Enabling Energy Transparency for Deeply Embedded Programs

By

KYRIAKOS GEORGIOU

Department of Computer Science

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

MARCH 2017

Word count: forty-nine thousand

Abstract

The evolution of pervasive computing into what is today known as Interent of Things (IoT) sparked a whole new world of deeply embedded applications. Most of these applications are based on deeply embedded systems that have to operate on limited or unreliable sources of energy, such as batteries or energy harvesters. Meeting the energy requirements for such applications is a hard challenge, which threatens the future growth of the IoT.

Software has the ultimate control over hardware. Therefore, its role is significant in meeting the strict energy budget of energy-critical applications. Currently, programmers do not have any feedback about how their software affects the energy consumption of a system. Such feedback, can be enabled by the concept of energy transparency. This makes a program's energy consumption visible, from hardware up to software, through the different system layers. Enhancing development tools with such transparency can enable energy optimizations at each layer and between layers and will allow programmers to make energy-aware decisions.

This thesis introduces a framework of novel techniques that can be integrated into development toolchains to enable energy transparency on deeply embedded devices, typically used for IoT applications. The techniques are significantly faster than simulation-based energy estimation and less costly to deploy than hardware energy measurements. They also provide fine-grained energy characterization of software, such as at instruction level.

A new target-agnostic mapping technique is introduced to allow energy characterization of the compiler's Intermediate Representation (IR). This is a significant step beyond existing Instruction Set Architecture (ISA)-level energy estimation techniques. The introduced mapping technique gives powerful insights to the compiler's optimizer regarding the execution time and the energy consumption of a program.

A Static Resource Analysis (SRA) technique is developed to provide energy consumption bounds. The analysis is introduced at both ISA and IR levels. The results demonstrate that the mapping technique enables SRA at the IR level with a small accuracy loss in the range of only 1%, compared to SRA at the ISA level. SRA-based energy estimation has been also applied to a set of multi-threaded programs for the first time. The analysis is successfully used for time-energy design space exploration of task farms and pipelined programs.

To estimate the actual energy consumption of a program under specific input parameters, a target-agnostic profiling technique is introduced. The technique estimates energy consumption at the IR level by utilizing the mapping technique. It is designed to ensure that the instrumentation code required for profiling does not lead to energy overheads. The experimental evaluation shows an average absolute error of 3.2% compared to hardware measurements.

The work performed in this thesis significantly advances the state of the art in energy-aware software development. It reduces the level of expertise needed for developing energy efficient software. Therefore, it allows energy consumption to be treated as a first-class citizen during the software development process.

Dedication and acknowledgments

The completion of this thesis has been a long journey, which I would not be able to walk on my own. I am thankful for the support of many people, that allowed me to stay focused and committed to my objectives through all these years. This maximized the gains made out of this Ph.D. journey. I would like to express my gratitude to my family, and especially my parents, Antonia and Christakis, which have been supporting me by any means possible through every single step of my life, and showed me the way to pursuit my dreams.

I owe a big thank you to my supervisor, Professor Kerstin Eder, who has given me the opportunity to pursue a Ph.D. and secured the funding needed to sustain myself through all these years. Her patient, advice, and guidance enhanced me with valuable skills and knowledge that I will carry for the rest of my life. I will also like to thank my colleagues and friends, including Steve Kerrison, Jeremy Morse, Craig Blackmore, Dejanira Araiza Illan, Yanni Ou, Tian Qin, Themis Omirou, Zbigniew Chamski, and many others. Our fruitful discussions, collaboration, and their tolerance made my research experience both more rewarding and fun at the same time. Many thanks to my external examiners, professors Michael Engel and Björn Lisper, as well as my internal coordinator Dr.Jose Luis Nunez-Yanez.

My Ph.D. was financially supported by a University of Bristol Ph.D. scholarship, the Whole-Systems Energy Transparency (ENTRA) project, funded by the EU 7th Framework Programme Future and Emerging Technologies (FET), and the EMC^2 project, an ARTEMIS Joint Undertaking project in the Innovation Pilot Programme "Computing platforms for embedded systems (AIPP5). I am grateful for these funding sources as without them this thesis and all the related collaboration activities would not have been possible. Finally, I will like to thank the European Network on High Performance and Embedded Architecture and Compilation (HIPEAC) for awarding me with a HIPEAC collaboration grant.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

Table of Contents

List of Tables

LIST OF FIGURES

List of Listings

Thesis conventions and terminology

Deeply embedded systems: Systems that are simpler than general purpose processors and favor predictability and low energy consumption over maximizing performance. Predictability makes them ideal for critical applications.

Mission critical application: A mission critical application (or just critical application) in the context of embedded systems is an application that has to meet some predefined constraints at runtime. Failing to do so can lead to severe consequences, such as damaging equipment, injuring people, or cause a of substantial financial loss. The predefined constraints can vary significantly depending on the application, but when using this term in the context of this thesis, it refers to applications that are resource constrained.Such applications have to operate without exhibiting the worst case scenario for execution time and/or within a predefined energy budget. Critical applications typically utilize deeply embedded processors to guarantee predictability.

Xcore: The XMOS Xcore processor in the context of this thesis refers to the one that is based on the XS-L1 architecture. The terms Xcore and XS-L1 might be used interchangeably in this thesis.

Actual case: In the scope of this thesis, the actual case refers to the prediction of the exact usage of the resource under investigation, time or energy, rather than bounds. The actual case for a program can depend on its input parameters.

Tight bounds: Such bounds, on the execution time or the energy consumption of an application, are safe to be used in mission-critical applications as they guarantee to overestimate the actual worst case, but at the same time, they are tight. Tight means that the amount of overestimation is within reasonable levels. In the scope of this thesis, bounds, for both execution time and energy consumption, are considered tight when their overestimation is less than 10% on the actual case.

Loose upper bound: In the context of this work, a loose upper bound is a bound that is not safe to be used for guarantees in mission critical applications.

A list of acronyms is given at the end of this thesis, while each acronym is also defined at its first occurrence in the text.

Chapter 1

Introduction

"Optimism is an occupational hazard of programming; feedback is the treatment."
— Kent Beck, *[11]*

The Interent of Things (IoT) is no longer just a buzzword in the media; it is becoming a reality. The emergence of IoT leads into a new era of innovation and creativity. This sets high expectations on both the research and industry communities for delivering the necessary technological advancements, which will enable the materialization of new IoT applications.

IoT, aims to connect the physical world to the Internet. Thus, "things" refers to anything around us, such as machines, buildings, trees, and even humans. IoT will enable the sensing of environment, collecting and transmitting of live data, locating and remote controlling of things. The application space covers a wide range of industries, from healthcare, energy management, transportation to environmental sensing. By 2020, it is predicted that around 50 billion things will be connected to the Internet [34].

Several elements are required for making an object part of the IoT. These include the assignment of a unique identity to the object, the ability to connect to the Internet or other objects, the capacity to sense the environment, some processing capabilities and finally an energy source. Internet Protocol Version 6 (IPV6) allows us to assign a unique address to every object on the earth without any limitations. Existing wireless communication infrastructure provides IoT the necessary connectivity, while embedded processors become the objects' brains to perform any processing needed. All these elements are meaningless without an adequate energy source to bring them to life.

Powering billions of embedded devices deployed into the environment is one of the biggest

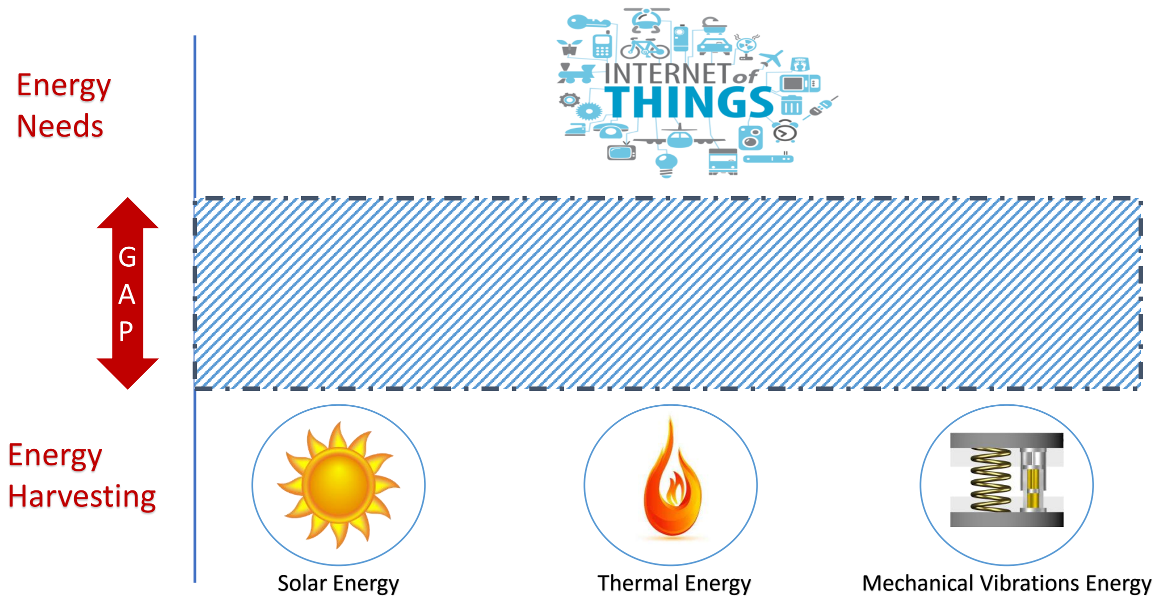


Figure 1.1: IoT energy needs vs. what energy harvesting can offer.

challenges that IoT faces. Battery-based solutions tend to be impractical and costly due to the need of replacement. Therefore, energy harvesting seems to be the only option for many IoT applications. Solar, thermal, and mechanical vibrations energy can be converted to electrical energy that is used to power the IoT devices. But energy harvesting comes with two caveats. Firstly, it is an unreliable source of energy, and secondly, there is still a large gap between the energy it can deliver and the required energy budget for many of the IoT applications (see Figure 1.1). For mission critical IoT applications, such as health-care, it is vital to complete a critical task before running out of the available energy budget. Many other energy-constrained deeply embedded application areas, such as robotics and aerospace, are facing similar challenges with the IoT. Solving the IoT energy challenge is an emerging research field.

Traditionally, for any power/energy challenge in the Information Communication Technology (ICT), hardware innovation has been the safe heaven to achieve energy savings. Similarly, hardware innovation is currently the prominent response to tackle the energy challenge that IoT faces. The hardware community has been introducing new ultra-low-energy embedded devices and customizing existing technologies to create more energy efficient versions, such as the Bluetooth Low Energy (BLE) [14], which are well suited for IoT energy-critical applications. But is that all we can do to tackle this challenge? Steve Furber, the principal designer of the ARM microprocessor, in one of his article in 2010 [1] stated:

"If you want an ultimate low-power system, then you have to worry about energy usage at every level in the system design, and you have to get it right from top to bottom, because any level at which you get it wrong is going to lose you perhaps an

order of magnitude in terms of power efficiency."

Software has the ultimate control over hardware. Inefficient software can drive energy-efficient hardware to waste the system's energy budget. Custom software that is well suited for a particular platform is always more resource efficient than more generic software versions which perform the same set of tasks. Therefore, focusing solely on optimizing the energy consumption of hardware is only part of the story. Then, in the same article, [1], Steve Furber continues with:

> *"Programmers will not be able to afford to be ignorant about the energy cost of the programs they write ... You need tools that give you feedback and tell you how good your decisions are. Currently the tools don't give you that kind of feedback."*

These tools are now needed more than ever to overcome the IoT energy challenge. However, there is still too little done to expose how different coding styles and algorithms affect the energy consumption on hardware. Therefore, very few programmers at present have much of an idea of how much energy their programs consume, and which parts of a program use the most energy. This has many implications for the development of IoT applications:

1. There is much guesswork, and thus bad energy-related choices are only identified at a late stage when the system malfunctions due to a failure to meet the system's energy requirements. Inadequate energy supply to a system can cause unexpected behavior, or even make the system crash;
2. The high level of expertise needed and the lack of energy-aware development tools reduce significantly the number of embedded developers who are able to deliver energy-critical systems;
3. Current solutions, such as end-to-end measurements with highly accurate oscilloscopes, are inadequate and extremely expensive to deploy and use on a routine basis.

All of the above conditions make the whole process of developing energy-constrained deeply embedded applications difficult and costly.

The first step towards solving a problem is to understand the source of the problem. Various abstraction layers have been introduced through the system stack to abstract away the hardware details and therefore make programming easier. At the same time, these abstraction layers took away the transparency that allowed the software developers to understand the impact of their coding choices on the way hardware is utilizing the various resources. This was not a major issue while the only resource of interest has been the execution time of programs. Compilers have been long focused and specialized in improving execution time and developers are left assured that a compiler will do the best possible job on optimizing their code for performance. On the other hand, improving energy consumption is generally treated as a side effect of improving time. Although in most cases, indeed, the most energy-efficient solution is also achieved while optimizing for performance, recent work has demonstrated that exposing the energy consumption of software can lead to new energy-specific optimizations. Moreover, in some cases these optimizations have no effect on execution time [97, 98].

Furthermore, with the introduction of many-, multi-core and multi-threaded architectures, and the ability to apply voltage and frequency scaling, there is a large number of different configurations available to the software developer. Through these various configurations, energy and time can now be interchanged to achieve the optimal operating conditions according to the system's specifications. To enable this, the developer needs to know the impact of each configuration on both execution time and energy. Although significant progress has been made in the area of predicting execution time, there is still a lack of techniques to estimate the energy consumption.

Predicting energy consumption can be more challenging than predicting execution time. In contrast to timing, energy consumption is more data sensitive. Considering an entirely time-predictable embedded architecture, a timing model can be constructed to capture the exact execution time of each ISA instruction. In contrast, it is hard to construct a similar model for energy consumption since the energy cost of executing an instruction varies depending on (the circuit switching activity caused by) the operands used. The data effect on the systems' energy consumption is insignificant when considering complex architectures which target performance. This is because performance-enhancing hardware components, such as caches, can have significantly larger energy consumption than the one caused by the processors switching activity [37, 46]. In contrast, deeply embedded architectures lack of such performance-enhancing complexity at the hardware level in favor of predictability and low-energy consumption. Thus, the data effect on the energy consumption of such deeply embedded architectures can be significant and must be taken into account. Accounting for energy through execution time can not capture this effect [55].

The recently introduced concept of energy transparency [31] aims to make a program's energy consumption visible, from hardware up to software, through the different systems layers. Enhancing system development tools and methods with energy transparency capabilities is the key to enable energy optimizations at each layer and between layers, and help both programmers and operating systems make energy-aware decisions.

The energy consumption of a program on a specific hardware can always be determined through physical measurements. Although this is potentially the most accurate method, it is often not easily accessible. Measuring energy consumption can involve sophisticated equipment and special hardware knowledge. Custom modifications may be needed to probe the power supply. Even though energy monitoring counters are becoming increasingly popular in modern processors, their number and availability are still limited in deeply embedded systems. Moreover, it is difficult to achieve fine-grained energy characterization of software, such as Control Flow Graph (CFG) basic blocks, with energy measurements.

An alternative to energy measurements is simulation-based energy consumption estimation. Execution statistics gathered from simulating a program for a particular hardware are input to an energy model to retrieve energy consumption estimates. Such an approach is typically slow and therefore, difficult to incorporate into the iterative process of software development. This is because cycle-accurate simulators are needed to achieve good estimation accuracy. Also, building a cycle-accurate simulator for an architecture is not a trivial task.

When building an energy-critical application, beyond the actual case energy consumption, developers also have to consider the worst-case energy consumption scenarios. Traditionally for embedded applications, the standard practice to account for the worst-case scenario is by the use of the worst-case power dissipation of the chip, as taken from the device datasheet. This can easily lead to the wrong choice of hardware as there is no indication how the software will eventually utilize the chosen device. An alternative is to apply end-to-end measurements to retrieve the worst-case energy consumption of a new system. It is well known in the Worst Case Execution Time (WCET) community that such an approach is inappropriate in most cases, as the whole input space of a program needs to be exhaustively searched [140]. This also applies when targeting the worst-case energy consumption with end-to-end measurements.

The IoT energy challenge urges the research community to develop new energy transparency techniques that can expose both the bounds and the actual energy consumption of software written for a specific platform. Such techniques need to be readily applicable during the whole embedded systems software development life-cycle. This will enable programmers, toolchains, and runtime systems to make energy-aware decisions in order to meet their strict energy constraints.

1.1 Thesis research questions

The primary objective of this thesis is to introduce energy-transparency techniques that can be easily integrated into the daily software development life-cycle of deeply embedded systems. This will help to tackle the energy challenge faced by IoT and other similar energy-constraint applications. Therefore, the research question that motivates the work carried out is:

How can the new concept of energy transparency be realized to enable energy-aware software development of deeply embedded programs?

Energy-aware software development has a significant role in tackling the energy challenge many embedded applications are facing today. Energy transparency is the fundamental prerequisite to developing energy-aware software development tools. The present challenge is to find appropriate techniques that can materialize the desired energy transparency. The problem can therefore be elaborated further in the following statements:

Energy consumption has to be treated as a first-class citizen during software development. Currently, most software developers have no idea about how their choices affect the energy consumption of their applications. Optimizing energy consumption has been long treated as a side effect of optimizing the execution time. This perception created a long-standing comfort zone that prevents the research of appropriate energy transparency techniques that can enable energy-aware development and optimizations.

There is a lack of adoption of the state of the art techniques for software energy consumption estimation by the existing deeply embedded system development tools. Although there is a substantial amount of research on energy modeling of hardware and energy simulation techniques (see Chapter 3), there is a failure in adopting them into the daily software development life-cycle of deeply embedded applications. This thesis concern is to identify and address the limitations of existing approaches that provide software energy consumption estimations, to enable the emergence of energy-aware development tools and practices.

The research performed in the scope of this thesis tackles the above issue by providing a framework of energy transparency techniques that are advancing the state of the art of software energy-aware development for deeply embedded systems. The main research directions that drive the work contacted in this thesis are:

Both the actual energy consumption and bounds are needed: Energy consumption bounds will help developers to meet the strict energy budget requirements of deeply embedded applications. Actual energy consumption estimates are necessary to serve as a benchmark for potential energy optimizations.

Energy transparency is needed at multiple levels of software abstraction: Energy consumption information needs to be provided at all levels of abstraction at which energy-aware development and optimizations can be enabled. There are three main software abstraction levels at which software developers and development tools can significantly influence the energy consumption of a program on a particular architecture. They are the source code level, the compiler's IR level, and the ISA level. The trade-off between estimation accuracy and program information availability at each level needs to be considered to establish the objectives of performing energy consumption estimation at each level.

Fine-grained energy characterization of software is needed: Identifying energy hotspots at any software level of abstraction requires the ability to attribute the energy consumption estimates to the various basic software components of the program under examination, such as instructions, CFG basic blocks and loops.

Energy transparency techniques have to be target and programming language agnostic: Energy transparency techniques that are target and programming language agnostic will provide a common framework for energy-aware development that can be adopted by a large number of deeply embedded architectures. Furthermore, supporting the same framework for a large variety of embedded architectures is more practical and cost-effective, than maintaining different solutions for each architecture.

The multi-threaded and multi-core case must be considered: Although, multi-threaded and multi-core is not yet that common in deeply embedded systems, some novel multi-core, multi-threaded but still predictable architectures emerged during the last years. Such architectures are needed due to the increasing demand for more computing power by deeply embedded applications. An example of such an architecture is the Xcore [86], which is the architecture under investigation in this thesis. As this trend of developing new deeply embedded multi-core, multi-threaded architectures is expected to grow in future, it is important to consider them when developing energy transparency techniques.

Energy transparency techniques must enable design space exploration: Having a number of different resources at hand, developers and toolchains can apply multi-objective optimizations to find the optimum balance between the available resources, such as execution time, energy and code size, according to the application's requirements. To enable this, energy transparency techniques have to provide feedback on the effect that each different configuration has on the resources of interest.

1.2 Thesis contributions

Software development tools need to be enhanced with capabilities that will enable energy transparency from hardware to software, to expose the energy consumption of software written for a particular platform. This will enable an iterative process of verifying energy consumption against available energy budgets, optimizing energy consumption and performing design space exploration.

This thesis provides a holistic framework of energy transparency techniques for deeply embedded systems that can be easily integrated into existing toolchains to enable energy-aware software development. To achieve this, the state of the art techniques that provide software energy estimates are reviewed, and the best practices are adopted and extended into the proposed framework. Furthermore, new techniques are introduced, where needed, to advance the state of the art of energy transparency techniques and address this thesis research question. The main contributions of this thesis are:

1. A low-level analysis for the target architecture that captures its energy consumption behavior statically. This includes a refinement of an existing ISA-level energy model, improving its accuracy by around four percentage points (Chapter 5);
2. High-level analysis which can lift existing low-level analysis to higher levels of software abstraction (Chapter 6). At the heart of this analysis lies the formalization and implementation of a novel target-agnostic mapping technique that lifts an ISA-level energy model to a higher level, the intermediate representation of the Low Level Virtual Machine (LLVM) compiler infrastructure (Section 6.3);
3. Energy estimation techniques that avoid the need for hardware energy measurements and simulation-based energy consumption estimation that are typically costly to implement and deploy. More specifically:
   a) SRA energy estimation at the ISA level and at the LLVM IR level using the introduced high-level analysis (Section 7.1). The experimental evaluation demonstrated that the mapping technique enables SRA at the LLVM IR level with a small accuracy loss in the range of only 1%, compared to SRA at ISA level (Section 8.2);
   b) A new target-agnostic profiling-based energy estimation technique that retrieves estimations at the LLVM IR level by the use of the introduced high-level analysis, and with an accuracy close to that achievable by a cycle accurate Instruction Set Simulator (ISS)-based estimation (Section 7.2). The experimental evaluation demonstrated an average absolute error of 3.1% for the profiling-based estimations when compared to hardware measurements (Section 8.3). The introduced profiling-based estimation is more flexible and significantly more efficient than ISS-based estimation: The technique can provide at least the same performance or significantly outperform the estimation speed of an ISS-based estimation, depending on the complexity of the algorithm implemented and the size of the resulting program. This makes it more suitable for iterative optimizations during software development (Section 8.3.1).
4. Comprehensive evaluation of the SRA and profiling energy estimation techniques and the accuracy of the mapping technique on a large set of representative benchmarks (Chapter 8);
5. Both the introduced SRA-based and profiling-based estimations support fine-grained energy characterization of software, from a single instruction up to a whole function (Chapter 7);
6. SRA and profiling extension for a set of multi-threaded programs, focusing on task farms and pipelines, two commonly used concurrency patterns in embedded computing (Section 7.3);
7. Parametric energy cost functions were extracted for a number of benchmarks using regression analysis (Section 8.2.1). The equations retrieved are analogous to those retrieved in [44], [69] and [70] by the use of automatic complexity analysis (Section 3.2.2.2);

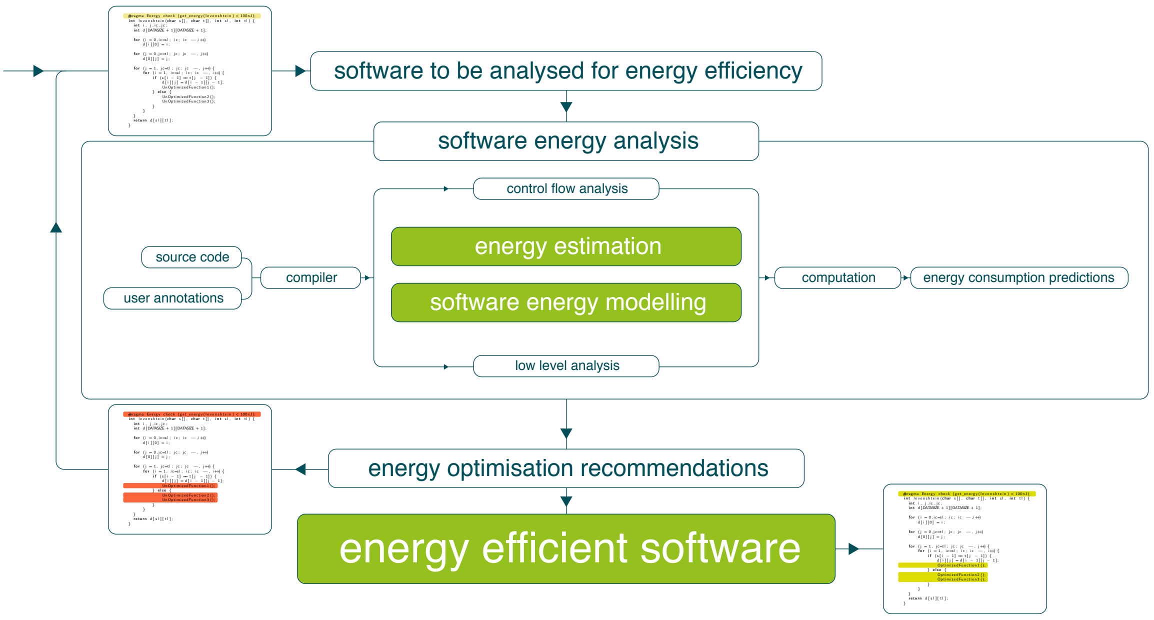

software to be analysed for energy efficiency
software energy analysis
control flow analysis
source code
user annotations
compiler
energy estimation
software energy modelling
low level analysis
computation
energy consumption predictions
energy optimisation recommendations
energy efficient software

Figure 1.2: A vision of the iterative process of energy-aware software development.

8. Design space exploration of a set of deeply embedded programs enabled by using only the SRA-based estimation introduced in this thesis (Section 8.2.3);
9. A thorough investigation of the implications of using Implicit Path Enumeration Technique (IPET) with data insensitive energy models, which contributes to gaining knowledge on the infeasible problem of retrieving tight worst-case energy consumption bounds and on how to utilize the best currently available practices (Section 8.4).

Figure 1.2 provides a visualization of how the techniques introduced in this thesis can be integrated into an existing deeply embedded toolchain to enable energy-aware development. The energy-aware development process starts with the user providing his/her source code as an input to his/her standard development toolchain, which is now tailored to support energy transparency. The source code can include annotations or directives to query or direct the analysis (e.g. yellow highlighted lines in upper-left code sample in Figure 1.2). The standard compilation toolchain is now enhanced with the new energy transparency utilities, including the CFG analysis, low-level analysis, energy estimation and software energy modeling (that includes both the high- and low-level analysis). The enhanced toolchain can now provide energy estimates at various software levels of abstraction, and attribute the estimated energy consumption to various basic software elements, such as CFG basic blocks. The results can then be passed back to the user as energy hot spots indications (e.g. red highlighted lines in low-left code sample in Figure 1.2), or used from the toolchain itself for optimization purposes. The whole process can be repeated multiple times until the application energy requirements are met.

This thesis focuses on energy transparency techniques. Such techniques can help to introduce energy-specific optimizations. Investigating such energy-oriented optimizations is out of the scope of this thesis and forming future work.

1.3 Related publications

- K. Georgiou, S. Kerrison, Z. Chamski, K. Eder. "Energy Transparency for Deeply Embedded Programs." In: *ACM Transactions on Architecture and Code Optimization (TACO)*([39].

  This paper introduces the SRA, mapping and profiling techniques used to achieve energy transparency for the Xcore processor. The paper also includes the experimental evaluation of the proposed methods and some design space exploration achieved for a number of multi-threaded programs by the use of SRA. The same work is part of this thesis but described in more depth.

- K. Georgiou , S. Kerrison , K. Eder , "On the value and limits of multi-level energy consumption static analysis for deeply embedded single and multi-threaded programs." In: *ArXiv e-prints, 2015* [41].

- K. Georgiou , S. Kerrison , K. Eder , "A multi-level worst case energy consumption static analysis for single and multi-threaded embedded programs." In: *Tech. Report CSTR- 14-003, University of Bristol, December 2014* [40].

  The above two technical reports are preliminary versions of the paper listed first, [39]. [40] is more focused on the worst-case energy consumption of a program, while [41] is focused on the effect of data on the energy consumption of a program.

- K. Eder, J. P. Gallagher, P. Lopez-Garca, H. Muller, Z. Bankovi, K. Georgiou, R. Haemmerl, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger, and M. Rosendahl, "Entra: Whole-systems energy transparency." In: *Microprocessors and Microsystems, 2016* [31].

  This journal presents the concept of energy transparency, which was investigated in the ENTRA EU funded project, [33], and also in the scope of this thesis. The paper demonstrates how energy transparency can enable energy-aware software development. The energy modeling techniques and the energy analysis techniques developed during the ENTRA project are presented. The article describes how the LLVM IR mapping techniques introduced in this thesis can be used to enable the energy analysis at higher levels, such as the LLVM IR and the source code level.

- N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, “Static analysis of energy consumption for llvm ir programs.” In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES ’15, (New York, NY, USA), ACM, 2015* [44].

  An early version of the mapping techniques introduced in this thesis was used in this paper to perform static resource analysis at the LLVM IR level. The static analysis techniques introduced by the author are based on traditional automatic complexity analysis and applied directly at the LLVM IR level. The mapping techniques from this thesis were used to leverage an existing ISA energy model to the LLVM IR level and enable the energy consumption analysis at that level. Furthermore, the hardware results presented in the paper were also contributed by the measurement and experimental setup introduced in this thesis.

- U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder, “Energy Consumption Analysis of Programs based on XMOS ISA-level Models.” In: *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR’13), 2014* [70].

  In this paper a generic resource analyzer [47, 93], was instantiated to perform energy analysis of XC programs [135] at the ISA level based on ISA-level energy models. A comparison to actual hardware measurements is included. The hardware measurements were achieved by the experimental setup introduced in this thesis, including the hardware energy measurement configuration and the test harness.

- U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder, “Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR.” In: *Foundational and Practical Aspects of Resource Analysis. Fourth International Workshop FOPARA 2015, Revised Selected Papers (M. V. Eekelen and U. D. Lago, eds.), Lecture Notes in Computer Science, Springer, 2016. In press* [69].

  The analysis introduced at the ISA-level in [70] is also applied at the LLVM IR level by the author of this paper. To enable energy estimations at this level, an initial version of the mapping techniques introduced in this thesis were used to leverage an existing ISA energy model to the LLVM IR level. The analysis on the two levels, ISA and LLVM IR, was tested on a number of benchmarks and compared against hardware measurements. A comparison between the two levels results demonstrated a small loss of accuracy at the LLVM IR level compared to the ISA ones, on an average less than three percentage points, but with the LLVM IR analysis able to cope with more and more complex benchmarks. This is because more program information is preserved at the LLVM IR level than the ISA level;

information which is essential to enable the static analysis employed. The experimental evaluation was performed by the use of the experimental setup introduced in this thesis, including the hardware energy measurement configuration and the test harness.

- K. Georgiou and U. Liqat, "Towards LLVM-Based Energy Consumption Analysis of Programs." In: *ICT Energy Letters, pp. 16–17, July 2014* [42].

  This work demonstrated some preliminary results of applying the static analysis techniques, introduced in [69], at the LLVM IR level. Again here an early version of the mapping techniques introduced in this thesis is used to convey energy information to the LLVM IR level. The experimental evaluation was performed by the use of the experimental setup introduced in this thesis.

The research conducted in the scope of this thesis contributed significantly to a number of related projects. Furthermore, this thesis work was exposed through several less research oriented activities with the aim of creating awareness on the importance of software energy-aware development beyond the research community. A list of the most significant contributions is given Appendix B.

1.4 Thesis structure

The rest of this thesis is organized into two parts. Part I focuses on related work and some background concepts that are related to this thesis. More specifically, Chapter 2 gives an overview of the CMOS power consumption and general guidelines for energy efficient software. Chapter 3 thoroughly investigates existing approaches for measuring or estimating the energy consumption of software on a specific hardware platform. Furthermore, the disadvantages of the existing approaches, which make it difficult to incorporate energy transparency in the daily software development life cycle, are identifying and taken into consideration to be addressed by the energy estimation techniques introduced in this thesis.

