, constants, instructions, basic blocks and functions are all Values. A Value also carries information of what other Values it uses, and what other Values use it.

1.1.4 LLVM Metadata

The LLVM-IR allows metadata to be attached to Instructions, Functions, Global Variables or Modules. Metadata can convey extra information about the code to the optimizers and code generator. The main use of metadata is debug information, but they may also carry information about loop boundaries or other assumption that are useful during the various stages of the compilation process.

Metadata can either be a simple string attached to an instruction, or they can be a Metadata Node (MDNode). MDNodes can reference each other and are specified by other classes in the LLVM APIs. See section 1.2 or the LLVM Language Reference [22] for more details.

1.1.5 LLVM Passes

LLVM passes are where most of the interesting parts of the compilation process take place. Passes perform the transformations and optimizations that make up the

[1] LLVM Doxygen documentation

compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code.

Passes are categorized in two ways: by the granularity at which they operate, and by the fact that they perform changes on the module or not.
By the first categorization, passes are identified as:

- Module Passes: operate on an entire Module.
- Function Passes: operate on a single Function.
- Loop Passes: operate only on loops.

By the second categorization, passes are identified as:

- Analysis Passes: passes that only perform an analysis of the given entity, without modifying it.
- Transformation Passes: passes that may modify the given entity. They exploit the results of the Analysis Passes, often (but not only) in order to perform optimizations: they may add, remove, move or replace instructions and basic blocks, with the ultimate goal of improving performances or reduce the size of the binary.

Passes may depend on other passes, for instance a pass that performs an optimization may require the results of a pass that performs a specific analysis. They are therefore handled by a Pass Manager that schedules the passes, ensuring that all the dependencies for a pass are met before executing it.

1.2 Debug Information in LLVM-IR

The LLVM class hierarchy associated to debug information resembles the structure of the DWARF standard, described in 1.4.2.

This hierarchy is represented via specific LLVM metadata objects.

1.2.1 Metadata Classes

An LLVM Instruction may correspond to zero or one DILocation. Each DILocation contains information about the Line, Column and Scope of the source location that corresponds the given Instruction.

The scope of the location is represented by a generic metadata node, it usually corresponds to either a lexical block or a function. Functions have metadata associated to them, represented by the DISubProgram class, which contains mainly the Line and Column and File where the function is defined. This structure allows to

map an LLVM Instruction to a location in the source file, uniquely identified by the triple $\langle Line, Column, File \rangle$.

Like the DWARF standard, LLVM contains other metadata classes, such as the ones needed to represent data types, which are less useful for our purposes and therefore not discussed here.

1.2.2 Transformation Passes Guidelines

As we have seen in section 1.1.5, during the compilation a module may undergo some changes: instructions may be removed, moved, merged together, and replaced with new instructions, all in order to improve the performances of the resulting program.

This transformations have the side effect of obfuscating the correspondence between source code and binary code: before the optimization occurs, debug information provide a very clear, one to many relation between source location and LLVM-IR instructions. But as the module progresses into the optimization pipeline, it becomes more and more difficult to maintain this relation.

In general it is not possible to unambiguously map source locations to optimized code, but the LLVM project provides a set of guidelines that specify how to correctly update debug info when implementing transformation passes [14].
Here we provide a short summary of such guidelines [2], highlighting some behaviors that, even when following them, lead to a loss of information regarding source-binary mapping. This behaviors are not bug or mistakes of the people who provided the guidelines, but are instead related to the fact that they want to provide a debugging experience as close as possible to the one that a user would have while debugging the unoptimized code.

The guiding principles for a developer that wants to update debug info are the following:

1. Do not provide misleading information: a developer should not speculate, and providing no information is better than providing wrong information that may lead a developer to wrong considerations about the behavior of his program.

2. Provide as much information as possible: when it is not misleading, information should be preserved.

In order to achieve this, when choosing what do to with the debug information of a given instruction, a developer has three alternatives:

- Preserve the original location.

[2] Provided at a speech at the 2020 LLVM Conference by Adrian Pranti and Vedant Kumar

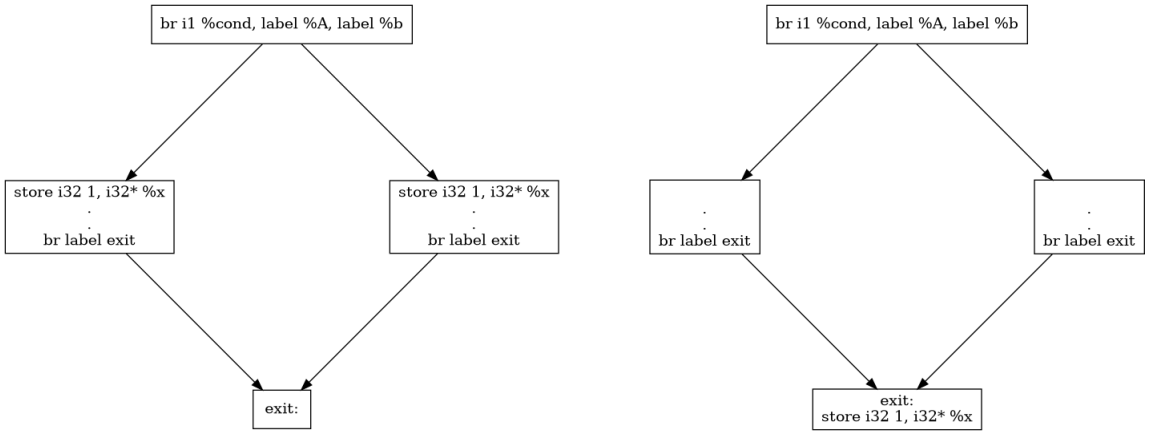

Figure 1.1: Example of optimization with merged debug locations.

- Merge two locations: two debug locations can be merged together. Locations merge is performed by computing the intersection of the two locations: the resulting location will contain only the information that the original two had in common.

- Delete the location.

Locations can be safely preserved when the modified instruction either remains in the same basic block, or its basic block is folded into a predecessor that branches unconditionally. For instance, an optimization the replaces the instruction `add x x` with a binary shift to the left , `shl x 1`, can safely keep the location of the original add.

Location should be merged when two instructions are replaced with a new instruction. An example of that is shown in Figure 1.1, in which the two stores can be merged into a new one, inserted in the exit basic block. The new instruction effectively replaces the original two, and therefore its location will be the merged location of the old ones.

In all cases where the previous rules do not apply, locations should be dropped. In particular, they should be dropped whenever an instruction is moved from a basic block with multiple predecessors, to one of the predecessors. This is done to avoid situations in which, while debugging, the program seems to have taken a branch in a conditional, while the actual conditions are not the one that would have resulted in the branch being taken.

Dropping locations and merging locations is a very reasonable course of action when dealing with debugging: they lead to a debugging experience that is as close as possible to debugging the unoptimized version. But they also lead to a loss of information in the source-binary mapping: when two locations are merged, we will most likely lose the information on the original source code lines, as they probably will

not be equal, and when a location is dropped we will of course lose the information it carried.

We have therefore developed a methodology that allows to propagate debug locations through the optimization pipeline, while also bringing to the developers a view of the optimizations performed on their program, so that they can understand how it has been optimized by the compiler.

1.3 Program Instrumentation

Instrumenting a program means to inject additional code that was not originally in the program's source, typically in order to produce additional information (regarding some functional or non functional properties) during the program's runtime. It can be performed directly on the source code, on the executable binary, or during the compilation. Examples of widely employed instrumentations are the many sanitizers that are part of the LLVM project, used to make runtime checks about memory and thread safety.

LLVM provides some helper classes to perform instrumentation on an IR Module, and in general a user may define his own transformation pass that inserts new code into the program being compiled.

Instrumentation often introduces a performance overhead, due of course to the fact that additional tasks are executed while the program is running, so it is usually performed only during the development stage of an application.

1.4 Debugging

A debugger is a computer program used to test and debug other programs. It allows a programmer to run the target program in controlled conditions, pause the program's execution, check the state of variables and more.

1.4.1 Debug Information

The main feature of a debugger, over which more advanced features can be built, are setting break points and accessing the content of a variable defined in the source code.
This is achieved by means of debug information: information stored by the compiler in the program's executable, with the purpose of providing a correspondence between source level entities (variable, source code locations, data types) and low level entities (assembly instructions and memory locations).

The format used to store them may vary with the compiler/operating system used, but the stored information is mainly:

- Definition of the data types employed in the program and their layout in memory, both language-defined (eg. int, float, unsigned in C) or user defined (eg. C structs or C++ classes).
- Mapping between variables defined in the source code and memory locations in which they are stored. This allows a debugger to output the value of a variable given its name.
- Mapping between source code locations and instruction's addresses in the executable. This allows the debugger to pause the program's execution when a given source code location is reached.

This information is useful not only for debugging purposes, but it may also be employed by any other tool that requires a mapping between source code and binary executable, such as a profiler or a test coverage tool.

1.4.2 DWARF Format

The DWARF format [32] is a debugging file format used by many compilers and debuggers to support source-level debugging. It is designed to be extensible with respect of the source language, and to be architecture and operating system independent.

The main data structure used to store debug information is the DIE (Debug Information Entry). DIEs are used to describe both data types and variables, and can reference each other creating a tree structure.

Another data structure that is very useful for our purposes is the Line Number Table: it contains the mapping between memory addresses of the executable code, and the source line corresponding to those addresses.
Each row of the table contains the following fields:

- Address: the program counter value of a machine instruction.
- Line: the source line number.
- Column: the column number within the line.
- File: an integer that identifies the source file.
- Statement: boolean value indicating if the current instruction is the beginning of a statement.
- Block: boolean indicating if the current instruction is the beginning of a basic block.

Other fields are described in the DWARF documentation.

Chapter 2

State of the Art

2.1 Overview

This chapter will provide an overview of the state-of-the-art methods to measure, estimate and visualize the energy consumption of software.

In general, there is no unique solution to this problem: the proposed techniques vary by both the applicative domain, the properties of the result and the procedure with which the result is obtained.

For the applicative domains, we have identified three main cases:

- Embedded systems: embedded systems are often employed as sensors in contexts where the only source of electricity is their own battery. Therefore, energy consumption has always been a concern of both hardware and software developers.

- Smart phones: similarly to embedded systems, smart phones have to rely on their own battery to operate. The current rate of battery improvement is around 5% a year, but the workloads that smart phones have to withstand increases by an order of magnitude every 5 years [25]. This means energy consumption has to be also tackled from the software prospective.

- Multicore CPU: energy consumption is not a big concern from the point of view of PC users. But it is a primary concern in large datacenters where heat dissipation requires good engineering solutions, and whose impact on global CO2 emission and energy consumption is non-negligible.

In terms of the properties of the result of the measurement/estimate, solutions differ by their granularity: some provide a single quantity (the total amount of energy consumed by a program), other are finely grained, allowing to attribute energy measures to either source code entities or hardware components.

The procedure adopted to obtain the result are widely different, [27] provides an overview of some techniques, and groups them in simulation-based and measurement-based. Simulation based technique require a model of the target architecture, and, provided a segment of binary code, perform a cycle-accurate simulation of the events that occur during the program's run.
Measurement-based techniques, instead, can be further sub-grouped in roughly three categories:

- Direct measurement: the measure can be taken by plugging the device into an instrument that allows to measure the current or power absorbed by the device while running the program.

- Performance-counter based: some hardware architectures provide special registers that store information about the energy consumption, and a set of APIs that allow to read their contents.

- Modeling based: some solutions propose using a mathematical model of the energy behavior of the target architecture. First, the parameters of the model are determined experimentally, then the resulting parameterized model is used in order to obtain an estimate of the consumed energy.

The dimensions that we have indicated are not completely orthogonal. In particular, modeling based approaches usually target embedded devices, since they have simpler underlying architectures that are inherently easier to analyze.

Performance counter based method, instead, are bound to specific architectures that provide such counters, such as Intel's Running Average Power Limit (RAPL, [28]), for their Sandy Bridge architectures, the Intel System Management Controller (SMC[1]) for the Xeon Phi processor, or the NVidia Management Library framework (NVML[2]) that allows to obtain energy consumption of NVidia's GPUs.

2.2 Simulation

Simulation based methods are methods that provide an estimate of the energy consumption by running the assembly code of the program in an architectural simulator. Said simulator must also have been provided with some energy/power model of the target.

The first proposal for such a technique has been published by Tiwari et al. in 2000 [6]. The simulator that they developed, Wattch, is the first simulator to operate at the architectural level: it does not require the full RTL design (usually expressed

[1] Xeon Phi's datasheet
[2] https://developer.nvidia.com/nvidia-management-library-nvml

in the Verilog language), but relies, instead, on a more high-level description of the CPU.

Given such a description – that includes functional units, caches, register files, memories, TLB and other components – Wattch employs a parameterizable power model that, through a cycle-accurate simulation, outputs an estimate of the consumed energy. The simulation is run by interfacing with the SimpleScalar [3] architectural simulator.

Being more high-level than RTL-based simulations, Wattch is faster and does not rely on the Verilog description of the target (usually not disclosed by companies), to the detriment of the accuracy of the result, since it does not model in full detail the entire logic of the target.

Despite being faster than RTL-based simulators, running a binary file with Wattch is about 10000 orders of magnitudes slower [4] than running the actual program, even if the latter has been instrumented.

Since its original publication, the original work on Wattch has been expanded in several ways. The main contribution in that sense has been made by Li et al., who developed a completely new power simulator, McPat [19], offering more modern and advanced features than Wattch.

McPat provides the support to compute power-area integrated metrics (energy-delay-area product), models of static, dynamic, and short-circuit power dissipation (whereas Wattch only modeled dynamic power dissipation), and support for modeling multicore architectures, which have become increasingly widespread. It also provides an XML interface to the simulator, that allows McPat to be ported to different performance simulators.

Validating the correctness of the output of these tools is not an easy task: they provide a very fine grained output (the power/energy estimates for each of the CPU's sub-components), but hardware manufacturers often to not disclose design data with such a level of detail. In [36], the authors, that work for the IBM corporation, have access to such data, and therefore they can provide a more insightful validation of the estimates emitted by McPat. They conclude that, while the procedure employed by McPat to obtain such results is sound, the power models that it exposes are often incomplete, too high-level, or represent an implementation of the structure that differs from the core at hand. The authors provide also some guidelines to improve power modeling accuracy, but ultimately state that academic researches would greatly benefit from the availability of validated power models for contemporary commercial chips, emitted by the hardware producers themselves.

To conclude, simulation based power models are very interesting as they can characterize an hardware architecture with great detail, but their (slow) simulation speed, and some concerns regarding their accuracy, make them not practical for software developers. Such tools are more suited to hardware/compiler developers

that want to characterize the power/energy behavior of a target architecture, not to programmers that want to characterize the energy behavior of software.

2.3 Direct Measuring

Directly measuring the current drawn by a device during the program's execution is the method the provides the greatest accuracy, but it is also the one that has the greatest "overhead" (the amount of time and resources invested in the measuring process) for a developer that wants to assess the energy consumption of his software.

Given its accuracy, it is often used as "ground truth" when evaluating the performances of other methods (simulations, performance counter or modeling).

Experimental setups may differ, depending on the target architecture and the tools at disposal, but they usually consist in a measuring point placed between the device and the power supply. For example in [29], Roth et al. state that their experimental setup consist in a precision current-sense amplifier that amplifies the voltage drop across a shunt resistor, the output signal is then sampled by an Analog to Digital Converter, and sent to a PC.

Assuming a direct current voltage supply, given the measured current, I, and the supply voltage V_{cc} the power drawn by the running target is given by $P = I \times V_{cc}$. The total energy consumed is given by $E = P \times T$, where T is the running time, which can be further decomposed in $T = N \times \tau$, where N is the is the number of clock cycles taken by the program, and τ is the clock period [34].

In [11], instead, Fahad et al. employed a power meter located between the target's power socket and the A/C outlet, in order to establish the energy consumption of servers running Intel multicore CPUs.

As we mentioned, the main drawback of directly measuring the energy consumption is the fact that a whole experimental setup is required, with appropriate tools that a software developer may not even have at his disposal. Another drawback of this approach is that it provides only a raw quantity (program X during this run consumed Y Joule), but a software developer may also desire some clues about which source-code entities had the largest contribution to the amount of energy consumed.

2.4 Performance Counters

Performance counters are a feature of some hardware architectures. These architectures expose some registers that store information about the energy consumption of a program (among other metrics). In this section we will give an overview of Intel's RAPL as an example of such a feature.

RAPL provides a set of counters providing energy and power information. It is not an analog power meter, but rather uses a software power model that estimates

power consumption by means of performance counters and hardware power models [30]. The RAPL counters can be accessed by a user by reading appropriate files on the target machine.

A typical usage of RAPL is to read the energy measures, perform a task, and then read again the energy measures, taking the difference between the two readings as the estimate of the energy consumed by the task.

RAPL provides a fine-grained view of the energy consumption, with respect of the hardware components, offering separate estimates for each of the following domain:

- Package: the whole CPU.
- Core: the central components of the CPU, such as ALU, FPU and L1 and L2 cache.
- Uncore: components that are shared between cores, such as L3 cache and the memory controller.
- DRAM: the main memory.

A typical use case example of RAPL is given in [21], in which Liu et al. first provide a Java library that allows to easily access the RAPL energy estimates, and then use said library to benchmark several data access and data organization patterns, providing some guidelines for application-level energy optimizations. They follow the typical usage pattern of measurement $\rightarrow$ task $\rightarrow$ measurement.

Regarding RAPL's accuracy, there are some discording opinions: in [28], it's authors state that the prediction provided by their power model matches actual measurements, showing high correlation between the two. This is partially disproved in [11], where Fahad et al. compare results obtained with RAPL to results obtained via direct measurement. They show that the average error between RAPL's estimate and the actual energy consumption ranges from 8% to 73%. The discrepancy between the estimate and the ground truth is also non-constant: it varies greatly depending on both the performed task and the configuration of the machine. They even show that optimizing an application using data collected from RAPL as benchmark leads to an effective increase in the total (directly measured) consumed energy.

