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

Thesis Title

Advisor: Prof. Giovanni AGOSTA

Thesis by:
Pietro Ghiglio Matr. 920491

Academic Year 2019–2020

To someone very special...

Acknowledgments

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Sommario

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Chapter 1

Background

1.1 LLVM

The LLVM Project [24] 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 its uses.

A basic block is a sequence of instructions with no branch, jump or returns between them. Intuitively, a when the first instruction in a basic block is executed, all the others are executed as well (assuming that all the calls return and no exceptions are thrown). 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:

```
int example(int n){
  int x;
  if(n > 0)
    x = n+1;
  else
    x = n-1;
  return x;
}
```

It is translated in the following LLVM-IR segment:

```
define dso_local i32 @example(i32 %0) {
  %2 = icmp sgt i32 %0, 0, !ID !1
  br i1 %2, label %3, label %5, !ID !2

3:                                                ; preds = %1
  %4 = add nsw i32 %0, 1, !ID !3
  br label %7, !ID !4

5:                                                ; preds = %1
  %6 = sub nsw i32 %0, 1, !ID !5
  br label %7, !ID !6

7:                                                ; preds = %5, %3
  %.0 = phi i32 [ %4, %3 ], [ %6, %5 ], !ID !7
  ret i32 %.0, !ID !8
}
```

There we see that:

1. The resulting code is in SSA form, since to each assignment to variable x in the source code, corresponds 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 Doxygen Documentation. 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 it's 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 [15] for more details.

1.1.5 LLVM Passes

LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the 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 How LLVM handles debug information

1.2.1 Metadata classes

The LLVM class hierarchy associated to debug information resembles the structure of the DWARF standard, described in 1.4.2.
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've 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 map unambiguously 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 [9].
Here we provide a short summary of such guidelines [1], 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's 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.

- 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.

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

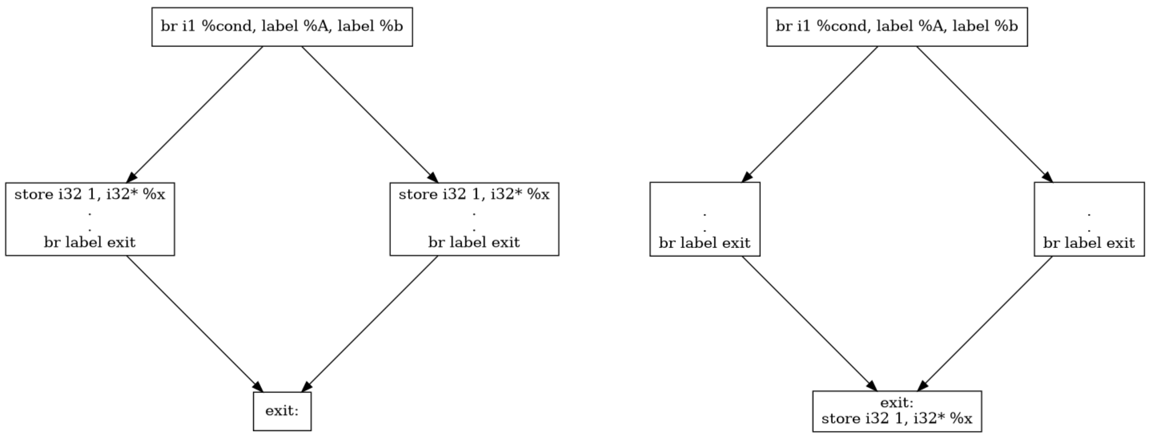

Figure 1.1: Example of optimization with merged debug location

Location should be merged when two instructions are replaced with a new instruction. An example of that is 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 the cases in which 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 insert 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. An example of instrumentation are the many sanitizers that are part of the LLVM project, they 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 more instructions 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 functionality 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 are 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 assembly instruction. This allows the debugger to pause the program's execution when a given source code location is reached.

These information are useful not only for debugging purposes, they 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, that may be able to annotate the source code with the information that they have gathered.

1.4.2 DWARF format

The DWARF format [23] 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 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.