Chapter 4 gives an overview of the Xcore architecture, the architecture under investigation in this thesis, and the initial version of the ISA energy model [59] for the Xcore processor, which is being redefined for the purpose of this thesis in Chapter 5.

Part II describes the research carried out in the scope of this thesis. Chapter 5 covers the low-level analysis performed in this thesis, which tries to capture the dynamic behavior of the processor in regards to its energy consumption characteristics. The chapter describes the experimental setup introduced in this thesis to perform energy hardware measurements and redefines the adopted ISA energy model to meet the requirements of the introduced low-level analysis.

Chapter 6 introduces the high-level analysis which aims to leverage the low-level analysis to a higher level, namely the LLVM IR, and enables the energy estimation at that level. At the heart of the high-level analysis lies a novel mapping technique that can lift ISA-level defined resource models, for execution time, energy consumption and code size, to the LLVM IR level. The chapter provides the formalization of the mapping technique and its instantiation for the energy consumption of the Xcore processor.

Chapter 7 details the two estimation methods, SRA- and profiling-based. A running example is used across the whole chapter to demonstrate both the techniques features and important implementation details.

The experimental evaluation methodology, benchmarks, and results are presented and discussed in Chapter 8. This chapter also covers the applicability of SRA-based estimation to a number of alternative use cases, including design space exploration for deeply embedded applications and the performance evaluation of the profiling technique.

Finally, Chapter 9 concludes the thesis, outlines opportunities for future work and lists some research questions, raised by this thesis, to stimulate further research in the area of energy transparency.

Part I

BACKGROUND

CHAPTER 2

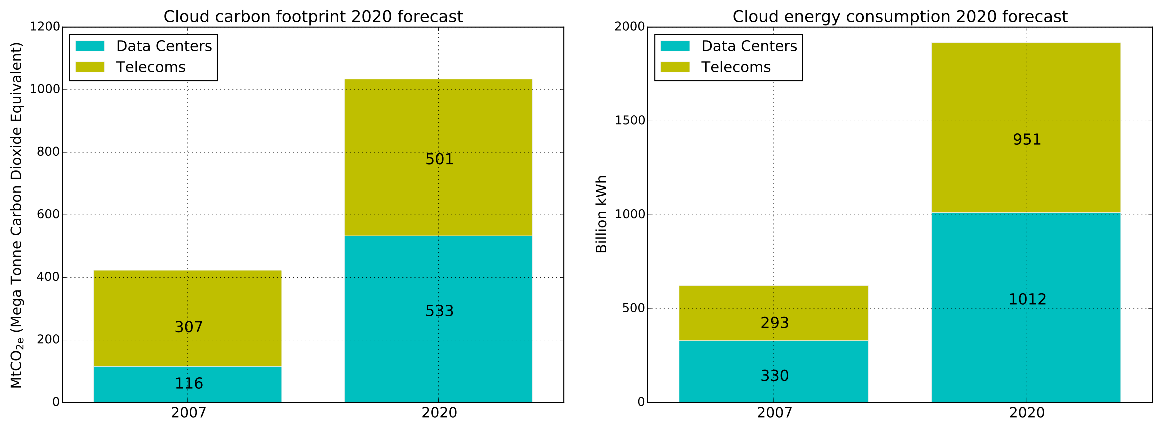
FUNDAMENTALS

This chapter provides the necessary background on the energy consumption of computing. At the same time, it further motivates the need for energy transparency techniques that can tackle the energy challenges faced by the ICT today. The chapter first provides an overview of the current ICT energy challenges, followed by a brief overview of the fundamental concepts of power dissipation and energy consumption of Complementary Metal–Oxide–Semiconductor (CMOS) devices; the dominant technology used to build today's microprocessors. Finally, the chapter concludes by explaining the significant impact that software has on the energy consumption of embedded systems, and by providing a set of guidelines on how to create energy efficient software for deeply embedded applications.

2.1 The ICT energy consumption problem

Green Computing is becoming critically important. However, the energy efficiency problem of computing is a multidimensional one and tough to tackle as a whole. Different ICT sectors and specific application constraints can yield a demand for energy efficient computing for various reasons. For example, the increase in battery-dependent mobile systems, the large energy-related operational cost of big data centers, and the negative impact on the environment due to the growing Carbon Dioxide (CO_2) emissions of ICT are all urging the scientific community for more energy efficient computing.

The long period of achieving further performance gains through Moore's law came to an end. Increasing the silicon density and at the same time scaling up the operating frequency is no longer an option due to overheating and quantum effects [133]. Quantum effects will make silicon transistors unreliable. Overheating affects the reliability of computation, decreases the lifetime of electronic components, and makes computing more expensive as advanced cooling technologies



(a) Cloud carbon footprint 2020 forecast. (b) Cloud energy consumption 2020 forecast.

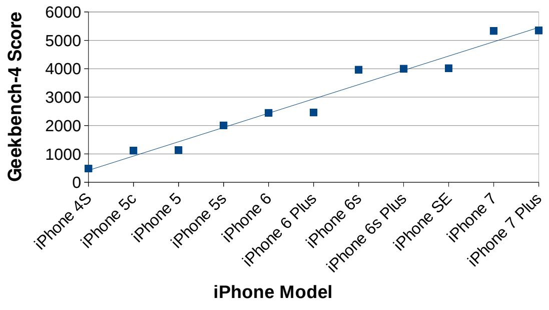
Figure 2.1: Data on the contribution of `cloud` computing to the climate change from the `GreenPeace` report released in 2014 [2].

are needed. Although the shift to multi-core solutions and three-dimensional stacked silicon allowed for further performance gains, the overheating problem remains a significant limiting factor [28, 63].

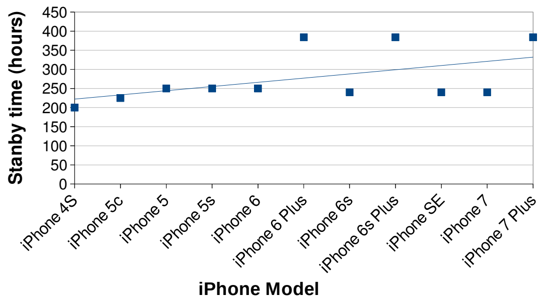
The replacement of the old, bulky desktops and laptops with smaller, more energy efficient devices such as smartphones and tablets, contributes to the reduction of the ICT energy consumption. However, new energy efficiency challenges emerged. Mobile computing sparked the rapid growth of the `cloud` online services that serve billions of mobile devices. Almost every mobile device application involves some interaction with the `cloud`. This includes video streaming, news feeding, social networks usage, software updates *etc.*

`GreenPeace` released a report in 2014 on the contribution of `cloud` computing to climate change [2]. Figure 2.1(a), represents the 2020 forecast for the carbon footprint of the two main `cloud` infrastructure elements; data centers and telecoms. A similar figure, Figure 2.1(b), represents the 2020 energy consumption forecast for the same `cloud` components. According to these figures, the `cloud` carbon footprint will at least double and the `cloud` energy consumption will at least triple, from 2007 to 2020. The research concluded that the energy consumed by the ICT sector is much larger than previously estimated, on the scale of around 70%. Another study conducted in 2013, [84], concluded that more than 10% of the world's energy consumption is spent on computing and that ICT's consumes 50% more than the aviation sector. Therefore, ICT's carbon footprint is becoming a threat to the environment. This can no longer be ignored.

The dramatic shift towards mobile computing comes with another great energy-related challenge. Mobile computing depends on limited portable energy sources; mainly batteries. Figure 2.2(a) shows the trend of the performance improvement for the `iPhone` devices released over the last four years. The figures are based on the `GeekBench-4` multi-core benchmark [3]. Figure 2.2(b) shows the standby time trend, retrieved from [4], for the same range of devices



(a) Geekbench-4 benchmark scores for the iPhone devices released in the last 4 years [3].



(b) Standby time for the iPhone devices released in the last 4 years [4].

Figure 2.2: The disproportional growth of mobile devices' performance compared to the standby time that batteries can offer.

found in Figure 2.2(a). While the performance of the devices has been growing significantly, the standby time growth has not been able to make substantial gains over the same period of time. Standby time captures the battery lifetime while the device is probably in its lowest power consumption mode. Therefore, when a device is under normal usage, the time a battery can keep the device alive is dramatically reduced compared to the standby time. This demonstrates the disproportional technological advancements between the mobile computing performance and the battery technology. Chemistry constraints limit the energy densities of electrochemical batteries [6]. Experts in the area, have been struggling for a breakthrough for decades.

To mitigate the above problem, the energy efficiency of mobile devices has been mainly improved by hardware innovation. It is a remarkable achievement that the battery lifetime of a mobile device, such as the `iPhone`, is kept at the same levels through the last four years, while significantly improving the device performance. However, there is still a lot to be done to expand the operational time of a mobile device; the time before recharging is needed. Any increase in the devices operational time will significantly improve the overall user experience. Therefore, energy efficiency is one of the most important selling points of a portable electronic device, so much so that it can determine the market success or failure of the device.

The challenge is even bigger when it comes to energy-critical deeply embedded applications. In this case, an energy-related failure will not just cause an inconvenience to the user. Instead, it can have severe consequences. For example, a health-monitoring device that fails to deliver crucial medical data due to inadequate energy budget can cause the loss of a human life.

After pervasive computing transformed into today's IoT, the number of energy-critical deeply embedded applications has increased dramatically. The majority of the computation performed by such applications depends on limited or unreliable sources of energy, such as energy harvesting. Striving for energy efficiency became the primary goal for electronic system engineers. This thesis aims to tackle the energy challenge that IoT and other energy-critical deeply embedded applications face, by enabling energy-aware software development.

2.2 CMOS energy consumption

CMOS technology of integrated circuit has been the dominant technology for building processors since 1981 where the first 32-bit microprocessor has been constructed using CMOS [147] transistors. One of the main reasons of CMOS success is the very low power consumption of the technology. [125]. This section explains the major sources of power consumption for processors built with digital CMOS circuits.

Equation (2.1) expresses the three main sources of power dissipation for a CMOS-based microprocessor; dynamic, static and short-circuit power dissipation.

$$P_{total} = P_{dynamic} + P_{static} + P_{short-circuit} \tag{2.1}$$

Dynamic power dissipation: This is the dominant source of power dissipation in CMOS circuits. It occurs due to capacitance charging and discharging of transistors when switching happens at CMOS gates. Equation (2.2) expresses the dynamic power dissipation as a product of the supply voltage, V, the effective capacitance, C, which is a function of wire length and transistor size, the clock frequency, F, and the activity factor which captures how often transistors are switching.

$$P_{dynamic} = a \cdot C \cdot V^2 \cdot F \tag{2.2}$$

Generally, dynamic power consumption is decreasing when:

- running on lower operating voltage and/or frequency;
- decreasing switching activity;
- decreasing switching capacitance.

For the last decades, supply voltage has been decreasing while the operating clock frequency has been increasing through successive microprocessors generations. Therefore, dynamic power dissipation has been improved.

Static power dissipation: The main source of static power dissipation is leakage current. Equation (2.3) express the leakage power dissipation in CMOS devices as the product of the leakage current, $I_{leakage}$, and supply voltage, V. Leakage power has been a major concern since the introduction of microprocessors with gate lengths smaller than 65 nm, because it can now be as high as 30-50% of the total integrated-circuit's power consumption [81].

$$P_{leakage} = I_{leakage} \cdot V \tag{2.3}$$

Generally, static power consumption is decreasing when:

- Lowering operating voltage;
- Having fewer leaking transistors (turning transistors off).

Short-circuit power dissipation: This occurs when both p-type Metal-Oxide-Semiconductor logic (pMOS) and n-type Metal-Oxide-Semiconductor logic (nMOS) transistors (the transistor types CMOS gates are built from) are on at the same time as a CMOS gate switches. For a short period of time, while the gates are switching, a direct path is established from the power supply to the ground, causing the short-circuit power dissipation. Many methods have been proposed for the estimation of the short-circuit power dissipation of CMOS circuits [49, 130]. The energy model used in this thesis factors the short-circuit dissipation into the dynamic and static power consumption [59].

Energy consumption: Energy, E, is the integral of the power dissipation, P, during a period of time, T, as given by Equation (2.4).

$$E = \int_0^T P(t)\,dt \tag{2.4}$$

Therefore, the energy consumption of a microprocessor, E_{mp}, can be given by Equation (2.5) as the product of the average power dissipation, P_{avg} during a certain time period T, with the duration of that time period. Therefore decreasing the execution time and/or the power dissipation of a system decreases the energy consumption.

$$E_{mp} = P_{avg} \cdot T \tag{2.5}$$

2.3 Writing energy efficient software for embedded applications

Software is the main source of energy consumption on an embedded system. It is estimated that up to 80% of the total energy consumption of an embedded system is contributed by software related activities [82]. This is analogous to the relation of a car and its driver. No matter how fuel-efficient a car is, the car utilization by the driver will determine the energy consumed. Thus, people with different driving behavior will cause a different fuel consumption, even in the case they both follow the same route by driving the same car. In the case of an embedded application, software is the driver and hardware is the vehicle. Therefore, writing energy-efficient software is critical to the overall energy efficiency of an embedded system.

A survey on software practices for creating energy-efficient software for embedded systems was contacted by Roy and Johnson in 1997 [108]. The authors introduced a set of guidelines to enable embedded software developers to create energy-efficient software. Although the guidelines are almost twenty years old, they are still applicable to today's embedded systems. A summary of these guidelines is provided below:

- **Choose the best algorithm:** The choice of algorithm is considered to be the most important step for an energy-efficient embedded application. The algorithm has to both fit the selected hardware and be optimal for the problem under consideration. For an algorithm that does not meet the above two conditions, it is highly unlikely that it will utilize any of the hardware's energy-saving features. Moreover, architecture-specific compiler optimizations will always perform better with an algorithm that better fits the target architecture.

- **Optimize memory usage:** Memory is one of the most energy hungry components of an embedded processor. After choosing the appropriate algorithm to tackle the problem at hand, it is important to optimize its memory usage. Moving up through the memory hierarchy is both expensive in time and energy. Therefore, for an energy-efficient software the memory size and the memory accesses have to be the minimum possible. This can be achieved by utilizing the registers and the memory bandwidth as efficiently as possible.

- **Use available parallelism:** Optimizing for performance can also yield significant energy savings in the majority of cases, either through putting the system to sleep earlier (run to idle) or by utilizing voltage and frequency scaling. Therefore, increasing the parallelism of the software to gain performance can at the same time save energy.

- **Use the hardware's power-management features:** Many of the embedded processors provide a variety of operating modes and the ability to apply voltage and frequency scaling. Putting the processor in a low operating mode when it is idle, no computation happens, can minimize the static power dissipation. Furthermore, the gains in performance made by using a multi-core system can be interchanged with energy savings through voltage and frequency scaling.

- **Energy efficient instruction selection:** As explained in Section 2.2, the dynamic power dissipation of an embedded processor heavily depends on the switching activity of its CMOS transistors. Therefore, reordering instructions in a way that can minimize the switching activity of the Central Processing Unit (CPU) and data-paths can reduce the overall systems' energy consumption. This is more applicable to deeply embedded systems, where the processor's energy consumption is significant in regards to the whole system.

The above guidelines touch all the software abstraction levels of an embedded system, presented in Figure 2.3. At the higher levels of software abstraction, the programmer has to make the decisions. These include the choice of algorithm and then optimization of the selected algorithm at the source code level. At the lower levels of software abstraction, such as the compiler's IR

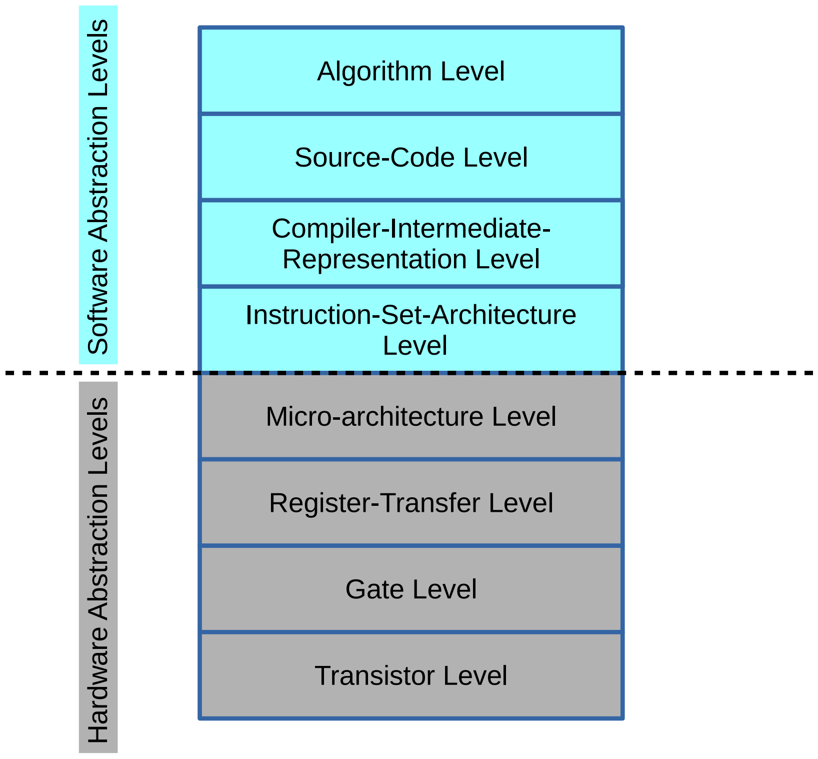

Software Abstraction Levels
Hardware Abstraction Levels
Algorithm Level
Source-Code Level
Compiler-Intermediate-Representation Level
Instruction-Set-Architecture Level
Micro-architecture Level
Register-Transfer Level
Gate Level
Transistor Level

Figure 2.3: The main software and hardware abstraction levels of deeply embedded systems.

and the ISA, the compiler performs the optimizations. However, compilers' optimizations are targeting performance and are only improving energy consumption as a side effect. Experienced deeply embedded developers will often manually implement and optimize their application at the ISA level, to achieve the optimal code in regards to execution time, code size, and energy consumption. This needs a high degree of understanding of the architecture at hand. While moving up into the embedded systems software stack, there is a reduction in the amount of feedback on how different choices or code transformations impact the execution on hardware. When it comes to energy, programmers, compilers, and embedded system designers have little understanding of how their choices affect the hardware's energy consumption. Roy and Johnson, [108], acknowledged this in their paper and stated that the key element of energy-aware development is the availability of techniques that can attribute the energy consumption of a system to its software components at each level of software abstraction. Many others supported this idea in the following years [1, 17, 31, 55, 82, 131].

2.4 Conclusion

This chapter examined the various energy-related challenges that ICT faces today. Although green computing is important at all areas of ICT, the energy-critical nature of some deeply embedded systems urges for more immediate attention. Energy-related failures of such embedded systems can cause severe consequences, such as in the case of health-care IoT applications. Therefore, this thesis focuses on addressing the energy challenge of critical deeply embedded systems.

Although the importance of developing energy-efficient software has been acknowledged for more than 20 years, there is still a lack of techniques that can enable energy-aware software development. The next chapter investigates the reasons behind this problem and establishes the requirements for new energy transparency techniques that will enable the energy-aware development needed.

Chapter 3

Software Energy Consumption Estimation

This chapter takes a close look at existing approaches used to reason about the energy consumption of software. The different techniques are presented together with examples of previous work, highlighting both the advantages and disadvantages in each case. The chapter continues with identifying the gaps in the existing approaches, which make it difficult to incorporate energy transparency in the daily software development life cycle. Then it concludes by explaining the rationale behind the decisions made in this thesis.

Two main approaches exist to reason about the energy consumption of a program written for a particular hardware platform. The first one is by performing physical measurements on real hardware while the program is being executed. The second one is by the use of estimation techniques. Such estimation techniques fall under two main categories; either profiling based estimation or estimation based on static analysis. Table 3.1, summarizes the main advantages and disadvantages of using each one of the aforementioned methodologies.

The following sections will give a more detailed view on each of the approaches listed in Table 3.1.

3.1 Physical measurements

Measuring the energy consumption during a program's execution is the most accurate method, subject to the accuracy of the measuring equipment used, compared to any other method that estimates the energy consumption. The most common methodology for measuring the energy consumption of embedded systems is by the use of a shunt resistor placed in the power supply circuit. The voltage drop across the resistor depends on the value of the resistor. This value should not be too large to avoid a high voltage drop that could affect the device operation.

Method	Pros	Cons
Physical Measurements (Section 3.1)	• Most accurate method; • Data sensitive; • Can account for peripherals energy consumption.	• Hardware modification to access power supply is needed; • Not always possible to access a specific component's power supply; • Needs hardware knowledge, not common for software developers; • Software fine-grained energy characterization is impractical; • Difficult to capture energy consumption bounds.
Profiling (Section 3.2.1)	• More accurate than Static Analysis; • Can scale to account for system-level energy consumption; • Easier to integrate into software development life cycle, depending on the estimation speed of the method.	• Less accurate than direct measurements; • Depends on energy modeling or hardware counters; • Needs a method to collect execution statistics; • Difficult to capture energy consumption bounds.
Static Analysis (Section 3.2.2)	• Can give information about energy consumption bounds; • Usually easier to integrate into software development life cycle; • More suitable for deeply embedded systems, due to the more predictable nature of the architectures.	• Needs energy modeling that is suitable for static analysis; • Hard to scale to system level; • Impractical for capturing the actual energy consumption;[2] • Data insensitive.[1]

[1] Does not capture the instruction energy cost variation that depends on (the circuit switching activity caused by) the operands used.
[2] Given a specific set of inputs to a program, static analysis will retrieve always bounds instead of the actual energy consumption.

Table 3.1: The advantages and disadvantages of the two main methods used to reason about the energy consumption of a program written for a specific hardware platform; a) direct measuring and b) estimation by either profiling or static analysis techniques.

Because the current through the resistor is proportional to the observed voltage drop, by applying Equation (3.1) the power consumption can be periodically calculated.

$$P = I \cdot V_{\text{bus}} \quad \text{where} \quad I = \frac{V_{\text{bus}} - V_{\text{shunt}}}{R_{\text{shunt}}} \tag{3.1}$$

where V_{bus} and V_{shunt} are the voltage measurements taken on the two sides of the resistor.

Measuring can be applied to a system as a whole by accessing the main system's power supply, or to individual components of a system, such as the CPU, memory or any hardware accelerator, by monitoring the power supply of the specific component. The main drawback of this method is that the majority of devices do not come with a built in access to the power supply of their individual components. Therefore, custom hardware modifications and advanced hardware knowledge are needed for being able to measure the energy consumption. This kind of skills is not common for the majority of software developers. All this makes it hard to integrate energy measurements into the daily software development activities.

Even though measuring is the most accurate method for retrieving the energy consumption of a program, it is not very practical when it comes to fine-grained energy characterization of software. Attributing the energy consumption to various software components, from individual instructions up to CFG basic blocks and whole functions, can be challenging. Doing so usually requires expensive measuring equipment such as oscilloscopes, that can support a high sampling frequency. Moreover, such solutions require huge storage to save the large amount of data collected for post-processing.

Although end-to-end measuring is the most common method adopted by industry to capture the best- and worst-case of a resource usage, such as time or energy, for a program, it is generally known that it does not provide guarantees [140]. The observed best and worst case are obtained by exercising only a subset of the possible execution test cases, and therefore it tends to overestimate the actual best case and underestimate the worst case. Figure 3.1 visualizes this. The lower curve demonstrates the distribution of energy consumptions that is usually captured by measuring the energy consumption for a set of a program's inputs. The dark upper curve represents the distribution of all possible energy consumptions for the same program. The observed energy consumptions failed to capture the actual Best Case Energy Consumption (BCEC) and Worst Case Energy Consumption (WCEC). Therefore, there is a need for techniques that can retrieve tight upper and lower energy consumption bounds. Such estimated tight bounds are closely overestimating the actual worst case or closely underestimating the actual best case, and are safe to be used in critical applications.

Retrieving the actual bounds with end-to-end measurements in most cases will require an exhaustive search through all the possible execution scenarios of a program. Brute forcing the whole input space of a program to exercise all the possible execution scenarios is in most cases impractical due to the large combination of possible inputs. Therefore, when using end-to-end measurements to provide some form of guarantees for the retrieved bounds an over-provision is added to the maximum observed time/energy. Building a critical system from a set of different components which all have their own over-provision to account for worst-case scenarios, might lead to a significant overestimation of the overall actual worst-case of the system as a whole. This is not desirable in the case of systems with limited resources, such as an IoT application running on a battery.

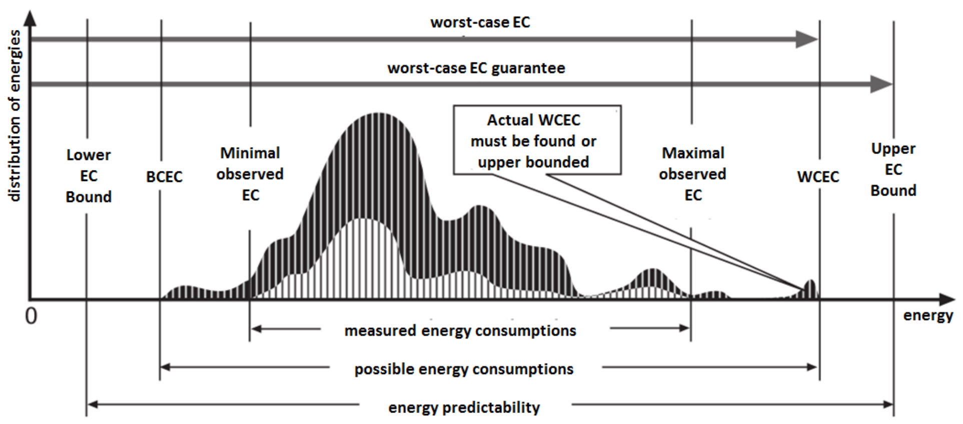


Figure 3.1: End-to-end measurements weakness to capture BCEC and WCEC: The lower curve demonstrates the distribution of energy consumptions that is usually captured by measuring the energy consumption for a number of a program's inputs. The dark upper curve represents the distribution of all possible energy consumptions for the same program. The observed energy consumptions failed to capture the actual BCEC and WCEC. (Modified from [140]).

3.2 Energy consumption estimation

Estimating the energy consumption of a program requires two main elements: an analysis technique that performs the estimation given a specific program and a specific hardware platform, and a way to convey energy information to the analysis. As already stated, there are two kinds of energy estimation techniques; profiling-based and SRA-based energy estimation techniques.

The most common way of conveying energy information to the analysis is via energy modeling. Energy modeling allows the energy consumption of a processor executing a program to be simulated or otherwise analyzed to predict how certain parameters will affect the total energy consumed. Energy modeling can be performed at various levels of abstraction, from gate- or transistor-level in detailed hardware simulation [15], up to high-level modeling of whole applications [8].

In the next sections, existing work on energy consumption estimation based on the two estimation techniques, profiling and static analysis, and the energy models used in each case, will be examined.

3.2.1 Profiling-based energy consumption estimation

For software energy profiling, there are two main requirements. First, a technique that instruments a program's execution and collects data that represent the program's dynamic behavior. Code instrumentation, simulation and hardware performance counters are some of the most popular techniques used to collect such data. The second requirement is a method of conveying energy information to the various entities, events in the set of the collected instrumentation data.

This is typically done either through an energy model or, in the case of hybrid techniques [1], by monitoring the power consumption during a program's execution. Software energy estimation using profiling is mainly done via five different approaches:

1. energy modeling at architecture level combined with cycle-level system simulators;
2. ISA level modeling and profiling;
3. compiler intermediate representation energy modeling and profiling;
4. software function-level modeling and profiling;
5. using Performance Monitoring Counters (PMCs) with profiling.

The rest of this section examines each one of the above approaches in more detail.

3.2.1.1 Architecture level energy estimation

In [19] the authors introduced *Wattch*, an architectural cycle-level simulator that estimates the CPU power consumption. Energy modeling was based on building parameterized models for common hardware structures of superscalar microprocessors, such as data and instruction caches, register files, functional units, clock buffers *etc.*These models were integrated into the SimpleScalar architecture simulator, which records the various units accessed per cycle and estimates the overall energy consumption of an application. The relative accuracy of the proposed models is within 10-13% on average. The approach was used for estimating the overall energy consumption of an application, and for exploring possible power/energy consumption optimizations at both the architecture and the compiler level.

A Register Transfer (RT) level power estimation tool, *SimplePower*, was introduced in [146]. This tool is based on transition-sensitive energy models for each functional unit, where the switching activity of a hardware block is considered based on a set of possible input transitions. The SimpleScalar compiler is used to translate high-level C code to *SimplePower* executables. These executables can be then simulated by *SimplePower* to provide cycle-by-cycle energy consumption estimations. Furthermore, it can provide energy estimations for the processor's datapath, memory, and on-chip buses. The framework was used to evaluate the effect of high-level algorithmic, compilation and architectural trade-offs on energy consumption.

Modeling and profiling at such low design levels is not practical for most commercial embedded processors since their lower level circuit information, such as effective capacitance of major architectural blocks, is not available. Moreover building an energy model at such low levels is a difficult and time-consuming task due to the high complexity involved. The estimation is slow and not practical for fine-grained energy characterization of software.

[1]Hybrid techniques combined both physical measurements and profiling-based energy estimation techniques with the aim of overcoming the limitations of the two.

3.2.1.2 Instruction set architecture modeling and profiling

In [127] an ISA level energy model was proposed that treated the hardware as a black-box and obtained energy consumption data through hardware measurements of large loops of individual instructions. Equation (3.2) shows Tiwari's energy model, in which the energy of a program, E_p, comprises a series of base costs, B, inter-instruction overheads, O, and external effects, E. Each ISA instruction i has a base power, B_i, and is executed a number of times, N_i. In addition, sequences of different instructions, i,j, cause processor activity, incurring an overhead $O_{i,j}$ multiplied by the number of times instruction sequence i,j occurs, $N_{i,j}$. Finally, external effects such as cache behavior (energy consumed during a hit versus a miss) are included in the energy consumption estimate for the program.

$$E_p = \sum_i (B_i \cdot N_i) + \sum_{i,j} \left(O_{i,j} \cdot N_{i,j}\right) + \sum_k E_k \tag{3.2}$$

The modeling technique was initially applied to x86 and SPARC architecture processors, capturing the energy cost of ISA instructions with an accuracy of within 10% of hardware measurements. This led to further research that combined ISA energy models with instruction set simulators and profilers to extract energy estimations [51, 109, 110, 122]. The above techniques deliver programs' energy estimations with an accuracy ranging between 1% to 10%.

While for the processors examined in [127], an average value (base cost) was sufficient to characterize the power consumption of instructions, the authors in [110] demonstrated that the variability of power due to internal switching activity, caused from different operands, is quite significant for simpler embedded processors such as Digital Signal Processing (DSP) processors. Thus, by using linear regression they formulated a model that can consider the effect of operands on various instructions. Similarly, in [122], the authors built an ISA energy model which takes into account the switching activity of busses. They also used regression analysis to extract their energy model, but they avoided the need for Very-High-Speed-Integrated-Circuit (VHSIC) Hardware Description Language (VHDL) simulations to collect their data. Avoiding such simulations is essential because detailed information about the internal structure of most commercial processors is not available. Both of the above methods need their simulators to be able to track the operands used during each instruction execution, in order to use their data-sensitive energy models.

In [102], the authors created two different flavors of ISA based energy models for the SPARC Leon 3 simulated processor, and compared them with each other to identify which energy model characterization yields the best accuracy. The first energy model considers the energy cost of individual ISA instructions and the second one the energy cost of pairs of instructions. They demonstrated that in 90% of the cases, a pair of instructions energy cost was lower than summing the energy costs of the two individual instructions using the single-instruction-based energy model. The average difference was -12%. Furthermore, they examined the effect of a cache presence, data switching activity, and register correlation between subsequent instructions.