In conclusion, RAPL offers a very interesting set of features: provides a fine granularity in terms of hardware components, can be accessed from source code, allowing to profile arbitrary regions of code, and has very little overhead for the programmer (he just has to add the calls to RAPL where he is interested), but it suffers of accuracy problems. Therefore, at least, it is not a good candidate to be used as ground truth to validate other estimation methods.

We have discussed RAPL as an example of performance counters methods and, as we said, it is bounded to Intel CPUs. Regarding other architectures, it would have been particularly interesting to have similar features implemented in RISC

architectures such as ARM, since they are often employed in the embedded systems applicative domain, where energy consumption is a primary concern. ARM provides some performance counters, related to instruction count, instructions per clock and cache performance [35], but, unfortunately, it does not provide energy-consumption oriented performance counters.

2.5 Instruction Level Energy Modeling

Given a target's Instruction Set Architecture (ISA), an energy model is a model of the energy consumed by each instruction. They have been introduced in 1996 by Tiwari et al. [34].

2.5.1 Characterization of an ISA Energy model

The main components of an energy model are:

- Instruction base cost (B_i, for each instruction i): the cost associated with the basic processing needed to execute an instruction.
- Effect of circuit state ($O_{i,j}$, for each pair of instruction i, j): the cost of the switching activity resulting from executing two consecutive instructions differing one from another.
- Other inter-instruction effects (E_k, for each additional effect k): any other effect that can occur in a real program, such as stalls or cache misses.

Given these components and a program P, the total energy consumed by it, E_p, is given by:

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k$$

Where N_i is the number of occurrences of instruction i, and $N_{i,j}$ is the number of times there has been a switch from instruction i to instruction j.

2.5.2 Why Employing an ISA Energy Model

The most common way to describe a processor's power consumption is through the average power consumption. This single number may not provide enough information to characterize the energy consumed by a program running on the target processor: different programs may employ the functional units of the CPU in different ways, leading to different measurements at equal running time.

ISA Energy Models offer a more detailed view of the energy profile of the target architecture. Therefore they allow to identify variations of consumed energy from one

program to another, and may also guide decision of both humans (hardware/software design) and software (compilers or operating systems).

2.5.3 Producing an ISA Energy Model

Energy models can be produced through an experimental procedure.
In order to obtain instruction base costs, a program consisting of a large loop of a repeated instruction is written. Then one can measure the average current drawn by the processor while executing the program, $\hat{i}$, and multiply it by the supply voltage V_{cc}, obtaining the base energy consumption.
Instructions may also be grouped together, since instructions with similar functionality will have similar base cost.

In order to obtain the circuit state effects, loop of pairs of instruction are required. The difference between the instruction's base costs and the average current measured provides the circuit state overhead.

A similar approach can be employed to obtain the costs of other inter-instruction effects: writing large loops in which the examined effect occurs several times, measuring the average current and subtracting the costs that are already known (base costs and circuit state).

The main disadvantage of this approach is that several different programs must be written: for an ISA with n instructions, $\mathcal{O}(n)$ programs are required to produce base costs and $\mathcal{O}(n^2)$ for circuit state effects.
Estimation of other inter-instruction effects also gets more difficult as the complexity of the architecture increases.

On the other hand, this approach has the big advantage of not requiring a model of the circuit of the target processor, information that is often not disclosed by the manufacturing companies.

In [24], the same authors of [34] employ their technique to model the instruction level energy consumption of a Digital Signal Processing (DSP) embedded system. They describe their experimental setup, consisting in a standard, off-the-shelf, dual-slope integrating digital ammeter connected between the power supply and the pins of the DSP chip. They exploit this power model in order to design a scheduling algorithm that minimizes the total energy consumed.

In this work, they also highlight some practical issues regarding the methodology employed to construct the energy model: impact of operand values and tables size. For the impact of operand values, they propose to make the measurement using a wide a range of operand values, and averaging the consumption values.

By table size, instead, they mean that the number of experiments that need to be carried out to model all the instruction can be overwhelming, and so they propose to group instructions by similar functionality, assigning an average cost to each group,

thus reducing the number of needed experiments. The variation of the energy cost of instructions grouped in this way is around 5%.

2.5.4 Extensions

The original work of Tiwari et al. has been extended in several ways through the years, both in terms of the complexity of the model, and in terms of how said model has been exploited.

Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor In [15], Eder et al. characterize the energy behavior of a multithreaded architecture. Their target processor is the XMOS XS1-L. In this processors threads are executing in a round robin fashion, this makes program execution time-deterministic and allows to easily model the multithreaded behavior.

According to their model, the energy consumption of a program, E_p, is given by:

$$E_p = P_{base} N_{idle} T_{clk} + \sum_{i=1}^{N_t} \sum_{i \in ISA} ((M_t P_i O + P_{base}) N_{i,t} T_{clk})$$

Where T_{clk} is the clock period, P_{base} is dissipated power when the processor is idle, N_{idle} is the number of clock cycles in which the processor was idle, N_t is the maximum number of running threads, M_t is a multiplier that depends of the level of concurrency, and, for each instruction i, P_i is the dissipated power and $N_{i,t}$ is the amount of times instruction the instruction has been executed in thread t, finally, O is the inter-instruction overhead, that they assume to be constant.

In order to perform their experiments, they have designed a software suite that allows to automatically generate benchmarks used to characterize the energy model, loading them on the target and monitor their execution. Tests are generated only for instructions with no effects on control flow and no non-deterministic timing. They obtain execution statistics by hardware simulation (this can be replaced by profiling). They observed that the number of operands has significant impact on power consumption, while data width has an also an impact on power consumption, but it is way lower. They also choose to generalize inter-instruction overhead, observing that it exhibits little variance between different couples of instructions.

For instructions that cannot be directly tested, they propose two solutions: either group instructions by number of operands, so that untestable instructions will be assigned with the cost of the appropriate group, or assign default cost to untestable instructions.

They conclude by stating that the model in which instructions are grouped by number of operands performs worse than the model in which instruction are considered individually: the first one exhibits an average error of 16%, the latter 7%).

Both the models provide a consistent underestimation.

An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors In [17], Lee et al. propose a different methodology to construct the energy model of RISC architectures, targeting embedded systems, that combines an empirical method with statistical data analysis.

They state that techniques such the one proposed by [34] are too simplistic, since they rely only on average current, and therefore cannot consider effects such as the operand specifiers or the instruction fetch address, which may give a contribution to the total energy consumption.

They firstly developed a linear model, then they estimate the unknowns of the model thanks to data from empirical observations, through linear regression.

In their model, they consider a pipelined processor with S stages, and $e_s(X, Y)$ is the energy consumed in pipeline stage s when instruction X is executed after instruction Y. This allows them to consider switching activity between different instructions. They indicate with $I_s(i)$ the instruction executed in pipeline stage s during clock cycle i, and compute the energy consumed in cycle i as the sum of the energy consumed by the pipeline stages: $E_i = \sum_{s \in S} e_s(I_s(i), I_s(I-1))$.

In order to estimate $e_s(X, Y)$, they consider a set V of *instruction level model variables*, such as instruction fetch address, instruction bit encoding, operand specifiers (registers numbers or immediates) and data values. Each instruction X is characterized by a set of natural numbers, one for each model variable, and they indicate one of such numbers as v_X. Given these variables and the corresponding quantities, they compute $e_s(X, Y)$ as the sum of the base cost of instruction X in stage s, B_s^X, and, for each model variable v, the variation of energy consumption contributed by it: $f_s^X(v_X, v_Y)$. So we have $e_s(X, Y) = B_s^X + \sum_{v \in V} f_s^X(v_X, v_Y)$.

The contribution of each model variable is itself expressed as a function of the Hamming distance between v_X and v_Y, $h(v_X, v_Y)$, and the number of bit with value 1 in the binary representation of v_X, called weight $w(v_X)$: $f_s(v_X, v_Y) = H_s^{v/X} \cdot h(v_X, v_Y) + W_s^{v/X} \cdot w(v_X$, where $H_s^{v/X}$ and $W_s^{v/X}$ are unknown coefficients.

Their model, therefore, has three sets of unknown parameters(B, H, and W) that will be estimated through linear regression. In order to do so, they proceed iteratively, by designing a set of programs where only one of the model variables changes, estimate the corresponding unknowns, and use the estimated parameters in next iteration of the measuring $\rightarrow$ estimating process.

To validate their work, they generate a random set of data processing operations, with random operands, and compared the result of their estimate with a direct measure, showing an error ranging from 1% to around 6%.

Source-Level Estimation of Energy Consumption and Execution Time of Embedded Software In [4], Brandolese et al. propose one the first methodologies to both estimate the energy consumption of a program and mapping it to source code level entities. The analysis is performed at level of the parse-tree. The parse tree of a C program is decorated by associating a cost-contribution (atom) to each node. The authors have also introduced *kernel instructions* as a form of target independent assembly instructions, to which energy costs can be assigned. Energy cost are estimated through least square fitting, obtained by comparing the estimate to the output of the ARMulator instruction set simulator. Following the grammar's rule of the C language, rules to combine instruction costs are defined. The parse tree is then instrumented in order to produce a trace during a run of the program, containing information about which nodes have been executed.

The final cost is obtained by combining the costs the of the atoms and the data from the output of the instrumentation, providing a view that maps to each node in the parse tree (which corresponds to a source level entity such as an operation, a function call or an assignment) its contribution to the overall energy cost.

This approach relies on analyzing the parse tree. This allows source code visualization but binds to the source language: in order to change source language, it would be required to perform a complete analysis of the grammar rule of the new language. Also, source level entities related to the grammar of a language may not have a trivial correspondence to assembly instructions, and therefore estimating their cost in terms of the energy associated to their assembly instructions is harder.

Software Energy Estimation Based on Statistical Characterization of Intermediate Compilation Code The same authors of [4], in [5] move their analysis to the LLVM-IR. They provide a technique to understand how LLVM-IR instruction are related to the target's assembly instructions, then, by means of an instrumentation that outputs a trace of the basic blocks executed during a run of a program, they gather data regarding its dynamic behavior. Finally, given a vector that maps energy costs to each assembly instruction, they are able to characterize the total energy cost of a program by summing the costs of the executed basic blocks. The cost of each basic block is obtained by summing the energy costs of all the assembly instructions corresponding to LLVM-IR instructions contained in the basic block.

Their technique to understand the LLVM-IR to assembly mapping is based on statistical analysis. The correspondence is given by an $L \times K$, matrix, T, where L is the number of LLVM instructions in the LLVM-IR language, and K is the number of assembly instructions in the target's instruction set. $T_{j,k} = n$ means that in the translation of the LLVM instruction j, there are n assembly instructions of type k. Given a dataset of several LLVM-IR programs $S = S_1, ..., S_N$, they compile them

to obtain assembly programs for the target architecture. Let $L_{i,j}$ be the numbers of LLVM instructions of type j in source program i, and $D_{i,k}$ the number of assembly instructions of type k in i. It is possible to build an over-constrained system of equations of type $L_{i,j} = \sum_{k=1}^{K} D_{i,k} \cdot x_{j,k}$. The unknowns $x_{j,k}$ are the elements of the sought after matrix T. By posing the constraint that they must be non-negative, they formulate a *non-negative least square* problem, whose solution provides the LLVM-IR to assembly mapping. This analysis has to be performed every time one wants to target a different architecture.

They also require the energy cost of each assembly instruction to be known. They acknowledge the fact that this information is often not disclosed by the manufacturers, and state that it may be approximated by a linear function of the clock cycles, since the current absorbed by each clock cycle exhibits very little variance. This a very interesting claim since data about the clock cycles taken by each instruction is often available, and it may allow a very easy way to provide an estimate of the energy cost of each assembly instruction.

Energy Transparency for Deeply Embedded Programs In [13], Eder et al. provide a LLVM-IR to assembly mapping technique that differs from [5], based on debug information and disassembly of the binary. Given this mapping, they provide a methodology to statically estimate the worst case energy consumption (WCEC) of a program, both at LLVM-IR level or the assembly level, and they also employ a profiling technique similar to [5] in order to obtain an energy consumption estimate given a run of the program.

Their LLVM-IR to assembly mapping technique is based on an LLVM pass that replaces the line number information, contained in the LLVM Debug Info classes, with an unique identifier of the instruction being considered. Then, after that the modified LLVM module has been compiled, by disassembling the binary and parsing the line table, for each entry in the line table, there will a pair *<addr, id>*, such that the assembly instruction at the address *addr*, can be mapped to the LLVM instruction with identifier *id*.

This procedure by itself does not suffice in providing a complete mapping: some assembly instructions do not have a corresponding entry in the line table, but they can be safely mapped to the closest preceding LLVM instruction in the line table.

Their static, worst-case energy consumption estimation is based on the Implicit Path Enumeration Technique (IPET) [20]. It requires the program's CFG, annotated with information regarding properties of the dynamic behavior of the program, such as loop bounds or mutually exclusive conditions. Given these annotations, that must be specified by the user, the problem can be formulated as an Integer Linear Programming problem, whose cost function is $\sum_{i=0}^{N} c_i \cdot x_i$, where, for each basic block i, c_i is the cost of executing the basic block, and x_i is the number of executions of

the basic block. c_i is obtained in similar fashion to [5]: by adding together all the costs of the instructions in the basic blocks, which are obtained by adding the costs of the assembly instructions mapped to the LLVM instructions.

Their profiling technique, instead, consist in an instrumentation that outputs the identifier of the basic block every time the basic block is run. Similarly to the static technique, the total cost is obtained by adding together the costs of the basic blocks executed during the run.

2.6 Source Code-Level Visualization

Between the aforementioned techniques, only a few of them allows for a mapping of the energy measures to source code level entities: Brandolese et al., [4] provide such a result, but at the cost of performing their analysis on the parse tree, and the same authors in [5] state that their methodology can provide source code level information, but without providing examples.

RAPL-based techniques, instead, allow to estimate the energy cost of arbitrary code segments by means of the measurement → task → measurement pattern, but they are coarse-grained, available only for some hardware architectures, and require the developer to manually mark the code region that they want to inspect.

All the other techniques provide just a raw measure of the total energy consumption. In [26], Pereira et al. propose an interesting method to map such raw results to source level entities, by what they call *SPectrum Based Energy Leak Localization* (SPELL). Their technique is based on spectrum based fault localization, a technique that uses statistical analysis to provide hints as to which software components may be responsible for a program's failure. The authors of [26] extend this approach, from program failures to energy leaks (a component that consumed *too much* energy). Unfortunately, it is clear to understand when a program fails (the output does not match the expected output), but it is not clear to understand when the energy consumption is too high, and the authors do not clearly state how this is performed. Nevertheless, their work represents an interesting effort, since it allows to map measures obtained with various techniques (direct measuring, simulation, or performance counters), to source level entities with arbitrary granularity (packages, classes, methods).

There are some commercially available tool, mentioned in [27], that allow for source level visualization. The development board developed by SiliconLabs [10] allows to perform energy measurements of ARM microcontrollers, and provides a per-method visualization. The board has integrated current and voltage sensors, and periodically sends the measured values to an host computer. Binaries are statically linked and therefore the program counter value alone provides enough information to determine instructions currently being executed.

Other commercially available tools are oriented towards the Android world, for instance Green Droid [9] provides estimates of the energy consumption by exploiting PowerTutor [38], a component power manager that employs the battery and current sensors commonly present in smartphones.

2.7 Final Remarks

We have offered a broad overview of some techniques employed to measure or estimate the energy consumption of software. Table 2.2 provides a summary of the described techniques, while table 2.1 focuses on instruction set energy model based techniques, briefly highlighting their main contributions. A complete listing of such techniques would go beyond the scope of this document.

Authors	Year	Static / Dynamic	Source code Attribution	Notes
Tiwari	1996	Dynamic	No	Introduces ISA energy models.
Tiwari	1997	Dynamic	No	Grouping instructions to reduce number of experiments.
Eder	2015	Dynamic	No	Multithreaded pipeline.
Lee	2001	Dynamic	No	Employs linear regression.
Brandolese	2008	Dynamic	Yes	First source code attribution.
Brandolese	2011	Dynamic	Yes	Analysis performed on an intermediate representation.
Eder	2017	Both	No	Worst case static analysis and profiling considerations.

Table 2.1: Summary of ISA energy model based techniques.

Name	Pros	Cons
Direct measurement.	Very precise. Can (in theory) be applied to any hardware architecture.	Requires a complex experimental setup.
Simulation.	Fine grained characterization of the energy consumption at hardware level.	Their accuracy depends on the accuracy of the target architecture's model and its power characterization.
Performance counters.	Simple APIs provide access to energy consumption estimates. Allows to profile arbitrary code regions.	Bound to a specific hardware vendor. Not always available. Concerns regarding their accuracy.
ISA energy models.	Accurate characterization of the target architecture's energy behavior. Allows for source-code level attribution.	Targets mostly only embedded devices. Producing an Instruction Set energy model is a very time-consuming operation.

Table 2.2: Summary of energy consumption estimation techniques.

These techniques are various and diverse, one common denominator that we have identified is the fact that they try to overcome a general reticence, from the hardware vendors, to provide clear and detailed information about the power consumption of their products, which is often reduced to a single average quantity (current or power), that may not provide enough information to fully characterize the hardware architecture.

Another key takeaway is the fact that there is not a methodology that is both target-agnostic and source language independent while providing estimates with a granularity that allows for a source level visualization. Therefore, in a similar fashion to some mentioned authors ([13], [5]), we will restrict the scope of our work to embedded systems, since they allow for a detailed model of the the underlying hardware architecture, that bears itself to source code-level attribution.

Chapter 3

Energy Consumption Estimation

This chapter will provide an overview of the proposed tool. We start by giving a more accurate definition of the goal that we want to accomplish, then we describe the architecture of the tool and finally we provide a piece-wise description of its components.

3.1 Problem Statement

When a program is compiled, it goes through a set of transformations. Generally speaking, starting from the source file, it is then translated into one or more intermediate representations that offer decreasing layers of abstraction going towards assembly language or machine code. LLVM-IR is one of these possible intermediate representations.