And other fields that 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 relay 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 [17]. This means that energy consumption has to be tackled also 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 have a finer grain, allowing to attribute energy measures to either source code entities or hardware components.

The procedure adopted to obtain the result are widely different, [19] 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 to 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 register that store information about the energy consumption, and a set of APIs that allow to read their contents.
- Modeling based: some solutions propose to model mathematically the energy behavior of the target architecture, perform some experiments in order to estimate the parameters of the model, and use the model 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), for their Sandy Bridge architectures, the Intel System Management Controller (SMC) for the Xeon Phi, or NVidia's NVidia Management Library (NVML), that allows to obtain energy consumption of their 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 [4]. The simulator that they developed, Wattch, is the first simulator to operate at the architectural level: it does not require the full RTL design (the Verilog of the target architecture), but relays 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 [1] architectural simulator.

Being more high-level than RTL-based simulations, Wattch is faster and doesn't 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 several order of magnitudes (around 10000 slower, as reported in [2]) slower than running the actual program, even if the latter has been instrumented.

Since it's original publication, the original work on Wattch has been expanded in several ways. The main contribution in that sense has been given by Li et al., who developed a completely new power simulator, McPat [12], offering more modern and advanced features than Wattch.
It provides the support to compute power-area integrated metrics (energy-delay-area product), models static, dynamic, and short-circuit power dissipation (whereas Wattch only modeled dynamic power dissipation), allows to model multicore architectures, that have become increasingly widespread, and 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 [26], 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 this section: 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: they 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's also the one that has the greatest "overhead" for a developer that wants to assess the energy consumption of his software.
Given it's 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 [21], they 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.
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 [25].

In [7], instead, they 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 said, the main the 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 lead to that energy consumption.

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 the following 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 [22]. 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 of RAPL is given in [14], in which the authors 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 [20], where RAPL is introduced to the general public, they state that the prediction provided by their power model matches actual measurements, showing high correlation between the two. This is partially disproved in [7], where the authors 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.

To summarize this section, 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. This means that, at least, it is not a good candidate to be used as ground truth to validate other estimation methods.

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. [25].

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 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. They therefore 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, $\hat{\imath}$, and multiply it by the supply voltage V_{cc}, obtaining the base energy consumption.

Instruction may also be grouped together, since instruction 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 [16], the same authors of [25] 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.

In [10], Eder et al. characterize the energy behavior of a multithreaded architecture. They 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 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, and put the untestable instr in the appropriate group or assign default cost to untestable instr.
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.

In [11], 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 [25] are too simplistic, since they relay 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 generated 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%.

In [2] 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 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 relays 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.

The same authors of [2], in [3] 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's possible to build an over-constraint 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 the 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.

In [8], Eder et al. provide a LLVM-IR to assembly mapping technique that differs from [3], 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 [3] 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 same LLVM instruction of the last previous assembly instruction with an entry in the line table.

Their static, worst-case energy consumption estimation is based on the Implicit Path Enumeration Technique (IPET) [13]. 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 indicates the cost of executing the basic block, and x_i indicates the number of executions of the basic block. c_i is obtained in similar fashion to [3]: 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 block 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., [2] provide such a result, but at cost of performing their analysis on the parse tree, and the same authors in [3] 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 $\rightarrow$ task $\rightarrow$ 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 [18], 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). They 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 [18] 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 and 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 [19], that allow for source level visualization. The development board developed by SiliconLabs [6] allows to perform energy measures of ARM microcontrollers, and provide 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 provides enough information to determine the currently executed method.

Other commercially available tools are oriented towards the Android world, for instance Green Droid [5] provides estimates of the energy consumption by exploting PowerTutor [27], a component power manager that employs the battery and current sensors commonly present in smartphones.

2.7 Final Remarks

Chapter 3

—

3.1 Instrumentation

3.2 Source level visualization

3.3 Compiler optimizations

Conclusions

Bibliography

[1] 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`.

[2] 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`.

[3] 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`.

[4] 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`.

[5] 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`.

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

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

[8] 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`.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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

First appendix

Appendix B

Second appendix

Appendix C

Third appendix