Although, they demonstrated that different data could create up to 60% variation on the energy cost of an instruction, they considered it impractical to construct a data sensitive energy model. Instead, they used a statistical approach to find the typical number of simultaneously switching bits and account for it in their energy model.

The authors also demonstrated that there is a significant impact on the energy consumption of a pair of instructions with register correlation, depending on the number of extra cycles caused by the correlation. This was not factored into their energy model. Their experimental evaluation for the 2-instruction-based model was done by the use of a cycle-accurate ISS. The evaluation demonstrated a smaller than 6% and 12% error, for the configuration with and without cache respectively, compared to gate-level energy simulation results. For the single-instruction-based energy model, there was a further error of 8.65% and 17.45% for the configuration with and without cache, respectively, compared to the two-instruction-based model.

In [115], it is shown that in the presence of caches and other architecture performance enhancements, their overheads overshadow instruction specific energy variation and therefore modeling inter-instruction effects is unnecessary. An alternative, instruction-class profiling was used and combined with statistics from an ISS to retrieve energy estimations.

Generally, for all the aforementioned ISA-based energy consumption estimation techniques, the modeling and profiling implementations heavily depend on the processor characteristics. For example for smaller processors switching activity must be considered but for larger embedded devices average values provide sufficient accuracy. Profiling can be done via hardware simulation or higher level ISA simulators, depending on the availability of the hardware details and the required estimation accuracy.

ISA based energy modeling and profiling approach became popular for embedded processors because a processor's ISA is the bridge between the hardware and software. Therefore, modeling at the ISA allowed for attributing energy costs to software components such ISA CFG basic blocks. For energy modeling to be useful to a software developer, the model must convey information that can be related to the code the developer is writing. Moreover, ISA energy modeling can convey energy information to compilers, making visible opportunities for energy targeted specific compiler optimizations. ISA-level energy modeling is more suitable for embedded devices, which focus on predictability and low power rather than performance. For more complex processors, such as superscalar processors, ISA energy modeling is impractical due to the inability of capturing their complex energy behavior at the ISA level. This is because for performance-enhancing hardware components, such as caches, is difficult to statically capture their behavior.

3.2.1.3 Compiler intermediate representation energy estimation

The majority of research done with the aim of enabling energy-aware compilation has been focusing on ISA energy consumption estimations rather than estimations at the IR of compilers. The IR can be considered the heart of a compiler as it is the means that enables or make it easier to apply compiler optimizations. Although IR-based energy estimation techniques can bring energy transparency directly into the compiler, there is a lack of such approaches. This is due to the difficulty of creating accurate energy models at the IR level, and associating execution statistics to the IR code.

In [17], energy characterization of LLVM IR code is performed by linear regression analysis. LLVM IR instructions statistics for a set of programs and the programs measured energy consumption were used to train the energy model. The energy model can be applied to instrumentation statistics collected during a program's execution on a host machine, to estimate the performance and the energy consumption of the program. The estimation quality heavily depends on the ability of the regression analysis to capture the energy behavior of the processor at the LLVM IR level. This is difficult to be adequately addressed by linear regression, which can not account for every possible compiler optimization or processor dynamic behavior. Furthermore, their instrumentation technique is based on collecting basic block execution counts, but this is done by executing programs on the host machine rather than the actual platform. Compiling and executing on a higher-end Personal Computer (PC) rather than the targeted deeply embedded device could yield a significantly different execution profile than the one expected from the target machine. For the set of simple benchmarks used to evaluate their techniques they achieved an average absolute error of around 6%.

3.2.1.4 Software function-level modeling and profiling

It is standard practice in software development to encapsulate frequently used code into functions and then into software libraries with a well-defined interface. Such library functions can be characterized in regards to their energy and timing behavior. This characterization helps the programmers to understand the implications of invoking function calls from software libraries into their code onto the relevant resources, such as energy. A choice between libraries with the same aim but different implementation can be made based on their energy/time characterization.

In [105], a methodology was proposed for estimating the energy consumption of embedded software, through characterizing the energy consumption of built-in library functions and basic instructions. According to the authors, for a specific architecture, a "power data bank" can be constructed that includes time and energy consumption information for a number of library functions and basic instructions. The energy and timing values can be obtained by the use of any architecture-level power simulator. Combining these values with execution statistics collected from profiling, they were able to obtain energy estimations on a set of benchmarks. The observed accuracy was within 3.5% of hardware measurements.

Estimating the energy consumption in the above way has some significant drawbacks. Firstly, user-defined functions can be the dominant part of an application, rather than library functions. In these cases, ignoring the energy consumption of user-defined code can lead to unrealistic estimations. Secondly, for library functions for which their energy consumption heavily depends on input parameters, a single-value energy profile will yield a low estimation accuracy. Finally, the approach limits the developer to use the same compilation flags that were used for the energy characterization phase. The choice of compilation flags has a significant impact on the energy consumption of a program, due to the different set of optimizations enabled [13].

3.2.1.5 Energy modeling and profiling based on performance monitoring counters

Energy modeling and profiling at the ISA level are insufficient for more complex architectures or system-level energy consumption estimation. In such cases, statistical PMC-based estimation is preferable.

A software energy profiler, called `Eprof`, was developed in [112] with the aim of attributing the consumed dynamic energy back to the software that caused this consumption. PMCs were used to create a linear energy model for the CPU and the memory of the two processors under consideration; the `Intel Atom N450` CPU and the `Intel Core 2 Quad Q6600` CPU. Selecting the appropriate set of PMCs to be used for the energy model construction, was critical to achieving a good estimation accuracy. PMC execution statistics are used to feed the energy model in order to retrieve energy consumption estimations. The average error for the CPU model is stated to be below 10%.

In [113], the authors created a many-core, multi-threaded energy model at the ISA level, for the `Intel Xeon Phi` processor. The instruction-level energy model is parametric to the number of used cores, the number of active threads per core, the instruction types and the operand sources. PMCs are used to feed the model with execution statistics for the relevant combinations of instruction types and operand sources. An accuracy of 5% was achieved when applying the estimation technique on a set of real-world benchmarks.

Run time power estimation can also be enabled by analytical energy modeling using PMCs. In [23], the authors introduced a linear power estimation model that uses PMCs to allow for on-the-fly energy consumption estimation. The underlying power model uses off-line estimated power weights that map PMC values to processor and memory power consumption. The power model is also parametric to the various voltage, frequency configurations available for the processor under consideration, the `Intel PXA255`, to allow for runtime energy-aware decisions. An average error of 4% was observed for the `SPEC2000` benchmark suite and a set of Java benchmarks.

For a given processor, energy modeling and profiling using PMCs can be challenging, due to the limited types of the events that can be monitored and the restricted number of counters that can be sampled simultaneously. The same constraints apply when trying to port the PMC-based modeling and estimation techniques to a new target. Moreover, runtime use of such PMC-based energy estimation methods can cause a significant overhead on the system's performance.

3.2.1.6 Hybrid approaches

Hybrid approaches are also researched [36, 136] to overcome the limitations of both physical measurements and profiling-based energy estimation techniques.

In [136], the authors extended a performance evaluation interface, called `PAPI`, to measure and report energy values. The new version of `PAPI` was extended with an interface able to accommodate measurements from a diverse set of hardware, such as external multi-meters or internal measurement mechanisms embedded in the hardware. This extension avoids the need for rewriting the instrumentation code when porting software to another machine with a different set of measurement hardware. Furthermore, the new `PAPI` version can collect profiling information from PMCs at the same time of measuring energy. This allows for a more comprehensive analysis of the performance and energy consumption behavior of a system. One of the main drawbacks of using `PAPI` is that it provides system-wide energy measurements which can not be attributed to the various software components.

In contrast, `PowerScope`, a tool introduced in [36] for estimating the energy usage of mobile applications, can map the energy consumption to program structure. The tool uses an external multimeter to sample the current drawn by the system under examination, during a program's execution. At the same time, the system's activity is sampled through appropriate kernel software support. Both energy measurements and system activity information are time-stamped, to allow a post-processing, off-line analysis that provides the energy profile of the various processes and procedures executed. The tool was used to identify energy optimizations on an adaptive movie player application. Applying these optimizations the overall energy consumption of the application was reduced by 46%.

Although hybrid techniques try to overcome the limitations of their combined underlying techniques, they are more expensive to implement and incorporate into the software development life-cycle. Software developers are usually not able to access power monitoring hardware. Moreover, fine-grained attribution of energy consumption to software structure is difficult, as it requires a large amount of space to store fine-grained power measurements and profiling data.

3.2.2 Energy consumption estimation based on static resource analysis

SRA is a methodology to determine usage bounds of a resource (usually time or energy or both) for a specific task when executed on a piece of hardware, without actually executing the task. This requires accurate modeling of the hardware in order to capture the dynamic functional and non-functional properties of task execution. Determining these properties accurately is known to be undecidable in general; meaning that no complete automatic tool can exist that can reason about a program's semantics and will guarantee both correctness and completeness at the same time. Therefore, to extract safe values for the resource usage of a task, a sound approximation is needed [18, 140], and thus analysis tools must fail in some instances. This is done by allowing the analysis tools not to terminate.

SRA has been mainly driven by the timing analysis community, with the objective of retrieving the WCET of programs executed on a specific platform. Therefore, in this section, a brief overview of the main SRA techniques used for WCET analysis is provided. Then the IPET technique is presented which is the one used for this thesis. The section will conclude with reviewing a number of works that used SRA techniques to retrieve energy consumption estimations instead of execution time.

3.2.2.1 WCET static analysis components

Generally, for performing an accurate WCET static analysis, there are four essential components [140]:

1. *Value analysis or data-flow analysis*: provides information that helps to estimate loop bounds, to identify infeasible paths and to analyze the behavior of the data cache by predicting the addresses of data accesses. Abstract interpretation is a technique that is often used to facilitate data flow analysis in the context of the WCET analysis [116, 126]. The technique dates back to 1970s seminal papers [26, 27]. Generally, abstract interpretation aims to reason about a property of a program, without performing all the calculations defined by the concrete semantics on the concrete domain. Instead, a partial execution is performed using a new set of abstract semantics on an abstract domain, that performs only the computation needed to reason about the property under examination. Abstract interpretation ensures a sound approximation of the semantics of a program. This guarantees the correctness of the program analysis, and hence the extraction of safe WCET bounds.

2. *Control-flow analysis*: aims to identify the dynamic behavior of a program. It takes as input the information from value analysis. Moreover, the call graphs and CFG of the program are constructed. The results of this stage is usually a number of flow facts that constrain the possible transitions on the program's CFG.

3. *Low-level analysis or processor-behavior analysis*: attempts to retrieve timing costs for each atomic unit on a given hardware platform, such as an instruction or a CFG basic block for a processor. Retrieving timing models is usually possible for simple architectures such as embedded processors. More complex processors that are mainly focused on performance are more prompt to timing anomalies and therefore retrieving a timing model is a hard challenge.

4. *Calculation*: uses the results from the two previous components to estimate the WCET. There are mainly two approaches used for this stage; automatic cost analysis techniques and worst case path analysis techniques.

3.2.2.2 SRA based on automatic cost analysis techniques

Automatic cost analysis techniques based on setting up and solving recurrence equations date back to Wegbreit's [137] seminal paper, and have been developed significantly in subsequent work [7, 29, 93, 107, 129]. Automatic cost analysis usually consists of two stages. The first stage takes as input a program and a cost model. A cost model for the resource of interest can be provided by performing low-level analysis on the architecture under consideration. The second step is the creation of Cost Relations (CRs). These are usually recursive equations which capture the cost of the program, in respect of the resource of interest, in terms of the size of its input data.

```
1 int fact(int i) {
2   if (i<=0)^a
3     return 1^b;
4   return (i *^d fact(i-1))^c;
5 }
```

(a) Original program.

a. $C_{fact}(i) = C_a + C_b \qquad if\ i <= 0$

b. $C_{fact}(i) = C_a + C_c(i) \qquad if\ i > 0$

c. $C_c(i) = C_d + C_{fact}(i-1)$

(b) Extracted cost relations.

Figure 3.2: CRs extracted for the factorial function written in the C programing language.

The right-hand side of Figure 3.2, gives an example of CRs extracted for the factorial function written in the C programming language and presented on the left-hand side of Figure 3.2. The i used in the cost functions stands for the input parameter of the factorial function. The constraints attached to the CRs (a) and (b) infer their applicability conditions, and they are extracted based on the program's branch statement at line 2. For instance, CR (a) captures the cost when the if branch at line 2 is taken (stated in the condition $if\ i <= 0$). This is the summation of the cost of executing the if statement at line 2, C_a, and the cost of executing the return statement at line 3, C_b. CR (b) captures the cost when the branch at line 2 is not taken, and therefore accounts for the cost of the if statement execution at line 2, C_a, and the cost of the return statement execution at line 4 C_c. CR (c) captures the cost of the return statement at line 4 as the cost of the multiplication operation, C_d, together with the cost of the recursive call $C_{fact}(i-1)$.

These CRs are recursive; iteration is needed for computing the cost of the program. To be able to compute bounds using recursive CRs, the relations need to be solved to obtain a closed-form solution; without recurrences. Solutions can be retrieved by using off-the-shelf solvers such as the Practical Upper Bounds Solver (PUBS) [7].

An example of such solution is given in Equation (3.3), where a closed-form energy consumption upper bound is retrieved for the program in Figure 3.2(a), using the CRs given in Figure 3.2(b). To obtain this parametric bound C_a, C_b and C_d are substituted with the energy costs required to execute their low-level instructions, as given by the ISA energy model introduced in Chapter 4. Then the resulted CRs are passed to the PUBS solver which obtains the Equation (3.3).

$$C_{fact}(i) = 4563 + 7878 \cdot i \qquad (in\ pJ) \tag{3.3}$$

Some of the most important advantages of the automatic cost analysis approach are:

- it is programming-language independent;
- it can be fully automatic;
- it can cover a variety of complexity classes;
- it provides good accuracy [44, 69, 70];
- it can be used for a variety of resources, such as memory, energy and time;

The main drawback of automatic cost analysis is that it is usually difficult or even not possible to extract a closed-form solution for the CRs of a program [44]. Therefore, the approach is limited and does not scale to large programs with complex structure.

3.2.2.3 SRA based on worst case path analysis techniques

The most common worst-case-path-based techniques, [32], used for calculation of the WCET are the path-based techniques [119], the tree-based methods [67] and the IPET [65, 66].

The path-based calculation retrieves the WCET by comparing the time cost of different execution paths of a program. The path with the longest execution time is selected as the WCET of the program. The search for the WCET path is usually done by the use of traditional graph algorithms for finding the longest path, such as the Dijkstra's algorithm. Use of such longest-path search algorithms requires the knowledge for the basic block execution counts prior the algorithm's application. Combining the basic block execution counts with the timing cost of each basic block will determine the overall execution cost of each basic block. Flow information is used to explore several sub-paths of the CFG to assign an execution count to each basic block. The main drawback of the above approach is that they do not scale well when considering programs with bigger and more complex CFGs, due to the larger number of paths that need to be explored by the longest path search algorithms. Finding the longest path is an Non-deterministic Polynomial-time (NP)-hard problem in general. Furthermore, with path-based approaches, it is difficult to account for flow information stretching across loop-nesting levels [61].

Tree-based method, also called "timing schema" [101], work by examining smaller parts of the program and then trying to estimate the WCET of larger parts of the program in a bottom-up approach [22]. This is a simple method with lower computation effort compared to the path-based methods. The main disadvantage of the tree-based methods is that it does not allow to consider generic flow facts in a direct way, since the analysis is usually performed on a syntax tree and lacks a holistic image of the overall global execution of the program [85].

The main advantage of the path- and tree-based methods is that they do not only report the execution time for each path but also explicitly report the examined paths.

IPET is the most commonly used technique for WCET calculation. It expresses the search of the WCET as an Integer Linear Programming (ILP) problem where the execution time is to be maximized under some constraints on the execution counts of the basic blocks. In contrast with the path- and tree-based methods, IPET defines the WCET path by its set of blocks and their respective execution counts but not the order which they are executed. IPET is also the technique adopted for the energy consumption SRA introduced in this thesis. Therefore IPET is examined in more depth in the following section.

3.2.3 Implicit Path Enumeration Technique (IPET)

Linear programming [16] is a technique for optimizing a mathematical problem whose requirements are represented by a system of linear constraints over a set of variables, together with an objective function. Constraints are essential to limit the values that the variables can take to a feasible region. By maximizing (or minimizing according to the problem requirements) the objective function, an optimal assignment of feasible values to the variables can be obtained. If the values need to be integers then the complexity of the problem changes from polynomial to NP-hard and its called ILP [140]. Linear Programming gained popularity due to its ability to express complex mathematical problems and its solve-ability strength. The methodology has been adopted in a wide range of applications, namely WCET estimation, resource assignment [35, 97], scheduling [24, 117, 141], modeling simple embedded processors [64, 66], and many more.

One of the most popular methods used for WCET analysis is the IPET [32, 65, 95, 103, 104, 126]. In this approach, the CFG of a program is expressed as an ILP system, where the objective function represents the execution time of the program. The problem then becomes a search for the WCET by maximizing the retrieved objective function under some constraints on the execution counts of the CFG's basic blocks. The main advantage of this technique is the ability to determine the basic blocks in the worst case execution path and their respective execution counts without the need to extract the explicit worst execution path (ordered list of the executed basic blocks). This is more efficient than the path-based techniques for retrieving WCET bounds [32].

IPET has been very successful for WCET estimation when applied to simple processors but does not scale well when used for the analysis of more complex processors. Processors with caches and complex pipelines can result in a high solving complexity of the extracted ILP system due to the complexity of modeling these micro-architectural components [126]. For example, in cache and branch prediction analysis, the estimated longest path might not be feasible, and the estimated WCET might be overly pessimistic [21]. Furthermore, the need to express basic blocks in different contexts, such as different options to reach them in a CFG, can make the problem too difficult to be mapped to an ILP system [139]. This was addressed by combining abstract interpretation [25] with IPET to provide tighter WCET bounds for more complex processors [126, 139].

The standard IPET formulation and constraints used for retrieving the WCET of a program are discussed next:

3.2.3.1 ILP formulation

Let x_i be the number of times the basic block B_i is executed. Let c_i be the timing cost assigned to block B_i using a timing model for the architecture under consideration. The energy cost of each CFG basic block, B_i, is obtained by aggregating the timing costs of all the ISA instructions included in the block. Combining these, the execution time of a program with N basic blocks is given by the expression:

$$E_{prg} = \sum_{i=1}^{N} c_i \cdot x_i \tag{3.4}$$

Considering that Equation (3.4) represents the execution time of a program, the number of values each x_i can take is constrained by the program structure and functionality and therefore by the data inputs. Solving this equation to get the upper bounds of the execution time requires maximizing it and taking into account the restrictions driven by the program structure and functionality, also called flow facts in more recent work [71]. Stating those restrictions in the form of linear constraints enables the use of ILP to maximize the retrieved equation.

An example of ILP formulation for the `jpegDct` function in Listing 3.1 is given in Equation (3.5). The equation can be automatically formulated using the function's CFG, given in Figure 3.3(a), and Equation (3.4).

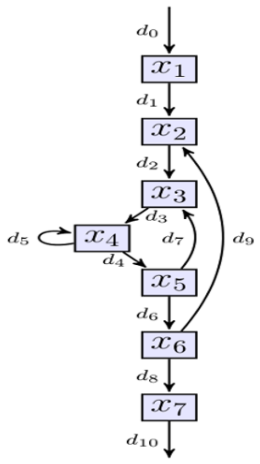
$$c_1 \cdot x_1 + c_2 \cdot x_2 + c_3 \cdot x_3 + c_4 \cdot x_4 + c_5 \cdot x_5 + c_6 \cdot x_6 + c_7 \cdot x_7 \tag{3.5}$$

3.2.3.2 Structural constraints

To extract the structural constraints of a program, a CFG is created and annotated. The edges are annotated with d_i and the basic blocks with x_i, which represents the number of times the edges and blocks are exercised during the program execution, respectively. For a program that does not deadlock, the number of times a basic block is entered must be equal to the number of times control exits this basic block. Hence, x_i is equal to the sum of d_i edges entering the block and the sum of d_i edges exiting the block. To illustrate this, the annotated CFG of the `jpegDct` function in Listing 3.1, is given in Figure 3.3(a) alongside with the retrieved structural constraints in the form of linear expressions, given in Figure 3.3(b). The edges, d_0 and d_{10}, representing the start and the exit of the CFG, respectively, are set to one in the case of analyzing a single function. Section 7.1.1 explains how the process of retrieving a function's CFG and the CFG's IPET-like annotation can be automated.

```c
void jpegDct(short d[], short r[])
{
    long int t[12];
    int v=0;
    short i, j, k, m, n, p, ic, ik;
    for (ik=2; ik; ik--) {
        for (i = 8; i; i--, v+=8) {
                  for (j = 3; j>=0; j--) {
                 // some code
            }
          // some code
       }
    }
}
```

Listing 3.1: JpegDct function



(a) jpegDct function's CFG

$$
\begin{aligned}
d_0 &= 1\\
x_1 &= d_1 = d_0\\
x_2 &= d_1 + d_9 = d_2\\
x_3 &= d_2 + d_7 = d_3\\
x_4 &= d_3 + d_5 = d_4 + d_5\\
x_5 &= d_4 = d_6 + d_7\\
x_6 &= d_6 = d_8 + d_9\\
x_7 &= d_8 = d_{10}\\
d_{10} &= 1
\end{aligned}
$$

(b) jpegDct structural constraints

$$
\begin{aligned}
1x1 &<= x2\\
x2 &<= 2x1\\
1x2 &<= x3\\
x3 &<= 8x2\\
1x3 &<= x4\\
x4 &<= 4x3
\end{aligned}
$$

(c) jpegDct functional constraints for loop bounds

Figure 3.3: jpegDct benchmark's IPET-annotated CFG and its structural and functional constraints formulated as linear expressions.

In the case of function calls, f-edges are used to connect the caller function to the start of the CFG of the callee function. The f-edge is then treated in the same way as a d-variable to construct the structural constraints in the caller function.

3.2.3.3 Functionality constraints

Functionality constraints are used to capture information that can affect the program's dynamic behavior, e.g. bounds on the number of iterations for loops, or path information (such as infeasible paths). Usually, this additional information can only be specified by the programmer, as it depends on the program's semantics and cannot be extracted by the static analysis in most cases. The minimum requirement for user input to enable the bounding of the problem is the specification of loop bounds. This is standard practice also in timing analysis [140]. The loop bounds constraints for the `jpegDct` function in Listing 3.1, are given by the user in the form of linear expressions, and shown in Figure 3.3(c). Further constraints, such as denoting an infeasible path in a CFG, can be provided to extract tighter bounds.

Using off-the-shelf Linear Programming (LP) solvers, Equation (3.5) can be maximized taking into account the constraints in Figure 3.3, in order to retrieve a WCET for the `jpegDct` function given in Listing 3.1.

3.2.4 Existing work on SRA-based energy consumption

Although significant research has been conducted in static analysis for the execution time estimation of a program, there is little on energy consumption. One of the few approaches [91] seeks to infer the energy consumption of Java programs as functions of input data sizes statically, by specializing a generic resource analyzer [47, 93] to Java bytecode analysis [92]. However, a comparison of the results to actual measurements was not performed. Later, in [70], the same generic resource analyzer was instantiated to perform energy analysis of XC programs [135] at the ISA level based on ISA-level energy models and including a comparison to actual hardware measurements. However, the scope of this particular analysis approach was limited to a small set of simple benchmarks because the information required for the analysis of more complex programs, such as program structure and types, is not available at the ISA level. A similar approach, using cost functions, was used in [44]. The analysis was performed at the LLVM IR level, using an preliminary version of the mapping technique that is being formalized and described in full detail in the Chapter 6 of this thesis. Although the range of programs that could be analyzed was improved compared to [70], the complexity of solving recurrence equations for analyzing larger programs proved a limiting factor.

In [55] the WCEC for a program was inferred by using the IPET first introduced in [65]. For the IPET formulation, instead of a timing cost model, the authors used an energy model that assigns energy values to blocks of ISA code. Although the authors manage to bound the WCEC on a simulated processor by maximizing the switching activity factor for each simulated component, they acknowledge the need to validate their estimation results against commercial embedded processors. Similarly, in [132] the authors attempt to perform static WCEC analysis for a simple embedded processor, the `ARM Cortex M0+`. This analysis is also based on IPET combined with a so-called absolute energy model; an energy model that is said to provide the "maximum energy consumption of each instruction". The authors argue that they can retrieve a safe bound. However, this is demonstrated on a single benchmark, `bubblesort`, only. They also acknowledge that using an absolute energy model can lead to significant overestimation making the bounds less useful.

All of the reviewed previous works combined SRA techniques together with energy models, to capture the WCEC path. Currently, there is no practical method to perform average-case static analysis [128]. One of the most recent works towards average-case SRA [111] demonstrates that compositionality combined with the capacity for tracking data distributions unlocks the average-case analysis, but novel language features and hardware designs are required to support these properties.

3.3 Thesis considerations

Although, there is a variety of techniques trying to retrieve the energy estimation of software, there are still some important gaps that make it difficult to incorporate energy transparency in the daily development life cycle of deeply embedded applications such as IoT applications. The main disadvantages identified from examining existing energy consumption estimation approaches are:

- end-to-end energy measurements can not provide safe energy usage bounds;
- it is too difficult to achieve fine-grained energy characterization of software with end-to-end energy measurements;
- all the techniques working at the architectural level are based on expensive hardware simulations, and therefore it is impractical to use them during an iterative software optimization process;
- energy modeling at lower levels than ISA is not always feasible because low-level information for most commercial architectures is not publicly available;
- although ISA energy estimations are the most accurate ones among the software levels in the system's stack, analysis and optimization at this level is constrained due to the loss of program's information, such as data structures;
- there is a lack of techniques that can provide energy consumption estimations directly at the compilers' IR level;
- there is too little done in the area of bounding energy consumption by the use of SRA techniques;
- to the best of my knowledge, there are no existing works targeting energy consumption SRA of multi-threaded programs;
- the majority of the current techniques are not easily transferable to new architectures.

In this thesis, to provide a holistic energy transparency framework for deeply embedded systems, the best practices of existing software energy estimation techniques are identified and adopted, and new techniques are introduced where this is needed. Special focus is given in making the underlying techniques of this framework as target agnostic as possible and able to provide fine-grained software energy characterization.

Since the focus of this thesis is to provide energy information on the software side of the system's stack, ISA level energy models are considered as a starting point to provide the low-level analysis required. ISA can be considered as the bridge that connects software and hardware, and therefore is the first practical level in the software stack that allows energy modeling without sacrificing too much accuracy.

IPET is the most successful technique used for WCET analysis of deeply embedded programs. To the best of my knowledge, only a couple of works applied IPET to perform bound energy consumption analysis; namely [55] and [132]. None of these works adequately investigated the applicability of IPET for retrieving bound energy consumption estimations for deeply embedded systems. In the first case, [55], IPET was applied on a simulated processor, where an activity factor of 1 can be set to retrieve the worst-case switching activity of the simulated processor. Such approach is not an option on an actual processor because the processors activity factor can not be configured. In the second case, [132], the authors used an ISA worst-case energy model, but their evaluation was limited to one benchmark. Worst-case energy models can be impractical to construct because the whole input space for each ISA instruction must be exercised and the energy must be measured for each set of different inputs to determine the instruction's worst case. Furthermore, such worst-case models can lead to significant overestimation, making the bounds less useful. In this thesis, IPET is also adopted at the ISA level, but a pseudo-random data characterized energy model is used, as empirical evidence shows that such models tend to be close to the actual worst case [100].

Energy transparency at the compiler level is essential to expose opportunities for energy-specific compiler optimizations. This thesis investigates techniques to enable accurate energy estimations at the intermediate representation level of the LLVM compiler. Beyond bound estimations, the actual case is also considered, as it is more appropriate to be used as an optimization basis rather than the use of the worst case. Because SRA can not capture the actual case, profiling techniques are also used in this thesis as a basis to achieve actual-case energy consumption estimations.

Furthermore, no existing work is considering SRA for bounding the energy consumption of multi-threaded programs. Deeply embedded multi-threaded processors, such as the XCore architecture, are ideal for a range of parallelizable applications, such as DSP. Therefore, such devices exhibit a greater number of design space exploration opportunities, since time, energy and hardware resources can be interchanged depending on the optimization targets. IPET-based energy analysis is investigated in this thesis for a selected set of multi-threaded deeply embedded programs from two commonly used concurrency patterns, task-farms and pipelines.

3.4 Conclusion

The review of existing energy consumption estimation techniques for software helped to set the objectives for the energy transparency techniques introduced in this thesis in two ways. Firstly, the advantages of the existing software energy estimation approaches were identified to be taken forward in this thesis. Secondly, the weaknesses of the existing methods, which make them hard to be adopted in the daily deeply embedded software development life cycle, were identified to be addressed by this thesis. Based on this analysis the requirements for the energy transparency techniques in this thesis together with their added benefits are listed in Table 3.2.

No	Requirement	Reasoning/ Added benefits
1	Avoid physical energy measurements.	• Enables energy-aware software development for less experienced software developers; • Avoids the costly and difficult setup needed for measurements.
2	Avoid the need for simulation.	• Improves the energy consumption estimation speed, making it more practical to use the energy consumption transparency techniques during an iterative software optimization process.
3	Provide energy transparency at different levels of the software stack.	• Enables understanding of how each level of software abstraction affects the energy consumption on a particular hardware; • Allows for potential energy-specific optimization and design-space exploration at each software abstraction level.
4	Provide fine-grained energy characterization of software.	• Allows to identify energy hot spots in software; • Allows for comparing the energy efficiency of different coding styles and algorithms.
5	Capture both bounds and actual case energy consumption.	• Energy consumption bounds will help developers to meet the strict energy budget requirements of deeply embedded applications; • The actual energy consumption can act as a benchmark for potential energy optimizations.
6	Target-agnostic techniques.	• Minimizes the required effort of transferring the techniques to new tool-chains and architectures; • A key element for the adoption of energy transparency techniques in the software development life-cycle of deeply embedded applications.
7	Enable design space exploration decisions.	• Enables multi-objective optimization or finding the optimum balance between different resources, such as execution time, energy and code size, according to the application's requirements.
8	Consider the multi-threaded case.	• Helps to explore the potentials and benefits of modern multi-treaded deeply embedded architectures in the scope of energy-critical applications.

Table 3.2: Requirements set for the energy transparency techniques in this thesis.

CHAPTER 4

THE XMOS XCORE PROCESSOR

This chapter first gives an overview of the XMOS XS1-L (Xcore) multi-treaded architecture used in this thesis. Then a detailed description of the energy model created in [59] for the same architecture is presented. This model forms the basis for the low-level analysis introduced in Chapter 5 to statically capture the dynamic behavior of the microprocessor, in regards to its energy consumption characteristics.

4.1 XMOS Xcore architecture and energy model overview

The Xcore XS1-L architecture manual [86] gives a detailed description of the architecture's features. In this section, an overview of the architecture's primary characteristics is provided, with a focus on the predictability of the microprocessor. Predictability is a key element for many IoT and mission critical deeply embedded applications. Furthermore, the level of predictability in an architecture defines how accurately static modeling and analysis can capture its dynamic behavior.

The Xcore is a 32-bit Reduced Instruction Set Computer (RISC), deeply embedded microprocessor, which is intended to allow hardware interfaces to be written in software. This makes the Xcore well suited to embedded applications that require multiple hardware interfaces with real-time responsiveness. Interfaces, such as SPI, I2C and USB, can be written efficiently in C-style software, rather than relying on hardware blocks provided in a System-on-Chip or synthesized onto Field-Programmable Gate Array (FPGA). To achieve this, the Xcore is designed to be a time deterministic hardware multi-threaded architecture, which provides inter-thread communication and Input/Output (I/O) port control directly in the ISA. In favor of predictability and low energy consumption over maximizing performance, the Xcore has no cache hierarchy, has a single-cycle access memory, and is event-driven; busy waiting is avoided by the use of hardware scheduled idle periods. It is integer only, with no floating point hardware. The timing-predictable behavior and efficiency features make this a good selection as a case study for the analysis techniques introduced in this thesis.