Figure 3.1 illustrates the compilation process. Each location in the source code can correspond to several instructions in the unoptimized intermediate representation, which, in our case, is the LLVM-IR.

At this stage the correspondence between source locations and IR instructions is very well defined, and it is provided by debug information, as mentioned in chapter 1. The module then goes through a set of optimization passes. These passes may greatly change its structure, with the side effect of "obfuscating" the relationship between high-level and low-level entities.

Finally, the optimized intermediate representation is lowered towards a target

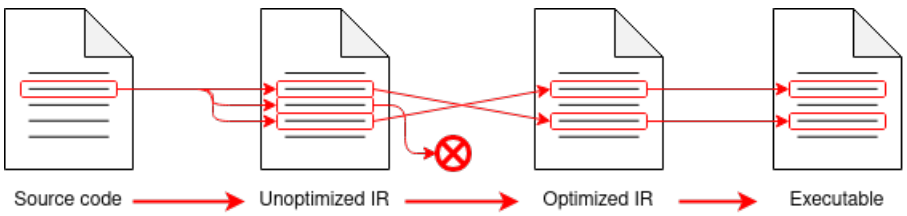

Figure 3.1: Illustration of the compilation steps.

architecture's assembly language. IR instructions do not correspond directly to assembly instructions, but since most of the transformations have been applied in the previous stages, and since the IR is basically a generic assembly language, a clear mapping between optimized IR and assembly instructions can be established.

Our Goal We want to develop a tool that is able to provide an estimate of the energy consumption of a program, and that can also provide insights to which source code entities led to the estimated consumption. In order to do so, there are several sub-requirements that we need to satisfy:

1. Reconstruct the mapping between source code locations and LLVM-IR instructions in the intermediate representation.
2. Reconstruct the mapping between unoptimized IR and optimized IR, understanding the changes made by the optimizer.
3. Reconstruct the mapping between optimized IR and assembly instructions.

This information will then be coupled with the runtime information gathered through program instrumentation, described in section 3.3, and with an ISA energy model whose quality will be assessed in chapter 5.

3.2 Architecture

The proposed tool is composed of several loosely coupled components, each of them is designed to be reusable and as independent as possible from the others. Figure 3.2 illustrates the general usage of the tool. Square nodes represent data (either on-disk files or in-memory), while round nodes represent components, either external (like the compiler front-end/back-end) or developed by us.

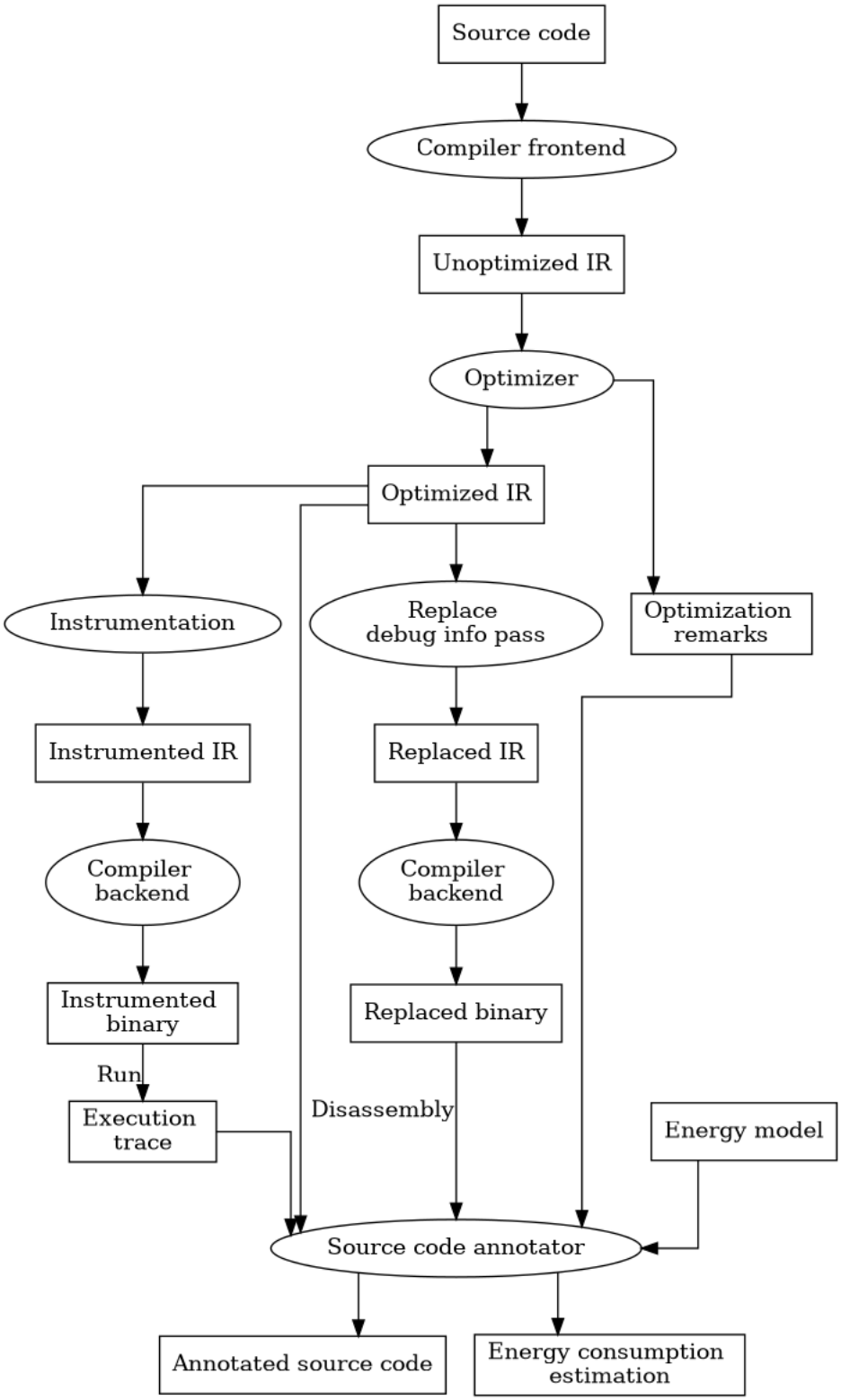

Figure 3.2: General workflow.

Starting from the source code of the program, we employ the compiler front-end in order to produce an LLVM Module. This allows to inherently support any high-level language that is compiled by means of the LLVM framework, and whose compilers allows for the emission of LLVM-IR.

The obtained (unoptimized) IR may then be optimized. Since we want to keep the source-code → LLVM-IR mapping as complete as possible, we modified the LLVM optimizer[1], in order to emit a description of the optimization process (the optimization remarks). This description is then employed to both preserve the aforementioned mapping, and to provide a visualization of the effects of compiler transformations. An accurate description of this component is provided in chapter 4.

The optimized module is then duplicated. One copy is instrumented, compiled and executed. The instrumentation has the purpose of producing a trace of the execution of the basic blocks, and it is described in greater detail in section 3.3, while the other copy is employed to obtain the LLVM-IR → Assembly mapping, following the procedure described in section 3.4. The modules can be compiled by two different compiler back-ends: since the basic block trace produced by the instrumentation works on the LLVM-IR, the sequence of executed blocks doesn't change by changing the target architecture, therefore the trace can be produced by running the program on a machine, but the final energy estimation is obtained by employing an energy model of a different target architecture.

Finally, the execution trace, the binary with replaced debug information, the optimized module, the execution trace and the optimization remarks are fed as input to the source code annotator. By employing an ISA energy model, this component can:

- Provide an estimate of the total energy consumption of the program.
- Propagate this estimate at source code level, showing how each source code entity contributed to the total energy consumption, with the granularity of a source code line.

The source code attribution process is independent from the employed metric: in this work we had a strong focus on energy consumption, but any "cost function" that may correspond to high-level entities could be employed, such as time, instruction count or events like cache misses.

3.3 Instrumentation

In the following section we will describe the instrumentation technique used in order to provide a rich source code-level visualization of the energy consumption of a

[1] http://llvm.org/docs/CommandGuide/opt.html

program. The proposed methodology is general, in the sense that it is not dependent of either the cost function employed, the target architecture, or the source language.

Our goal is to assign to each source code location the energy (or any other cost) consumed by the low level instructions that correspond to the location. Moreover, all modern programming languages have the concept of function call. The source code location where a function call takes place is called *callsite*. We want to associate to each callsite not only the cost of performing the call, but the cost of the entire computation performed during the function call, providing to the user a more natural and informative annotation.

We will start by describing an instrumentation technique similar to the ones employed in [5] and [13], that we will call *SimpleInstr*, then we will describe another instrumentation, *ComplexInstr*, that is able to provide more information than *SimpleInstr*, but has a bigger runtime overhead. Finally, we will describe an algorithm that allows to obtain the same amount of information provided by the latter, given the output of the former instrumentation. *ComplexInstr* provides enough information to associate to each callsite the total cost of the function call, but has a bigger runtime overhead, therefore, employing the algorithm that we will present allows to obtain the need information without incurring in an additional runtime overhead.

3.3.1 SimpleInstr

This instrumentation aims at providing as output a trace of the basic blocks executed during a run of the program. Thus it basically consists of:

- The assignment of an unique identifier to each basic block.
- For each basic block, the injection, at its beginning, of an instruction that outputs (to standard output or to a file) its unique id.

Given the class hierarchy presented in section 1.1.3 the pseudo code of the instrumentation is the following:

Algorithm 3.1 SimpleInstr.

```
   Input Module M
   Output Instrumented Module M
1: id := 0
2: M.addFunction(output)
3: for Function F ∈ M do
4:     for BasicBlock BB ∈ F do
5:         BB.push_front(new callInstruction(output, id))
6:         id++
7:     end for
8: end for
```

Where *output* is a generic function that allows to output the basic block's id. At line

5 we are inserting, at the beginning of the basic block a new call Instruction, that calls function *output* with parameter id, allowing to produce the basic block trace once the program has been run.

The actual run time overhead of this instrumentation depends on the implementation of the *output* function, which will have to run every time a basic block is run. Therefore, the overhead is multiplicatively constant, where the constant depends on the run time of the *output* function.

Example Consider the following snippet of C code

```
1  int fact(int n){
2    if(n <= 1)
3      return 1;
4    return n * fact(n-1);
5  }
6
7  int main(){
8    //BB 3
9    fact(2);
10   fact(3);
11 }
```

When lowered to LLVM-IR, it has the following form, the basic blocks have been commented [2] with their identifier:

```
1  define dso_local i64 @fact(i32 %0)  {
2    ;bb0
3    call void @llvm.dbg.value(metadata i32 %0, metadata !12,
         metadata !DIExpression()),
4    %2 = icmp sle i32 %0, 1,
5    br i1 %2, label %8, label %3,
6
7  3:                                                ; preds = %1
8    ;bb1
9    %4 = sub nsw i32 %0, 1,
10   %5 = call i64 @fact(i32 %4),
11   call void @llvm.dbg.value(metadata i64 %5, metadata !19,
         metadata !DIExpression()),
12   %7 = mul i64 %6, %5,
13   br label %8,
14
15 8:                                                ; preds = %1,
       %3
16   ;bb2
17   %.0 = phi i64 [ %7, %3 ], [ 1, %1 ],
```

[2] Comments in LLVM-IR begin with a semicolon

```
18    ret i64 %.0,
19  }
20
21
22  define dso_local i32 @main()  {
23    ;bb3
24    %1 = call i64 @fact(i32 2),
25    %2 = call i64 @fact(i32 3),
26    ret i32 0,
27  }
```

By instrumenting the program and running it, we obtain the following execution trace: 3, 0, 1, 0, 2, 2, 0, 1, 0, 1, 0, 2, 2, 2.

3.3.2 ComplexInstr

The information provided by *SimpleInstr* is enough to associate to each location of the source code the computational cost that arose from it, but it does not suffice for providing the total cost of the computation initiated in a callsite.

In order to perform this, we need not only to know which basic block has been executed, but also to know which callsites have been executed before the execution of the said basic block. *ComplexInstr* provides this information, by basically storing a stack of callsites, and output-ing said stack every time a basic block is executed. The pseudo code of the instrumentation is the following:

Algorithm 3.2 ComplexInstr.

```
    Input Module M
    Output Instrumented Module M
 1: id := 0
 2: var stack;
 3: M.addGlobalVariable(stack)
 4: M.addFunction(output)
 5: M.addFunction(print_stack)
 6: M.addFunction(push)
 7: M.addFunction(pop)
 8: for Function F ∈ M do
 9:     for BasicBlock BB ∈ F do
10:         BB.push_front(new callInstruction(print_stack, id))
11:         id++
12:         for Instruction I ∈ BB do
13:             if isCallInstruction(I) then
14:                 loc := I.getLocation()
15:                 I.add_before(new callInstruction(push, loc))
16:                 I.add_after(new callInstruction(pop))
17:             end if
18:         end for
19:     end for
20: end for
```

We start by inserting in the module a new global variable: a pointer to the head of the stack. Then we insert four functions: *output*, which is the same as in *SimpleInstr*, *print_stack*, that receives as input an id, and iteratively calls *output* in order to print the id and then the whole stack, *push* and *pop*, two functions that respectively add or remove an element from the stack.

This instrumentation has a larger overhead than the one previously presented: it requires to manage a stack at runtime, which requires memory allocation and deallocation, and also outputs the entire stack every time a basic block is run, operation that requires linear time with respect of the size of the stack.

Example Given the same input program of the previous section, *ComplexInstr* produces the following output:

```
3
0 <Line: 13 Col: 3 File: fact.c>
1 <Line: 13 Col: 3 File: fact.c>
0 <Line: 7 Col: 21 File: fact.c> <Line: 13 Col: 3 File: fact.c>
2 <Line: 7 Col: 21 File: fact.c> <Line: 13 Col: 3 File: fact.c>
2 <Line: 13 Col: 3 File: fact.c>
0 <Line: 14 Col: 3 File: fact.c>
1 <Line: 14 Col: 3 File: fact.c>
0 <Line: 7 Col: 21 File: fact.c> <Line: 14 Col: 3 File: fact.c>
1 <Line: 7 Col: 21 File: fact.c> <Line: 14 Col: 3 File: fact.c>
0 <Line: 7 Col: 21 File: fact.c> <Line: 7 Col: 21 File: fact.c>
  <Line: 14 Col: 3 File: fact.c>
2 <Line: 7 Col: 21 File: fact.c> <Line: 7 Col: 21 File: fact.c>
  <Line: 14 Col: 3 File: fact.c>
2 <Line: 7 Col: 21 File: fact.c> <Line: 14 Col: 3 File: fact.c>
2 <Line: 14 Col: 3 File: fact.c>
```

As we can see, each source location is identified by a $\langle Line, Column, File \rangle$ triple, that refers to the original C source file. We have three callsites in the source file, at lines 13, 14 and 7, and we can clearly see which callsites have been traversed at the execution of each basic block. This is the information needed to associate to each callsite the total cost of the function call, but the non-constant overhead of the *ComplexInstr* makes this kind of instrumentation not usable in practical contexts.

3.3.3 Trace Expansion Algorithm

Due to the the big runtime overhead of *ComplexInstr*, here we propose an algorithm that, given a trace produced by *SimpleInstr*, allows to produce a trace equivalent to the one produced by *ComplexInstr*. The algorithm is based on the following considerations:

- The sequence of calls performed by a basic block does not change at runtime (due to the definition of basic blocks).
- When a function returns, the last basic block executed by a function will have a *return* as the last instruction.

It is therefore possible to, provided the trace of the execution of the basic blocks, perform the same sequence of *push* and *pop* that would have been performed by *ComplexInstr*, but this can be done in post processing, greatly reducing the overhead of the instrumentation, but retaining the same information.

The first step of the algorithm is to analyze all the basic blocks contained in the module. For each basic block bb_i, we want to determine

- The function calls performed in the block. We will refer to them as $calls(bb_i)$, which returns an array of callsites $\langle c_1, ..., c_n \rangle$.
- Whether or not the basic block is terminated by a *return* instruction, expressed by the predicate $isReturn(bb_i)$.

During the algorithm's execution, we will manage a stack-based data structure, the element that we will store in the stack are pairs $\langle location, bb_i \rangle$, where $location$ is the source location of a callsite, and bb_i is a reference to the basic block in which the callsite took place. The reference is needed since a basic block may have multiple calls within itself, and when we pop a callsite, we may need to push the next one in the basic block.

The pseudo code of the algorithm is the following. A trace contains a sequence of basic blocks: bb_i means "basic block with id i", not "basic block with position i in the trace".

Algorithm 3.3 Trace Expansion.

Input Trace T, produced by *SimpleInstr*
Output Trace L, equivalent to one produced by *ComplexInstr*

```
1:  function expandTrace(Trace T)
2:      var stack                                   ▷ stack variable storing the pairs
3:      for bb_i ∈ T do
4:          L.push_back(bb_i, stack)                ▷ Add an entry to output trace
5:          if calls(bb_i) == ∅ then                ▷ If a block has no calls
6:              if isReturn(bb_i) then              ▷ If it is return-terminated
7:                  pop(stack)                      ▷ Pop from callsites stack
8:              end if
9:          else                                    ▷ If a block has calls
10:             ⟨c_1, ..., c_n⟩ := calls(bb_i)
11:             push(stack, ⟨c_1, bb_i⟩)            ▷ push a new pair on the stack
12:         end if
13:     end for
14:     return L
15: end function

16: procedure pop(stack)
17:     if stack == ∅ then
18:         return
19:     end if
20:     ⟨b_i, c_j⟩ = stack.pop_back()               ▷ Remove last element of the stack
21:     ⟨c_1, ..., c_n⟩ := calls(bb_i)
22:     if c_j == c_n then          ▷ If the popped call was the last of the basic block
23:         if isReturn(bb_i) then
24:             pop(stack)                          ▷ Recursively pop from the stack
25:         end if
26:     else                        ▷ The popped call was not the last of the basic block
27:         push(stack, ⟨bb_i, c_{j+1}⟩)
28:     end if
29: end procedure
```

The pop_back() function is the usual function that removes the head of the stack and returns it. The recursive call at line 24 is needed because a basic block may both have function calls and be return-terminated. Return-terminated blocks with no function calls are handled at line 7, while return-terminated blocks with function calls lead to the corresponding *pop* only after all the calls performed by them have returned.