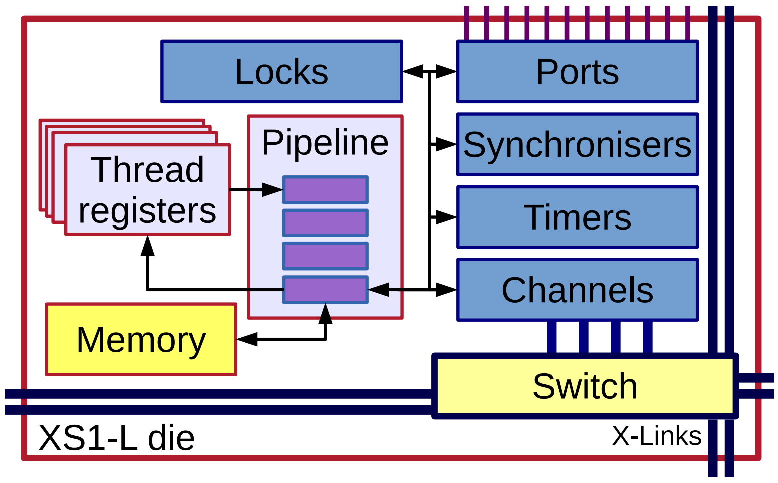


Figure 4.1: Overview of the Xcore XS1-L microprocessor (Reproduced from [59],with permission).

Figure 4.1 shows an overview of the main elements of the Xcore architecture. The processor supports up to 8 threads, with machine instructions and hardware resources dedicated to thread creation, synchronization, and destruction. Each thread has its own instruction buffer and bank of registers, to allow for fast thread creation and concurrent thread execution. The processor's hardware resources include:

- a scheduler for dynamic thread scheduling;
- a set of synchronizers for thread synchronization;
- a set of ports to perform I/O operations;
- a set of timers for controlling the real time execution;
- a set of channels for communication among threads.

The XS1-L ISA has in total 203 instructions. Beyond the typical instructions found in an RISC-like architecture, such as arithmetic, logic, memory and branching instructions, a new set of instructions is provided to give direct access to the processor hardware resources. These instructions support thread handling, thread communication, event handling, accessing timers and I/O operations. Furthermore, some DSP instructions are supported.

The processor features a short four-stage pipeline, designed to provide time-deterministic execution and to maximize responsiveness. By design, the architecture avoids the need for forwarding between pipeline stages, speculative instruction issue, and branch prediction and allows for zero-time overhead in thread context switching.

4.1.1 Thread scheduling

Figure 4.2, demonstrates how instructions are scheduled by the hardware thread scheduler, and executed through the Xcore pipeline for different counts of active threads. Threads are executed round-robin through a four-stage pipeline. The thread scheduler maintains a pool of runnable threads in order to pick the next instruction to be executed. Each thread can have only one instruction occupying the pipeline at any time to avoid data hazards. Therefore, if fewer than four threads are active, there will be clock cycles with inactive pipeline stages. These inactive pipeline stages are represented by the white boxes appeared after the $T_{0,0}$ diagonal in Figures 4.2(a) to 4.2(c). This means that to reach the maximum computation power of the processor four or more threads have to be active to fully occupy the processors pipeline. Figures 4.2(d) and 4.2(e) demonstrate this case. When more than four threads are active, maximum pipeline throughput is maintained, but compute time is divided between the active threads, such as in Figure 4.2(e). Generally, when n threads are active, the scheduler guarantees that each one will at least get $1/n$ processor cycles.

Clk	**Pipeline Stages**			
1	$T_{0,0}$			
2		$T_{0,0}$		
3			$T_{0,0}$	
4				$T_{0,0}$
5	$T_{0,1}$			
6		$T_{0,1}$		
7			$T_{0,1}$	
8				$T_{0,1}$

(a) 1 active thread

Clk	**Pipeline Stages**			
1	$T_{0,0}$			
2		$T_{0,0}$		
3	$T_{1,0}$		$T_{0,0}$	
4		$T_{1,0}$		$T_{0,0}$
5	$T_{0,1}$		$T_{1,0}$	
6		$T_{0,1}$		$T_{1,0}$
7	$T_{1,1}$		$T_{0,1}$	
8		$T_{1,1}$		$T_{0,1}$

(b) 2 active threads

Clk	**Pipeline Stages**			
1	$T_{0,0}$			
2	$T_{1,0}$	$T_{0,0}$		
3	$T_{2,0}$	$T_{1,0}$	$T_{0,0}$	
4		$T_{2,0}$	$T_{1,0}$	$T_{0,0}$
5	$T_{0,1}$		$T_{2,0}$	$T_{1,0}$
6	$T_{1,1}$	$T_{0,1}$		$T_{2,0}$
7	$T_{2,1}$	$T_{1,1}$	$T_{0,1}$	
8		$T_{2,1}$	$T_{1,1}$	$T_{0,1}$

(c) 3 active threads

Clk	**Pipeline Stages**			
1	$T_{0,0}$			
2	$T_{1,0}$	$T_{0,0}$		
3	$T_{2,0}$	$T_{1,0}$	$T_{0,0}$	
4	$T_{3,0}$	$T_{2,0}$	$T_{1,0}$	$T_{0,0}$
5	$T_{0,1}$	$T_{3,0}$	$T_{2,0}$	$T_{1,0}$
6	$T_{1,1}$	$T_{0,1}$	$T_{3,0}$	$T_{2,0}$
7	$T_{2,1}$	$T_{1,1}$	$T_{0,1}$	$T_{3,0}$
8	$T_{3,1}$	$T_{2,1}$	$T_{1,1}$	$T_{0,1}$

(d) 4 active threads

Clk	**Pipeline Stages**			
1	$T_{0,0}$			
2	$T_{1,0}$	$T_{0,0}$		
3	$T_{2,0}$	$T_{1,0}$	$T_{0,0}$	
4	$T_{3,0}$	$T_{2,0}$	$T_{1,0}$	$T_{0,0}$
5	$T_{4,0}$	$T_{3,0}$	$T_{2,0}$	$T_{1,0}$
6	$T_{0,1}$	$T_{4,0}$	$T_{3,0}$	$T_{2,0}$
7	$T_{1,1}$	$T_{0,1}$	$T_{4,0}$	$T_{3,0}$
8	$T_{2,1}$	$T_{1,1}$	$T_{0,1}$	$T_{4,0}$

(e) 5 active threads

Pipeline Stages: Decode Execute 1 Execute 2 Memory Access

Clk: Clock Cycle $T_{i,\,j}$: i = Tread Number, j = Instruction Number

Figure 4.2: The Xcore round robin thread scheduling for different counts of active threads.

There is a number of conditions under which a thread needs to pause its execution, and the scheduler will remove the paused thread from the pool of runnable threads. These conditions are:

- waiting for thread synchronization to continue or terminate (sync or join);
- waiting on a timer;
- waiting on an input channel or port, with no data available;
- waiting on an output channel or a port, with no room for data;
- waiting on a number of events.

If all threads become inactive, an Xcore can be put into deep sleep mode to save energy. In the case of more than four active threads, if any thread pauses its execution then the performance of the remaining active threads is increased. When a paused thread becomes active again, it will be placed back into the scheduler's pool of runnable threads and continue executing from the point where it paused.

4.1.2 Fetch No-Operation (FNOP) instruction

The instruction buffer of each thread is 64-bit and therefore able to hold up to four 16-bit instructions or two 32-bit instructions. Instructions are 16-bit aligned in memory. The majority of the instructions are 16-bit. For a typical program execution, more than 80% of the instructions executed are 16-bit instructions to allow the processor to fetch two instructions each cycle. This is critical for the performance of the processor as instruction fetch happens at the last stage of the pipeline and has to compete with memory accesses. The implications of this pipeline behavior are discussed as follows:

In addition to the instructions defined in the ISA, a further operation, FNOP can be issued internally by the processor. An FNOP is issued when the instruction buffer of a thread does not contain the next instruction to be released into the pipeline for execution. The fetch stage of the pipeline is then used to fetch the missing data, at the cost of the other stages being idle for the affected thread. Thus, the thread effectively performs a no-op during its scheduled slot. The conditions that produce FNOPs are documented in [86, pp. 8–10]. These are deterministic and can be statically identified. In summary, an FNOP is needed when:

- Repeated sequences of memory operations prevent fetches from occurring normally at the memory access stage of the execution pipeline, requiring FNOP operations to re-fill the instruction buffer;
- If a branch is taken the instruction buffer of the relevant thread is flushed, and the branch target instructions are fetched during the memory access stage. In the case the fetched instructions also require a data access they will leave the instruction buffer empty and an FNOP will be required;
- In the case a branch instruction is executed and flushes the instruction buffer, it may require an FNOP if the branch target is unaligned and the target instruction is long (32 bits).

FNOPs can be usually eliminated by the compiler instruction scheduling; e.g. avoiding long sequences of load and store instructions.

4.1.3 Scalability

The Xcore is many-core scalable, with 2- or 5-wire "X-Links" and a network switch embedded into each core. Communication between threads on the same or different cores is done via synchronous channel-based message passing. Threads on the same core can communicate without contention, whereas the links used for multi-core communication may be contended. Resources such as memory, processing power, network throughput and event-handling scale with cores. To demonstrate the scalability of the architecture, in [50] the authors created a 480-core system using the XMOS Xcore XS1-L architecture as the basic unit for the computation and networking infrastructure.

4.1.4 Programming language

XC is a high-level imperative programming language, endorsing the syntax of C programming language and introducing extensions to support concurrency, communication, I/O operations, and real-time behavior. The primary objective of XC is to allow programmers to have direct control on the hardware's capabilities of the Xcore's architecture. This allows for the creation of concurrent and multi-core real-time programs. The real-time performance is guaranteed both by the architecture and the tools.

The XC language is directly compiled to bare metal. The desirable target system configuration is described in a separate file and passed to the compiler together with the source code. Then the compiler ensures that a sufficient number of hardware resources are available to execute the program being compiled. A detailed description of the XC language version used in this work is provided in [135].

4.1.5 XMOS Xcore timing behavior

Most instructions in the Xcore processor are time-deterministic, i.e. the number of cycles taken is a single figure, not a range. These instructions complete in four clock cycles — the length of the processor pipeline. Exceptions include the signed and unsigned division and remainder operations, which perform bit-serial operations and may take up to 32 clock cycles, but have early-completion capabilities. Furthermore, instructions that interact with resources, such as ports and channel communication, may block, which can result in a thread being de-scheduled until the blocking condition is satisfied. This thesis focuses on fully predictable instructions, with timing disturbances from communication forming future work.

In order to have timing analysis that is valid at the processor level, these predictable instructions must be considered with respect to the thread scheduling and pipeline of the processor. As explained in the previous section, the Xcore can run between one and eight threads, time-slicing access to its 4-stage pipeline. A *thread cycle* is the length of time between subsequent instructions in an active thread. A thread can only have a single instruction in the pipeline at a time, resulting in under-utilization of the pipeline if fewer than four threads are active. Thus, the minimum thread cycle time is $4 \times \frac{1}{F}$, where F is the processor clock frequency. Similarly, if five or more threads are active, there is pipeline contention, which is resolved with round-robin scheduling. This increases the thread cycle time, relative to the number of active threads. Equation (4.1) defines the total Instructions Per Second of the processor, IPS_p, for N_t active threads, and Equation (4.2) defines the execution rate of an active thread, IPS_t.

(4.1) $$\text{IPS}_p = F\frac{\min(4, N_t)}{4}$$

(4.2) $$\text{IPS}_t = \frac{F}{\max(4, N_t)}$$

These rules are applied in the energy model described in the next section, as first proposed in [59]. They are also utilized in the energy consumption estimation techniques performed in this thesis.

4.2 The Xcore multi-threaded ISA energy model

The underlying Xcore energy model for this work was first introduced in [59]. The modeling technique is built upon [127], which is adapted and extended to consider the scheduling behavior and pipeline characteristics of the Xcore. The model is captured at the ISA level. Individual instructions from the ISA are assigned a single cost each. These can then be used to compute power or energy for sequences of instructions, which are obtained by a simulator.

The cost associated with an instruction represents the average energy consumption obtained from measuring the energy consumed during instruction execution. This is based on constrained pseudo-randomly generated operands using the setup described in [59]. For any instructions that their anergy costs were unable to be captured by direct measurements, such as the FNOP instruction, linear regression, and regression-based decision trees were used. The regression analysis used the already known values of instructions, characterized via physical energy measurements on the actual platform, to find an energy profile that matches an uncharacterized instruction. This was done based on a number of statically determined parameters. Furthermore, to account for the cost of hardware thread scheduling, a series of profiling tests and measurements were performed. Accounting for the hardware thread scheduling is critical for the accuracy of the energy model because it influences the energy consumption of programs' execution significantly.

The initial model is targeted in estimating the energy consumption of a program based on the execution statistics retrieved from a cycle-accurate ISS. The accuracy of the model was evaluated on 11 benchmarks, yielding an average error margin of 7%. The initial model formulations is described by Equation (5.1), as given in [59].

(4.3) $$E_{prg} = P_{base} \cdot N_{\text{idle}} \cdot T_{clk} + \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} \left((M_t \cdot P_i \cdot O + P_{base}) \cdot N_{i,t} \cdot T_{\text{clk}} \right)$$

In Equation (5.1), E_{prg} is the energy of a program, formed by adding the energy consumed at idle to the energy consumed by the pipeline activity. The energy consumed at idle is estimated by multiplying the number of cycles with no active threads, N_{idle}, by the clock period, T_{clk}, and base power, P_{base}. P_{base} is the power dissipated for idle processor's periods.

The pipeline activity is taken into consideration by the remaining part of the equation: for each possible number of concurrent threads up to the maximum, N_t, a multiplier, M_t, is applied for that level of concurrency. Then, for all the instructions in the ISA, each instruction i at concurrency level t is assigned an instruction power, P_i, that is scaled by a constant overhead, O, which accounts for inter-instruction overheads. Finally, the base power, P_{Base}, is added, and the overall result is multiplied by the clock period, T_{clk}, and the number of times this instruction is executed, $N_{i,t}$, at concurrency level t.

4.3 Conclusion

This chapter gave a broad overview of the Xcore XS1-L architecture, focusing on the architecture's features that can affect its energy consumption. Xcore is a multi-threaded, time predictable architecture that can be extended to many-core systems. This makes the architecture an interesting research case to explore techniques that can predict the energy consumption of software and can enable design space exploration. This is why the Xcore processor is selected as the focus of this thesis among other more popular deeply embedded processors used for IoT applications, such as the `ARM Cortex M` series [9].

The energy model presented in this chapter will form the basis of the low-level analysis, introduced in Chapter 5. The low-level analysis provides the basic energy costs needed, including instruction and CFG basic block costs, by the energy consumption estimation techniques developed in this thesis.

Part II

ENERGY TRANSPARENCY TECHNIQUES

Chapter 5

Low-Level Analysis

"If you cannot measure it, you cannot improve it."
— Lord Kelvin, *[58]*

This chapter covers the low-level analysis performed in this thesis. Low-level analysis captures the dynamic behavior of the processor in regards to its energy consumption characteristics. In other words, it tries to associate energy costs for each atomic unit on a given hardware platform, such as an instruction or a CFG basic block for a processor. These costs are the means of conveying energy information to the energy consumption estimation methods developed in this thesis.

5.1 Experimental setup

Energy measurements and simulation-based energy estimations are essential both for establishing the low-level analysis but also for validating this thesis energy estimation techniques.

5.1.1 Energy-measurement setup

Conventional methods are used to measure the power dissipation of the target device over a period of time, similar to those described in Section 3.1. The voltage drop over a shunt resistor is measured in order to determine the current provided to the device. Figure 5.1 illustrates the experimental setup of retrieving energy measurements for the platform under consideration. A control board collects power readings and averages them before storing the results on a PC. The control board's measurements can be triggered, so that measurements are only taken when a test-run is active. The actual parts of the measurement setup illustrated in Figure 5.1 are:

- **Monitored Board:** This is the Device Under Test (DUT). For this thesis the XMOS XP-SKC-A16 [145] development board is used. The board's processor consists of two interconnected Xcores, with 16 threads available in total and supports voltage and frequency scaling. The board was modified to allow access to the cores' power supply.
- **Power Measurement Board:** An `INA219` high side Direct Current (DC) shunt and power monitor chip [123], is used for sampling the the power dissipation of the two cores. The sample rate can be up to 1.8 K samples per second with a resolution of 11-bit and an accuracy of 100µA per least significant bit when using a shunt resistor of 0.1Ω.
- **Control Board:** This board collects the power samples from the power measurement board and calculates the average power dissipation of the cores during a program's execution. The board also triggers the DUT, to ensure the synchronization between the execution of a program and the power sampling. An *Intel Edison* [52] development board is used as the control board in this thesis.
- **Data Collection:** Data collection is performed on a host machine. This machine is also used for flashing the programs onto the DUT.

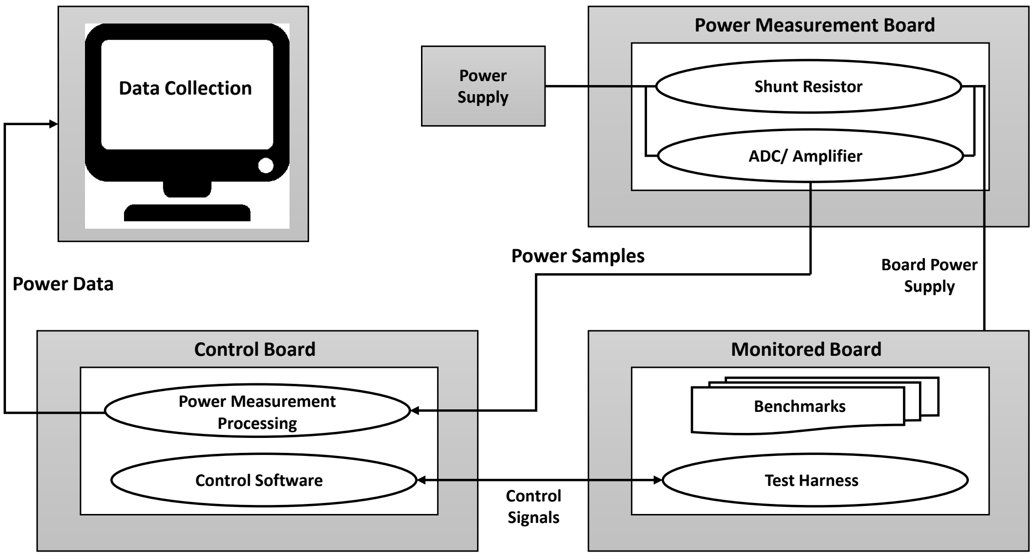


Figure 5.1: Illustration of the measurement setup, hardware and software harness, for acquiring energy measurements from the platform under consideration.

5.1.2 Simulation-based energy estimation

The evaluation for the initial version of the energy model, introduced in [59] and discussed in Section 4.2, was performed by retrieving energy estimations based on execution statistics produced from the cycle-accurate *axe* [106] ISS. Similar simulation-based estimations are used in this thesis for a) validating the newer version of the energy model, as discussed in the next sections, and b) as a baseline for the best achievable accuracy by the introduced estimation methodologies because ISS produces more accurate execution information than any analysis techniques.

5.2 Xcore energy model adjustment

The low-level analysis uses as a starting point the multi-threaded energy model developed in [59] for the Xcore architecture, and described in detail in Section 4.2. This initial version of the model was put under significant stress by a series of evaluation and adjustment rounds to ensure the maximum accuracy possible. This is important as any error in the energy model will be inherited in the results of any estimation technique which uses this model.

Carefully drafted synthetic benchmarks were used to isolate the energy cost of specific instructions. This was done for the majority of instructions for which their energy cost could not be established by executing the instruction in a loop and measuring the energy consumption. Using these synthetic benchmarks in both hardware measurements and ISS-based estimation, the energy cost of an instruction could be validated. The validation information was passed in an iterative process to the author of [59] and creator of the model, who kindly reformulated his initial model to achieve a higher level of accuracy. The newer version represents energy in terms of static and dynamic power components to better reflect inter-instruction and inter-thread overheads. This has improved model accuracy by an average of four percentage points.

The resulted model is well suited for the energy estimation techniques developed in this thesis, but also it can still be utilized by a cycle-accurate ISS to perform energy estimations at the ISA level. The new model is the one used in this work and explained as follows:

$$E_{\text{prg}} = (P_s + P_{di}) \cdot T_{\text{idl}} + \sum_{i \in \text{prg}} \left(\frac{P_s + P_i \cdot M_{N_p} \cdot O}{N_p} \cdot 4 \cdot T_{\text{clk}} \right), \text{ where } N_p = \min(N_t, 4) \tag{5.1}$$

In Equation (5.1), E_{prg} is the energy of a compiled version of a program, prg, formed by adding the energy consumed at idle to the energy consumed by every instruction, i, executed in the program. At idle, only a base processor power, the sum of its static, P_s, and dynamic idle power, P_{di}, is dissipated for the total idle time, T_{idl}. For each instruction, static power is again considered, with additional dynamic power for each particular instruction, P_i. The dynamic power contribution is then multiplied by a constant inter-instruction overhead, O, that has been established as the average overhead of instruction interleaving. This is then multiplied by a scaling factor to account for the number of threads in the pipeline, M_{N_p}. The result is divided by the number of instructions in the pipeline, which is at most four and is dependent upon the number of active threads, N_t. Each instruction completes in four cycles, so $4 \cdot T_{clk}$ gives the energy contribution of the given instruction, based on the calculated power.

When more than four threads are active, the issue rate of instructions per thread will be reduced. The energy model accounts for this with the min term in Equation (5.1). From a purely timing perspective, the latency between instruction issues for a thread is $\max(N_t, 4) \cdot T_{clk}$. This property means that instructions are time-deterministic, provided the number of active threads is known.

5.2.1 Making the energy model parametric to voltage and frequency

One of the main objectives of this thesis is to explore if the energy estimation techniques introduced are capable of supporting design space exploration decisions at development time. Execution time, energy and hardware resources can be interchanged to achieve the desired optimal point according to the application specifications. Scaling the frequency and the voltage of the core is a desirable feature; it allows for a broad range of configuration options in the design phase, with multiple optimization targets. If the energy model used by the estimation techniques can be parametric to voltage and frequency, then there is a potential for the estimation techniques to provide feedback on the effects of scaling these two parameters, for a particular program.

The dynamic power consumption of each one of the processor's instructions can be characterized by its effective capacitance expressed in farad. This is presented in Equation (5.2) for a specific operating mode j, in terms of the actual clock frequency and core's supply voltage. The power consumption $P_{i,j}$ of an instruction i is the product of the instruction switching activity factor, α multiplied by the instruction's effective capacitance, c_i, the square of the core's supply voltage, V_j^2, and the core's operating frequency, F_j. The switching activity factor represents the fraction of total capacitance that switches each cycle, and it is typically set to 0.5, which means random switching activity [19]. The switching activity factor for this thesis' energy model is not set to a fixed number. Instead, it is taken into account by the energy costings extracted by the energy characterization phase for each ISA instruction.

$$P_{i,j} = \alpha \cdot c_i \cdot V_j^2 \cdot F_j \tag{5.2}$$

The static power consumed by the processor can be also described in terms of the current leakage, I_{leak}, and the core's supply voltage, V_j, as shown in Equation (5.3).

$$P_{s,j} = I_{leak} \cdot V_j \tag{5.3}$$

It is very uncommon for chip vendors to release any information on the effective capacitance of the processor's ISA instructions. To parametrize the power consumption of the Xcore ISA instructions in terms of their effective capacitance, the energy model introduced in the previous section was again reformulated by the author [59] and creator of the energy model. Using the ISA energy costs provided by the energy model and the core supply voltage and operating frequency used to establish the energy model, Equation (5.1) was solved to extract the product of the activity factor, α, and the effective capacitance, c_i, for each ISA instruction. Similar process was used to identify the I_{leak} value and therefore parametrize the P_s from Equation (5.1), according to Equation (5.3). This process resulted in a new formulation of the energy model, which is parametric to the core supply voltage and operating frequency. The new energy model is given in Equation (5.4).

(5.4)
$$E_{\text{prg}} = \left(I_{leak} \cdot V + C_s \cdot V^2 \cdot F\right) \cdot T_{\text{idl}} + \sum_{i \in \text{prg}} \left(\frac{I_{leak} \cdot V + C_i \cdot M_{N_p} \cdot O \cdot V^2 \cdot F}{N_p} \cdot 4 \cdot T_{\text{clk}} \right), \text{ where } N_p = \min(N_t, 4)$$

The new model decouples the estimation of the effective capacitance and the number of clock cycles for the execution of individual functions, from the overall voltage and frequency dependent time and energy figures. In this thesis, the new model was again passed through a series of validation and adjustment rounds to ensure the maximum accuracy possible.

The original energy model was created using a bespoke XMOS board containing an XS1-L core. Transferring the energy model to a different development board requires accounting for the base processor power, P_s and P_{di} in Equation (5.1), of the new board. The base processor power captures the power dissipation when there are no active threads. For the XS1-L-based development board used in this thesis, the XP-SKC-A16 [145], the base cost was established in the same way as it was done for the original model in [59]. The main idea is the use of a test harness where the processor is set to wait on a hardware timer. This keeps the processor active while there are no active threads or instructions being executed.

5.3 FNOP modeling

The FNOP instruction has no effect on the actual result of the program but does affect time and energy consumption. Thus, taking into account the FNOPs is necessary for accurate execution time or energy consumption estimations.

With ISS-based energy estimation, the FNOPs occurred in a program's execution are known, and their energy effect is taken into consideration. In contrast, the estimation techniques introduced in this thesis, do not have this dynamic information. Both static analysis and profiling techniques introduced in this thesis, work on the programs' CFGs and not on a list of profiled instructions. As mentioned in Section 4.1.2, FNOPs occur deterministically and can be statically modeled.

To account for the FNOPs of a program in the introduced analysis techniques, the program's CFG at ISA level is analyzed. The instruction buffer logic that determines where in a basic block FNOPs will occur, has been modeled in the scope of this thesis. In the case of static analysis, a potential inaccuracy in this approach is when an FNOP insertion depends on the dynamic behavior of the program. When a basic block is reached by a branch instruction, the instruction buffer is flushed before loading the target address. If the target is unaligned, then an FNOP may be required for long instructions, or one may be required in the subsequent cycle if the target instruction includes a memory operation. For blocks that have more than one predecessor, and one of the predecessors passes execution to the block without branching, then there is a dynamic decision as to whether an FNOP is required upon entry to the block. This is analogous to assigning FNOPs to edges of the CFG, rather than within the basic blocks.

The effect of an FNOP arising from such branches is significant if the basic block is a loop entry point. As static analysis techniques are targeting safe upper bounds, an assumption has been made that there is always an FNOP at the entry of a basic block which has multiple predecessors and its first instruction is unaligned and either long or a memory operation. The experimental evaluation, presented in Chapter 8, demonstrates that this does not have a significant effect on the tightness of the retrieved bounds. This is because the vast majority of static FNOP predictions will be correct during the repeated iteration of a loop, typically with a single mis-prediction upon entry to the loop.

In the case of the profiling, the sequence of basic blocks executed is recorded. Therefore, the above case of dynamic FNOP occurrence can be correctly predicted, by observing the program's Basic Block (BB) execution traces retrieved by the profiler.

5.4 Utilizing the Xcore energy model for energy consumption estimations

To determine the energy consumption of a program based on Equation (5.1) the program's instruction sequence, $\langle i_1, \ldots, i_n \rangle$, the idle time T_{idl}, and the number of active threads N_p during instruction execution must be known. In [59] an ISS was used to gather full trace data or execution statistics to obtain these parameters. In this work the ISS is only used as a reference for comparison of SRA and profiling results, with a second reference being direct hardware measurement.

For the two estimation techniques introduced in this thesis, in Chapter 7, SRA and profiling, it is needed to extract the CFGs for each thread and identify the inter-leavings between them. This allows for each instruction in the program to identify the N_p component in Equation (5.1). It also allows to estimate the total idle time, T_{idl} of the program. For single-threaded programs the energy characterization of the CFG is straightforward, as there is no thread interleaving. In the case of SRA, the IPET can be directly applied to the energy characterized CFG to extract a path that bounds the energy consumption of the program, as described in Section 7.1. For arbitrary multi-threaded programs, using static analysis to energy characterize the CFG of each thread is challenging. Therefore, in this thesis, the focus is on two commonly used concurrency patterns, task farms and pipelines, which have evenly distributed workloads across threads. For these classes of programs the number of active threads across the whole execution is constant and equal to the number of threads used to implement the task farm or the pipeline. Similarly, the profiling technique does not need to account for thread interleavings for the single- and multi-threaded programs investigated in this work.

Finally, the model does not require any hardware counters to be simulated, such as cache hit rates, thus making static analysis possible.

5.5 Conclusion

The low-level analysis covered in this chapter provides the energy information needed for the energy estimation techniques, introduced in the next chapters. The basis of the low-level analysis is the energy model introduced in [59]. This model is further refined in this chapter to improve its accuracy and to become parametric to voltage and frequency. The model provides an energy cost for each ISA instruction under a particular operating mode, defined by the clock frequency and core supply voltage. The accuracy of the new version of the model will be evaluated in Chapter 8.

Furthermore, the FNOP instruction issuing mechanism of the processor is statically modeled to capture the instruction's contribution to the energy consumption when statically estimating the energy consumption of a program. This is important as empirical evidence demonstrated that FNOPs can attribute up to 10% of the energy consumption of a program (observed in measurements conducted in the scope of this thesis for benchmarks with FNOPs introduced in loops).

The cost associated with an instruction represents the average energy consumption obtained from measuring the energy consumed during instruction execution and it is based on constrained pseudo-randomly generated operands. Thus, this model does not explicitly consider the range of input data values and how this may affect energy consumption. Empirical evidence indicates that such factors can contribute to the dynamic energy consumption [100]. The implications of using a random data constructed single value energy model for the low-level analysis will be investigated in Chapter 8.

Chapter 6

High-Level Analysis

The low-level analysis presented in the previous chapter enables energy estimation at the ISA level. In this chapter a novel mapping technique is introduced, that can lift the low-level analysis at a higher level, the intermediate representation of the compiler, namely LLVM IR [40], implemented within the LLVM toolchain [49]. This enables energy consumption analysis techniques to be performed at the LLVM IR level, thus introducing energy transparency directly into the heart of the LLVM compiler optimizer. As seen in Section 3.2.1.3, very little is done for exposing the energy consumption to a compiler's IR. Therefore, the technique is a significant step beyond the state of the art techniques that can enable energy transparency at the compiler's IR level.

6.1 Energy estimation at different software levels of abstraction

There are three major software levels of abstraction for deeply embedded programs that a developer usually has to interact with; source code, the compiler's IR and ISA. Figure 6.1 demonstrates the trade-off between the energy consumption estimation accuracy and the level of software abstraction. To perform energy estimation at a software level of abstraction without involving any physical energy measurements, an energy model has to exist at the same level. This will statically provide energy information to the estimation technique. An energy estimation technique automatically inherits any precision loss factored into the energy model that the technique utilizes.