Example Consider again the snippet of LLVM-IR code:

```
1  define dso_local i64 @fact(i32 %0)  {
2    ;bb0
3    call void @llvm.dbg.value(metadata i32 %0, metadata !12,
        metadata !DIExpression()),
4    %2 = icmp sle i32 %0, 1,
5    br i1 %2, label %8, label %3,
6
7  3:                                                ; preds = %1
8    ;bb1
9    %4 = sub nsw i32 %0, 1,
10   %5 = call i64 @fact(i32 %4),
11   call void @llvm.dbg.value(metadata i64 %5, metadata !19,
        metadata !DIExpression()),
12   %7 = mul i64 %6, %5,
13   br label %8,
14
15 8:                                                ; preds = %1,
       %3
16   ;bb2
17   %.0 = phi i64 [ %7, %3 ], [ 1, %1 ],
18   ret i64 %.0,
19 }
20
21
22 define dso_local i32 @main()  {
23   ;bb3
24   %1 = call i64 @fact(i32 2),
25   %2 = call i64 @fact(i32 3),
26   ret i32 0,
27 }
```

. We have seen that, when instrumented, it produces the following trace: bb_3, bb_0, bb_1, bb_0, bb_2, bb_2, bb_0, bb_1, bb_0, bb_1, bb_0, bb_2, bb_2, bb_2.

We have three callsites, at line 7, 13 and 14 of the C source file, that we will

identify as c_7, c_{13}, c_{14}, so we have $calls(bb_1) = \langle c_7 \rangle$, $calls(bb_3) = \langle c_{13}, c_{14} \rangle$ [3], basic blocks bb_2 and bb_3 are return-terminated.

Given this characterization of the basic blocks, we can begin to apply the algorithm. We start with a trace entry of bb_3, we add the corresponding entry in the resulting trace and we push on the stack the pair $\langle c_{13}, bb_3 \rangle$. This means the next basic block will be executed within the function call performed at line 13.

---

worklist = bb_0, bb_1, bb_0, bb_2, bb_2, bb_0, bb_1, bb_0, bb_1, bb_0, bb_2, bb_2, bb_2
$L = bb_3$
$stack = \langle c_{13}, bb_3 \rangle$

Next we have an entry for bb_0, it does not contain calls and its not return-terminated, therefore we simply add an entry to the resulting trace, containing the id of the block and the current stack.

---

worklist = bb_1, bb_0, bb_2, bb_2, bb_0, bb_1, bb_0, bb_1, bb_0, bb_2, bb_2, bb_2
$L = bb_3, \langle bb_0, c_{13} \rangle$
$stack = \langle c_{13}, bb_3 \rangle$

When processing entry bb_1 we push the pair $\langle c_7, bb_1 \rangle$, and then we process bb_0, emitting the corresponding entry.

---

worklist = bb_2, bb_2, bb_0, bb_1, bb_0, bb_1, bb_0, bb_2, bb_2, bb_2
$L = bb_3$, $\langle bb_0, c_{13} \rangle$, $\langle bb_1, c_{13} \rangle$, $\langle bb_0, c_{13}, c_7 \rangle$
$stack = \langle c_{13}, bb_3 \rangle$, $\langle c_7, bb_1 \rangle$

Next we have two entries of bb_2. It is return-terminated and does not contain any call, so we pop from the stack. In the first pop we remove the pair $\langle c_7, bb_1 \rangle$, since bb_1 does not contain any call besides c_7, we do not have any additional push.
With the second bb_2 entry we remove $\langle c_{13}, bb_3 \rangle$. This means we have terminated the part of computation associate to c_{13}. bb_3 contains also call c_{14}, so we push a pair $\langle c_{14}, bb_3 \rangle$ on the stack, therefore the following basic block will belong to the computation of call c_{14}. The algorithm continues and it is possible to verify that it will produce a trace equivalent to the one produced by *ComplexInstr*.

3.3.4 CountInstr

The aforementioned instrumentation techniques provide a trace of the basic blocks executed by the program. This allows to estimate the energy consumption of the program, annotate the source code and, eventually, the call sites. The main downside of this approach is that the size of a trace grows linearly with the length of

[3] In the pseudo code we have used c_i to identify the callsite in position i in the calls of a block, here we use i as the line number that identifies the call.

the program's run, eventually becoming quite large, reaching hundreds of megabyte of size after around a minute of execution.

The last instrumentation technique that we have developed aims at providing basic source code annotation (with no callsites attribution), and total energy estimation, while having lower performance overhead and more importantly, generating a smaller, more manageable output. We achieve this by simply *counting* the amount of times each basic block is executed.

The pseudo code of the instrumentation is the following, where the *countBasicBlocks* function provides the number of basic blocks in a module, the *increase* function is an LLVM Function that increases by one the execution count of a given basic block, and the *output* function is a function that outputs the entire vector of execution counts:

Algorithm 3.4 CountInstr.

Input Module M
Output Instrumented Module M

```
1: numBB := countBasicBlocks(M)
2: M.addGlobalVariable(Vector(int, numBB))   ▷ Inject a vector of length = # of basic blocks
3: id := 0
4: M.addFunction(increase)  ▷ Function that increases by one the execution count for the specified basic block
5: for Function F ∈ M do
6:     for BasicBlock BB ∈ F do
7:         BB.push_front(new callInstruction(increase, id))
8:         id++
9:     end for
10: end for
11: M.addCallOnExit(output)              ▷ Inject a call to output when the program terminates.
```

Example Given the same program execution of the previous section, with trace 3, 0, 1, 0, 2, 2, 0, 1, 0, 1, 0, 2, 2, 2, the *CountInstr* will provide as output:

```
0: 4
1: 3
2: 5
3: 1
```

In other words, basic block 0 has been executed four times, basic block 1 three times, and so on.

3.4 Replacing Debug Info

Replacing debug info in the module is necessary in order to obtain the LLVM-IR → Assembly mapping following the procedure described in 2.5.4. This procedure exploits the debug information found in the executable: we implemented an LLVM Pass that replaces the line information in the debug info with an identifier of the LLVM Instruction, then, by generating an executable with debug information, we will find the identifiers of the LLVM Instruction in the DWARF Line table. The pass simply iterates over all the instructions, inserting the identifier:

Algorithm 3.5 Replace Debug Info.

Input Module M
Output Module with replaced debug info

```
1: id := 0
2: for Function F ∈ M do
3:     for BasicBlock BB ∈ F do
4:         for Instruction I ∈ BB do
5:             I.setDebugLine(id)
6:             id++
7:         end for
8:     end for
9: end for
```

Example Consider the following C code:

```
1 int f(int n){
2   return n+1;
3 }
```

It is compiled to the following LLVM-IR code (the `!dbg !n` signature indicates a pointer to the corresponding metadata node, only the useful nodes have been included)

```
1 define dso_local i32 @f(i32 %0) !7 {
2   %2 = add nsw i32 %0, 1, !dbg !13
3   ret i32 %2, !dbg !14
4 }
5
6 !13 = !DILocation(line: 2, column: 11, scope: !7)
7 !14 = !DILocation(line: 2, column: 3, scope: !7)
```

By running the pass we assign identifier 0 to the *add* instruction, and identifier 1 to the *return* instruction. The identifiers are found in the line attribute of the debug locations:

```
1 define dso_local i32 @f(i32 %0) !7 {
2   %2 = add nsw i32 %0, 1, !dbg !13
```

```
3   ret i32 %2, !dbg !14
4 }
5
6 !13 = !DILocation(line: 0, column: 11, scope: !7)
7 !14 = !DILocation(line: 1, column: 3, scope: !7)
```

Finally, after compiling, in the resulting assembly we can find the identifiers, in the Line attribute of the DWARF line table.:

```
0000000000401110 <f>:
  401110: 55                              pushq %rbp
  401111: 48 89 e5                        movq  %rsp, %rbp
  401114: 83 c7 01                        addl $1, %edi
  401117: 89 f8                           movl  %edi, %eax
  401119: 5d                              popq  %rbp
  40111a: c3                              retq
  40111b: 0f 1f 44 00 00                  nopl (%rax,%rax)
```

```
Address            Line   Column File   ISA Discriminator Flags
------------------ ------ ------ ------ --- ------------- -------------
0x0000000000401114      0     11      1   0             0  is_stmt prologue_end
0x0000000000401117      1      3      1   0             0  is_stmt
```

There are some observations to make:

1. Not all the instructions have a corresponding entry in the line table.
2. There are some assembly instructions, the ones corresponding to the assembly function prologue, that cannot be correctly mapped to LLVM-IR instructions.

Point number 1 comes from the fact that the DWARF format is designed to allow for source code level debugging. The line table provides the mapping between source code and addresses in the executable, and its main purpose is to allow to set breakpoints. When setting a breakpoint, the debugger stops *before* any of the effects of the instructions corresponding to the specified location occurs, therefore only the entry of the first address of the set of instructions corresponding to the source location is needed in the line table. Therefore, the assembly instructions that do not have an entry in the line table can be mapped to the location (or instruction id, in our case) of the closest instruction that has an entry in the line table and has address lower than the other instruction, similarly to how Eder el at. performed this operation, as described in section 2.5.

Point number 2 comes from the fact that the function prologue and epilogue conventions are specified in the instruction set of the target architecture. While LLVM provides a *return* instruction, that allows to map the instructions of the

epilogue in the assembly to the return instruction of the function, there is not a *begin* instruction, and therefore we cannot map the instructions in the prologue to any LLVM Instruction. To overcome this problem, we keep track of which instructions belong to the function prologue, and assign them to source code line in which the function is defined, which is an information available in the LLVM Metadata.

This technique provides a complete mapping between LLVM-IR and assembly instructions. By "complete" we mean that every assembly instruction corresponds to one LLVM-IR instruction. In the following we will refer to this relation as $map_{IR \to Asm}$. Given an LLVM instruction i, $map_{IR \to Asm}(i)$ represents the (possibly empty) set of assembly instructions corresponding to i.

3.5 Disassembler

In order to be read from the executable, the replaced debug information must be extracted from the binary produced by the compiler backend. The process of converting the binary machine in human-readable assembly language is called *disassembling*. There are several off-shelf tools that can be employed for disassembling, but most of them aim at providing a string representation of the assembly.

We need an in-memory representation that can be manipulated through a set of APIs, therefore we provide a custom implementation of a disassembler, based on the llvm-objdump tool [4].

[4] https://www.llvm.org/docs/CommandGuide/llvm-objdump.html

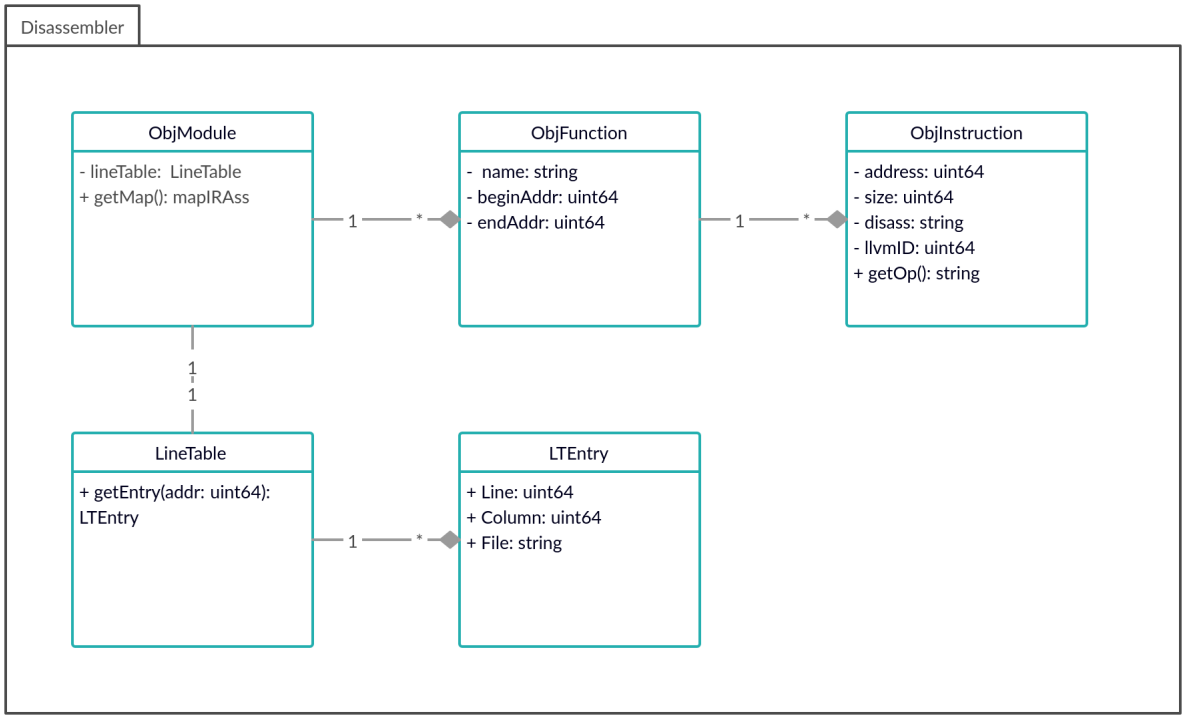

Figure 3.3: UML diagram of the disassembly classes.

Class Hierarchy The goal of the disassembler is to retrieve the $map_{IR \to Asm}$ relation, given the binary executable of a program. To do so, we have designed a set of classes that resemble the LLVM class hierarchy (Modules, Functions, Instructions) but at the assembly level, represented in Figure 3.3. Each ObjInstruction represents an assembly instruction, identified by its address in the executable, and its size. The class is basically a wrapper around the string representation of the instruction, containing the name of the operation and the operands, and provides methods to easily access them. An ObjFunction is a sequence of ObjInstructions, characterized by its begin and end address in the executable, the function name is the same name that the function has in the LLVM Module. An ObjModule contains all the disassembled functions and provides a method to compute the $map_{IR \to Asm}$ relation.

Extracting the Map In order to disassemble a function, given its name in the LLVM Module, we need to find the corresponding *symbol* in the binary. LLVM provides some facilities to extract all the symbols form a binary file, then the symbols can be sorted by comparing the respective addresses. Functions are stored contiguously in the *.text* section of a program, and by sorting we can find the end address of a function as the address of the next symbol in the sorted sequence of symbols. This allows us to understand the boundaries of a function, and then we can disassemble all the instructions within those boundaries, and add them to the corresponding

ObjFunction.

Each ObjModule contains also an in-memory representation of the DWARF line table contained in the executable. This line table will of course have the line number replaced with the the id of the LLVM Instructions, and therefore it provides the inverse relation of $map_{IR \to Asm}$. This means we simply have to iterate over it to compute its inverse. The inverse, coupled with the information contained in each ObjInstruction, provides us the sought after $map_{IR \to Asm}$.

3.6 Energy Model

This section describes the energy model employed in order to estimate the energy consumption of the program being analyzed. We have developed several models, with increased complexity and accuracy. In doing so we have been guided by the results of the experimental evaluation described in chapter 5. Please note that the choice of the energy model employed is largely orthogonal with respect to the source code attribution of the execution cost. From the point of view of the attribution, the energy model is simply a function that, given an assembly instruction, provides the energy consumed while executing it. Therefore, the lack of accuracy of the energy model does not invalidate the rest of this work: a more precise model can be plugged in the system, increasing the accuracy of the estimation.

As we detailed in chapter 2, there are several methods that allow for the modeling of the energy consumption of a target architecture. Our main goal is to provide a tool that allows a software developer to gain insights in the energy performances of his application, while investing as less time and resources as possible in constructing an experimental setup that allows for the measurement of the energy consumption.

Clock-Cycles Based Model As we already mentioned in the final remarks of our overview of the state of the art, a lot information that could be useful in order to provide our desired result is not disclosed to the general public by the hardware manufacturers. The approach employed by Brandolese et al. in [5] is particularly interesting since it is based mainly on the clock cycles needed by each instruction in order to complete, and this information is often available, since it is used by the compiler in the code generation phase. This would allow a developer who is targeting an hardware architecture whose energy characterization is not readily available, to construct the energy characterization by simply inserting the data regarding the architecture's CPI (clocks per instruction). Given an assembly instruction j, $cpi(j)$ provides the clock cycles of the instruction. Knowing the average power consumption of the target architecture, $\overline{P}$, and its operating frequency, f_{clk}, we can define the energy consumed by j as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}}$.

Inter-Instruction Overhead The clock-cycles based model's main advantage is the fact that it can be constructed using readily available information. Unfortunately, it provides a large underestimation – when compared to the direct measurement – of the energy consumption. The main cause of this underestimation can be found in the fact that the model doesn't take into account the cost of the switching activity that occurs during the program's execution. In order to account for this phenomena, but also to keep the model as simple as possible, we have adopted a solution similar to the one employed by Eder et al. [15]: define a single parameter, O, constant with respect to the executed instructions. Given this addition to the model, the energy consumed by an assembly instruction can be modeled as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}} + O$. In order to estimate the inter-instruction overhead, we have resorted to a linear regression technique: the estimation of the total energy consumed by a program P, can be expressed as $E(P) = \sum_{j \in P}(energy(j) + O)$. Since O is constant with respect to j, we can rewrite the previous equation as $E(P) = \sum_{j \in P} energy(j) + IC(P) * O$, where $IC(P)$ indicates the number of instruction executed during a run of program P. Given a direct measures of the energy consumption of P, $\hat{E}(P)$, the inter instruction overhead can be estimated by minimizing the function $J = \sum_P (E(P) - \hat{E}(P))^2$.

Memory Access Coefficient The main assumption made when employing the clock cycles in order to model the energy consumed by an instruction is that the current drawn by the CPU when executing the instruction (and therefore the power) is constant. During the experimental evaluation we have realized that this is, in general, not true. By comparing the current draw while executing memory-bound sections of a program (such as the initialization phase) to sections that are more CPU-bound, we have empirically observed that memory operations draw around 10 % less current. A possible cause of this phenomena are CPU stalls that may occur while the processor is waiting for the memory, combined with the fact that the RAM draws less current that the CPU.