On the one hand, modeling at a lower software abstraction level is always more accurate as it is closer to the hardware where the actual power dissipation happens. On the other hand,

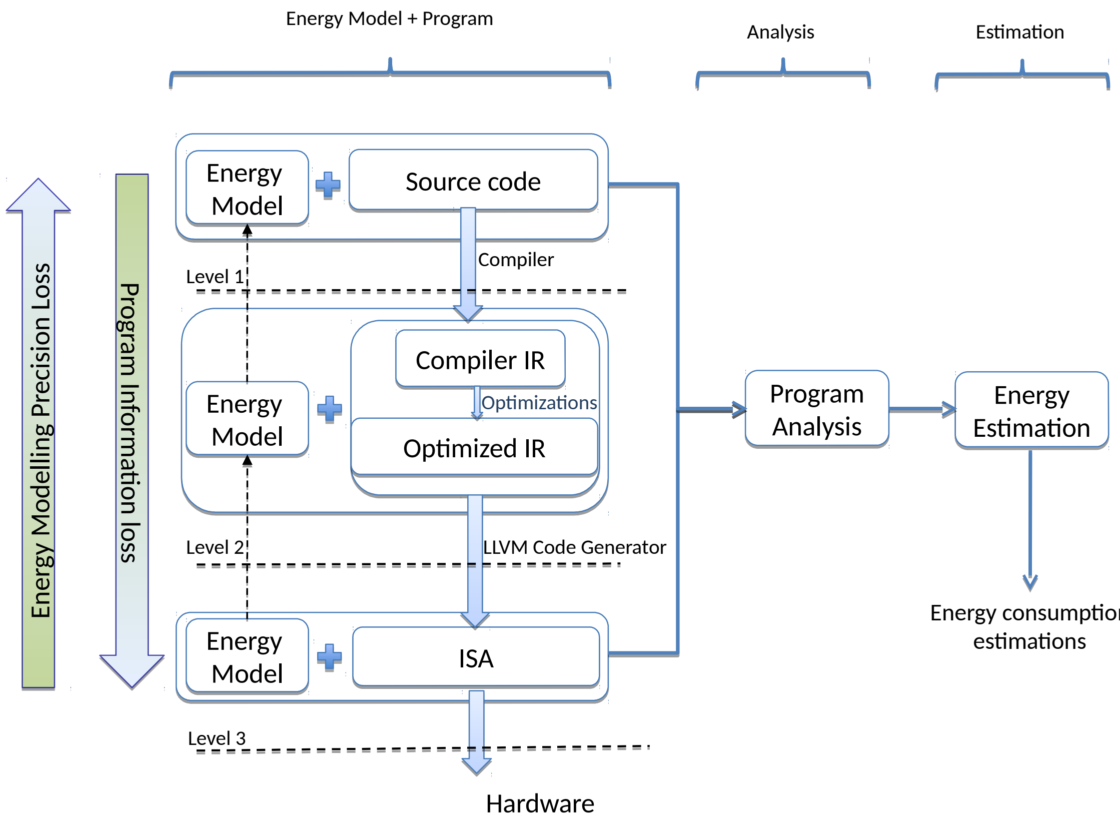


Figure 6.1: Software abstraction level and energy estimation accuracy trade-off.

when moving at higher levels of software abstraction, there is an increasing amount of program information, such as types and loop structures. The availability of program information is significantly reduced when moving from source code to ISA, due to the numerous transformation and optimization passes occurring through the different levels of the software stack. Such information can be crucial for any program analysis that is needed to support the energy estimation. For example, indirect jumps that are introduced at the ISA level can make it impossible to extract a CFG at that level, which is needed by the majority of SRA-based techniques operating at the ISA level. While this prevents ISA-level SRA, at the compiler-IR level, SRA can be performed for these programs. Furthermore, the availability of program information is vital for code optimizations.

Considering the above trade-off, this thesis targets energy estimations techniques at two levels; the ISA level and the compiler-IR level. The reasons for this choice are:

- The accuracy of energy consumption bounds is crucial for energy-critical deeply embedded applications. Therefore, the lowest level in the software stack, ISA, is chosen for SRA as energy modeling precision increases when moving to lower levels of software abstraction.
- For any potential energy optimizations, the availability of program information, such as

types or loop structures, is more important than the energy estimation accuracy. Therefore, performing energy estimation at a compiler's IR level is preferable than at the ISA level. Furthermore, the compiler's optimizer, which works at the IR level, provides the infrastructure to utilize existing optimizations or even introduce new for energy efficiency with lower effort than at any other software abstraction level.

- The optimized version of the compiler's IR is chosen. This is because the optimized IR is closer to the hardware than the unoptimized IR and therefore can yield better accuracy in the energy estimations.

Although energy estimation at the source code level, in theory, is feasible, the expected high loss of precision precludes from performing energy estimation at this level. Instead, in this thesis, the aim is to accurately map the energy estimation information retrieved from lower levels of software abstraction up to the source code.

6.2 The LLVM compiler infrastructure

The LLVM, [72] is an open source compiler and toolchain project. The main aim of the LLVM project is to enable easy adaptation of its technologies by new languages and architectures. This is achieved by keeping its underlying techniques modular and reusable. Among other tools, LLVM provides a framework to build compiler front-ends for new languages and back-ends for new architectures and a modern source and target-independent optimizer. The optimizer works on well-specified code representation, the LLVM IR [20], which is well suited for performing optimizations.

LLVM supports the three-phase compiler design as demonstrated in Figure 6.2. This makes it easier for a single compiler to support multiple source languages or target architectures. The strength of the whole design comes from the LLVM IR common code representation in its optimizer. For each source language only a front end that compiles to the LLVM IR is needed, and then the existing LLVM target-independent optimizer and back-end can be reused.

Due to the retargetability enabled by the three-phase design, all the supported source languages and target architectures can get advantage of the continuous enhancements and improvements to the compiler by the open source community. Also, there is a large number of useful tools, [94], working directly on the LLVM IR.

Although LLVM can be considered a less mature compiler infrastructure than the GNU Compiler Collection (GCC) [124], LLVM modern design and flexibility make it a more attractive choice. Furthermore, the old and difficult to maintain codebase of GCC together with its poor documentation impose a steep learning curve for new developers. In contrast, the modular design and good documentation make LLVM a better choice for new developers. Hence, many tool vendors are switching from GCC to LLVM for the creation of their compiler toolchains.

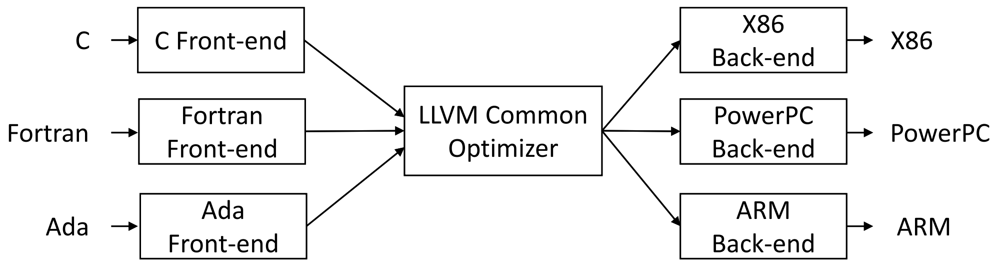


Figure 6.2: Three-phase compiler design using the LLVM core libraries (recreated from [20]).

In this thesis, the LLVM compilation infrastructure is utilized to provide energy transparency techniques that are as target and language agnostic as possible. The LLVM support for the Xcore architecture is also one of the reasons that the Xcore architecture is considered under this thesis. Furthermore, an XC LLVM oriented front-end exists. In the following sections, a novel mapping technique is introduced which provides the means of energy characterizing the LLVM IR of a program and therefore it enables the energy consumption analysis at the LLVM IR level.

6.3 Mapping overview

Directly modeling at a higher level of software abstraction, such as the LLVM IR, has a number of disadvantages:

- The modeling process needs to be repeated for a new architecture;
- Usually a static model at such levels of software abstraction can not account for the compiler's dynamic and architecture-specific behavior (such as FNOPs in the case of the Xcore processor).

Instead of building a static energy model directly at the LLVM IR level, this thesis introduces a novel, dynamic mapping technique that can leverage an existing ISA energy model to convey energy information to higher levels of software abstraction than the ISA. In the case of LLVM IR, the mapping technique aims to link each LLVM IR instruction of a program with its corresponding machine specific ISA emitted instructions. Such a mapping can give powerful insights to the LLVM IR optimizer regarding code size, execution time and the energy consumption of a program. Furthermore, the LLVM IR mapping technique is dynamic, in the sense that it does not involve any statistical analysis, unlike in other approaches [17], but rather it is an on-the-fly technique, that allows taking into consideration the compiler behavior and the program's CFG structure. The mapping is also complete, in the sense that it guarantees no loss of any resource usage between the ISA and the LLVM IR level. The technique is fully portable and target agnostic. It requires only the adjustment of the LLVM mapping pass to the new architecture.

In the next sections, the generic mapping technique is firstly being formalized. Since this work focus on programs energy consumption, the technique is then specialized to determine the energy characteristics of LLVM IR instructions. This specialization propagates ISA level energy models up to LLVM IR level, enabling energy consumption estimation of programs at that level. Finally, the mapping technique is being instantiated and tuned for the architecture under consideration, the Xcore.

6.4 Formal specification of the mapping

The main idea of the mapping technique is to monitor the back-end of a compiler to establish a $1:m$ relation between the optimized LLVM IR and the emitted ISA. The goal of this mapping is to associate a single LLVM IR instruction with all the ISA instructions that originated from it, whenever possible. Then, by aggregating the energy costs of these ISA instructions, an energy cost can be assigned to their single corresponding LLVM IR instruction. The mapping technique also guarantees that there is no loss of energy between the two levels as each ISA instruction will be mapped to one LLVM IR instruction. The mapping is formalized as follows. For a program P, let

$$\text{IRprog}_L = \{1,2,...,n\} \tag{6.1}$$

be the ID numbers of P's LLVM IR instructions after the LLVM transformation and optimizations passes, with

$$\text{IRprog} = \langle ir_1, ir_2, ..., ir_n \rangle \tag{6.2}$$

being the sequence of LLVM IR instructions for P. An architecture-specific compiler back-end T_{arch} translates the IR program into ISA code:

$$\text{T}_{arch}(\text{IRprog}) = \text{ISAprog} = \langle isa_1, isa_2, ..., isa_k \rangle \tag{6.3}$$

producing a sequence of machine instructions $\langle isa_1, isa_2, ..., isa_k \rangle$ that represents the program P at ISA level. Let

$$\text{M}(\text{IRprog}) = \langle (isa_1, m_1), (isa_2, m_2), ..., (isa_l, m_l) \rangle \text{ where } m_1, m_2, ..., m_l \in \text{IRprog}_L \tag{6.4}$$

be the mapping process that monitors T_{arch} and creates a relation between the sequence of ISA instructions for P and the IDs of the LLVM IR instructions, with the aim to associate an ISA instruction with the LLVM IR instruction it originated from, whenever this is possible. For ISA instructions that are not related to any LLVM IR instruction (ISA injected instructions) or whose origin LLVM IR instruction is ambiguous, the mapping process has to make an implementation-specific choice as to which LLVM IR instruction an ISA instruction should be associated with.

This will preserve the 1:m relation and ensure that such instructions are accounted at the LLVM IR level. The mapping function

$$\begin{aligned} &\mathrm{R}(i) = \{isa_j | ir_i \in \mathrm{IRprog} \land isa_j \in \mathrm{ISAprog} \land (isa_j, i) \in \mathrm{M(IRprog)}\} \text{ with} \\ &\text{the property } \forall\ ir_n, ir_k \in \mathrm{IRprog} \land n \neq k \text{ then } \mathrm{R}(ir_n) \cap \mathrm{R}(ir_k) = \emptyset \end{aligned} \tag{6.5}$$

captures a $1:m$ relation from IRprog to ISAprog instructions. The energy consumption of an LLVM IR instruction can be retrieved by

$$\mathrm{E}(ir_i) = \sum_{isa_j\ \in\ \mathrm{R}(i)} \mathrm{E}(isa_j) \text{ where } ir_i \in \mathrm{IRprog} \tag{6.6}$$

as the sum of the energy consumed by all ISA instructions mapped to that LLVM IR instruction. Equation (6.6) can also be used to associate timing or code size information with LLVM IR instructions, by replacing the ISA instructions' energy costings with the resource of interest costings.

6.5 Xcore mapping instantiation and tuning

In the Xcore case, the T_{arch} function is the XMOS toolchain lowering phase that translates the LLVM IR to Xcore-specific ISA. The real challenge is to create from scratch a mechanism that can monitor T_{arch} and create the mapping given by Equation (6.4) which will have the property of Equation (6.5). This would require an extensive knowledge of the back-end of the architecture under consideration and a vast engineering effort to take into account every possible transformation and optimization that happens between the optimized LLVM IR and the final ISA emitted code. Instead, this thesis demonstrates a simple yet powerful technique that can provide sufficiently accurate results. An overview of the mapping technique is given in Figure 6.3. The three mapping stages are described in the following subsections.

6.5.1 Stage 1: LLVM IR annotation

The goal of this stage is to enable the property of Equation (6.5) that ensures a $1:m$ relation between the LLVM IR and ISA code. To achieve this, the mapping implementation leverages the debug mechanism in the XMOS compiler toolchain. This is typically used by the programmer to identify and fix problems in application code. Symbols are created during compilation to assist with debugging. These symbols are propagated to all intermediate code layers and down to the ISA code. Debug symbols can express which programming language constructs generated a specific piece of machine code in a given executable module. In the Xcore case, these symbols are generated by the front-end of the XMOS compiler in standard Debugging With Arbitrary Record Format (DWARF) [5]. These are transformed to LLVM metadata [62] and attached to the LLVM IR. LLVM 2.7 and upwards uses this metadata format as the primary means of storing debug

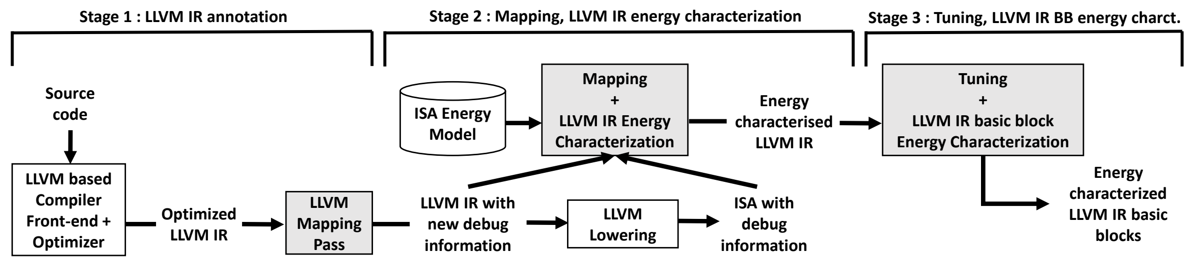

Stage 1 : LLVM IR annotation
Stage 2 : Mapping, LLVM IR energy characterization
Stage 3 : Tuning, LLVM IR BB energy charct.
Source code
LLVM based Compiler Front-end + Optimizer
Optimized LLVM IR
LLVM Mapping Pass
LLVM IR with new debug information
LLVM Lowering
ISA with debug information
ISA Energy Model
Mapping + LLVM IR Energy Characterization
Energy characterised LLVM IR
Tuning + LLVM IR basic block Energy Characterization
Energy characterized LLVM IR basic blocks

Figure 6.3: LLVM IR energy characterization overview.

information. The LLVM community has put significant effort into preserving metadata nodes through code transformations and optimizations, so using this metadata allows us to benefit from this preservation.

During the lowering phase of compilation, LLVM IR code is transformed to a target ISA by the back-end of the compiler, with debug information stored alongside in the DWARF standard format. Naturally, the accuracy of debug information in the output executable is reduced if the number of optimization passes is increased. This is due to portions of the initial LLVM IR either being discarded or merged during these passes. Tracking source code debug information gives an $n : m$ relationship between instructions at the different layers, because several source code instructions can be translated to many LLVM IR instructions, and these again into many ISA instructions.

An example of this $n : m$ relation is given in Figure 6.4(a). The left hand side of this figure shows part of the LLVM IR CFG of a program, along with the debug location for each LLVM IR instruction. On the right hand side the corresponding ISA CFG is given, together with the debug location for each ISA instruction. The coloring of the instructions demonstrates the mapping between the two CFGs instructions using their debug locations. This $n : m$ relation prevents a fine-grained energy mapping needed for accurate energy estimations.

In this thesis, to achieve a $1 : m$ mapping between the optimized LLVM IR instructions and the ISA instructions using the debug mechanism, an LLVM pass is created that traverses the optimized LLVM IR and replaces source location information with LLVM IR location information, or adds new location information to LLVM IR instructions without any debug data. This LLVM pass runs after all optimization passes, just before emitting ISA code, because the optimized LLVM IR is closer in structure to the ISA code than the unoptimized version. An example output of this process is given on the left-hand side of Figure 6.4(b). This represents a part of the LLVM IR CFG of a program after the LLVM optimizations, along with the unique debug location, IRprog_L in Equation (6.1), assigned to each LLVM IR instruction.

The XMOS compiler debug mechanism applies the following four rules to preserve the debug information through the different transformations and optimizations applied in the back-end:

1. If an instruction is eliminated, its debug location information is also eliminated.

2. If one instruction is transformed into another one, the debug location of the original instruction is assigned to the new instruction.

3. If multiple instructions are merged into one, the debug location of one of the initial instructions, the first one in the case of this thesis, is assigned to the new instruction.

4. If one instruction is transformed into multiple ones, then all the new instructions are being assigned the debug location of their origin instruction.

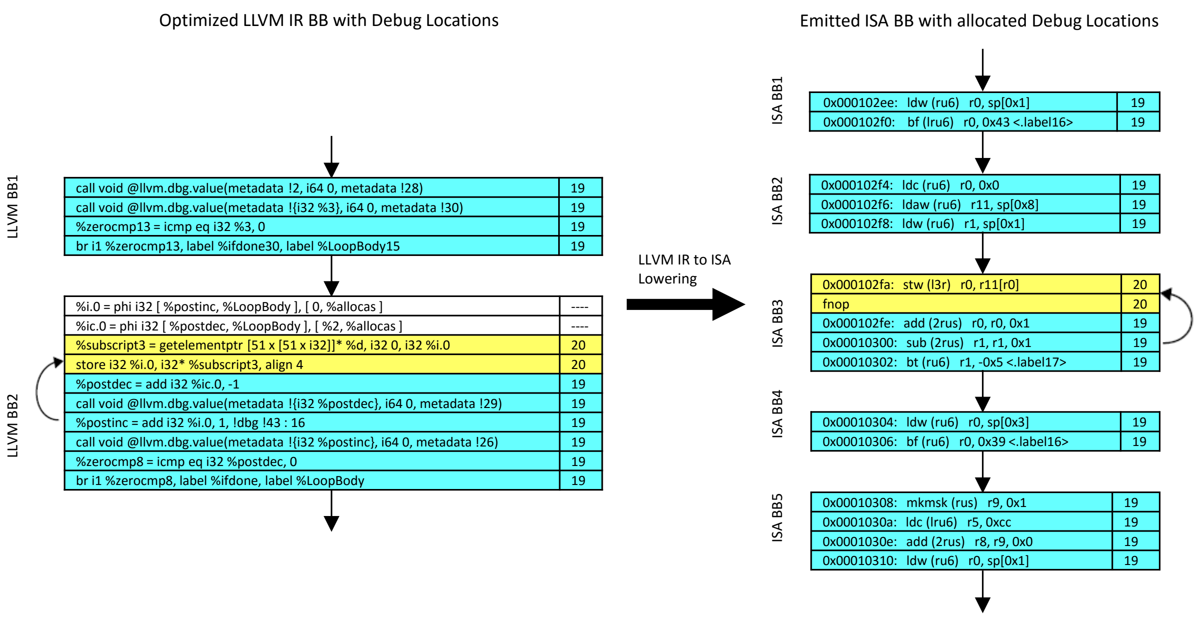
These rules iteratively preserve the $1{:}m$ relationship between the LLVM IR instructions in IRprog and the final ISA in ISAprog, with two exceptions.

The first is when instructions are introduced at the ISA level and they do not correspond to any LLVM IR instruction (ISA injected instructions). In such cases the mapping process can assign to them the debug location of an adjacent ISA instruction in the same BBs. This ensures that they are accounted for in the mapped LLVM IR block. The second is when a transformation takes as an input multiple instructions and converts them to another set of multiple instructions in a single step. An example of such transformations are peephole optimizations. In such cases the handling of debug information depends on the compiler implementation. For the back-end under investigation, ISA instructions generated by such transformations are left without any debug information. To account for these instructions at the LLVM IR level, the mapping treats them as ISA injected instructions.

6.5.2 Stage 2: Mapping and LLVM IR energy characterization

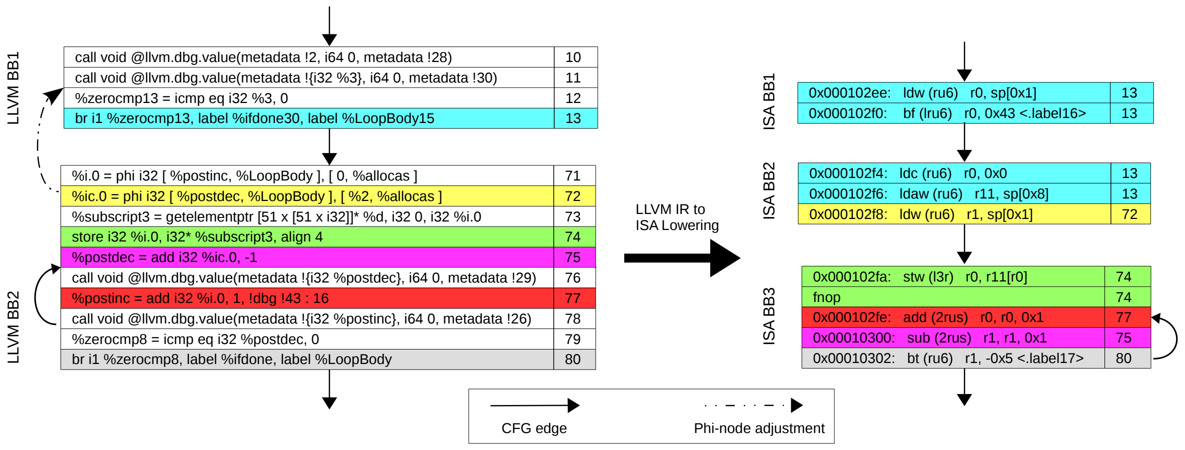
Once the LLVM IR annotation has been performed for a program, the back-end lowering phase translates the LLVM IR code to target-specific ISA code. Then the mapping phase, which implements Equation (6.5), runs and maps LLVM IR instructions with the new debug locations to the emitted ISA instructions which carry the same debug location ID. Finally, the energy values for groups of ISA instructions are aggregated and then associated with their single corresponding LLVM IR instruction, as described by Equation (6.6).

An example mapping is given in Figure 6.4(b). This is the same part of the CFG used in Figure 6.4(a). On the left-hand side is a part of the LLVM IR CFG of a program, which represents the IRprog in Equation (6.2), along with the new debug location assigned to each LLVM IR instruction by the mapping pass and represented by Equation (6.1). The right-hand side shows the corresponding ISA CFG, together with the debug locations for each ISA instruction, given by Equation (6.4). The coloring of the instructions demonstrates the mapping between the two CFGs' instructions using Equation (6.5). Now, one LLVM IR instruction is associated with many ISA instructions, but each ISA instruction is mapped to only one LLVM IR instruction. Some LLVM IR instructions are not mapped, because they are removed during the lowering phase of the compiler. This mapping also guarantees that all ISA instructions are mapped to the LLVM IR, so there is no loss of energy between the two levels.


Optimized LLVM IR BB with Debug Locations
Emitted ISA BB with allocated Debug Locations
LLVM IR to ISA Lowering
LLVM BB1
LLVM BB2
ISA BB1
ISA BB2
ISA BB3
ISA BB4
ISA BB5

```
LLVM BB1
call void @llvm.dbg.value(metadata !2, i64 0, metadata !28)                   19
call void @llvm.dbg.value(metadata !{i32 %3}, i64 0, metadata !30)            19
%zerocmp13 = icmp eq i32 %3, 0                                                19
br i1 %zerocmp13, label %ifdone30, label %LoopBody15                          19

LLVM BB2
%i.0 = phi i32 [ %postinc, %LoopBody ], [ 0, %allocas ]                       ----
%ic.0 = phi i32 [ %postdec, %LoopBody ], [ %2, %allocas ]                     ----
%subscript3 = getelementptr [51 x [51 x i32]]* %d, i32 0, i32 %i.0            20
store i32 %i.0, i32* %subscript3, align 4                                     20
%postdec = add i32 %ic.0, -1                                                  19
call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !29)      19
%postinc = add i32 %i.0, 1, !dbg !43 : 16                                     19
call void @llvm.dbg.value(metadata !{i32 %postinc}, i64 0, metadata !26)      19
%zerocmp8 = icmp eq i32 %postdec, 0                                           19
br i1 %zerocmp8, label %ifdone, label %LoopBody                               19

ISA BB1
0x000102ee:  ldw (ru6)   r0, sp[0x1]                19
0x000102f0:  bf (lru6)   r0, 0x43 <.label16>        19

ISA BB2
0x000102f4:  ldc (ru6)   r0, 0x0                    19
0x000102f6:  ldaw (ru6)  r11, sp[0x8]               19
0x000102f8:  ldw (ru6)   r1, sp[0x1]                19

ISA BB3
0x000102fa:  stw (l3r)   r0, r11[r0]                20
fnop                                                20
0x000102fe:  add (2rus)  r0, r0, 0x1                19
0x00010300:  sub (2rus)  r1, r1, 0x1                19
0x00010302:  bt (ru6)    r1, -0x5 <.label17>        19

ISA BB4
0x00010304:  ldw (ru6)   r0, sp[0x3]                19
0x00010306:  bf (ru6)    r0, 0x39 <.label16>        19

ISA BB5
0x00010308:  mkmsk (rus) r9, 0x1                    19
0x0001030a:  ldc (lru6)  r5, 0xcc                   19
0x0001030e:  add (2rus)  r8, r9, 0x0                19
0x00010310:  ldw (ru6)   r0, sp[0x1]                19
```

(a) LLVM IR to ISA $n:m$ mapping using debug information without the mapping pass.



```
LLVM BB1
call void @llvm.dbg.value(metadata !2, i64 0, metadata !28)                   10
call void @llvm.dbg.value(metadata !{i32 %3}, i64 0, metadata !30)            11
%zerocmp13 = icmp eq i32 %3, 0                                                12
br i1 %zerocmp13, label %ifdone30, label %LoopBody15                          13

LLVM BB2
%i.0 = phi i32 [ %postinc, %LoopBody ], [ 0, %allocas ]                       71
%ic.0 = phi i32 [ %postdec, %LoopBody ], [ %2, %allocas ]                     72
%subscript3 = getelementptr [51 x [51 x i32]]* %d, i32 0, i32 %i.0            73
store i32 %i.0, i32* %subscript3, align 4                                     74
%postdec = add i32 %ic.0, -1                                                  75
call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !29)      76
%postinc = add i32 %i.0, 1, !dbg !43 : 16                                     77
call void @llvm.dbg.value(metadata !{i32 %postinc}, i64 0, metadata !26)      78
%zerocmp8 = icmp eq i32 %postdec, 0                                           79
br i1 %zerocmp8, label %ifdone, label %LoopBody                               80

ISA BB1
0x000102ee:  ldw (ru6)   r0, sp[0x1]                13
0x000102f0:  bf (lru6)   r0, 0x43 <.label16>        13

ISA BB2
0x000102f4:  ldc (ru6)   r0, 0x0                    13
0x000102f6:  ldaw (ru6)  r11, sp[0x8]               13
0x000102f8:  ldw (ru6)   r1, sp[0x1]                72

ISA BB3
0x000102fa:  stw (l3r)   r0, r11[r0]                74
fnop                                                74
0x000102fe:  add (2rus)  r0, r0, 0x1                77
0x00010300:  sub (2rus)  r1, r1, 0x1                75
0x00010302:  bt (ru6)    r1, -0x5 <.label17>        80
```

(b) Fine grained $1:m$ mapping including the LLVM mapping pass.

Figure 6.4: LLVM IR to ISA mapping before and after the use of the LLVM mapping pass.

This Xcore instantiation of the mapping technique, using the debug mechanism, can be ported to any architecture supported by the LLVM IR compiler for which an ISA-level energy model exists. Typically, these are deeply embedded processors, such as the one used here and the `ARM Cortex M` series, which are ideal for IoT applications.

6.5.3 Stage 3: Tuning and basic block energy characterization

Without any further tuning, the mapping instantiation for the Xcore architecture provides an average deviation of 6% between the SRA prediction at the LLVM IR level and that at the

ISA level. An additional tuning phase is introduced after the mapping to account for specific architecture behavior and to facilitate this thesis BB level energy analysis. In the case of the bound analysis, this improved the LLVM IR SRA accuracy, narrowing the gap between ISA and LLVM IR energy predictions to an average of 1% as demonstrated in this thesis results in Section 8.2. This tuning had a similar positive effect on the accuracy of the LLVM IR profiling-based energy estimation.

As discussed in Section 5.4, FNOPs can be issued by the processor and this can be statically determined at the ISA level. This can not be represented in LLVM IR. Ignoring FNOPs can therefore lead to a significant underestimation of energy at the LLVM IR level. To account for FNOPs at the LLVM IR level, the mapping treats them similarly to the ISA injected instructions; by assigning them the debug location of an adjacent ISA instruction in the same BB. Figure 6.4, provides an example of such FNOP treatment at debug location number 74.

For an energy estimation method to work on the LLVM BB level, the energy cost of each BB is needed. This can be obtained by accumulating the energy costs of all LLVM IR instructions in an LLVM IR BB. When estimating energy at the BB level, the position of the LLVM IR instructions in BBs is critical, and tuning may be needed to transfer mapped energy costs to different BBs to reflect more accurately where the energy is consumed during program execution. *Phi-nodes*, for example, benefit from such tuning.

Phi-nodes can be introduced at the start of a BB as a side effect of the Single Static Assignment (SSA) used for variables in the LLVM IR. A *phi-node* takes a list of pairs, where each pair contains a reference to the predecessor block together with the variable that is propagated from there to the current block. The number of pairs is equal to the number of predecessor blocks of the current block. A *phi-node* can create inaccuracies in the mapping when LLVM IR is lowered to ISA code that no longer supports SSA, because it can be hoisted out from its current block to the corresponding predecessor block at the ISA level. For blocks in loops this can lead to a significant estimation error. Such cases can be detected by examining the mapping. Similar inaccuracies can be introduced by branching LLVM IR instructions with multiple targets, if the ISA of the target processor supports only single-target branches.

Algorithm 1 detects cases where the ISA energy values mapped to *phi-nodes* in a looping BB are accumulated into the wrong LLVM IR BB. It then hoist these costs out to the appropriate LLVM IR predecessor BBs. The algorithm detects the problematic cases by using the mapping and the BB loop depth. Then, by examining the mapping, function `FindIRBB` at Line 10 finds the LLVM IR predecessor BB that matches the ISA block holding the ISA *phi-node* generated instruction. The ISA's instruction energy cost is then added to the cost of the selected predecessor LLVM IR BB and subtracted from the *phi-node's* LLVM IR BB. Performing the mapping after the LLVM IR optimization passes and just before the lowering phase increases the possibility of similar CFG structures between the two levels. This improves the ability of `FindIRBB` to find the correct BB and therefore improves the results of the *phi-node* tuning.