To keep this phenomena into account, we have extended the model by adding the function $memAcc : ISA \rightarrow [0, 1]$, that returns the memory access coefficient for an instruction j, for instance, for a specific target architecture, we could have $memAcc(add) = 1$, since the add operation doesn't access memory, and $memAcc(load) = 0.9$, since the load operation accesses memory. The energy consumed by an instruction J can therefore be modeled as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}} \times memAcc(j) + O$.

Tool Integration The required information for the energy model can be provided to the tool through a JSON file. The required fields are:

- name: the name of the CPU in the LLVM framework, used for the program's disassembling.

- freq: the operational frequency of the CPU.
- power: the average power consumed by the CPU.
- iiover: the inter-instruction overhead for the target CPU.
- memacc: the memory access coefficient for the target CPU.
- cpi: the list of instructions in the CPU's instruction set, containing for each instruction the name, the cost (in clocks per instruction) and a boolean field that states if the instruction accesses memory or not.

If either the memory access coefficient or the inter instruction overhead are unknown by a user, they can be set respectively to 0 and 1. An example of a JSON file containing the configuration for the CPU employed during the experimental evaluation can be found in Appendix B.

3.7 Source Code Annotator

The source code annotator is the final component in the pipeline. It takes as input the results of all the components that we have described so far, producing an annotated version of the original source code.

Cost function In the following we will refer to the metric employed to determine the cost as a generic cost function $cost$: LLVM Instruction $\rightarrow \mathbb{R}$, and we assume that, given a trace T of the program execution produced by one of the instrumentations described in section 3.3, the total cost of executing the program, C_p, can be expressed as the sum of the costs of all the instructions in the executed basic blocks: $C_p = \sum_{bb \in T} \sum_{i \in bb} cost(i)$. Some examples of possible cost functions are:

- The LLVM-IR instruction count (the number of executed LLVM-IR instructions), that can be defined as simply $cost_{LLVM}(i) = 1$.
- The assembly instruction count, that, given the definition of $map_{IR \rightarrow Asm}$ provided in 3.4, can be defined as: $cost_{Asm}(i) = |map_{IR \rightarrow Asm}(i)|$, where $|.|$ represents the set cardinality.
- The energy cost, that, given an energy model $energy(j)$, for each assembly instruction j, can be defined as $cost_{energy}(i) = \sum_{j \in map_{IR \rightarrow Ass}(i)} energy(j)$.

Cost Attribution Given the definition of cost function that we have provided, in order to propagate the cost at source code level, we simply have to exploit the debug information contained in the LLVM instructions. Since the execution trace is produced by running the optimized version of the program, the attribution works

by mapping directly from the optimized module to the source code, without the intermediate step of the unoptimized module.

The debug information of the optimized version may be integrated with the propagation technique described in chapter 4. The only caveat is that some instructions, even in the unoptimized module, do not have debug information attached to them. Examples of such instructions are the *alloca* instructions that are used in LLVM-IR to allocate stack memory, or *phi* instructions that, since most of the high-level language are not in SSA form, can rarely be mapped to source code locations.

In order to propagate also the cost of this location-less instructions to the source code, we will assign their cost the function's definition location.

Assigning this cost to the function's definition location is an heuristic approach that allows us to signal to the developer that the cost can be attributed, in general, to the function, even if there is not a precise line to which it can be assigned.

As described in section 3.3, the data regarding the execution of a program can either be a trace (sequence of basic blocks), or the profile (execution count). We have, therefore, to distinguish between attribution with trace and attribution with counts. The pseudo code of the cost attribution with trace is the following (where SCM is a data structure that stores the cost associated to each source location) :

Algorithm 3.6 Source level cost attribution - Trace.

Input Trace T, Cost function *cost*
Output Source Location $\rightarrow \mathbb{R}$ map SCM.

```
1:  for BasicBlock BB ∈ T do
2:      funLoc := BB.getFunction().getLocation()      ▷ Function definition location
3:      for Instruction I ∈ BB do
4:          if I.hasDebugLocation() then
5:              loc := I.getLocation()
6:              SCM[loc] += cost(I)
7:          else
8:              SCM[funLoc] += cost(I)
9:          end if
10:     end for
11: end for
```

The pseudo code for the cost attribution, provided the execution counts of the basic blocks, is shown in Algorithm 3.7. The execution counts are symbolized by a *count* function, that receives a basic block and returns the amount of times it has been executed.

Algorithm 3.7 Source level cost attribution - Counts.

Input Execution Counts *counts*, Cost function *cost*
Output Source Location → ℝ map SCM.

```
1: for BasicBlock BB ∈ M do                    ▷ Iterate over each basic block once
2:     funLoc := BB.getFunction().getLocation()    ▷ Function definition location
3:     for Instruction I ∈ BB do
4:         if I.hasDebugLocation() then
5:             loc := I.getLocation()
6:             SCM[loc] += cost(I) * count(BB)
7:         else
8:             SCM[funLoc] += cost(I) * count(BB)
9:         end if
10:    end for
11: end for
```

Example The following snippet of C code has been commented with the LLVM instruction count associated to each line of code.

```
1  unsigned long fact(int n){ //6.500000e+01 LLVM instr
2    if(n <= 1) //3.900000e+01 LLVM instr
3      return 1; //4.000000e+00 LLVM instr
4    unsigned long a = fact(n-1); //5.500000e+01 LLVM instr
5    return n*a; //6.600000e+01 LLVM instr
6  } //2.600000e+01 LLVM instr
7
8  int main(){
9    fact(3); //1.000000e+00 LLVM instr
10   fact(10); //1.000000e+00 LLVM instr
11 } //1.000000e+00 LLVM instr
```

Please note how both the function calls at line 9 and 10 have a cost of 1 associated to them (simply the *callinstr* of the LLVM-IR).

3.7.1 Callsites Attribution

As we stated in section 3.3, we want to provide a rich source code level visualization, that also associates to each callsite the total cost of the computation performed in the function call. The instrumentation techniques that we have developed ultimately produce a trace that consists in a sequence of entries. Each entry has the form $\langle bb_i, callsites \rangle$, where bb_i is the executed basic block, and $callsites = \langle c_1, ..., c_n \rangle$ are the callsites on the stack when the block was executed. Given such a trace, all we have to do to obtain the cost of the function calls is to assign to each c_j in *callsites* the cost of executing all the instructions in bb_i. Of course, *CountInstr* (3.3.4) cannot be employed for callsite attribution, due to the fact that it stores only the execution counts of each basic block, and therefore we lose information about the actual path which led to the execution of the blocks, and so we cannot reconstruct the callsites

stack as we do with *SimpleInstr* and the Trace expansion algorithm.

Recursive Functions Recursive functions must be treated slightly differently. A recursive function is a function that (directly or indirectly) calls itself. Therefore, when executing a basic block that belongs to the recursive function, the *callsites* vector will likely contain several times the callsite corresponding to the recursive call. If we simply assign the cost of the instructions to all the callsites, the location corresponding to the recursive call will have a disproportionate cost associated to it, for instance:

```
1  unsigned long fact(int n){ //6.500000e+01 LLVM instr
2    if(n <= 1) //3.900000e+01 LLVM instr
3      return 1; //4.000000e+00 LLVM instr
4    unsigned long a = fact(n-1); //9.640000e+02 LLVM instr
5    return n*a; //6.600000e+01 LLVM instr
6  } //2.600000e+01 LLVM instr
7
8  int main(){
9    fact(3); //5.500000e+01 LLVM instr
10   fact(10); //2.020000e+02 LLVM instr
11 } //1.000000e+00 LLVM instr
```

Here we see that the recursive call at line 4 has 969 LLVM instructions assigned to it, whereas the two calls at line 9 and 10, have a total cost of (202+55) 207 LLVM instructions. This means the the cost of executing the instructions the *fact* function has been counted too many times. We want to ensure that, for every function, the sum of the costs associated to locations in the function equals the sum of the costs associated to calls to that function. This is trivially satisfied for non recursive functions. For recursive functions, instead, we need to not assign the cost to the callsite, but treat the function call as a normal instruction, assigning it the cost of, for instance, just 1 LLVM instruction.

Call Graph In order to achieve this result we have to automatically identify recursive calls in the program. To so so, we build the *Call Graph*: a graph where each node represents a function, and there is an edge $\langle i, j \rangle$ if function i calls function j. Each edge is annotated with the source location corresponding to the function call, and since a function may call another function in multiple locations, this is a *multigraph*: a graph where two nodes may be connected by more than one edge.

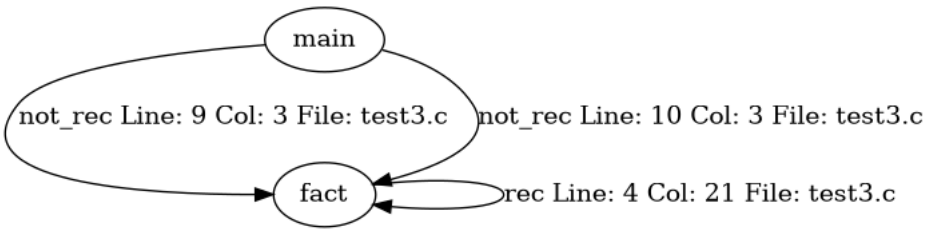

Figure 3.4: Example of call graph.

Figure 3.4 provides an example of a call graph. Edges corresponding to recursive calls are marked as "rec". A call is marked as recursive if the corresponding edge in the call graph is within a cycle. This can be easily checked by performing a depth first traversal of the graph. The pseudo code of the cost attribution with callsites is the following:

Algorithm 3.8 Source level cost attribution with callsites.

Input Trace T, Cost function *cost*, Call Graph CG
Output Source Location $\to \mathbb{R}$ map SCM.

```
1: for ⟨bb_i, callsites⟩ ∈ T do
2:     funLoc := bb_i.getFunction().getLocation()      ▷ Function definition location
3:     for Instruction I ∈ BB do
4:         if I.hasDebugLocation() then      ▷ Same cost attribution as in without callsite
5:             loc := I.getLocation()
6:             SCM[loc] += cost(I)
7:         else
8:             SCM[funLoc] += cost(I)
9:         end if
10:        for callLocation ∈ callsites do                    ▷ Callsite attribution
11:            if not isRecursive(callLocation, CG) then
12:                SCM[callLocation] += cost(i)
13:            end if
14:        end for
15:    end for
16: end for
```

Example The following snippet shows the callsite attribution with the exclusion of recursive calls.

```
1 unsigned long fact(int n){ //6.500000e+01 LLVM instr
2   if(n <= 1) //3.900000e+01 LLVM instr
```

```
3       return 1; //4.000000e+00 LLVM instr
4     unsigned long a = fact(n-1); //5.500000e+01 LLVM instr
5     return n*a; //6.600000e+01 LLVM instr
6   } //2.600000e+01 LLVM instr
7
8   int main(){
9     fact(3); //5.500000e+01 LLVM instr
10    fact(10); //2.020000e+02 LLVM instr
11  } //1.000000e+00 LLVM instr
```

As we can easily check, the sum of the costs of the locations associated to the $fact$ function equals the sum of the costs of the calls to the function. This heuristic allows us to achieve the previously specified goal: the sum of the costs associated to locations in the function equals the sum of the costs associated to calls to the function.

Chapter 4

Tracing and Visualizing the Impact of Compiler Optimizations

As we mentioned in section 1.1.5, during the compilation, a program undergoes some changes due to operations performed by transformation passes. This changes obfuscate the mapping between LLVM instructions and source code locations, therefore, in order to correctly attribute energy costs to the program's source code, we need to:

- Assess the impact of the effects of compiler optimizations on the LLVM → source code mapping.
- Eventually, try to complete the mapping when the transformations lead to a loss of information, by means of the heuristic approach that we have developed.

We have therefore developed a tool that allows to automatically propagate debug information through the passes pipeline, while also allowing developers to visualize the effects of the optimizations on the module. The tool is implemented on top of the LLVM optimizer [1].

Debug Information vs Metadata In the following section, and also in this whole work, we make heavy usage of debug information. Debug information is designed with a precise objective in mind: allow for source code level debugging. The way in which we employ debug information, instead, concerns only the correspondence between instructions and source locations: we want to track down the (source code) origin of each LLVM instruction. The two goals largely overlap, but there are some discrepancies, largely discussed in 1.2. This means the semantics that, in this work, we attribute to debug information, differ slightly from their usual semantics. Therefore, defining a new form of metadata that completely matches our need would have

[1] http://llvm.org/docs/CommandGuide/opt.html

been appropriate. Nevertheless, since more often than not our goals match, we have decided to keep employing debug information, and the definition of proper metadata classes has been left for future work.

Overview Our analysis of the effects of compiler optimizations is based on the observation that each pass may greatly change the structure of the module, but, ultimately, the operations that it performs can be decomposed in a sequence of simple and atomic operations performed on single instructions. Instructions are either created and inserted in the module, removed from the module or moved from a position in the module to a new one. Creating and removing often has the purpose of replacing one or more instruction with a new one.
The tool basically provides a log of the changes performed by the transformation, expressing them in terms of the aforementioned atomic operations. It is implemented by assigning to each instruction an unique identifier. This identifier represents a given instruction *in a given position in the module*: if the instruction is moved, the identifier is changed to a new one. This means we cannot employ the memory address of the instruction as identifier (as it is usually done in LLVM, where instructions are address-comparable), but we have to add a new metadata node to the instruction. The entries of the log produced by the tool use these identifiers to refer to the instructions in the module. The following snippet provides an example of a module where an identifier metadata (!ID) has been assigned to each instruction (all the other metadata have been omitted for brevity).

```
1  define dso_local i32 @main() #0 !dbg !8 {
2    %1 = alloca i32, align 4, !ID !12
3    call void @llvm.dbg.declare(metadata i32* %1, metadata !13, metadata
         !DIExpression()), !dbg !14, !ID !15
4    store i32 10, i32* %1, align 4, !dbg !16, !ID !17
5    %2 = load i32, i32* %1, align 4, !dbg !18, !ID !19
6    %3 = add nsw i32 %2, 1, !dbg !18, !ID !20
7    store i32 %3, i32* %1, align 4, !dbg !18, !ID !21
8    ret i32 0, !dbg !22, !ID !23
9  }
10
11
12 !12 = !{i64 0}
13 !15 = !{i64 1}
14 !17 = !{i64 2}
15 !19 = !{i64 3}
16 !20 = !{i64 4}
17 !21 = !{i64 5}
18 !23 = !{i64 6}
```

Implementation The tool is implemented as a patch to the LLVM code base. The log is produced by modified versions of the LLVM APIs: the methods for creating, removing and moving instructions have been patched in order to, besides doing the original operation, signal that they are performing the action. This allows to obtain a list of events containing some, but not all, the operations performed by a pass. Some events, in order to be logged, required the inspection of the actual source code of the passes. The patch to LLVM is described in more detail in appendix A. Besides modifying the LLVM API and some of the passes, we also implemented two simple utility passes that allow to create the log of the transformations: a pass that adds the unique identifier metadata to the instructions, and a pass that outputs which pass is going to be executed next. This second pass is necessary in order to automatically understand which pass performed the logged changes.

Example Consider the snippet of C code shown in section 3.3, and the corresponding LLVM-IR translation, where each instruction has been commented with the corresponding ID:

```
1  define dso_local i32 @fact(i32 %0) #0 !dbg !8 {
2    call void @llvm.dbg.value(metadata i32 %0, metadata !12, metadata !
         DIExpression()), !dbg !13, !ID !14 ;0
3    %2 = icmp sle i32 %0, 1, !dbg !15, !ID !17 ;1
4    br i1 %2, label %3, label %4, !dbg !18, !ID !19 ;2
5
6  3:                                                ; preds = %1
7    br label %8, !dbg !20, !ID !21 ;3
8
9  4:                                                ; preds = %1
10   %5 = sub nsw i32 %0, 1, !dbg !22, !ID !23 ;4
11   %6 = call i32 @fact(i32 %5), !dbg !24, !ID !25 ;5
12   %7 = mul nsw i32 %0, %6, !dbg !26, !ID !27 ;6
13   br label %8, !dbg !28, !ID !29 ;7
14
15 8:                                                ; preds = %4, %3
16   %.0 = phi i32 [ 1, %3 ], [ %7, %4 ], !dbg !13, !ID !30 ;8
17   ret i32 %.0, !dbg !31, !ID !32 ;9
18 }
```

We want to turn this recursive implementation of the factorial function into an iterative one, by running compiler optimizations. This can be achieved by running the *tailcallelim* pass, that will remove the recursive call (line 11, id 5), replace it with a branch instruction, and add an accumulator variable that stores the result of the computation. The result of the optimization is the following

```
1  define dso_local i32 @fact(i32 %0) #0 !dbg !8 {
2    br label %tailrecurse, !dbg !12, !ID !13 ;12
3
4  tailrecurse:                                      ; preds = %4, %1
```

```
5    %accumulator.tr = phi i32 [ 1, %1 ], [ %6, %4 ], !ID !14 ;16
6    %.tr = phi i32 [ %0, %1 ], [ %5, %4 ], !ID !15 ;13
7    call void @llvm.dbg.value(metadata i32 %.tr, metadata !16, metadata !
         DIExpression()), !dbg !17, !ID !18 ;0
8    %2 = icmp sle i32 %.tr, 1, !dbg !19, !ID !21 ;1
9    br i1 %2, label %3, label %4, !dbg !22, !ID !23 ;2
10
11 3:                                                ; preds = %
       tailrecurse
12   br label %7, !dbg !24, !ID !25 ;3
13
14 4:                                                ; preds = %
       tailrecurse
15   %5 = sub nsw i32 %.tr, 1, !dbg !26, !ID !27 ;4
16   %6 = mul nsw i32 %.tr, %accumulator.tr, !dbg !28, !ID !29 ;6
17   br label %tailrecurse, !dbg !12, !ID !30 ;17
18
19 7:                                                ; preds = %3
20   %accumulator.ret.tr = mul nsw i32 1, %accumulator.tr, !dbg !28, !ID
         !31 ;18
21   ret i32 %accumulator.ret.tr, !dbg !32, !ID !33 ;9
22 }
```

As we can see, the pass has inserted two phi nodes in order to disambiguate the values of the accumulator variable and the loop counter (id 16 and 13), has removed the function call and has added a branch instruction (id 17). The log of this transformation is the following:

```
Running: Tail Call Elimination
Replacing i64 8 with value i32 1
Removing i64 8
Removing i64 7
Creating i64 12
Creating i64 13
Replacing Argument i32 %0 with i64 13
Creating i64 14
Creating i64 15
Creating i64 16
Replacing operand in i64 6 from i64 5 with i64 16
Creating i64 17
Removing i64 5
Removing i64 14
Removing i64 15
Creating i64 18
```

The log clearly shows which instructions have been created or removed, and it also

shows that in instruction 6 (that in the recursive version corresponds to the multiplication n*fact(n-1)), the result of the function call (id 5) has been replaced with the content of the accumulator variable (id 16, line 5).