ALGORITHM 1: *phi-node* tuning

```
input : IRprog for a program $P$ as described by Equation (6.2)
input : IRprogCFG, the CFG for IRprog, with the total energy cost for each BB
input : ISAprogCFG, the CFG for the ISAprog given by Equation (6.3) with BB total energy info
input : $M$, the $1:m$ mapping retrieved for $P$ using the mapping as described in this section
1  for each $ir_n \in$ IRprog do
2      if $ir_n$ is a phi-node then
3          ISAmap ← GetISAmap($ir_n$, $M$) // all the ISA instructions mapped to $ir_n$
4          irBBloopDepth ← GetInstrBBLoopDepth($ir_n$) // degree of nested loops for $ir_n$'s BB
5          FromIRBB ← GetCurrentIRBB (IRprogCFG, $ir_n$) // the $ir_n$'s BB
6          for each $isa_k \in$ ISAmap do
7              isaBBloopDepth ← GetInstrBBLoopDepth($isa_k$) // degree of nested loops for $isa_k$'s BB
8              if irBBloopDepth > isaBBloopDepth then
9                  ISABB ← GetISABB (ISAprogCFG, $isa_k$) // the $isa_k$'s BB
10                 ToIRBB ← FindIRBB (IRprogCFG, ISABB) // finds the IR predecessor BB of FromIRBB
                   that matches ISABB by examining the mapping of their instructions
11                 if ToIRBB not NULL then
12                     ISAcost ← GetCost ($isa_k$) // the energy cost of $isa_k$ from the ISA energy model
                       introduced in Section 5.2
13                     RemoveCostFromBB (FromIRBB, ISAcost)// remove cost from initial block
14                     AddCostToBB (ToIRBB, ISAcost) // add cost to the chosen predecessor block
15                 end
16             end
17         end
18     end
19 end
```

An example of a *phi-node* adjustment is given in Figure 6.4 at debug location 72. Its corresponding ISA instruction is hoisted out from the loop BB `ISA BB3` and into `ISA BB2`. A similar hoisting is performed from `LLVM BB2` and into `LLVM BB1` by Algorithm 1, thus correctly assigning energy values to each LLVM IR block.

6.5.4 Limitations

Our mapping approach between the LLVM IR and the ISA guarantees that no energy is lost between the two levels, as all the ISA instruction energy costs are propagated to the LLVM IR level. Therefore, any inaccuracies introduced in the LLVM IR energy estimations from the mapping are a consequence of attributing ISA energy costs to the wrong LLVM IR BBs. This is because the LLVM IR estimation techniques work at a BB level. In that respect, two cases can affect the estimation accuracy.

In the first case, for instructions at the boundaries of LLVM IR BBs, their corresponding ISA instructions may cross these boundaries at the ISA level. The mapping will, however, still associate the energy costs of these ISA instructions with their original LLVM IR instruction, and thus with their original LLVM IR BB. If such LLVM IR instructions belong to a BB that is part of a loop, when the ISA instruction has been hoisted out of that loop, then, with no proper adjustment, an overestimation will occur due to the mapping. An example of such a case are the *phi-node* instructions, described in Section 6.5.3, where Algorithm 1 is introduced to adjust their mapping. Such cases can be statically identified by examining the mapping. In the second case, a difference between the two CFG structures can occur when BBs are introduced/eliminated at the ISA level. If this makes the structures of the two CFGs significantly different, then the energy costs allocated to the LLVM IR BBs by the mapping can be inaccurate. Performing the mapping after the LLVM IR optimization passes and just before the lowering phase increases the possibility of having similar CFG structures between the two levels and therefore the accuracy of the mapping. The impact of the above issues is investigated in Section 8.1.

6.6 Mapping interface

For the scope of this thesis, the mapping techniques are implemented by a Proof-of-Concept (PoC) tool. The mapping information for a program is emitted by this PoC tool in a file using the JavaScript Object Notation (JSON) [56]. JSON is at the same time human-readable and easy for machines to parse and generate notation. Many languages have support for parsing and generating this format. Therefore, the JSON mapping output file was selected to serve as an interface between the mapping PoC tool and any other tool that needs the mapping information. [44] and [69] used this interface to utilize the mapping information in their energy consumption static analysis approaches to retrieve energy consumption estimations at the LLVM IR level. The energy consumption analysis techniques presented in this thesis also use this interface.

Listing 6.1 demonstrates the format of the mapping output for a program using JSON. The top level represents the program entity and has entries for the program's filename and the path that the program's files are stored on disk. The next inner level is a list of all the program's functions. For each function, its name, starting line and ending line at the source code are stored. Inside each function object, there is a list of all the basic blocks. For each basic block its name, starting line and ending line at the source code are stored. Each basic block also holds a list of its LLVM IR instructions together with their ISA associated instructions. The total energy cost for each LLVM IR instruction and LLVM IR basic block are also saved.

The starting and ending source code line entries for the LLVM IR basic blocks and functions are kept by the mapping tool to enable a three-way mapping between the source code, the optimized LLVM IR, where the mapping is working upon, and the ISA. By using this three-way mapping, the energy estimation techniques introduced in this thesis can attribute the retrieved energy consumption estimations to various software components at the source code level, such as loops. This source code mapping is similar to the source-level annotation done for the purpose of Software-Level Simulation (SLS) [80, 87, 120, 121]. SLS is used as an alternative of ISS as simulating at the host machine is significantly faster. In a SLS scenario, the source code of a program is annotated based on the matching of the CFGs between the source-code level and

```
{"program": {
  "functions": [{
    "blocks": [
      {
        "mapping": [
          {
            "ISAMap": [
            ISA_1,
            ...
            ],
            "LLVMIRIns": LLVMIR_1,
            "LLVMIRInstrEnergy": "\gls{LLVM} \gls{IR} instruction's energy cost
                 in pJ"
          },
          ...
        ],
        "name": "block's name",
        "lineStarts": "block's source code starting line number",
        "lineEnds": "block's source code ending line number",
        "energy": "block's total energy cost in pJ"
      },
      ...
      ],
    "name": "function's name",
    "lineStarts": "function's source code starting line number",
    "lineEnds": "function's source code ending line number"
  }
  ...
  ],
  }
  "name": "program's file name",
  "fileLocation": "file's path on disk"
}
```

Listing 6.1: The JSON format of the ISA to LLVM IR mapping

the ISA level of a program. The source code annotation is then translated into instrumentation code when the program is compiled for the host machine. Executing the compiled code on the host machine yields an approximation of the program's execution path, which was going to be obtained if the program had been executed on the target processor [121]. SLS is beyond the scope of this thesis, and it is future work to determine the effectiveness of our source code annotation for retrieving SLS-based energy estimations.

An actual example of a JSON file created by the PoC mapping tool together with its corresponding source code is given in Appendix A, for an implementation of the matrix multiplication function.

6.7 Conclusion

In this chapter, the trade-off between the different software abstraction levels and the energy estimation accuracy is investigated. The LLVM IR level is chosen for performing energy estimations beyond the ISA level. This is because, the LLVM IR level offers a good compromise between the availability of program information and the loss of energy consumption estimation accuracy, in comparison to performing energy consumption estimations at the ISA level. Instead of directly energy modeling at the LLVM IR level, a different approach is taken using a novel mapping technique. This is a dynamic technique that uses an ISA cost model to provide energy costs for the LLVM IR code of a program. Therefore, the technique benefits from the accuracy that an ISA model can offer, as it is the software abstraction level that is closer to the hardware. The technique can also account for particular compiler and architecture behavior. The generic technique has been specialized for energy and implemented for the Xcore in a PoC tool. JSON files serve as an interface to this tool. The energy estimations techniques introduced in the next chapters will use this interface to perform energy consumption estimations at the LLVM IR level.

Chapter 7

Energy Estimation Techniques Introduced

Although avoiding physical energy measurements means sacrificing some accuracy in the retrieved energy estimations, it is the key element for unlocking the energy-aware development of software to a greater number of less experienced developers. Therefore, it helps to establish the energy consumption of computing as a first class design goal for software. In this chapter, two energy consumption analysis techniques are introduced. The first technique performs SRA based on IPET, to retrieve the energy consumption bounds of a program. This works both at the ISA level and at the LLVM IR level using the mapping technique introduced in the previous chapter. The second technique uses profiling to estimate the actual energy consumption of a program rather than bounds. It also uses the mapping technique to retrieve estimations at the LLVM IR level. Both techniques are designed to be target agnostic and to provide fine-grained energy consumption characterization of software. A running example will be used across the whole chapter to demonstrate the application of the analysis techniques introduced; an implementation of the matrix multiplication function shown in Listing 7.1.

7.1 SRA-based energy consumption estimation

SRA can be utilized for bounding the energy consumption of software, as an alternative to end-to-end physical energy measurements and simulation-based energy consumption estimation. As mentioned in Chapter 3, IPET has been very successful in the area of WCET for deeply embedded processors. Therefore, IPET is also adopted in this work with the aim of bounding the energy consumption of deeply embedded programs. Although the first steps towards energy consumption SRA using IPET have been made in [55] and [132], none of these works adequately investigated the applicability of the technique for retrieving energy consumption bounds; the work performed in [55] was focused on a simulated processor and in [132] the evaluation was limited to only one benchmark. In the scope of this thesis, IPET is applied to an actual processor, the Xcore, and its applicability and usefulness in the context of energy transparency are thoroughly investigated through a large number of benchmarks and some industrial case studies.

```c
void Multiply(matrix A, matrix B, matrix Res, int size)
{
   register int Outer, Inner, Index;
   size--;
   for (Outer = size; Outer >= 0; Outer--)
      for (Inner = size; Inner >= 0; Inner--)
      {
         Res [Outer][Inner] = 0;
         for (Index = size; Index >= 0; Index--)
            Res[Outer][Inner]  += A[Outer][Index] * B[Index][Inner];
      }
}
```

Listing 7.1: Matrix multiplication function used as a running example in this chapter.

Figure 7.1 shows the SRA process for both analysis at the ISA and at the LLVM IR level. The main stages of the process are based on the three elements required for performing an IPET style SRA: assigning resource costings to CFG basic blocks, ILP problem formulation, and constraining and solving the formulated ILP problem, as described in Section 3.2.3. For the ISA level energy consumption estimations the following stages are applied:

- low-level analysis, introduced in Chapter 5, is used for assigning energy costings to the CFG basic blocks;
- analysis of program control flow is used to formulate and constraint the ILP problem;
- user input to provide the loop bound information;
- IPET-based computation is used to retrieve energy consumption bounds.

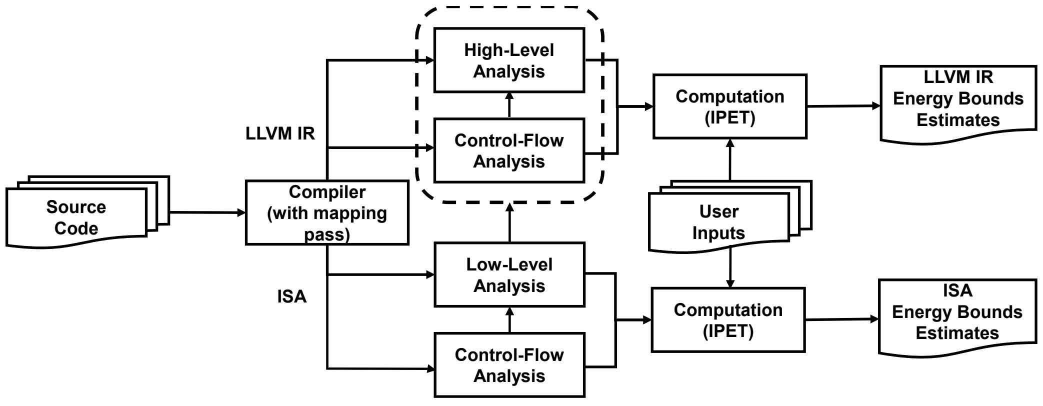


Figure 7.1: Overview of the energy consumption static resource analysis.

In the case of the LLVM IR analysis an extra step is required; the high-level analysis. As described in Section 6.5, the high-level analysis requires the input from both control-flow analysis and low-level analysis. Furthermore, the mapping pass needs to be invoked at the compilation stage. Beyond enabling the analysis at the LLVM IR level, the mapping pass also produces source code location information for linking the SRA-based results, generated by both the ISA and LLVM IR SRAs, back to the source code, as discussed in Section 6.6.

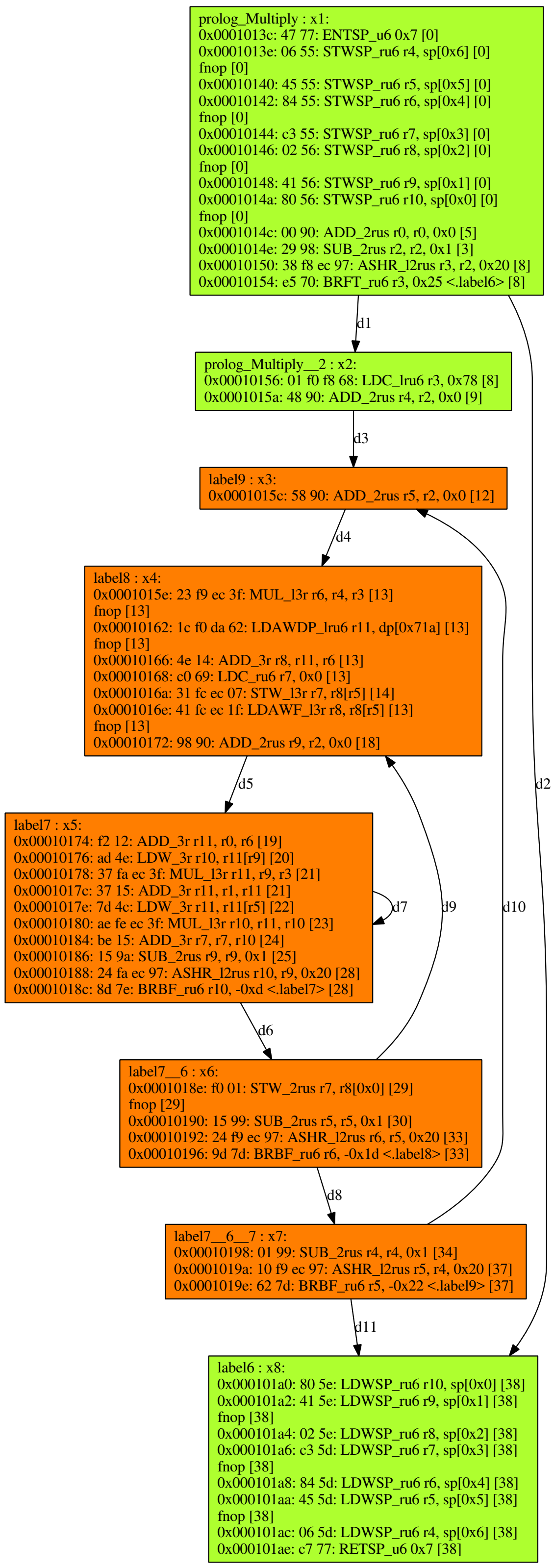
A PoC tool has been created to implement the IPET-like SRA demonstrated in Figure 7.1. In the next section, the important implementation details and the flow of extracting bound estimations by the use of this tool are illustrated through the chapter's running example.

7.1.1 IPET implementation

IPET requires the CFG and call graph of a program to be constructed at the same level as the analysis. At the LLVM IR level, the compiler can generate them. At the ISA level a tool was created to construct them in this thesis. This works on the disassembled code of the programs' executables. To detect basic blocks that belong to a loop or recursion, the algorithm in [138] is adopted. The CFGs are annotated according to the needs of IPET, as described in Section 3.2.3.

The GraphViz tool [43] is used to visualize the annotated CFGs. Figure 7.2 gives an example of this visualization at the ISA level for the function in Listing 7.1. For each basic block in the graph, at its top, its actual name and its name allocated by the IPET annotation is given in the format "Actual block name: IPET-based name". Then the list of all the BB's ISA instructions is given as they appear in the disassembled code. The format of each of these lines is "instruction address: instruction encoding: instruction mnemonic: [mapping ID]". The mapping ID is extracted from the Dwarf debug information. This is the information that the mapping technique uses to enable the mapping between the ISA and the LLVM IR levels, as described in Section 6.5.1. A green colored basic block denotes that the block is not part of any loop and an orange colored block denotes a looping basic block;a basic block that is a member of a loop. Beyond the instructions defined in ISA, the implicit FNOP operations are also introduced in each basic block by the low-level analysis as described in Section 5.3. Finally, the CFG's edges are also annotated according to the IPET needs.

The PoC tool can automatically infer and partially constrain the ILP problem formulation by using the produced IPET-annotated CFGs. The ILP formulated function, and its constraints are stored in the form of an LP file format [79], which can be used as an input to the LP solver open source tool [78]. Listing 7.2 shows the LP extracted file for the IPET-annotated CFG in Figure 7.2. The first line of the file represents the formulated ILP objective function, as described in Section 3.2.3.1. The directive *max* at the begin of the line instructs the LP solver tool to maximize the solution, in order to retrieve an energy consumption upper bound. In the case that the lower bound is required, the *min* directive can be used. To infer the energy consumption, instead of using the time cost of a CFG basic block, traditionally done for WCET analysis, the



```
prolog_Multiply : x1:
0x0001013c: 47 77: ENTSP_u6 0x7 [0]
0x0001013e: 06 55: STWSP_ru6 r4, sp[0x6] [0]
fnop [0]
0x00010140: 45 55: STWSP_ru6 r5, sp[0x5] [0]
0x00010142: 84 55: STWSP_ru6 r6, sp[0x4] [0]
fnop [0]
0x00010144: c3 55: STWSP_ru6 r7, sp[0x3] [0]
0x00010146: 02 56: STWSP_ru6 r8, sp[0x2] [0]
fnop [0]
0x00010148: 41 56: STWSP_ru6 r9, sp[0x1] [0]
0x0001014a: 80 56: STWSP_ru6 r10, sp[0x0] [0]
fnop [0]
0x0001014c: 00 90: ADD_2rus r0, r0, 0x0 [5]
0x0001014e: 29 98: SUB_2rus r2, r2, 0x1 [3]
0x00010150: 38 f8 ec 97: ASHR_l2rus r3, r2, 0x20 [8]
0x00010154: e5 70: BRFT_ru6 r3, 0x25 <.label6> [8]

prolog_Multiply__2 : x2:
0x00010156: 01 f0 f8 68: LDC_lru6 r3, 0x78 [8]
0x0001015a: 48 90: ADD_2rus r4, r2, 0x0 [9]

label9 : x3:
0x0001015c: 58 90: ADD_2rus r5, r2, 0x0 [12]

label8 : x4:
0x0001015e: 23 f9 ec 3f: MUL_l3r r6, r4, r3 [13]
fnop [13]
0x00010162: 1c f0 da 62: LDAWDP_lru6 r11, dp[0x71a] [13]
fnop [13]
0x00010166: 4e 14: ADD_3r r8, r11, r6 [13]
0x00010168: c0 69: LDC_ru6 r7, 0x0 [13]
0x0001016a: 31 fc ec 07: STW_l3r r7, r8[r5] [14]
0x0001016e: 41 fc ec 1f: LDAWF_l3r r8, r8[r5] [13]
fnop [13]
0x00010172: 98 90: ADD_2rus r9, r2, 0x0 [18]

label7 : x5:
0x00010174: f2 12: ADD_3r r11, r0, r6 [19]
0x00010176: ad 4e: LDW_3r r10, r11[r9] [20]
0x00010178: 37 fa ec 3f: MUL_l3r r11, r9, r3 [21]
0x0001017c: 37 15: ADD_3r r11, r1, r11 [21]
0x0001017e: 7d 4c: LDW_3r r11, r11[r5] [22]
0x00010180: ae fe ec 3f: MUL_l3r r10, r11, r10 [23]
0x00010184: be 15: ADD_3r r7, r7, r10 [24]
0x00010186: 15 9a: SUB_2rus r9, r9, 0x1 [25]
0x00010188: 24 fa ec 97: ASHR_l2rus r10, r9, 0x20 [28]
0x0001018c: 8d 7e: BRBF_ru6 r10, -0xd <.label7> [28]

label7__6 : x6:
0x0001018e: f0 01: STW_2rus r7, r8[0x0] [29]
fnop [29]
0x00010190: 15 99: SUB_2rus r5, r5, 0x1 [30]
0x00010192: 24 f9 ec 97: ASHR_l2rus r6, r5, 0x20 [33]
0x00010196: 9d 7d: BRBF_ru6 r6, -0x1d <.label8> [33]

label7__6__7 : x7:
0x00010198: 01 99: SUB_2rus r4, r4, 0x1 [34]
0x0001019a: 10 f9 ec 97: ASHR_l2rus r5, r4, 0x20 [37]
0x0001019e: 62 7d: BRBF_ru6 r5, -0x22 <.label9> [37]

label6 : x8:
0x000101a0: 80 5e: LDWSP_ru6 r10, sp[0x0] [38]
0x000101a2: 41 5e: LDWSP_ru6 r9, sp[0x1] [38]
fnop [38]
0x000101a4: 02 5e: LDWSP_ru6 r8, sp[0x2] [38]
0x000101a6: c3 5d: LDWSP_ru6 r7, sp[0x3] [38]
fnop [38]
0x000101a8: 84 5d: LDWSP_ru6 r6, sp[0x4] [38]
0x000101aa: 45 5d: LDWSP_ru6 r5, sp[0x5] [38]
fnop [38]
0x000101ac: 06 5d: LDWSP_ru6 r4, sp[0x6] [38]
0x000101ae: c7 77: RETSP_u6 0x7 [38]
```

Figure 7.2: The IPET annotated CFG at the ISA level, for the function in Listing 7.1 as produced by the proof-of-concept tool that implements the SRA-based energy consumption estimation.

block's energy cost is used in this thesis. The low-level analysis provides the basic blocks energy costs. Therefore, for each basic block i there is a term in the objective function which is the product of the basic block's energy consumption cost and of the times the basic block needs to be executed, *x_i*, to achieve the maximum, or minimum, energy consumption of the program.

```
 1  max:18498 x_1 + 2320 x_2 + 1161 x_3 + 11766 x_4 + 12175 x_5 + 5800 x_6 +
      3503 x_7 + 12719 x_8;
 2  x_1 = 1;
 3  x_1 = d_1 + d_2;
 4  x_2 = d_1;
 5  x_2 = d_3;
 6  x_3 = d_3 + d_10;
 7  x_3 = d_4;
 8  x_4 = d_4 + d_9;
 9  x_4 = d_5;
10  x_5 = d_5 + d_7;
11  x_5 = d_6 + d_7;
12  x_6 = d_6;
13  x_6 = d_8 + d_9;
14  x_7 = d_8;
15  x_7 = d_10 + d_11;
16  x_8 = d_2 + d_11;
17  x_8 = 1;
18
19  x_5 >= bound_1 d_6;
20  x_5 <= bound_1 d_6;
21  x_6 >= bound_2 d_8;
22  x_6 <= bound_2 d_8;
23  x_7 >= bound_3 d_11;
24  x_7 <= bound_3 d_11;
25
26  int x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,d_1,d_2,d_3,d_4,d_5,d_6,d_7,d_8,d_9,d_10
      ,d_11;
```

Listing 7.2: The Integer Linear Programming (ILP) formulation and constraints for the Matrix multiplication function given in Listing 7.1.

Lines 2-17, in Listing 7.2, represent the structural constraints as described in Section 3.2.3.2. These are inferred automatically from the IPET-annotated CFGs. The user needs to provide the loop bound information to enable bounding of the ILP problem. To assist with this, the PoC tool provides the loop bound functional constraints and lets the user input the relevant iteration count for each loop. Line 19-24, in Listing 7.2, represent the functional constraints for the function in Listing 7.1. The keyword *bound_no* needs to be replaced with actual iteration numbers for the corresponding loop in the function. This is done by the tool printing the loop's function name and source code line for each loop in the program and prompting the user to provide an iteration count. In the case of the function in Listing 7.1, all three loop iteration counts are controlled by the function's parameter *size*. Using the value ten as an input example for this parameter, and solving the ILP problem with the LP solver tool, it will output the solution showed in Listing 7.3. The first line of this output represents the estimated upper bound for the function (in pJ). This is followed be the list of the execution counts for each CFG basic block and edge under the selected worst case scenario. Line 26 in Listing 7.2 instructs the solver that the basic block execution counts, *x_i*, and the edges execution counts, *d_j*, should take only integer values.

```
Value of objective function: 1.40118e+07

Actual values of the variables:
x_1                               1
x_2                               1
x_3                              10
x_4                             100
x_5                            1000
x_6                             100
x_7                              10
x_8                               1
d_1                               1
d_2                               0
d_3                               1
d_10                              9
d_4                              10
d_9                              90
d_5                             100
d_7                             900
d_6                             100
d_8                              10
d_11                              1
```

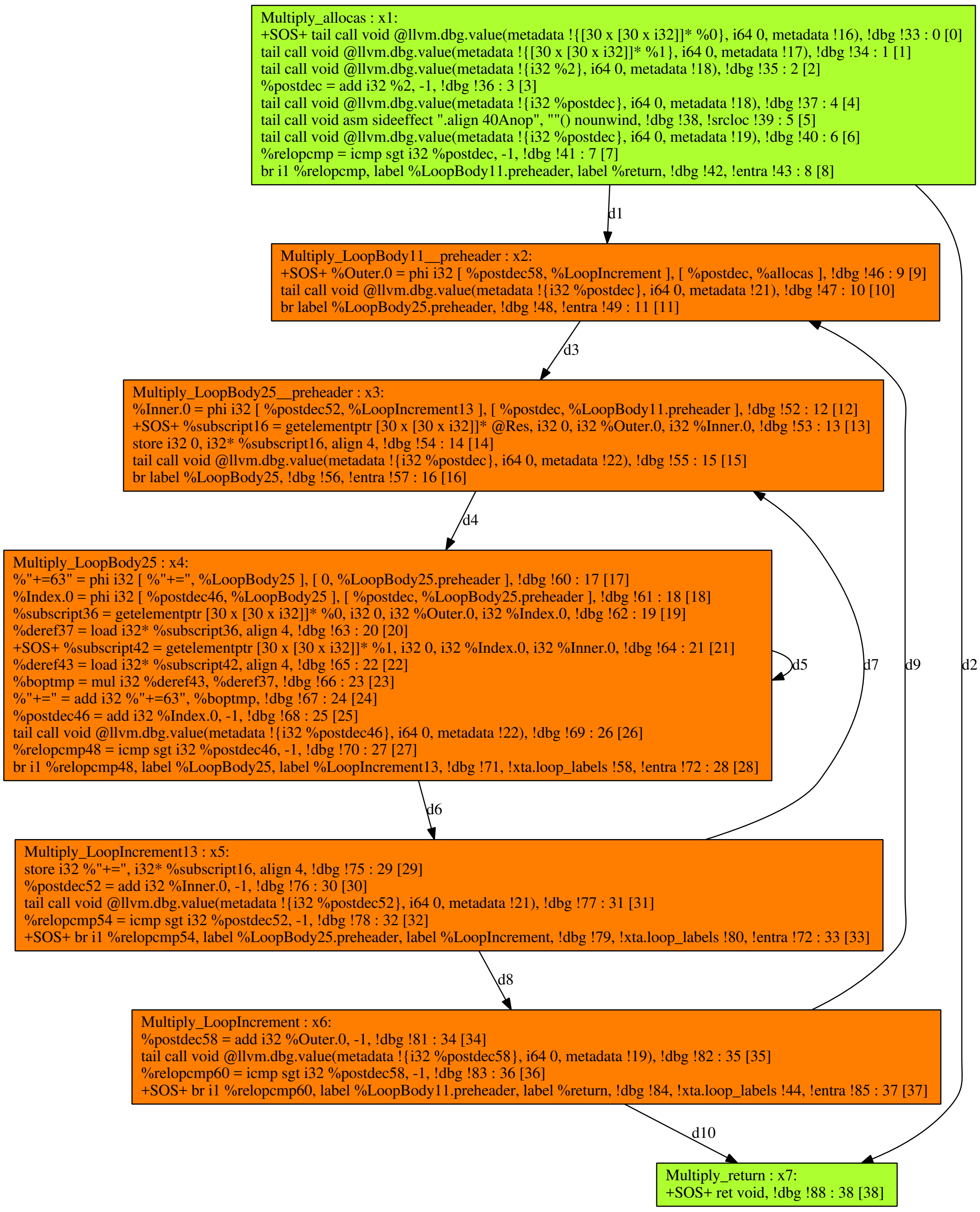
Listing 7.3: The solution to the ILP formulation in Listing 7.2. This is an upper bound energy consumption estimation using the value of ten for the *size* parameter of the matrix multiplication function in Listing 7.1.

The PoC tool can also retrieve IPET-annotated CFG graphs at the LLVM IR level, similar to the ones retrieved at the ISA level. An example of such a graph for the function in Listing 7.1 is given in Figure 7.3. The mapping ID is given at the end of each LLVM IR instruction, in the same way as in the corresponding ISA graph, Figure 7.2. These mapping IDs offer a visualization of the association created by the mapping technique, linking instructions between the ISA and the LLVM IR level. This provides some insights on how the compiler's back-end treats the code during the lowering phase, which can be valuable for exploiting opportunities to optimize code size, execution time or energy. Furthermore, the LLVM IR instructions annotated, by the PoC tool, with the keyword `+SOS+` are the ones with the largest energy costs in their BB. This information can be used by the compiler instruction selection phase to search for alternative less energy expensive options, in future.

Similarly to the ISA level analysis, LP files are also created for the LLVM IR level and solved to retrieve the energy consumption bounds estimations at the LLVM IR level. For this, the mapping process is used to assign energy costs to the LLVM IR basic blocks, as explained in Section 6.5.3.

To account for function calls, the PoC tool uses the call graphs at the targeted level of analysis. It recursively retrieves bound estimations for each called function and adds the retrieved bound cost to the cost of the appropriate BB of the caller function; the BB that calls the function.

Further functional constraints, such as denoting CFG infeasible paths, can be provided in the form of user source code annotations, to extract more accurate estimations. In the case of retrieving upper bounds, infeasible paths can lead to significant over-approximations but never



```
Multiply_allocas : x1:
+SOS+ tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %0}, i64 0, metadata !16), !dbg !33 : 0 [0]
tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %1}, i64 0, metadata !17), !dbg !34 : 1 [1]
tail call void @llvm.dbg.value(metadata !{i32 %2}, i64 0, metadata !18), !dbg !35 : 2 [2]
%postdec = add i32 %2, -1, !dbg !36 : 3 [3]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !18), !dbg !37 : 4 [4]
tail call void asm sideeffect ".align 40Anop", ""() nounwind, !dbg !38, !srcloc !39 : 5 [5]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !19), !dbg !40 : 6 [6]
%relopcmp = icmp sgt i32 %postdec, -1, !dbg !41 : 7 [7]
br i1 %relopcmp, label %LoopBody11.preheader, label %return, !dbg !42, !entra !43 : 8 [8]
```

```
Multiply_LoopBody11__preheader : x2:
+SOS+ %Outer.0 = phi i32 [ %postdec58, %LoopIncrement ], [ %postdec, %allocas ], !dbg !46 : 9 [9]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !21), !dbg !47 : 10 [10]
br label %LoopBody25.preheader, !dbg !48, !entra !49 : 11 [11]
```

```
Multiply_LoopBody25__preheader : x3:
%Inner.0 = phi i32 [ %postdec52, %LoopIncrement13 ], [ %postdec, %LoopBody11.preheader ], !dbg !52 : 12 [12]
+SOS+ %subscript16 = getelementptr [30 x [30 x i32]]* @Res, i32 0, i32 %Outer.0, i32 %Inner.0, !dbg !53 : 13 [13]
store i32 0, i32* %subscript16, align 4, !dbg !54 : 14 [14]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !22), !dbg !55 : 15 [15]
br label %LoopBody25, !dbg !56, !entra !57 : 16 [16]
```

```
Multiply_LoopBody25 : x4:
%"+=63" = phi i32 [ %"+=", %LoopBody25 ], [ 0, %LoopBody25.preheader ], !dbg !60 : 17 [17]
%Index.0 = phi i32 [ %postdec46, %LoopBody25 ], [ %postdec, %LoopBody25.preheader ], !dbg !61 : 18 [18]
%subscript36 = getelementptr [30 x [30 x i32]]* %0, i32 0, i32 %Outer.0, i32 %Index.0, !dbg !62 : 19 [19]
%deref37 = load i32* %subscript36, align 4, !dbg !63 : 20 [20]
+SOS+ %subscript42 = getelementptr [30 x [30 x i32]]* %1, i32 0, i32 %Index.0, i32 %Inner.0, !dbg !64 : 21 [21]
%deref43 = load i32* %subscript42, align 4, !dbg !65 : 22 [22]
%boptmp = mul i32 %deref43, %deref37, !dbg !66 : 23 [23]
%"+=" = add i32 %"+=63", %boptmp, !dbg !67 : 24 [24]
%postdec46 = add i32 %Index.0, -1, !dbg !68 : 25 [25]
tail call void @llvm.dbg.value(metadata !{i32 %postdec46}, i64 0, metadata !22), !dbg !69 : 26 [26]
%relopcmp48 = icmp sgt i32 %postdec46, -1, !dbg !70 : 27 [27]
br i1 %relopcmp48, label %LoopBody25, label %LoopIncrement13, !dbg !71, !xta.loop_labels !58, !entra !72 : 28 [28]
```

```
Multiply_LoopIncrement13 : x5:
store i32 %"+=", i32* %subscript16, align 4, !dbg !75 : 29 [29]
%postdec52 = add i32 %Inner.0, -1, !dbg !76 : 30 [30]
tail call void @llvm.dbg.value(metadata !{i32 %postdec52}, i64 0, metadata !21), !dbg !77 : 31 [31]
%relopcmp54 = icmp sgt i32 %postdec52, -1, !dbg !78 : 32 [32]
+SOS+ br i1 %relopcmp54, label %LoopBody25.preheader, label %LoopIncrement, !dbg !79, !xta.loop_labels !80, !entra !72 : 33 [33]
```

```
Multiply_LoopIncrement : x6:
%postdec58 = add i32 %Outer.0, -1, !dbg !81 : 34 [34]
tail call void @llvm.dbg.value(metadata !{i32 %postdec58}, i64 0, metadata !19), !dbg !82 : 35 [35]
%relopcmp60 = icmp sgt i32 %postdec58, -1, !dbg !83 : 36 [36]
+SOS+ br i1 %relopcmp60, label %LoopBody11.preheader, label %return, !dbg !84, !xta.loop_labels !44, !entra !85 : 37 [37]
```

```
Multiply_return : x7:
+SOS+ ret void, !dbg !88 : 38 [38]
```

Figure 7.3: The IPET annotated CFG at the LLVM IR level, for the function in Listing 8.1 as produced by the proof-of-concept tool that implements the SRA-based energy consumption estimation.

to underestimation. Therefore excluding infeasible paths from the SRA can only improve the tightness of the bounds. Introducing constraints to the ILP problem to account for infeasible paths can increase the problem's solving time [65]. In this thesis, all the retrieved IPET solutions, selected worst case paths in regards of energy consumption, were empirically checked to verify their feasibility. This was done either by manually inspecting the code or by using appropriate test data in combination with the profiler technique introduced in the next section. Detecting infeasible paths is an active research area [10, 12], which is beyond the scope of this work.

Finally, the execution counts estimation for each CFG basic block retrieved by the IPET allows for fine-grained energy characterization of the program at the ISA level and at the LLVM level. Furthermore, the use of the source code locations provided by the mapping JSON file, described in Section 6.6, enables the assignment of energy costs to software components at the source code level, such as loops and functions.

7.2 Profiling-based energy consumption estimation

Because currently, there is no practical method to perform actual case static analysis [128], a new target-agnostic profiling technique is also introduced. In contrast to the SRA-based energy consumption estimation, given a specific set of input parameters, the profiling technique aims to provide the actual energy consumption of a program. To achieve this, the technique collects LLVM IR BB execution counts. The technique is target-agnostic in the sense that the instrumentation code is inserted at the architecture-independent LLVM IR level and does not require the modification of a program's object code to insert instrumentation instructions, unlike in other approaches [118]. The energy consumption estimations are also collected at the LLVM IR level. This is enabled by the energy characterization of the LLVM IR code using the mapping techniques introduced in Chapter 6.

Profiling is chosen over ISS-based estimation for two reasons. Firstly, building an ISS for an architecture is a significantly bigger task than transferring a target-agnostic profiling technique to a new architecture. Secondly, in general, profiling-based estimation methods tend to outperform significantly the estimation speed achieved by the ISS-based approaches. The performance of an ISS usually depends on the complexity of the algorithm implemented for the program under execution. For the profiling technique introduced in this thesis, the profiling statistics are gathered with negligible overhead on the execution time, when the program executes on the actual platform. Thus the profiling performance depends mainly on its off-line energy analysis stages and therefore on the code size of the program under analysis. This makes the profiler more suitable for iterative optimizations during software development.

A unique characteristic of the profiling technique introduced in this thesis is the ability to provide energy estimation directly at the LLVM IR level. One of the few works that investigated energy consumption estimation at the LLVM IR level based on profiling is [17]. Their instrumen-

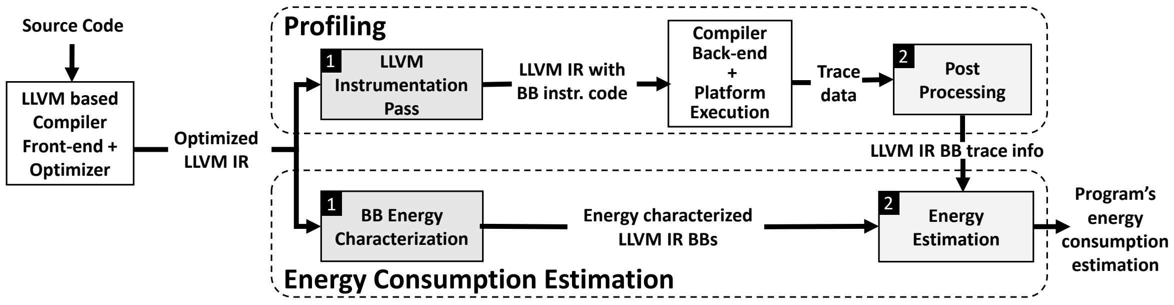

Source Code
LLVM based Compiler Front-end + Optimizer
Optimized LLVM IR
Profiling
1 LLVM Instrumentation Pass
LLVM IR with BB instr. code
Compiler Back-end + Platform Execution
Trace data
2 Post Processing
LLVM IR BB trace info
1 BB Energy Characterization
Energy characterized LLVM IR BBs
2 Energy Estimation
Program's energy consumption estimation
Energy Consumption Estimation

Figure 7.4: Overview of the profiling-based energy estimation.

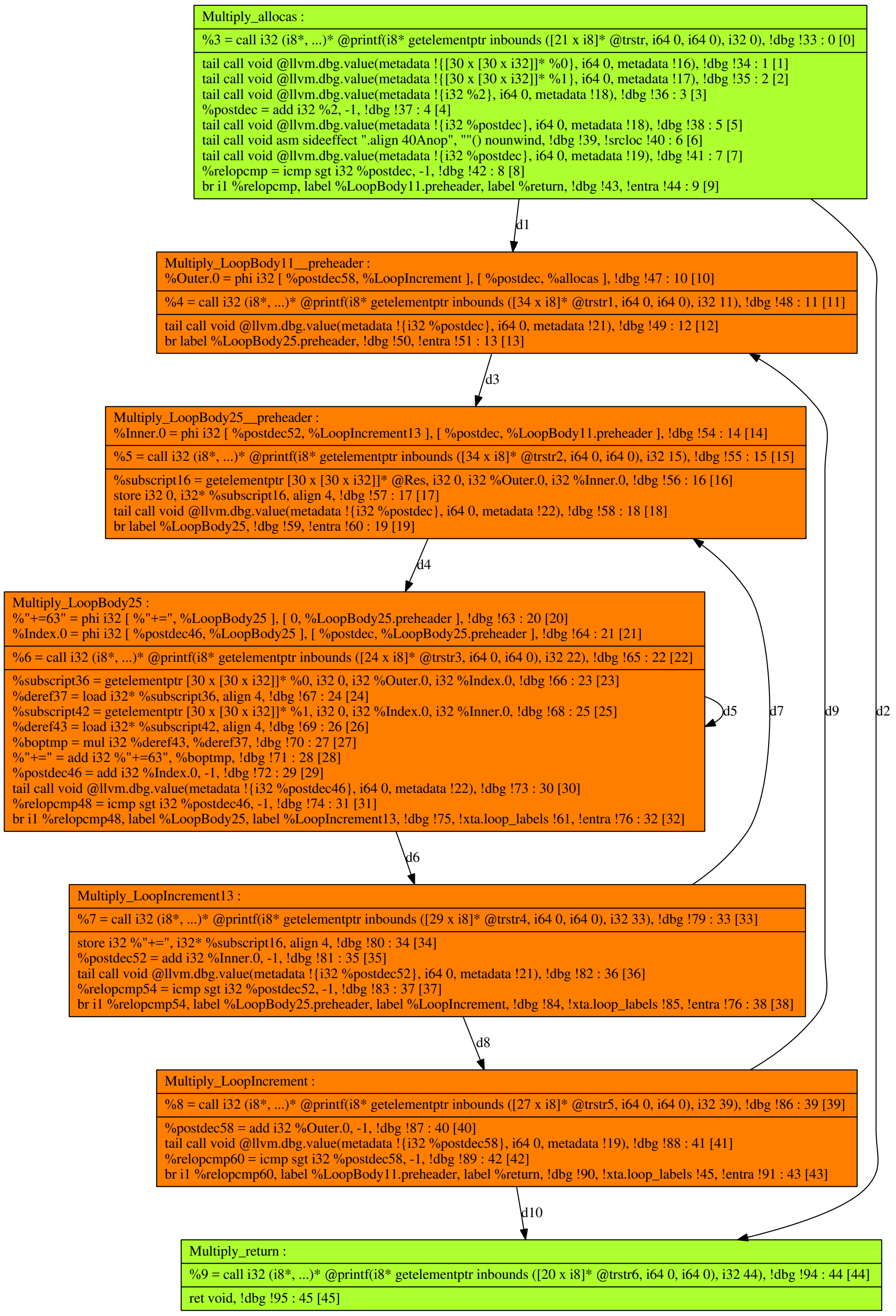
tation technique is also based on collecting execution BB counts, but this is done by executing programs on a host machine rather than the actual platform. However, compiling and executing on a higher-end PC rather than the target deeply embedded device could yield an execution profile significantly different from the one that would be obtained on the target machine. The profiling technique introduced, overcomes this issue by collecting execution counts on the target machine, and mapping them back to their corresponding LLVM IR BB.

7.2.1 Methodology and implementation

Figure 7.4 outlines the profiling-based energy consumption estimation process. The process is split in two phases: a profiling phase, which collects BB execution traces, and an energy estimation phase that performs the LLVM IR energy characterization and combines it with the profiling information to obtain energy estimations. Both phases take as input the same LLVM IR; the LLVM IR retrieved after all the optimization passes and just before emitting the target's ISA code. Using the optimized LLVM IR allows the energy estimation to benefit from a more accurate energy characterization via the mapping technique, since it is closer in structure to the ISA code than the unoptimized version. This also avoids possible negative impact on the LLVM optimizations' abilities from the injected instrumentation code. The stages for each of the two phases are annotated with numbers in black boxes in Figure 7.4 and are explained next.

Profiling Phase:

1. **LLVM Instrumentation Pass**: A new LLVM pass is built into the LLVM compiler that runs after all the LLVM optimization passes and just before the ISA lowering. The pass injects each LLVM IR BB with instrumentation instructions. To implement the BB instrumentation two choices are considered. First, using BB execution counters, one for each BB, and second using instructions that emit a unique ID, identifying the BB and its origin function, every time the BB is executed. The second option puts less stress on the limited memory size, typically available on deeply embedded devices, as there is no need to keep global variables. Therefore, a choice was made to use the *print* instruction supported in the



```
Multiply_allocas :
%3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([21 x i8]* @trstr, i64 0, i64 0), i32 0), !dbg !33 : 0 [0]
tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %0}, i64 0, metadata !16), !dbg !34 : 1 [1]
tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %1}, i64 0, metadata !17), !dbg !35 : 2 [2]
tail call void @llvm.dbg.value(metadata !{i32 %2}, i64 0, metadata !18), !dbg !36 : 3 [3]
%postdec = add i32 %2, -1, !dbg !37 : 4 [4]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !18), !dbg !38 : 5 [5]
tail call void asm sideeffect ".align 40Anop", ""() nounwind, !dbg !39, !srcloc !40 : 6 [6]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !19), !dbg !41 : 7 [7]
%relopcmp = icmp sgt i32 %postdec, -1, !dbg !42 : 8 [8]
br i1 %relopcmp, label %LoopBody11.preheader, label %return, !dbg !43, !entra !44 : 9 [9]

Multiply_LoopBody11__preheader :
%Outer.0 = phi i32 [ %postdec58, %LoopIncrement ], [ %postdec, %allocas ], !dbg !47 : 10 [10]
%4 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([34 x i8]* @trstr1, i64 0, i64 0), i32 11), !dbg !48 : 11 [11]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !21), !dbg !49 : 12 [12]
br label %LoopBody25.preheader, !dbg !50, !entra !51 : 13 [13]

Multiply_LoopBody25__preheader :
%Inner.0 = phi i32 [ %postdec52, %LoopIncrement13 ], [ %postdec, %LoopBody11.preheader ], !dbg !54 : 14 [14]
%5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([34 x i8]* @trstr2, i64 0, i64 0), i32 15), !dbg !55 : 15 [15]
%subscript16 = getelementptr [30 x [30 x i32]]* @Res, i32 0, i32 %Outer.0, i32 %Inner.0, !dbg !56 : 16 [16]
store i32 0, i32* %subscript16, align 4, !dbg !57 : 17 [17]
tail call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !22), !dbg !58 : 18 [18]
br label %LoopBody25, !dbg !59, !entra !60 : 19 [19]

Multiply_LoopBody25 :
%"+=63" = phi i32 [ %"+=", %LoopBody25 ], [ 0, %LoopBody25.preheader ], !dbg !63 : 20 [20]
%Index.0 = phi i32 [ %postdec46, %LoopBody25 ], [ %postdec, %LoopBody25.preheader ], !dbg !64 : 21 [21]
%6 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([24 x i8]* @trstr3, i64 0, i64 0), i32 22), !dbg !65 : 22 [22]
%subscript36 = getelementptr [30 x [30 x i32]]* %0, i32 0, i32 %Outer.0, i32 %Index.0, !dbg !66 : 23 [23]
%deref37 = load i32* %subscript36, align 4, !dbg !67 : 24 [24]
%subscript42 = getelementptr [30 x [30 x i32]]* %1, i32 0, i32 %Index.0, i32 %Inner.0, !dbg !68 : 25 [25]
%deref43 = load i32* %subscript42, align 4, !dbg !69 : 26 [26]
%boptmp = mul i32 %deref43, %deref37, !dbg !70 : 27 [27]
%"+=" = add i32 %"+=63", %boptmp, !dbg !71 : 28 [28]
%postdec46 = add i32 %Index.0, -1, !dbg !72 : 29 [29]
tail call void @llvm.dbg.value(metadata !{i32 %postdec46}, i64 0, metadata !22), !dbg !73 : 30 [30]
%relopcmp48 = icmp sgt i32 %postdec46, -1, !dbg !74 : 31 [31]
br i1 %relopcmp48, label %LoopBody25, label %LoopIncrement13, !dbg !75, !xta.loop_labels !61, !entra !76 : 32 [32]

Multiply_LoopIncrement13 :
%7 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([29 x i8]* @trstr4, i64 0, i64 0), i32 33), !dbg !79 : 33 [33]
store i32 %"+=", i32* %subscript16, align 4, !dbg !80 : 34 [34]
%postdec52 = add i32 %Inner.0, -1, !dbg !81 : 35 [35]
tail call void @llvm.dbg.value(metadata !{i32 %postdec52}, i64 0, metadata !21), !dbg !82 : 36 [36]
%relopcmp54 = icmp sgt i32 %postdec52, -1, !dbg !83 : 37 [37]
br i1 %relopcmp54, label %LoopBody25.preheader, label %LoopIncrement, !dbg !84, !xta.loop_labels !85, !entra !76 : 38 [38]

Multiply_LoopIncrement :
%8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([27 x i8]* @trstr5, i64 0, i64 0), i32 39), !dbg !86 : 39 [39]
%postdec58 = add i32 %Outer.0, -1, !dbg !87 : 40 [40]
tail call void @llvm.dbg.value(metadata !{i32 %postdec58}, i64 0, metadata !19), !dbg !88 : 41 [41]
%relopcmp60 = icmp sgt i32 %postdec58, -1, !dbg !89 : 42 [42]
br i1 %relopcmp60, label %LoopBody11.preheader, label %return, !dbg !90, !xta.loop_labels !45, !entra !91 : 43 [43]

Multiply_return :
%9 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([20 x i8]* @trstr6, i64 0, i64 0), i32 44), !dbg !94 : 44 [44]
ret void, !dbg !95 : 45 [45]
```

Figure 7.5: The LLVM IR CFG of the function in Listing 7.1 with the Basic Block (BB) instrumentation instructions inserted by the LLVM instrumentation pass. Calls to the instrumentation function are indicated by enclosing them in a box for each LLVM IR BB.

LLVM IR assembly language. These *print* instructions need to be translated by the compiler back-end to target-specific tracing functions. In the Xcore case, *printf* real-time debugging is supported by the `xTAG` debug adapter [144], which emits the data to the host machine via buffering, with negligible impact on program execution time. The only requirement for other embedded devices to benefit from this profiler is the implementation of a target-specific instrumentation function to emit profiling data to the host machine, if not already in place. On most targets, this can be implemented using I/O ports, with very low impact on the program's execution time. Although any execution time overhead will not affect the accuracy of the retrieved estimation, keeping this overhead low makes the profiling-based energy consumption estimation more favorable than an ISS-based estimation due to the significant estimation performance benefits of profiling over ISS. This will be thoroughly investigated in Section 8.3.1.

2. **Post Processing**: The program is then executed and the BBs execution trace is collected. Based on the unique IDs emitted, the post-processing stage can associate execution counts with each BB of the optimized LLVM IR code.

Energy Consumption Estimation Phase:

1. **BB Energy Characterization**: A clean copy of the optimized LLVM IR code, with no instrumentation instructions, is energy-characterized using the BB energy characterization mapping techniques introduced in Chapter 6. This ensures that no energy overheads appear in the energy consumption estimations due to instrumentation code.
2. **Energy Estimation**: This stage takes as input the BB energy-characterized LLVM IR and the BBs execution counts collected from the Profiling phase, and produces the program's total energy consumption estimation. Having the LLVM IR BBs' execution counts and the LLVM IR instruction energy costings, a fine-grained software energy characterization can be achieved, where energy consumption can attributed to specific programming constructs, e.g. BBs, loops and functions.

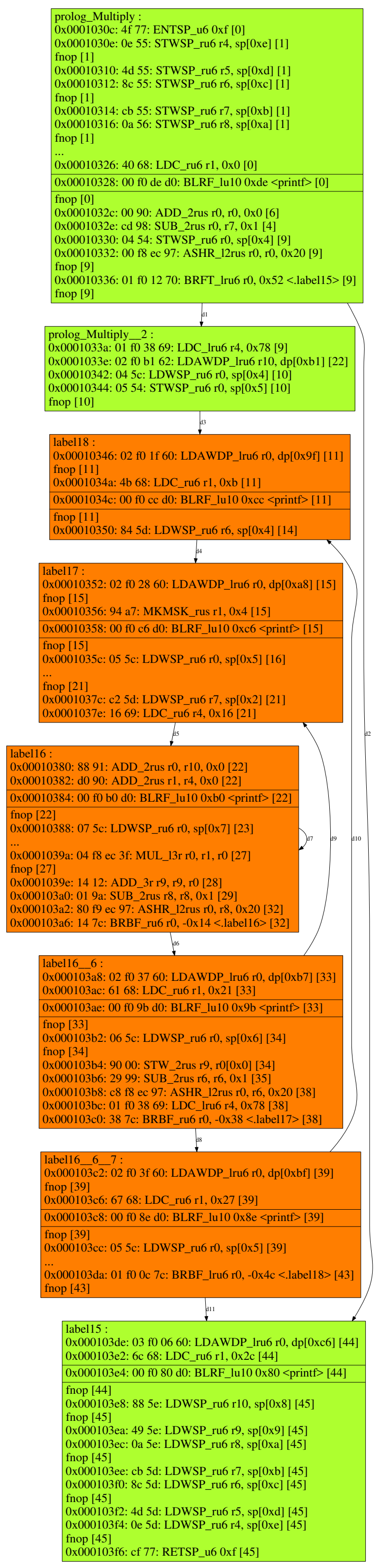
Figure 7.5 demonstrates the LLVM IR CFG for the program in Listing 7.1 after the LLVM instrumentation pass is executed as part of the `Profiling` phase. This CFG is similar to the one in Figure 7.3. The only difference is that each BB in Figure 7.5 includes an extra LLVM IR command, inserted by the LLVM instrumentation pass. These extra commands are indicated in each BB of the figure by enclosing them within a box, and they are always placed after any *Phi-nodes* instructions. This conforms with the LLVM IR optimizer and code emitter expectation that all *Phi-nodes* instructions are always placed at the start of their LLVM IR BB. The LLVM IR CFG in Figure 7.5 is only used for the generation of the ISA instrumented code. For the `Energy Consumption Estimation` phase of the profiler, the BB energy characterization will be performed on a clean version of the optimized LLVM IR. Thus, for this chapter's running example, the clean version LLVM IR used is the one shown by the CFG in Figure 7.3.

Figure 7.6 demonstrates the ISA CFG, created from the LLVM IR instrumented code in Figure 7.5 by the LLVM lowering phase. Similarly to the Figure 7.5, the introduced profiling instructions are indicated by placing them into a box in each LLVM IR BB. These instrumentation instructions represent calls to the *printf* target specific function, which provides real-time debugging by the use of the `xTAG` debug adapter. The second CFG BB, starting from the top, with the name `prolog_Multiply__2` is the only BB with no instrumentation instruction. This BB does not correspond to any LLVM IR BB in its origin LLVM IR CFG Figure 7.5, and therefore to no LLVM IR BB into the clean from profiling code CFG in Figure 7.3. It is one of the cases that the compiler back-end was unable to eliminate all *Phi-nodes* instructions. The mapping process, described in Section 6.5.1, will firstly associate the costs of the instructions of such ISA BBs to their corresponding *Phi-nodes* instructions at the LLVM IR level. Then the BB tuning stage of the mapping, detailed in Section 6.5.3, will hoist the cost of these instructions out of their *Phi-nodes*' BBs and to their appropriate predecessors BB.

For this chapter's running example, the BB mapping tuning stage will cause the cost of the BB untracked by the profiler to be always taken into consideration; even in the case that the BB is not executed. This will usually cause the energy to be slightly overestimated. In the case of the IPET bound analysis, this overestimation is acceptable. Although in the case of the profiling-based estimation, which targets the actual energy consumption case, the introduced overestimation is negligible due to the loops energy cost dominance, it is still not desirable. To address this, the mapping phase needs to be disabled for such untracked by the profiler ISA CFG BBs. To automate this the profiler provides to the mapping phase a list of the untracked by the profiler BBs. Then using the profiling traces, the execution of such BBs can be inferred by examining the execution of their outgoing BBs. For example in the CFG of Figure 7.6 the execution count for `prolog_Multiply__2` is equal to the execution count of its unique outgoing BB, `label18`, which is tracked by the profiling. Therefore, the overall energy consumption cost of the `prolog_Multiply__2` ISA BB can be estimated and added to the final estimation retrieved at the LLVM IR level.

7.3 First steps towards analysis of multi-threaded programs

This thesis presents the first steps towards energy consumption analysis of multi-threaded programs. Two concurrency patterns are considered: replicated threads with no inter-thread communication, working on different sets of data (task farms), and pipelines of communicating threads. For both cases, evenly distributed, balanced workloads are considered. In the former case, an example use is simultaneously processing multiple independent data sets. In the latter case, pipelining enables parallelism to be used to improve performance when processing a single



```
prolog_Multiply :
0x0001030c: 4f 77: ENTSP_u6 0xf [0]
0x0001030e: 0e 55: STWSP_ru6 r4, sp[0xe] [1]
fnop [1]
0x00010310: 4d 55: STWSP_ru6 r5, sp[0xd] [1]
0x00010312: 8c 55: STWSP_ru6 r6, sp[0xc] [1]
fnop [1]
0x00010314: cb 55: STWSP_ru6 r7, sp[0xb] [1]
0x00010316: 0a 56: STWSP_ru6 r8, sp[0xa] [1]
fnop [1]
...
0x00010326: 40 68: LDC_ru6 r1, 0x0 [0]
0x00010328: 00 f0 de d0: BLRF_lu10 0xde <printf> [0]
fnop [0]
0x0001032c: 00 90: ADD_2rus r0, r0, 0x0 [6]
0x0001032e: cd 98: SUB_2rus r0, r7, 0x1 [4]
0x00010330: 04 54: STWSP_ru6 r0, sp[0x4] [9]
0x00010332: 00 f8 ec 97: ASHR_l2rus r0, r0, 0x20 [9]
fnop [9]
0x00010336: 01 f0 12 70: BRFT_lru6 r0, 0x52 <.label15> [9]
fnop [9]
```

```
prolog_Multiply__2 :
0x0001033a: 01 f0 38 69: LDC_lru6 r4, 0x78 [9]
0x0001033e: 02 f0 b1 62: LDAWDP_lru6 r10, dp[0xb1] [22]
0x00010342: 04 5c: LDWSP_ru6 r0, sp[0x4] [10]
0x00010344: 05 54: STWSP_ru6 r0, sp[0x5] [10]
fnop [10]
```

```
label18 :
0x00010346: 02 f0 1f 60: LDAWDP_lru6 r0, dp[0x9f] [11]
fnop [11]
0x0001034a: 4b 68: LDC_ru6 r1, 0xb [11]
0x0001034c: 00 f0 cc d0: BLRF_lu10 0xcc <printf> [11]
fnop [11]
0x00010350: 84 5d: LDWSP_ru6 r6, sp[0x4] [14]
```

```
label17 :
0x00010352: 02 f0 28 60: LDAWDP_lru6 r0, dp[0xa8] [15]
fnop [15]
0x00010356: 94 a7: MKMSK_rus r1, 0x4 [15]
0x00010358: 00 f0 c6 d0: BLRF_lu10 0xc6 <printf> [15]
fnop [15]
0x0001035c: 05 5c: LDWSP_ru6 r0, sp[0x5] [16]
...
fnop [21]
0x0001037c: c2 5d: LDWSP_ru6 r7, sp[0x2] [21]
0x0001037e: 16 69: LDC_ru6 r4, 0x16 [21]
```

```
label16 :
0x00010380: 88 91: ADD_2rus r0, r10, 0x0 [22]
0x00010382: d0 90: ADD_2rus r1, r4, 0x0 [22]
0x00010384: 00 f0 b0 d0: BLRF_lu10 0xb0 <printf> [22]
fnop [22]
0x00010388: 07 5c: LDWSP_ru6 r0, sp[0x7] [23]
...
0x0001039a: 04 f8 ec 3f: MUL_l3r r0, r1, r0 [27]
fnop [27]
0x0001039e: 14 12: ADD_3r r9, r9, r0 [28]
0x000103a0: 01 9a: SUB_2rus r8, r8, 0x1 [29]
0x000103a2: 80 f9 ec 97: ASHR_l2rus r0, r8, 0x20 [32]
0x000103a6: 14 7c: BRBF_ru6 r0, -0x14 <.label16> [32]
```

```
label16__6 :
0x000103a8: 02 f0 37 60: LDAWDP_lru6 r0, dp[0xb7] [33]
0x000103ac: 61 68: LDC_ru6 r1, 0x21 [33]
0x000103ae: 00 f0 9b d0: BLRF_lu10 0x9b <printf> [33]
fnop [33]
0x000103b2: 06 5c: LDWSP_ru6 r0, sp[0x6] [34]
fnop [34]
0x000103b4: 90 00: STW_2rus r9, r0[0x0] [34]
0x000103b6: 29 99: SUB_2rus r6, r6, 0x1 [35]
0x000103b8: c8 f8 ec 97: ASHR_l2rus r0, r6, 0x20 [38]
0x000103bc: 01 f0 38 69: LDC_lru6 r4, 0x78 [38]
0x000103c0: 38 7c: BRBF_ru6 r0, -0x38 <.label17> [38]
```

```
label16__6__7 :
0x000103c2: 02 f0 3f 60: LDAWDP_lru6 r0, dp[0xbf] [39]
fnop [39]
0x000103c6: 67 68: LDC_ru6 r1, 0x27 [39]
0x000103c8: 00 f0 8e d0: BLRF_lu10 0x8e <printf> [39]
fnop [39]
0x000103cc: 05 5c: LDWSP_ru6 r0, sp[0x5] [39]
...
0x000103da: 01 f0 0c 7c: BRBF_lru6 r0, -0x4c <.label18> [43]
fnop [43]
```

```
label15 :
0x000103de: 03 f0 06 60: LDAWDP_lru6 r0, dp[0xc6] [44]
0x000103e2: 6c 68: LDC_ru6 r1, 0x2c [44]
0x000103e4: 00 f0 80 d0: BLRF_lu10 0x80 <printf> [44]
fnop [44]
0x000103e8: 88 5e: LDWSP_ru6 r10, sp[0x8] [45]
fnop [45]
0x000103ea: 49 5e: LDWSP_ru6 r9, sp[0x9] [45]
0x000103ec: 0a 5e: LDWSP_ru6 r8, sp[0xa] [45]
fnop [45]
0x000103ee: cb 5d: LDWSP_ru6 r7, sp[0xb] [45]
0x000103f0: 8c 5d: LDWSP_ru6 r6, sp[0xc] [45]
fnop [45]
0x000103f2: 4d 5d: LDWSP_ru6 r5, sp[0xd] [45]
0x000103f4: 0e 5d: LDWSP_ru6 r4, sp[0xe] [45]
fnop [45]
0x000103f6: cf 77: RETSP_u6 0xf [45]
```

Figure 7.6: The CFG of the ISA instrumented code for the function in Listing 7.1. Calls to the target specific instrumentation function are indicated by enclosing them in a box for each ISA BB.

data stream.

There is a fundamental difference when statically predicting the case of interest (worst, best, average case) for time and for energy for multi-threaded programs. Generally, for time only the computations that contribute to the path forming the case of interest must be considered. For energy, all computations taking place during the case of interest must be considered. For instance, in an unbalanced task farm (tasks with different amount of workload), the WCET will be equivalent to the longest running thread. To bound energy, the energy consumed by each thread needs to be aggregated. Thus, the static analysis needs to determine the number of active threads at each point in time in order to apply the energy model from Equation (5.1) and characterize the CFG of each thread. Then, IPET can be applied to each thread's CFG, extracting energy consumption bounds. Aggregating these together will give a loose upper bound on the program's energy consumption, meaning that the safety of the bound cannot be guaranteed (as defined in this thesis terminology on page xvii).

In the balanced task farm examples used in this thesis, all task threads are active in parallel for the duration of the test. Thus, the number of active threads is constant, giving a constant N_t, used to determine the pipeline occupancy scaling factor, M, in Equation (5.1). For balanced pipelined programs the continuous, streaming data use case is considered, so that the same constant thread count property holds. In both cases IPET can be performed on each thread's CFG and the results aggregated to retrieve the total energy consumption. The energy model covers the core-local instructions used for thread communication. Although off-core communication uses the same instructions as core-local communication, an additional energy cost needs to be accounted for external-link usage. This cost was determined empirically for the development board used in Section 8.2.3.

For multi-threaded programs with synchronous communications, to retrieve a WCET, IPET can be applied on a global graph, connecting the CFGs of all threads along communication edges [103]. The IPET can treat the communication edges as normal CFG edges, and WCET can be extracted by solving the formulated problem. This will return a single worst case path across the global graph. Bounding energy in this way is not possible, as parallel thread activity over time needs to be considered. Here the task is harder in comparison to programs without communication, as activity can be blocked if the threads' workloads are unbalanced, due to the synchronous blocking message passing. In this case, statically determining the number of active threads at each point in time is a hard challenge.

Similarly to SRA, profiling-based energy estimation does not need to infer the number of active threads at each execution stage, for the balanced task farms and pipelined test cases under investigation. Therefore, retrieving the BB execution counters is sufficient for energy consumption estimations.