4.1 Visualization

Our goal is also to provide developers with a view of how their code have been optimized. This is achieved by inspecting the compiler transformation's log, and annotating the textual representation of the module with the information found in the log. For each pass that has been run, we provide a view of the module before and after the pass, marking in red instructions that have been removed, and in green instructions that have been inserted. When instructions have been moved or replaced, we annotate the instruction in order to signal it. The tool outputs a set of HTML pages, that can be easily opened with any web browser. Each HTML page contains a navigation bar that lists all the passes that have been run. Passes are identified using the name provided by the LLVM APIs, a numeric identifier is added since a pass may run multiple times in the optimization pipeline. Figure 4.2 shows an example of the contents of the navigation bar, while figure 4.1 shows the differential view of a module.

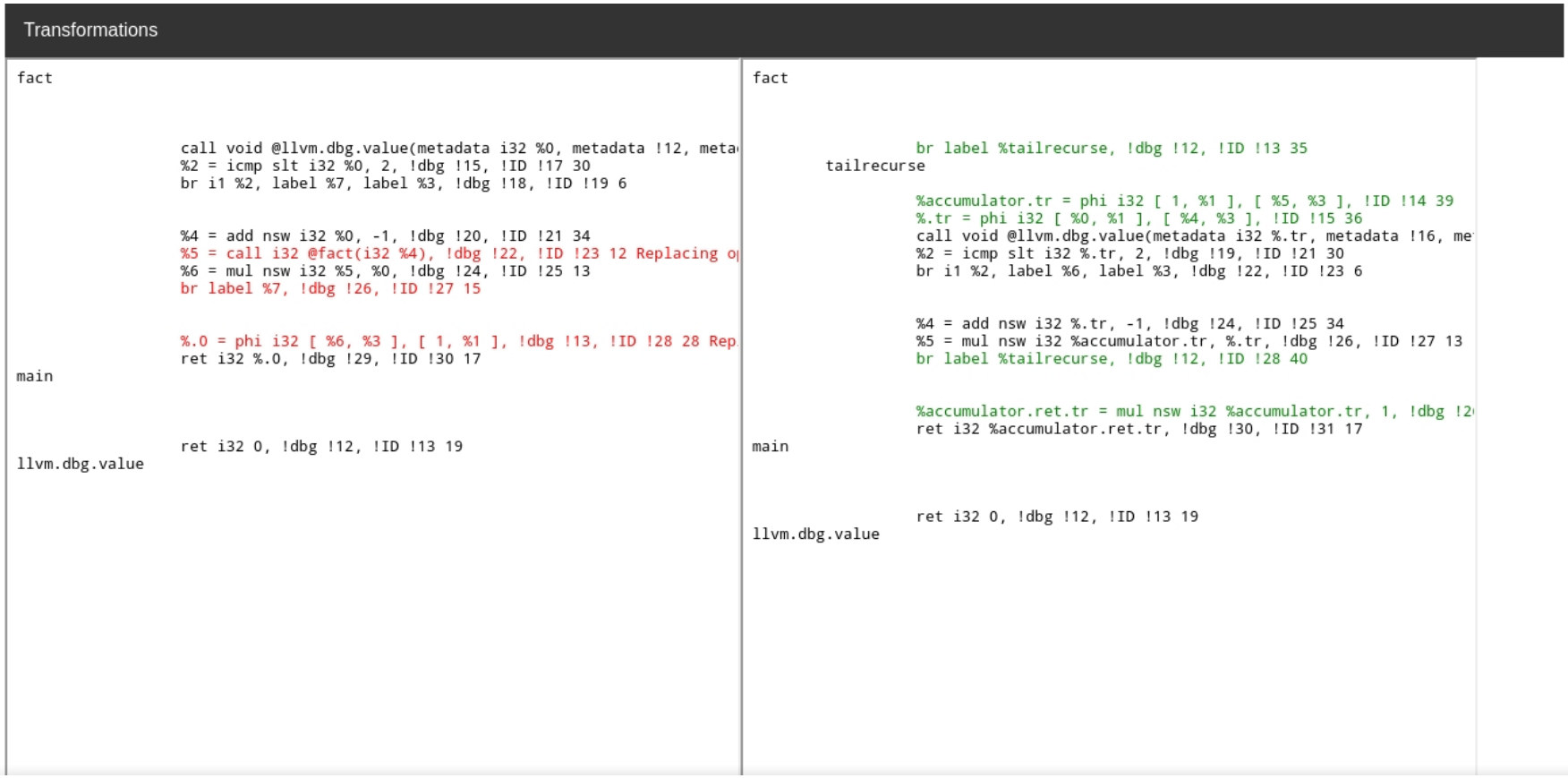

Figure 4.1: Differential view of a module.

Figure 4.2: List of the scheduled passes.

4.2 Debug Location Propagation

The second goal of this tool is to automatically propagate debug locations across passes of the optimization pipeline. As we stated in section 1.2, this is largely already done by LLVM. The main result that we wanted to achieve is to not completely override the handling of debug information that has already been implemented in LLVM, but to overcome the discrepancies between the "standard" usage of debug information and our purposes.

The key insight that allows us to propagate locations is that if instruction A is replaced by (or moved to) instruction B, then instruction B should have the debug location of instruction A. As we already mentioned, this is not always true when employing debug information for source level debugging, but is it always true for our purposes, since we want to track down the source code origin of all the instructions in a module.

Our debug location propagation is performed in two steps:

1. Build a graph (RepGraph) that represents the "replaced by" relation: the graph has one node for each identifier, and an edge $\{i, j\}$ if instruction i is replaced by or moved to instruction j.

2. Use algorithm 4.1 to propagate debug locations across the graph.

To each node i of the RepGraph, we associate a function $locations(i)$, that returns a set of debug locations that constitute the origin of instruction i. That fact that the function returns a set (instead of a single location) allows us to handle the merge of instructions, described in section 1.2, by associating to the new instruction the union of the two original instructions.

The following is the algorithm used to perform location propagation. The *topologicalSort* function performs a topological sort of a graph g, providing an ordering of the nodes of g such that if $\{i, j\} \in g$, then i precedes j in the ordering. This ensures that locations must be updated only once for each edge in the graph, so the computational complexity of the algorithm is $\mathcal{O}(|V| + |E|)$, the cost of the topological sort.

Algorithm 4.1 Debug location propagation.

Input RepGraph g
Output RepGraph g with propagated debug information.

```
1: nodes := topologicalSort(g)
2: for i ∈ nodes do
3:     for each edge {i, j} ∈ g do
4:         locations(j) := locations(j) ⋃ locations(i)
5:     end for
6: end for
```

Example - Tail Call Elimination Consider the tail call optimization presented in the previous example. The phi-node for the accumulator variable in the optimized version (line 5, id 16) has no debug information attached (there's no !dbg node associated to it), but it replaces the result of the recursive function call, therefore we would like to assign its cost to the source code location of the function call. The log produced by the transformation contains the entry

```
Replacing operand in i64 6 from i64 5 with i64 16
```

Therefore, the RepGraph will contain an edge $5 \rightarrow 16$, and we will assign the debug location of instruction 5 to instruction 16. Propagating locations after a tail call optimization is particularly significant for the source code attribution, since without the propagation we would totally disregard the contribution of the recursive calls.

Example - Loop Invariant Code Motion The Loop Invariant Code Motion (LICM) pass performs a transformation that brings outside a loop all the instructions whose value remains unchanged among the loop iterations. It provides an example of the *hoisting* of an instruction into a preceding basic block. Consider the following snippet, where only relevant metadata has been included:

```
1  define void @testfunc(i32 %i) !dbg !7 {
2    br label %Loop, !dbg !16, !ID !17
3
4  Loop:                                             ; preds = %Loop, %0
5    %j = phi i32 [ 0, %0 ], [ %Next, %Loop ], !dbg !18, !ID !19
6    %i2 = mul i32 %i, 17, !dbg !21, !ID !22
7    %Next = add i32 %j, %i2, !dbg !24, !ID !25
8    %cond = icmp eq i32 %Next, 0, !dbg !27, !ID !28
9    br i1 %cond, label %Out, label %Loop, !dbg !30, !ID !31
10
11 Out:                                              ; preds = %Loop
12   ret void, !dbg !32, !ID !33
13 }
14
15 !21 = !DILocation(line: 3, column: 1, scope: !7)
16 !22 = !{i64 3}
```

The *mul* instruction at line 6 is a loop invariant, therefore it can be hoisted in the predecessor (at line 2):

```
1  define void @testfunc(i32 %i) !dbg !7 {
2    %i2 = mul i32 %i, 17, !dbg !16, !ID !17
3    br label %Loop, !dbg !18, !ID !19
4
5  Loop:                                             ; preds = %Loop, %0
6    %j = phi i32 [ 0, %0 ], [ %Next, %Loop ], !dbg !20, !ID !21
7    %Next = add i32 %j, %i2, !dbg !25, !ID !26
8    %cond = icmp eq i32 %Next, 0, !dbg !28, !ID !29
```

```
9     br i1 %cond, label %Out, label %Loop, !dbg !31, !ID !32
10
11  Out:                                              ; preds = %Loop
12    ret void, !dbg !33, !ID !34
13  }
14
15  !16 = !DILocation(line: 0, scope: !7)
16  !17 = !{i64 11}
```

As we can see, the line entry of its debug location has been set to 0, that by convention means "unknown", and the id of the instruction has changed. The log will therefore contain the entry

```
Moving i64 3 to i64 11
```

And the corresponding RepGraph will have the edge $3 \rightarrow 11$, allowing to propagate the original debug location.

Example - Simplify CFG The simplify CFG transformation pass simplifies the structure of the Control Flow Graph, merging basic blocks when some conditions are met. In the example in Figure 4.3, basic block b performs the same operations as basic block a, it is therefore removed, and all the uses of the instructions in b are replaced with the corresponding instructions in a. The log of the transformations contains an entry for each replacing that occurred, this allows us to assign to the instructions in basic block a not only their original source location, but also the locations of the instructions in block b that they replace.

```
test2(i64 %i0, i64 %i1)

    entry

            %and.i.i = and i64 %i0, 281474976710655, !ID !1 5
            %and.i11.i = and i64 %i1, 281474976710655, !ID !2 6
            %or.cond = icmp eq i64 %and.i.i, %and.i11.i, !ID !3 7
            br i1 %or.cond, label %c, label %a, !ID !4 8
    a

            %shr.i4.i = lshr i64 %i0, 48, !ID !5 9
            %and.i5.i = and i64 %shr.i4.i, 32767, !ID !6 10
            %shr.i.i = lshr i64 %i1, 48, !ID !7 11
            %and.i2.i = and i64 %shr.i.i, 32767, !ID !8 12
            %cmp9.i = icmp ult i64 %and.i5.i, %and.i2.i, !ID !9 13
            br i1 %cmp9.i, label %c, label %b, !ID !10 14
    b

            %shr.i13.i9 = lshr i64 %i1, 48, !ID !11 15 Replacing 15 11
            %and.i14.i10 = and i64 %shr.i13.i9, 32767, !ID !12 16 Replacing 16 12
            %shr.i.i11 = lshr i64 %i0, 48, !ID !13 17 Replacing 17 9
            %and.i11.i12 = and i64 %shr.i.i11, 32767, !ID !14 18 Replacing 18 10
            %phitmp = icmp uge i64 %and.i14.i10, %and.i11.i12, !ID !15 19
            br label %c, !ID !16 20
    c

            %o2 = phi i1 [ false, %a ], [ %phitmp, %b ], [ false, %entry ], !ID !17
            ret i1 %o2, !ID !18 22
```

```
i1 test2(i64 %i0, i64 %i1)

    entry

            %and.i.i = and i64 %i0, 281474976710655, !ID !1 5
            %and.i11.i = and i64 %i1, 281474976710655, !ID !2 6
            %or.cond = icmp eq i64 %and.i.i, %and.i11.i, !ID !3 7
            br i1 %or.cond, label %c, label %a, !ID !4 8
    a

            %shr.i4.i = lshr i64 %i0, 48, !ID !5 9
            %and.i5.i = and i64 %shr.i4.i, 32767, !ID !6 10
            %shr.i.i = lshr i64 %i1, 48, !ID !7 11
            %and.i2.i = and i64 %shr.i.i, 32767, !ID !8 12
            %cmp9.i = icmp ult i64 %and.i5.i, %and.i2.i, !ID !9 13
            %phitmp = icmp uge i64 %and.i2.i, %and.i5.i, !ID !10 33
            %not.cond = xor i1 %cmp9.i, true, !ID !11 34
            %and.cond = and i1 %not.cond, %phitmp, !ID !12 35
            br label %c, !ID !13 36
    c

            %o2 = phi i1 [ %and.cond, %a ], [ false, %entry ], !ID !14 21
            ret i1 %o2, !ID !15 22
```

Figure 4.3: Simplify CFG example.

4.3 Final Remarks

This section described a technique to reconstruct a mapping from optimized LLVM-IR to source code that is as complete as possible. Our main purpose was to overcome the discrepancies between the common usage of debug information and our goal (tracking the origin of LLVM instructions). While the debug info propagation part of this work is quite tied to out purposes, the logging of the changes performed by a pass, and the consequent visualization that we have built on top of it, is more general and could be useful to anyone who is interested in understanding how a module has been optimized.

There are similar efforts coming from the LLVM developers community, aiming at highlighting the changes occurred during optimization. They are mainly based on *diff-ing* between the string representation of a module, in a similar fashion to how, for example, version control tools check how a file has been modified. They have the advantage of being "lighter" since they do not require to modify the core LLVM APIs, but they are less powerful since they only allow to visualize the changes, whereas we needed a deeper understanding of the transformations in order to update the debug locations.

Chapter 5

Experimental Evaluation

This chapter will describe the experimental evaluation that we have conducted in order to assess and improve the accuracy of the energy model. During the experimental evaluation we have collected samples of the energy consumption of an embedded system running a set of benchmarks, comparing them to the estimate provided by our tool. We have followed an iterative approach, improving the energy model after analyzing the results of one experiment, and repeating the estimation process in order to assess the accuracy of the improved model.

5.1 Experimental Setup

In this section we provide an overview of how the experiments have been performed, detailing the selection of the employed benchmarks, the employed measuring instruments and target hardware architecture.

Benchmark Selection In order to evaluate the accuracy of the energy model, we have selected Polybench [37] as benchmark suite. It consist of a set of thirty benchmarks extracted from applications in various application domains, ranging from linear algebra to physics simulations and dynamic programming. They are all written in C and allow for compile time tuning of several parameters, such as the problem size (e.g. size of the matrices used in the computations or numbers of iterations) or stack allocations versus heap allocation.

Target Hardware Evaluation has been performed on an STM32F407VGT6 Discovery board. It is equipped with an ARM Cortex M4 CPU, with maximum frequency of 168 Mhz, and 512 Kbytes of flash memory and 192 Kbytes of SRAM. Appendix B provides the full characterization of the board, with average power, frequency and clocks per instruction.

Operating System The benchmarks have been executed by means of the Miosix OS kernel [18]. This GPL-licensed operating system allows to run both C and C++ programs, provides standard libraries and support for POSIX thread API, and, more importantly, gives developers access to the whole compilation pipeline, allowing us to employ our compiler passes in order to obtain the energy consumption estimation.

Measuring Instruments In order to directly measure the energy consumed by a benchmark, we have employed the Otii Arc power analyzer [1]. This product acts as the power supply for the board, sampling the voltage and the drawn current. It also automatically computes the power consumed at each sample, and integrates in order to provide the total energy consumed since the beginning of the measurement.

It allows to sample current with an accuracy of $\pm(0.1\% + 50$ nA), at a sample rate of 1 ksps, and voltage with an accuracy of $\pm(0.1\% + 1.5$ mV), at a sample of 1 ksps.

5.2 Analysis and Evaluation

In this section we present measurement data and estimated value for the different energy models that we have provided. The error has been computed as $\epsilon = |measure - estimate|/measure$.

Clock Cycles Based We remind that the clock cycle based model estimates the energy consumed by an assembly instruction j as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}}$. Table 5.1 contains the raw data, Figure 5.1 summarizes the comparison between measure and estimate. While the two values exhibit an high correlation, the average error is quite high, with a value of 0.61051.

[1] https://www.qoitech.com/otii

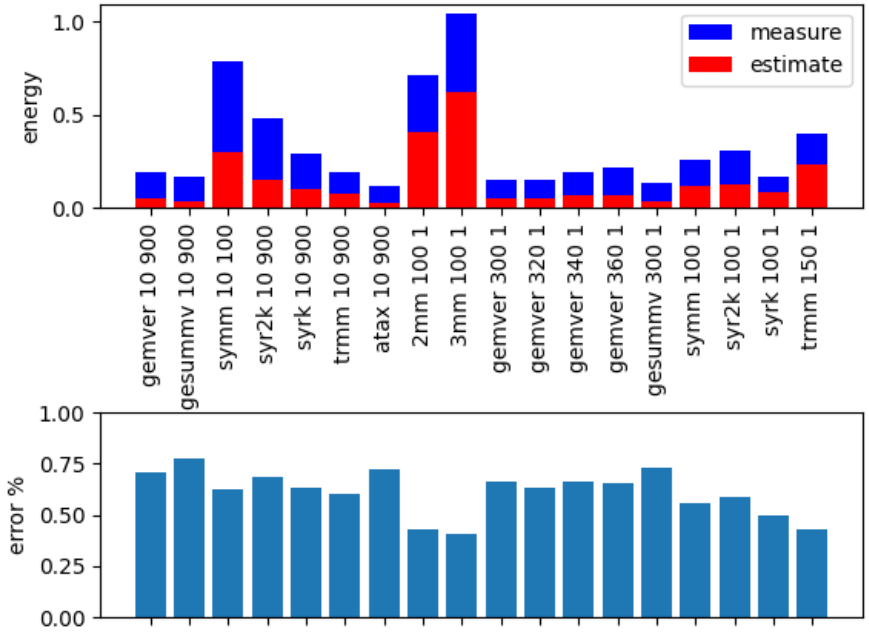

Figure 5.1: Measure and estimate for the clock cycle based model.

Name	Size	Iterations	Measure	Estimate	Error
gemver	10	900	0.1903	0.0552	0.7100
gesummv	10	900	0.1715	0.0389	0.7734
symm	10	100	3.1468	1.1917	0.6213
syr2k	10	900	0.4853	0.1535	0.6838
syrk	10	900	0.2887	0.1064	0.6315
trmm	10	900	0.1922	0.0769	0.5997
atax	10	900	0.1148	0.0317	0.7239
2mm	100	1	0.7172	0.4115	0.4263
3mm	100	1	1.0445	0.6194	0.4070
gemver	300	1	0.1512	0.0509	0.6636
gemver	320	1	0.1531	0.0562	0.6329
gemver	340	1	0.1922	0.0653	0.6603
gemver	360	1	0.2144	0.0732	0.6586
gesummv	300	1	0.1382	0.0375	0.7285
symm	100	1	0.2634	0.1164	0.5581
syr2k	100	1	0.3061	0.1265	0.5869
syrk	100	1	0.1677	0.0842	0.4979
trmm	150	1	0.4022	0.2310	0.4256

Table 5.1: Data for the clock cycles based energy model.

Inter Instruction Overhead The inter instruction overhead model introduces a constant that accounts for the switching activity of the CPU. The energy is estimated as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}} + O$. As we can see from the data reported in table 5.2 and Figure 5.2, the model constitutes a good improvement over the one based exclusively on clock cycles. The average error is 0.417.

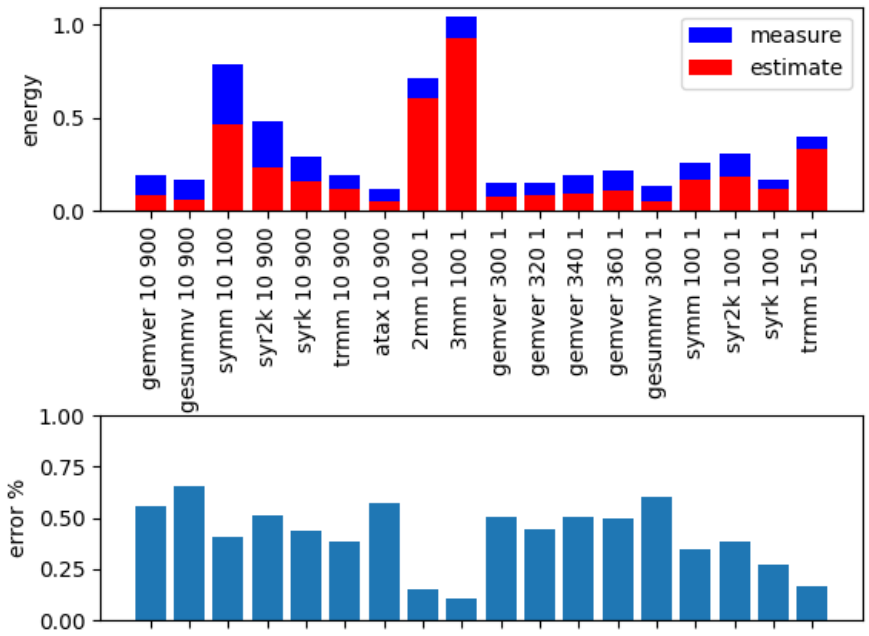

Figure 5.2: Plot for the inter instruction overhead model.

Name	Size	Iterations	Measure	Estimate	Error
gemver	10	900	0.1903	0.0839	0.5591
gesummv	10	900	0.1715	0.0587	0.6579
symm	10	100	3.1468	1.8579	0.4096
syr2k	10	900	0.4853	0.2377	0.5101
syrk	10	900	0.2887	0.1633	0.4344
trmm	10	900	0.1922	0.1182	0.3849
atax	10	900	0.1148	0.0494	0.5697
2mm	100	1	0.7172	0.6081	0.1522
3mm	100	1	1.0445	0.9324	0.1073
gemver	300	1	0.1512	0.0746	0.5065
gemver	320	1	0.1531	0.0849	0.4458
gemver	340	1	0.1922	0.0958	0.5017
gemver	360	1	0.2144	0.1074	0.4992
gesummv	300	1	0.1382	0.0547	0.6046
symm	100	1	0.2634	0.1726	0.3447
syr2k	100	1	0.3061	0.1890	0.3825
syrk	100	1	0.1677	0.1223	0.2704
trmm	150	1	0.4022	0.3344	0.1684

Table 5.2: Data for the inter instruction overhead energy model.

Memory Access Coefficient The memory access coefficient accounts for the fact that memory access instructions lead to a lower current drawn by the CPU, as exemplified in Figure 5.4, where the gap between the current drawn in the initialization and the computation phase of the benchmark is very visible. The memory access coefficient model estimates the consumed energy as $energy(j) = \frac{\overline{P} * cpi(j)}{f_{clk}} \times memAcc(j) + O$. It provides a further improvement over the inter instruction coefficient-based model, with an average error of 0.3931.

Estimate Upper Bound The analysis that we have conducted so far highlights two main characteristics of the error of out estimation: it exhibits a quite high variance, ranging from 5% to 64%, but it consistently always underestimates the energy consumption. From the second observation, it follows that our estimate provides a lower bound, and we can also introduce a multiplying factor α such that $estimate \times \alpha$ constitutes an upper bound of the true value of the energy consumption. Figure 5.5 shows the plot of the estimate and the upper bound for the memory access coefficient model, with $\alpha = 3.0$.

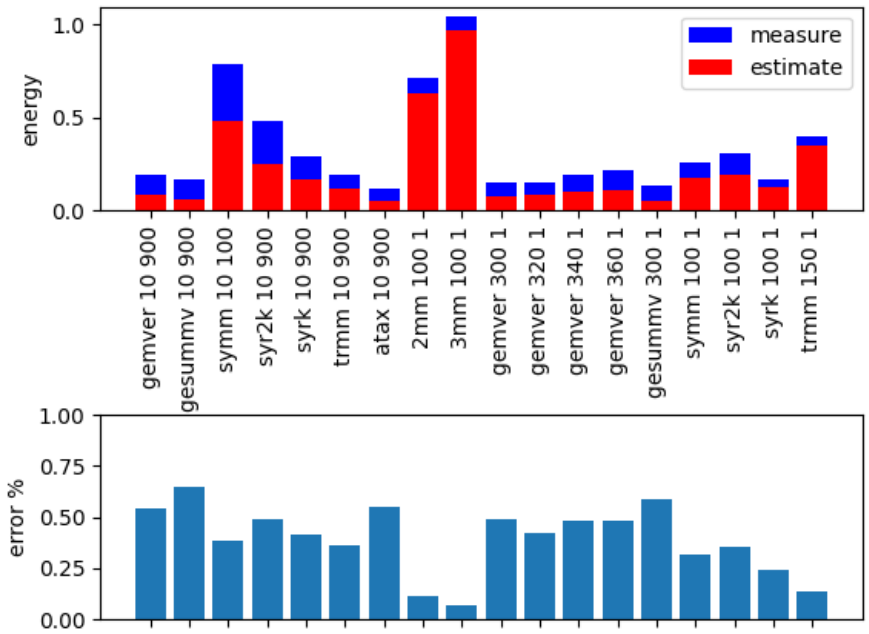

Figure 5.3: Plot for the memory access coefficient model.

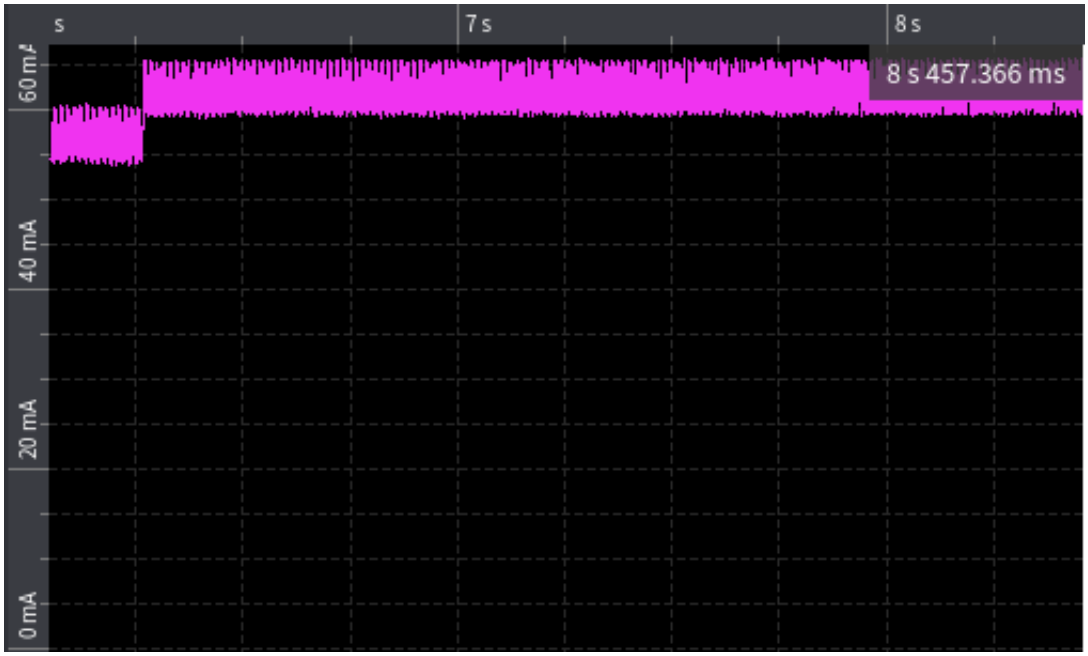

Figure 5.4: Example of the current profile of a benchmark.

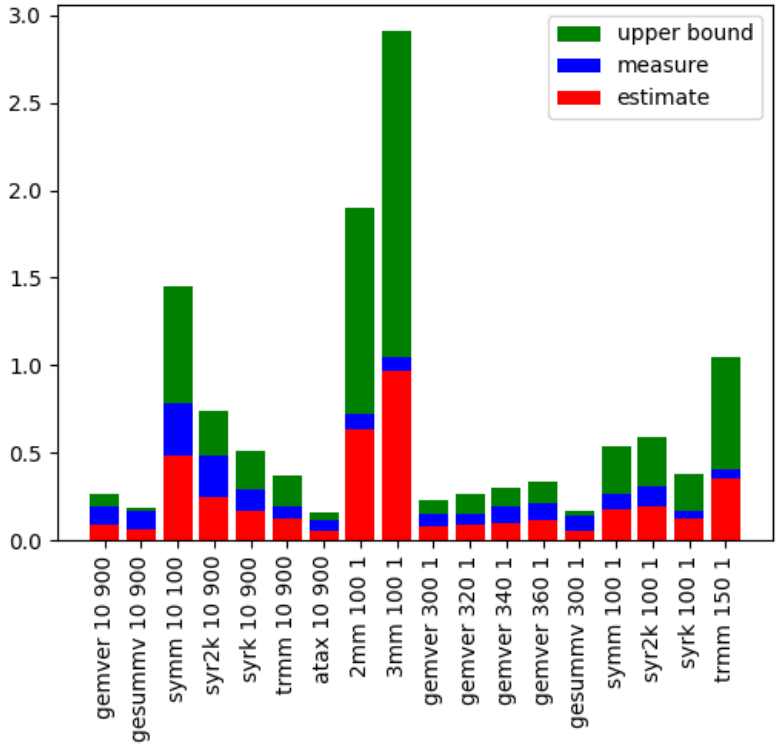

Figure 5.5: Plot of the estimate upper bound.

Name	Size	Iterations	Measure	Estimate	Error
gemver	10	900	0.1903	0.0872	0.5416
gesummv	10	900	0.1715	0.0610	0.6443
symm	10	100	3.1468	1.9355	0.3849
syr2k	10	900	0.4853	0.2474	0.4902
syrk	10	900	0.2887	0.1698	0.4117
trmm	10	900	0.1922	0.1230	0.3599
atax	10	900	0.1148	0.0513	0.5530
2mm	100	1	0.7172	0.6339	0.1162
3mm	100	1	1.0445	0.9719	0.0696
gemver	300	1	0.1512	0.0776	0.4868
gemver	320	1	0.1531	0.0883	0.4237
gemver	340	1	0.1922	0.0996	0.4818
gemver	360	1	0.2144	0.1116	0.4792
gesummv	300	1	0.1382	0.0568	0.5888
symm	100	1	0.2634	0.1800	0.3166
syr2k	100	1	0.3061	0.1971	0.3562
syrk	100	1	0.1677	0.1275	0.2396
trmm	150	1	0.4022	0.3487	0.1328

Table 5.3: Data for the memory access coefficient energy model.

5.3 Sources of Error

In this section we analyze the main factors that contribute to the error of our estimate.

5.3.1 Measuring Errors

The Otii Arc Power analyzer was sufficiently accurate for our purposes, but it exhibits a transitory period at the beginning of the measurement that would have partially invalidated the results. To overcome this problem, we have added an idle period of time at the beginning of each benchmark, and we start subtract the energy consumed in the idle period to the total energy measured by the instrument. The board that we have employed for the evaluation also reaches an idle state after the execution of the benchmark. In this state the board keeps drawing current and therefore we have to manually inspect the plot of the drawn current provided by the Otii Arc, and identify the beginning and end points of the actual benchmark. This has led to a noisier measure.

5.3.2 Modeling Errors

Our goal when defining the energy model was to allow developers to set its parameters using as much readily available information as possible. We have therefore employed clock cycles, which are usually already available, and then added only two

parameters that need an experimental evaluation in order to be determined. The final result is an energy model that cannot capture all the factors that have an impact on the energy consumption. Comparing our energy model with the models described in chapter 2, we do not take into account variables such as the values of the operands of an instruction, and the introduction of the inter instruction overhead as a single coefficient simplifies the modeling of the switching activity, whereas other contributions have employed either different coefficients depending of the pair of instructions, groups of instructions or as functions of the Hamming distance between the instruction's encoding.

Another assumption that we have made is that the clock cycles needed to complete an instruction are always constant. This is in general not true, since effects such as cache misses or pipeline flushes may delay the completion of an instruction. The ARM Cortex M4 CPU that we have employed does not have a cache, but instructions such as branches or arithmetical operations on the program counter may lead to a pipeline flush, that may change the total cost of the instruction from 1 up to 4 clock cycles.

Conclusions

In this thesis we have presented the prototype of a tool that allows to estimate and visualize the energy consumption of a program running on an embedded system. We have shown that it is possible to develop frameworks that enable software developers to better understand which components of their program are more responsible for the estimated consumption, without the need of an experimental setup.

We have also proposed a methodology that allows to explore the transformation phase of the compilation, allowing users to understand how their code has been optimized and improving the accuracy of mapping between source code locations and low level instructions.

We assessed the accuracy of our estimate through experimental evaluation, showing that out energy model provides useful bounds for the energy consumption. We have decided to employ a simple energy model, that can be built using mostly readily available information. More complex models may be employed in future works, increasing either the accuracy of the estimate, or targeting more complex hardware architectures.

Bibliography

[1] Anders SG Andrae and Tomas Edler. "On global electricity usage of communication technology: trends to 2030". In: *Challenges* 6.1 (2015), pp. 117–157.

[2] *ARM Cortex M4 technical reference manual*. URL: https://developer.arm.com/documentation/100166/0001/Programmers-Model/Instruction-set-summary/Table-of-processor-instructions?lang=en.

[3] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: an infrastructure for computer system modeling". In: *Computer* 35.2 (2002), pp. 59–67. DOI: 10.1109/2.982917.

[4] C. Brandolese. "Source-Level Estimation of Energy Consumption and Execution Time of Embedded Software". In: *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*. 2008, pp. 115–123. DOI: 10.1109/DSD.2008.43.

[5] C. Brandolese, S. Corbetta, and W. Fornaciari. "Software energy estimation based on statistical characterization of intermediate compilation code". In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. 2011, pp. 333–338. DOI: 10.1109/ISLPED.2011.5993659.

[6] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 83–94. ISBN: 1581132328. DOI: 10.1145/339647.339657. URL: https://doi.org/10.1145/339647.339657.

[7] *Cambridge Bitcoin Electricity Consumption Index*. URL: https://cbeci.org/.

[8] Javier Corral-Garcia et al. "Analysis of Energy Consumption and Optimization Techniques for Writing Energy-Efficient Code". In: *Electronics* 8.10 (2019), p. 1192.

[9] M. Couto et al. "GreenDroid: A tool for analysing power consumption in the android ecosystem". In: *2015 IEEE 13th International Scientific Conference on Informatics*. 2015, pp. 73–78. DOI: 10.1109/Informatics.2015.7377811.

[10] *Energy Debugging Tools for Embedded Applications*. URL: https://www.silabs.com/documents/public/white-papers/energy-debugging-tools.pdf.

[11] Muhammad Fahad et al. "A Comparative Study of Methods for Measurement of Energy of Computing". In: *Energies* 12 (June 2019). DOI: 10.3390/en12112204.

[12] Kyriakos Georgiou, Samuel Xavier-de-Souza, and Kerstin Eder. "The IoT energy challenge: A software perspective". In: *IEEE Embedded Systems Letters* 10.3 (2017), pp. 53–56.

[13] Kyriakos Georgiou et al. "Energy Transparency for Deeply Embedded Programs". In: *ACM Trans. Archit. Code Optim.* 14.1 (Mar. 2017). ISSN: 1544-3566. DOI: 10.1145/3046679. URL: https://doi.org/10.1145/3046679.

[14] *How to Update Debug Info: A Guide for LLVM Pass Authors*. URL: http://www.llvm.org/docs/HowToUpdateDebugInfo.html.

[15] Steve Kerrison and Kerstin Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015). ISSN: 1539-9087. DOI: 10.1145/2700104. URL: https://doi.org/10.1145/2700104.

[16] Steffen Lange, Johanna Pohl, and Tilman Santarius. "Digitalization and energy consumption. Does ICT reduce energy demand?" In: *Ecological Economics* 176 (2020), p. 106760.

[17] Sheayun Lee et al. "An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors". In: LCTES '01. Snow Bird, Utah, USA: Association for Computing Machinery, 2001, pp. 1–10. ISBN: 1581134258. DOI: 10.1145/384197.384201. URL: https://doi.org/10.1145/384197.384201.

[18] Alberto Leva et al. *Control-based operating system design*. Vol. 89. Institution of Engineering and Technology, 2013.

[19] Sheng Li et al. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: Association for Computing Machinery, 2009, pp. 469–480. ISBN: 9781605587981. DOI: 10.1145/1669112.1669172. URL: https://doi.org/10.1145/1669112.1669172.

[20] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration". In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers and Tools for Real-Time Systems*. New York, NY, USA: Association for Computing Machinery, 1995. ISBN: 9781450373081. DOI: 10.1145/216636.216666. URL: https://doi.org/10.1145/216636.216666.

[21] Kenan Liu, Gustavo Pinto, and Yu Liu. "Data-Oriented Characterization of Application-Level Energy Optimization". In: Apr. 2015. DOI: 10.1007/978-3-662-46675-9_21.

[22] *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.

[23] Irene Manotas et al. "An empirical study of practitioners' perspectives on green software engineering". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 237–248.

[24] Mike Tien-Chien Lee et al. "Power analysis and minimization techniques for embedded DSP software". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 5.1 (1997), pp. 123–135. DOI: 10.1109/92.555992.

[25] Jose Nunez-Yanez and Geza Lore. "Enabling accurate modeling of power and energy consumption in an ARM-based System-on-Chip". In: *Microprocessors and Microsystems* 37.3 (2013), pp. 319–332. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2012.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0141933113000021.

[26] R. Pereira. "Locating Energy Hotspots in Source Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 88–90. DOI: 10.1109/ICSE-C.2017.151.

[27] Felix Rieger and Christoph Bockisch. "Survey of Approaches for Assessing Software Energy Consumption". In: CoCoS 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017. ISBN: 9781450355216. DOI: 10.1145/3141842.3141846. URL: https://doi.org/10.1145/3141842.3141846.

[28] E. Rotem et al. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge". In: *IEEE Micro* 32.2 (2012), pp. 20–27. DOI: 10.1109/MM.2012.12.

[29] Mikko Roth, Arno Luppold, and Heiko Falk. "Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES '18. Sankt Goar, Germany: Association for Computing Machinery, 2018, pp. 86–89. ISBN: 9781450357807. DOI: 10.1145/3207719.3207729. URL: https://doi.org/10.1145/3207719.3207729.

[30] *Running Average Power Limit*. URL: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl.

[31] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and policy considerations for modern deep learning research". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 09. 2020, pp. 13693–13696.

[32] *The DWARF Debugging Standard*. URL: http://dwarfstd.org/.

[33] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[34] V Tiwari et al. "Instruction level power analysis and optimization of software". In: vol. 13. Feb. 1996, pp. 326–328. ISBN: 0-8186-7228-5. DOI: 10.1109/ICVD.1996.489624.

[35] *Using the ARM Performance Monitor Unit (PMU) Linux Driver*. URL: https://community.arm.com/developer/ip-products/system/b/embedded-blog/posts/using-the-arm-performance-monitor-unit-pmu-linux-driver.

[36] S. L. Xi et al. "Quantifying sources of error in McPAT and potential impacts on architectural studies". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 577–589. DOI: 10.1109/HPCA.2015.7056064.

[37] Tomofumi Yuki. "Understanding polybench/c 3.2 kernels". In: *International workshop on Polyhedral Compilation Techniques (IMPACT)*. 2014, pp. 1–5.

[38] Lide Zhang et al. "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones". In: CODES/ISSS '10. Scottsdale, Arizona, USA: Association for Computing Machinery, 2010, pp. 105–114. ISBN: 9781605589053. DOI: 10.1145/1878961.1878982. URL: https://doi.org/10.1145/1878961.1878982.

Appendix A

LLVM Patch Description

In this appendix we will provide a more detailed description of the LLVM patch that allows to trace and visualize the impact of compiler optimizations, as described in chapter 4.

The patch can be decomposed in two main contributions: a set of changes to the core LLVM library, and a new tool that outputs the HTML files, allowing for a better visualization of the transformations. The patched version of LLVM is publicly available on Github [1].

A.1 Changes to the Core Library

The core LLVM library consists of the classes that provide the in-memory representation of the LLVM-IR, see chapter 1 for a description of the IR. The first change that we have performed is the addition the method `Module::getNewID()`, that allows to retrieve a new identifier to be assigned to an `Instruction`. The next ID to be assigned is stored in the `Module` in the form of Metadata, in order to allow its serialization, and to preserve this information across different passes and opt runs. We have also defined the methods that allow to produce the log of the changes performed during the optimization: `Module::addCreateEntry`, `Module::addRemoveEntry`, `Module::addMoveEntry` and `Module::addReplaceEntry`. This methods print the entry to the standard debugging output, allowing to enable or disable the logging by means of flags passed to the opt tool.

After the definitions of this methods, we have modified the methods of the `Instruction` class that are used to create, delete and move instructions. As an example, consider the following constructor for the `Instruction` class, that creates an `Instruction` of the given `Type`, and eventually inserts it before another `Instruction`, `InsertBefore`:

[1] https://github.com/PietroGhg/tesi-ghiglio-llvm

```
1  Instruction::Instruction(Type *ty, unsigned it, Use *Ops,
       unsigned NumOps,
2                           Instruction *InsertBefore)
3    : User(ty, Value::InstructionVal + it, Ops, NumOps), Parent(
         nullptr) {
4
5    // If requested, insert this instruction into a basic block
         ...
6    if (InsertBefore) {
7      BasicBlock *BB = InsertBefore->getParent();
8      assert(BB && "Instruction to insert before is not in a
           basic block!");
9      BB->getInstList().insert(InsertBefore->getIterator(), this)
           ;
10
11     setID();
12   }
13 }
```

The call to `Instruction::setID()` at line 11 will retrieve a new identifier from the `Module` to which the `Instruction` belongs, and assign it to the `Instruction`.

Similar changes have been performed to methods such as `Instruction::removeFromParent`, `Instruction::insertBefore` or `Instruction::moveBefore`. In order to understand when an `Instruction` is replaced by another one, we have modified the `replaceAllUsesWith` method from the `Value` class.

Modifying this methods allowed to produce a log of the changes performed by a transformation pass, without having to go through the actual code of the transformation pass, greatly reducing the amount of time needed to understand which lines of code lead to a change in the structure of the `Module`. In the LLVM code base there are currently around fifty transformation passes used during the optimization phase, each of them counting thousands of lines of code, so checking all the passes would be very time consuming, and changes made to passes would be harder to maintain as new updates of LLVM are released by the community.

Nevertheless, due to the fact that, in the code base, not all the changes to a `Module` are performed through the well structured APIs that we have described so far, we have been forced to manually inspect the code of some of the transformations, and add methods calls producing the log entries where needed. For instance, some times an `Instruction` is not inserted in a `BasicBlock` through a call to the appropriate method, but it is inserted by retrieving the list of instructions of the basic block, and pushing the new instruction using the `push_back` method. This pattern, and similar ones, is quite recurring in the code base, and unfortunately led us to perform many small and sparse changes to the source code of the transformation

passes.

A.2 The opt-parser Tool

LLVM tools are stand alone executables within the LLVM repository, that make use of the library in order to provide a well defined service. There are tools for disassembling, dumping DWARF information, linking and other tasks.

The transformation log can be obtained by passing the *-pn* flag to the opt tool. This will output to the default debugging stream a set of `Modules` and log entries, and may become quite hard to read and understand for a user. To overcome this problem we have developed a tool within the LLVM repository that receives as input the output of the *-pn* flag, and renders it in HTML, providing the visualization described in chapter 5. The tool also performs the propagation of debug locations, creating the RepGraph, and outputting a textual representation of the module with replaced debug information.

Appendix B

ARM Cortex M4 Model

This appendix provides an example of the JSON file containing the characterization of the ARM Cortex M4 CPU. The average power consumption and the operational frequency reported refer to the board employed in the experimental evaluation, the STM32F407 Discovery. Frequency is expressed in Megahertz, power is expressed in milliWatt and the inter instruction overhead is expressed in nanoJoule. The cost of each instruction is measured in clocks per instructions, and it's derived from the Cortex M4 technical reference manual [2].

```
{
    "name": "cortex-m4",
    "freq" : 120,
    "power": 171,
    "iiover": 2.5,
    "memacc": 0.9,
    "cpi": [
 { "opname": "movw", "cost": 1, "memacc": 0},
 { "opname": "movs", "cost": 1, "memacc": 0},
 { "opname": "movt", "cost": 1, "memacc": 0},
 { "opname": "mov", "cost": 1, "memacc": 0},
 { "opname": "add", "cost": 1, "memacc": 0},
 { "opname": "addw", "cost": 1, "memacc": 0},
 { "opname": "adc", "cost": 1, "memacc": 0},
 { "opname": "adr", "cost": 1, "memacc": 0},
 { "opname": "sub", "cost": 1, "memacc": 0},
 { "opname": "subw", "cost": 1, "memacc": 0},
 { "opname": "subs", "cost": 1, "memacc": 0},
 { "opname": "sbc", "cost": 1, "memacc": 0},
 { "opname": "rsb", "cost": 1, "memacc": 0},
 { "opname": "mul", "cost": 1, "memacc": 0},
```

```
{ "opname": "muls", "cost": 1, "memacc": 0},
{ "opname": "smmla", "cost": 1, "memacc": 0},
{ "opname": "mla", "cost": 1, "memacc": 0},
{ "opname": "mls", "cost": 1, "memacc": 0},
{ "opname": "smull", "cost": 1, "memacc": 0},
{ "opname": "umull", "cost": 1, "memacc": 0},
{ "opname": "smlal", "cost": 1, "memacc": 0},
{ "opname": "umlal", "cost": 1, "memacc": 0},
{ "opname": "sdiv", "cost": 12, "memacc": 0},
{ "opname": "udiv", "cost": 12, "memacc": 0},
{ "opname": "ssat", "cost": 1, "memacc": 0},
{ "opname": "usat", "cost": 1, "memacc": 0},
{ "opname": "cmp", "cost": 1, "memacc": 0},
{ "opname": "cmn", "cost": 1, "memacc": 0},
{ "opname": "and", "cost": 1, "memacc": 0},
{ "opname": "eor", "cost": 1, "memacc": 0},
{ "opname": "orr", "cost": 1, "memacc": 0},
{ "opname": "orn", "cost": 1, "memacc": 0},
{ "opname": "bic", "cost": 1, "memacc": 0},
{ "opname": "mvn", "cost": 1, "memacc": 0},
{ "opname": "tst", "cost": 1, "memacc": 0},
{ "opname": "teq", "cost": 1, "memacc": 0},
{ "opname": "lsl", "cost": 1, "memacc": 0},
{ "opname": "lsr", "cost": 1, "memacc": 0},
{ "opname": "asr", "cost": 1, "memacc": 0},
{ "opname": "ror", "cost": 1, "memacc": 0},
{ "opname": "rrx", "cost": 1, "memacc": 0},
{ "opname": "clz", "cost": 1, "memacc": 0},
{ "opname": "ldr", "cost": 2, "memacc": 1},
{ "opname": "ldrh", "cost": 2, "memacc": 1},
{ "opname": "ldrb", "cost": 2, "memacc": 1},
{ "opname": "ldrsh", "cost": 2, "memacc": 1},
{ "opname": "ldrsb", "cost": 2, "memacc": 1},
{ "opname": "ldrt", "cost": 2, "memacc": 1},
{ "opname": "ldrht", "cost": 2, "memacc": 1},
{ "opname": "ldrbt", "cost": 2, "memacc": 1},
{ "opname": "ldrsht", "cost": 2, "memacc": 1},
{ "opname": "ldrsbt", "cost": 2, "memacc": 1},
{ "opname": "ldrd", "cost": 2, "memacc": 1},
{ "opname": "ldm", "cost": 5, "memacc": 1},
```

```
{ "opname": "str", "cost": 2, "memacc": 1},
{ "opname": "strh", "cost": 2, "memacc": 1},
{ "opname": "strb", "cost": 2, "memacc": 1},
{ "opname": "strsh", "cost": 2, "memacc": 1},
{ "opname": "strsb", "cost": 2, "memacc": 1},
{ "opname": "strt", "cost": 2, "memacc": 1},
{ "opname": "strht", "cost": 2, "memacc": 1},
{ "opname": "strbt", "cost": 2, "memacc": 1},
{ "opname": "strsht", "cost": 2, "memacc": 1},
{ "opname": "strsbt", "cost": 2, "memacc": 1},
{ "opname": "strd", "cost": 2, "memacc": 1},
{ "opname": "stm", "cost": 2, "memacc": 1},
{ "opname": "push", "cost": 2, "memacc": 1},
{ "opname": "pop", "cost": 5, "memacc": 1},
{ "opname": "ldrex", "cost": 2, "memacc": 0},
{ "opname": "ldrexh", "cost": 2, "memacc": 0},
{ "opname": "ldrexb", "cost": 2, "memacc": 0},
{ "opname": "strex", "cost": 2, "memacc": 0},
{ "opname": "strexh", "cost": 2, "memacc": 0},
{ "opname": "strexb", "cost": 2, "memacc": 0},
{ "opname": "clrex", "cost": 1, "memacc": 0},
{ "opname": "b", "cost": 4, "memacc": 0},
{ "opname": "bl", "cost": 4, "memacc": 0},
{ "opname": "bx", "cost": 4, "memacc": 0},
{ "opname": "blx", "cost": 4, "memacc": 0},
{ "opname": "cbz", "cost": 4, "memacc": 0},
{ "opname": "cbnz", "cost": 4, "memacc": 0},
{ "opname": "tbb", "cost": 5, "memacc": 0},
{ "opname": "tbh", "cost": 5, "memacc": 0},
{ "opname": "svc", "cost": 0, "memacc": 0},
{ "opname": "it", "cost": 1, "memacc": 0},
{ "opname": "cpsid", "cost": 2, "memacc": 0},
{ "opname": "cpsie", "cost": 2, "memacc": 0},
{ "opname": "mrs", "cost": 2, "memacc": 0},
{ "opname": "msr", "cost": 2, "memacc": 0},
{ "opname": "bkpt", "cost": 0, "memacc": 0},
{ "opname": "sxth", "cost": 1, "memacc": 0},
{ "opname": "sxtb", "cost": 1, "memacc": 0},
{ "opname": "uxth", "cost": 1, "memacc": 0},
{ "opname": "uxtb", "cost": 1, "memacc": 0},
```

```
{ "opname": "ubfx", "cost": 1, "memacc": 0},
{ "opname": "sbfx", "cost": 1, "memacc": 0},
{ "opname": "bfc", "cost": 1, "memacc": 0},
{ "opname": "bfi", "cost": 1, "memacc": 0},
{ "opname": "rev", "cost": 1, "memacc": 0},
{ "opname": "rev16", "cost": 1, "memacc": 0},
{ "opname": "revsh", "cost": 1, "memacc": 0},
{ "opname": "rbit", "cost": 1, "memacc": 0},
{ "opname": "sev", "cost": 1, "memacc": 0},
{ "opname": "wfe", "cost": 1, "memacc": 0},
{ "opname": "wfi", "cost": 1, "memacc": 0},
{ "opname": "nop", "cost": 1, "memacc": 0},
{ "opname": "isb", "cost": 4, "memacc": 0},
{ "opname": "dmb", "cost": 1, "memacc": 0},
{ "opname": "dsb", "cost": 1, "memacc": 0},
{ "opname": "beq", "cost": 4, "memacc": 0},
{ "opname": "bne", "cost": 4, "memacc": 0},
{ "opname": "bgt", "cost": 4, "memacc": 0},
{ "opname": "blt", "cost": 4, "memacc": 0},
{ "opname": "bge", "cost": 4, "memacc": 0},
{ "opname": "ble", "cost": 4, "memacc": 0},
{ "opname": "bcs", "cost": 4, "memacc": 0},
{ "opname": "bhs", "cost": 4, "memacc": 0},
{ "opname": "bcc", "cost": 4, "memacc": 0},
{ "opname": "blo", "cost": 4, "memacc": 0},
{ "opname": "bmi", "cost": 4, "memacc": 0},
{ "opname": "bpl", "cost": 4, "memacc": 0},
{ "opname": "bal", "cost": 4, "memacc": 0},
{ "opname": "bnv", "cost": 4, "memacc": 0},
{ "opname": "bvs", "cost": 4, "memacc": 0},
{ "opname": "bvc", "cost": 4, "memacc": 0},
{ "opname": "bhi", "cost": 4, "memacc": 0},
{ "opname": "bls", "cost": 4, "memacc": 0},
{ "opname": "smmul", "cost": 1, "memacc": 0},
{ "opname": "vmov", "cost": 2, "memacc": 0},
{ "opname": "vstr", "cost": 3, "memacc": 1},
{ "opname": "vldr", "cost": 3, "memacc": 1},
   ]
}
```