The concurrency patterns addressed here represent typical embedded use cases. As demonstrated in Section 8.2.3, for these use cases, SRA can provide sufficiently accurate information for design space exploration. Building on this, more complex programs will be analyzed in future work, such as unique non-communicating threads rather than replicated threads, unbalanced farm and pipeline workloads and other concurrency patterns. Such programs will feature varying numbers of active threads over the course of execution. In these cases the SRA must be extended to perform analysis that extracts all the possible combinations of thread interleaving. To achieve this, more advanced concurrency analysis techniques, such as the ones employed in [114], could be combined with the SRA techniques of this thesis. For these more complex concurrency patterns, the profiling technique introduced in this thesis is anticipated to scale better than SRA.

7.4 Conclusion

The energy consumption analysis techniques introduced in this chapter address the requirements set in Table 3.2 as:

- Both the SRA-based and profiling-based techniques introduced avoid the need for energy measurements and simulation.
- the novel mapping technique introduced in Chapter 6 enables energy consumption estimation at a higher level than the ISA, namely the LLVM IR level. The on-the-fly nature of the mapping technique has significant benefits over more static approaches, such as regression-based energy modeling; it allows the analysis techniques introduced to account for particular compiler's and architecture's dynamic behavior.
- Both of the analysis techniques introduced enable fine-grained energy consumption characterization of software at multiple levels, including the ISA, the LLVM IR, and the source code levels. At the LLVM IR and source code level, the fine-grained characterization is achieved due to the ability of the mapping technique to energy characterize the LLVM IR code.
- Both SRA-based and profiling-based energy consumption estimations are easily portable to new architectures and compilers. To apply the techniques on a new deeply embedded architecture one only needs an ISA energy model. Accurate ISA energy models are relatively easy to be constructed for deeply embedded architectures, as described in Section 3.2.1.2.
- The techniques introduced can currently account for time and energy consumption. Furthermore, the techniques can be easily account for code size estimation, if a code size ISA model is provided. This also applies to the analysis at the LLVM IR level since the mapping technique is a generic technique that can lift any ISA resource model to the LLVM IR level, as described in Section 6.3. This makes the techniques capable of supporting design space exploration decisions. This will be further investigated in Section 8.2.3(reference later to results section).

- The techniques utilize the low-level analysis introduced in Chapter 5, which is built on the top of a multi-threaded energy model. Therefore the techniques can account for the multi-threaded case. This thesis focuses on task farms and pipelines, two commonly used concurrency patterns in embedded computing, as described in Section 7.3.

Chapter 8

Experimental Evaluation

This chapter is dedicated to the experimental evaluation of all the energy transparency techniques introduced in this thesis including the low-level analysis, the high-level analysis, and the two energy consumption estimation methods (SRA-based and profiling-based). The experimental results show several features, influenced by the level of multi-threading, the properties of the benchmarks, and the levels at which estimation analysis and modeling are performed. All of these are examined to determine what influences estimate accuracy at each level, highlighting both strengths and limitations. Furthermore, beyond providing energy consumption bounds, the applicability of SRA-based estimation to a number of alternative use cases is examined, including design space exploration for deeply embedded applications.

8.1 Benchmarks and considerations

Two open source benchmark suites were used to evaluate the energy estimation techniques, SRA and profiling-based, and the mapping technique. The first one, `Mälardalen WCET` benchmark suite [45], is specially designed for WCET analysis. The second, the `BEEBS` benchmark suite [99], is targeted at evaluating the energy consumption of embedded processors. The selected benchmarks were modified to work with the test harness implemented for the evaluation. When necessary, benchmarks using floating point were modified to use integer values, since the Xcore does not support floating point operations. Some of the benchmarks were also extended to be multi-threaded task farms, where the same code runs on two or four threads. Furthermore, a range of industrial benchmarks was selected to demonstrate the value of the introduced estimation techniques to embedded developers.

Deeply embedded processors do not typically have hardware support for division or floating point operations, using software libraries instead. Software implementations are usually far less efficient than their hardware equivalents, both in terms of execution time and energy consumption. The effect of these software implementations on energy consumption should be known by developers, therefore software division and software-emulated floating point arithmetic benchmarks are included. A radix-4 software divider, `Radix4Div` [83], is used. A less efficient version in terms of execution time and energy consumption, `B.Radix4Div`, is added for comparison; it omits an early return when the dividend is greater than 255. As a consequence, CFG paths become more balanced, with less variation between the possible execution paths. The effect of this on the energy consumption bounds and its potential value for cryptographic applications is discussed in Section 8.2.2. For software floating point, single precision `SFloatAdd32bit` and `SFloatSub32bit` operations from [53] are analyzed.

To extend the introduced analysis to multi-threaded communicating programs, two common signal processing tasks written for the Xcore, `FIR` and `Biquad` [142], are analyzed. Both tasks are implemented as pipelines of seven threads. Such programs are the preferred form for Xcore, as spreading the computation across threads allows the voltage and frequency of the core to be lowered, significantly reducing energy consumption while retaining the same performance as the single threaded version.

Table 8.1 summarizes all of the benchmarks' attributes. The meaning of the columns is as follows: *Used* indicates the energy consumption estimation method(s) applied on the benchmark, *Source* indicates where the benchmark was retrieved from, *Description* provides insight into the benchmarks' functionality, *NCSL* is the number of non-commentary source lines of code, T is the number of threads used, and the remaining columns indicate the presence of loops (L), nested loops (N), arrays and/or matrices (A), bitwise operations (B), a complex CFG structure (CP), multiple functions (MF) and thread communications (C).

The benchmarks' source codes, can be found at [38]. Benchmarks were compiled with `xcc` version 12 [143] at optimization level `O2`; the most widely used by software developers.

The shunt resistor current sense circuit and data sampling hardware described in Section 5.1.1 is used for the hardware measurements used for the evaluation. This obtains power dissipation with sub-milliwatt precision and variation of less than 1 %. For the ISS-based estimation, the cycle-accurate simulator used for the energy model evaluation in Chapter 5 and described in Section 5.1.2 is used.

8.2 SRA results

For the evaluation of the SRA estimations together with the mapping technique, 20 benchmarks were used. These cover a broad spectrum of language features and code complexity. A combination of good understanding of the underlying algorithms, profiling information and brute forcing of the benchmarks' functions input space was necessary to identify tests covering the algorithmic

Benchmark	Used[1]	Source	Description	Code structure & characteristics								
				NCSL	T	L	N	A	B	CP	MF	C
base64	Both	Online[2]	Computes the base64 encoding	32	1	✓		✓	✓			
radix4Div	Both	Online[3]	Optimized Software Division for platforms without hardware support	63	1	✓			✓	✓		
B.Radix4Div	Both	Online,[3] modified	Software Division for platforms without hardware support	37	1	✓			✓	✓		
dijkstra	Prof.	Online[4]	Find shortest paths from source to all vertices in the given graph	34	1	✓	✓	✓	✓	✓	✓	
sha256	Prof.	Online[5]	Implementation of the SHA-256 hashing algorithm	168	1	✓		✓	✓		✓	
FIR	Both	XMOS	Finite Impulse Response filter	40	1	✓		✓	✓			
P. FIR_7T	Both			103	7	✓		✓	✓			✓
biquad	Both	XMOS	Signal equaliser using biquad filtering	49	1			✓	✓			
biquad_2T	SRA			55	2			✓	✓		✓	
biquad_4T	SRA			57	4			✓	✓		✓	
P. Biquad_7T	Both			94	7			✓	✓		✓	✓
SFloatAdd32bit	Prof.	SoftFloat	Single point F. addition for platforms without hardware support	359	1				✓	✓	✓	
SFloatSub32bit	Prof.	SoftFloat	Single point F. subtraction for platforms without hardware support	371	1				✓	✓	✓	
matMul	Both	MDH WCET	Matrix multiplication of two square matrices	15	1	✓	✓	✓				
matMul_2T	Both			25	2	✓	✓	✓				
matMul_4T	Both			27	4	✓	✓	✓				
jpegdct	Both	MDH WCET	Performs a JPEG discrete cosine transform	35	1	✓	✓	✓	✓			
jpegdct_2T	Both			43	2	✓	✓	✓	✓		✓	
jpegdct_4T	Both			45	4	✓	✓	✓	✓		✓	
ndes	Prof.	MDH WCET	Complex embedded code	253	1	✓		✓	✓		✓	
qsort	Prof.	MDH WCET	Non-recursive version of quick sort algorithm	101	1	✓	✓	✓			✓	
bs	Prof.	MDH WCET	Binary search	90	1	✓		✓	✓			
minver	Prof.	MDH WCET	Inversion of Matrix	155	1	✓	✓	✓	✓		✓	
ludcmp	Prof.	MDH WCET	LU decomposition algorithm	89	1	✓	✓	✓			✓	
nsichneu	Prof.	MDH WCET	Simulate an extended Petri Net	1469	1	✓						
cnt	Both	MDH WCET	Counts non-negative numbers in a matrix	29	1	✓	✓	✓				
st	Both	MDH WCET	Statistics program	85	1			✓	✓		✓	
mac	Both	MDH WCET	Dot product of two vectors and sum of squares	11	1	✓		✓				
crc	Prof.	BEEBS	Cyclic redundancy check computation on 40 bytes of data	18	1	✓		✓	✓			
recursion	Prof.	BEEBS	A simple example of recursive code	50	1						✓	
bsort100	Both	BEEBS	Bubblesort program	66	1	✓	✓		✓			
levenshtein	Both	BEEBS	Measures the difference between two strings	26	1	✓	✓	✓	✓		✓	

NCSL: Number of non-comment source-lines **T**: Number of threads **L**: Contains loops **N**: Contains nested loops **A**: Uses arrays/matrices **B**: Uses bitwise operations **CP**: Has complex CFG structure **MF**: Consist of Multiple functions **C**: Contains thread communications.

[1] Indicates the energy consumption estimation method(s) applied on the benchmark: **SRA**: Used only with Bound Static Analysis, **Prof.** Used only with Profiling, **Both**: Used with both SRA and profiling.

[2] Retrieved from `http://stackoverflow.com/questions/342409`, Nov 2014.

[3] Retrieved from [83], Nov 2014.

[4] Retrieved from `http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/`, Nov 2014. [5] Retrieved from `https://github.com/B-Con/crypto-algorithms`, Jul 2016.

Table 8.1: Description and attributes of benchmarks.

worst case execution path for each benchmark with certainty.

Figure 8.1 presents the error margin of using the same ISA energy model with three energy estimation techniques compared to hardware energy measurements for the benchmarks used. *Simulation* produces energy estimations based on instruction traces from ISS, *ISA SRA* uses the low-level analysis introduced in Chapter 5 for static analysis at the ISA level and *LLVM IR SRA*

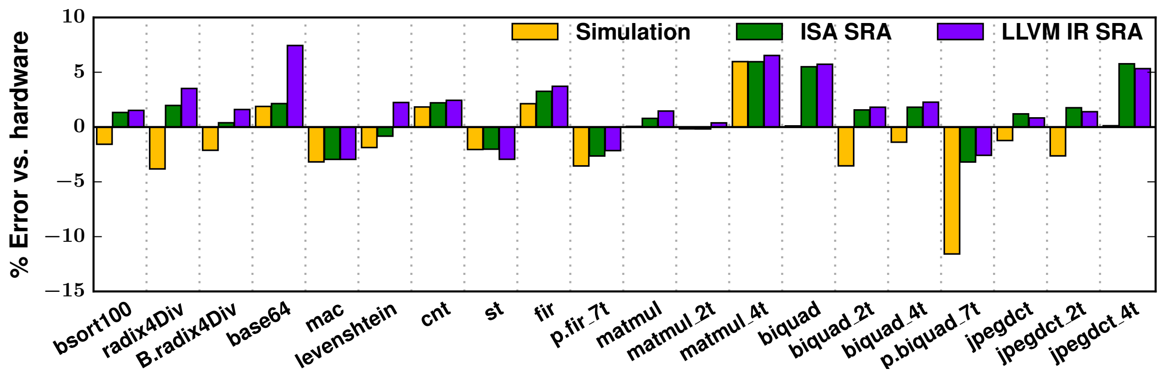


Figure 8.1: SRA and ISS against hardware measurements (worst case program inputs).

uses the high-level analysis introduced in Chapter 6 to apply the model and analysis at LLVM IR level.

In the case of LLVM IR SRA, the accuracy of the prediction is heavily dependent on the accuracy of the mapping techniques presented in Section 6.5. As shown in Figure 8.1, for all benchmarks the LLVM IR SRA results are within one percentage point error of ISA SRA results, except for the `Base64` and `levenshtein` benchmark with a further 4.3 and 2.0 percentage points error, respectively. In these cases the CFGs of the two levels were significantly different due to new BBs introduced from branches in the ISA level CFG. As discussed in Section 6.5.4, substantial differences between the two CFGs can reduce the effectiveness of the current tuning implementation.

Generally, for all results, a proportion of error is present in both forms of static analysis as well as simulation-based energy estimation. The error in the ISS-based estimation is a baseline for the best achievable accuracy in static analysis, as the ISS produces the most accurate execution statistics. For all the benchmarks, the ISA SRA results are over-approximating the trace-based energy estimations. This applies also to the LLVM IR SRA results with exception of the `st` benchmark. This over-approximation is a product of the bound analysis used, which is trying to select the most energy-costly CFG path based on the provided cost model.

The majority of SRA energy estimations are tightly overestimating the actual energy consumption measured on the hardware, up to 5.7% for ISA SRA and up to 7.4% for LLVM IR SRA. There are a number of cases for which the retrieved estimations underestimate the actual energy consumption (`mac`, `levenshtein`, `st`, `p.fir_7t`, `p.biquad_7t`). IPET is intended to provide bounds based on a given cost model. Therefore, the IPET-based SRA introduced in this thesis tries to select the worst case execution paths in terms of the energy consumption. However, energy consumption is data sensitive, i.e. the energy cost of executing an instruction varies, depending on (the circuit switching activity caused by) the operands used. Therefore, the most prominent cause for the observed underestimation is the use of a data-insensitive energy model and analysis. All the possible sources of errors in the retrieved estimates and their role to the observed underestimation will be examined in detail in Section 8.4.

The SRA estimations seen in Figure 8.1 represent a loose upper bound on the benchmarks' energy consumption. These bounds cannot be considered safe for use in mission-critical applications. However, the experimental results show a low level of SRA underestimation, less than 4%, and therefore the SRA can still provide valuable guidance to the application programmer, e.g. to compare coding styles or algorithms.

8.2.1 Parametric resource usage equations

Figure 8.2 compares energy estimates from SRA-based estimation at ISA and LLVM IR levels and ISS-based estimation at ISA level to hardware measurements for a range of parameters in six parametric benchmarks. In all six cases, the energy consumption follows the complexity of the algorithm implemented for each benchmark. This also indicates that there is a strong correlation between time and energy on the Xcore platform.

Regression analysis was applied to the ISA level static analysis results of the benchmarks `Base64`, `Cnt`, `Mac` and `MatMult 1,2,4`. The resultant upper bound equations are shown in Table 8.2. These are expressed in terms of a function over the number of loop iterations. Typically, loop iteration counts are determined by the input parameters of the functions in these benchmarks. The second column shows the retrieved equations which return the energy consumption predictions in nano-Joules (nJ) as a function over x, as defined in the third column.

The equations retrieved by regression analysis are analogous to those retrieved in [44], [69] and [70] by automatic complexity analysis Section 3.2.2.2. The regression analysis approach can handle the same complexity classes like the ones in the papers mentioned above. The limiting factor for the automatic complexity analysis is the solver's ability to provide a closed-form solution for the cost relations of a program. For the regression analysis approach, the critical step is to identify the appropriate curve fit model, e.g. linear regression, polynomial regression *etc.*, to be applied in each case. Manual inspection of the plots, similar to the ones given in Figure 8.2, and knowledge about the complexity of the underlying benchmarks' algorithms can help the user decide which curve fit model to apply. A potential approach to automating the process of retrieving cost functions with regression analysis is to use goodness-of-fit tests, [88], to identify the probability density function to which the retrieved estimation data belongs. This forms future work.

Benchmark	Regression Analysis (nJ)	x
Base64	$f(x) = 19x + 94.2$	string length
Mac	$f(x) = 15x + 21.1$	length of two vectors
Cnt	$f(x) = 19.9x^2 + 5.7x + 34.6$	matrix size
MatMul	$f(x) = 12.2x^3 + 17.5x^2 + 4.7x + 33$	size of square matrices
MatMul_2T	$f(x) = 19.3x^3 + 21.4x^2 + 5.9x + 96.8$	size of square matrices
MatMul_4T	$f(x) = 22.7x^3 + 25x^2 + 6.5x + 157.7$	size of square matrices

Table 8.2: Benchmarks with parametric energy consumption.

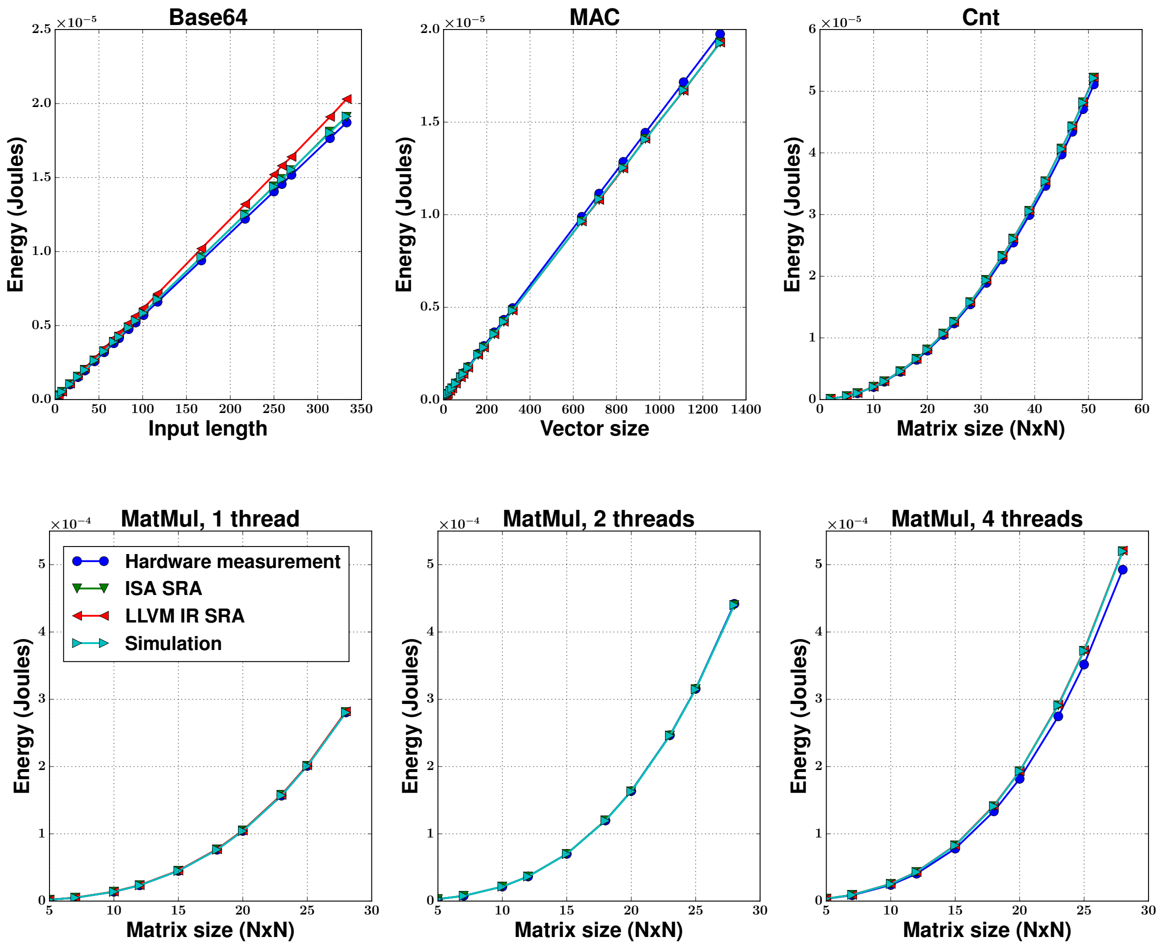


Figure 8.2: Benchmark results, including hardware measurement, SRA-based estimation at the ISA and at the LLVM IR levels and ISS-based estimation.

As seen in Section 8.2, the SRA-based retrieved energy consumption upper bounds may underestimate the actual worst case due to the data-insensitive energy model and analysis used. Therefore, the resource usage equations retrieved by the regression analysis, shown in Table 8.2, can not be considered safe to be used for mission-critical applications. This also applies to the energy consumption equations retrieved in [44, 69, 70] using automatic complexity analysis.

Parametric resource usage equations can still be valuable for a programmer or user to predict a loose upper bound of the energy consumption with specific parameter values. Such estimates can be valuable for comparing coding styles or algorithms and making energy-aware decisions. This is further investigated in Section 8.2.3. Moreover, embedding energy consumption equations into an operating system can enable energy-aware decisions for either scheduling tasks, or checking if the remaining energy budget is likely adequate to complete a task.

The effect of data on the upper bounds retrieved by the SRA-based estimation is further investigated in Section 8.4.

Beyond the use of SRA predictions for bounding the energy consumption, a number of other use cases were investigated and are detailed next.

8.2.2 SRA alternative use cases

The modified `B.Radix4Div` benchmark avoids an early return when the dividend is greater than 255. Omitting this optimization is less efficient, but balances the CFG paths. The effect of this modification can be seen in Figure 8.3. The ISA level energy consumption lower and upper bounds (the best and worst case retrieved by IPET) are shown. In the optimized version, `Radix4Div`, the energy consumption across different test cases varies significantly, creating a large range between the upper and lower energy consumption bounds. Conversely, the unoptimized version, `B.Radix4Div`, shows a lower variation, thus narrowing the margin between the upper and lower bounds, but has a higher average energy consumption.

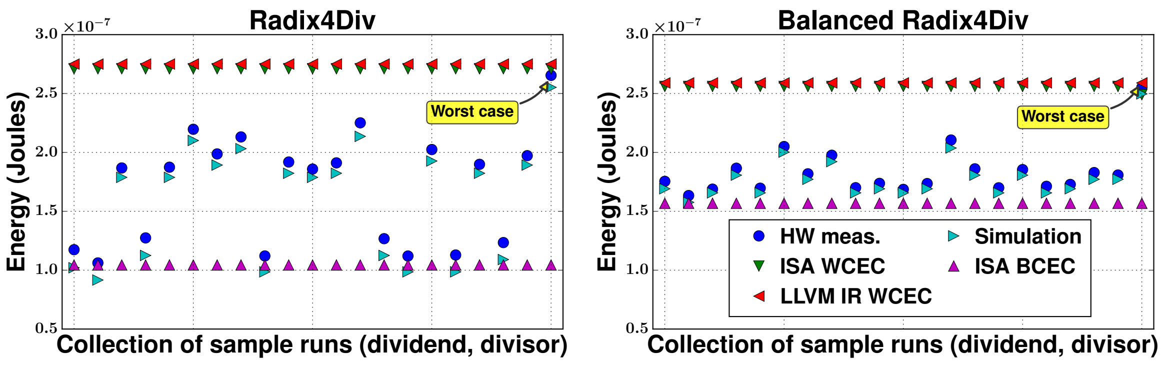


Figure 8.3: The energy consumption bounds of the optimized and unoptimized version of `Radix4Div` benchmark, captured by SRA.

Knowledge of such energy consumption behavior can be of value for applications like cryptography, where the power profile of systems can be monitored to reveal sensitive information in side channel attacks [60]. In these situations, SRA can help code developers to design code with low energy consumption variation, so that any potential leak of information that could be obtained from power monitoring can be obfuscated.

8.2.3 Design space exploration using SRA

This section examines how ISA SRA can be used as an alternative to measurements or simulation for design space exploration of task farms and pipelined programs. For the applications under consideration, the worst-case execution path is actually the dominant execution path. Having a number of available threads, a number of cores and the ability to apply voltage and frequency scaling, provides a wide range of configuration options in the design phase, with multiple optimization targets. This can include optimizing for quality of service, time and energy, or a combination of all three. For this experiment, the XMOS XP-SKC-A16 [145] development board is used, which consists of two interconnected Xcores, with 16 threads available in total and supports voltage and frequency scaling.

The left-hand side of Figure 8.4 shows different configurations for the same program; on the right-hand side these different versions are compared in regards of savings, if any, achieved for time and energy. For energy, both the hardware measurement-based, and the SRA-predicted percentages of savings are shown in order to assess the ability of SRA to support sound energy-aware decisions. The analysis can infer also the time figures but due to the time-deterministic nature of the architecture the predictions closely match the actual values and therefore they are omitted from the graph.

First, SRA was applied to replicated non-communicating threads. The user can make energy-aware decisions on the number of threads to use, with respect to time and energy estimations retrieved by the introduced analysis. As an example, consider four independent matrix multiplications on four pairs of equally sized matrices (30×30). For all three versions the number of cores, the core voltage and frequency are kept constant, but the number of threads is altered and the computation is split across them. The single thread version, `M1`, has an execution time of $4\times$ the time needed to execute one matrix multiplication. However, the two-thread version, `M2`, halves the execution time and, as indicated by both SRA and actual measurements, decreases the energy by 21%, compared to `M1`. The four-thread version, `M3`, halves the execution time again and, as predicted by SRA, the energy consumption is decreased by 40% compared to the two-thread version, `M2`. For the same case, `M3 vs M2`, the actual energy savings are 44%. Although there is a different estimation error between different numbers of active threads, the error range of 5% is small enough to allow comparisons between these different versions, by just using the energy estimations from SRA. The measured and the predicted values are strongly correlated and confirm that using more threads increases the power dissipation, but the reduction in execution time saves energy on the platform under investigation.

Benchmark	ID	Configuration			
		C	T	V	F
MatMult (4 pairs of 30x30 matrices)	M1	1	1	1	450
	M2	1	2	1	450
	M3	1	4	1	450
Biquad Filter	B1	1	1	1	450
	B2	1	7	0.75	150
	B3	2	7	0.7	75

C: Number of cores used
T: Number of threads used
V: Core(s) Voltage in Volts
F: Core(s) Frequency in MHz

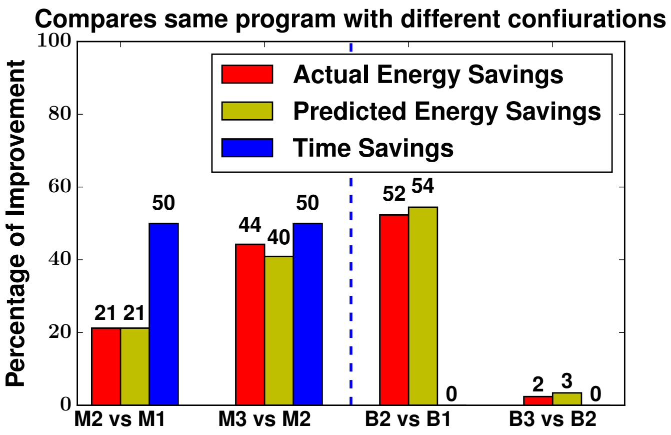


Figure 8.4: Design space exploration enabled by SRA.

Next, SRA was applied to streaming pipelines of communicating threads. There is a choice in how to spread the computation across threads to maximize throughput to either minimize execution time or lower the necessary device operating frequency and voltage while maintaining performance. The new SRA can take advantage of the fact that the energy model used is parametric to voltage and frequency, to statically identify the most energy-efficient configuration of the same program, among a number of different options that deliver the same required performance. This is demonstrated on the `Biquad` benchmark.

The `Biquad` benchmark implements an equalizer; it takes a signal and attenuates or amplifies different frequency bands. The `Biquad` equaliser uses a cascade of biquad filters. Each biquad filter attenuates or amplifies one specific frequency range; the signal is sequentially passed through all filters, eventually attenuating or amplifying all bands. In a standard equalizer setting, a seven-bank filter is used, of which the low four banks are set to amplify, and the top three banks are set to attenuate. These seven banks are implemented in a seven-stage pipeline.

Figure 8.4, on the left-hand side shows the three different versions explored for the `Biquad` filter. `B1` is the sequential version running at 450 MHz and 1V. The single-core parallel version, `B2`, uses 7 threads, one for each bank, and runs at 150 MHz and 0.75V. The multi-core parallel version, `B2`,again uses 7 threads but places 4 of these on the first core and 3 on the second core. It therefore allows for halving the core frequency to 75 MHz and a further reduction of the core voltage to 0.7V. All versions, `B1`, `B2` and `B3` deliver the same filter performance, therefore the percentage of time savings on the right hand side graph of Figure 8.4 is zero. In contrast, we can see a predicted energy saving on `B2 vs B1` of 54% and on `B3 vs B2` of 3%. The actual energy savings are 52% and 2%, respectively. The small error of the retrieved energy consumption estimations enables energy-aware decisions just by the use of SRA. In this case, version `B2` is the best solution since while maintaining the same performance, it halves the energy consumption. In comparison, the small additional energy savings realized by `B3` may not warrant the use of a second core. These energy-aware decisions can not be achieved by only examining the execution time of each one of the three configurations.

Both of the above examples demonstrate that the introduced SRA is sufficient to provide design space exploration guidance considering both time and energy, without the need of any simulation or hardware measurements. This has significant benefits since SRA is faster than simulation and less costly to deploy than hardware measurements.

8.3 Profiling-based analysis results

To evaluate the profiling-based energy estimation technique together with the mapping technique, 30 benchmarks were used. For these benchmarks, Figure 8.5 presents the error margin of using the same ISA energy model with the ISS-based estimations and the profiling-based estimations, compared to hardware energy measurements, respectively. The average absolute error obtained for the profiling-based estimations is 3.1%, and 2.7% for the ISS-based estimations. The ISS-based estimation is more reliable than profiling. This is because cycle-accurate ISS allows for a very precise energy consumption estimation for each particular program execution as the program is executed in simulation with a specific set of input parameters. The exact sequence of instructions can be recorded during simulation and then used to estimate energy consumption. In contrast, profiling is less precise as it operates further away from the hardware, at the LLVM IR level. Profiling estimation precision heavily depends on the quality of the mapping techniques introduced in Section 6.5, and on the quality of the profiling techniques used to collect execution counts of LLVM IR BBs, as described in Section 7.2. However, the profiling results demonstrate a high accuracy with an average error deviation of 1.8% from the ISS. This low error deviation together with the performance benefits over ISS-based estimation, investigated in the next section, makes the profiling technique a significantly better solution for energy-aware software development than the ISS.

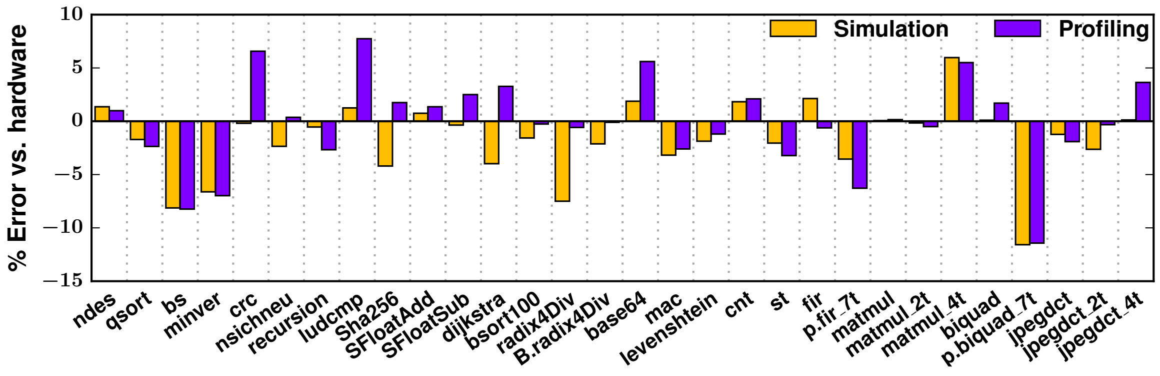


Figure 8.5: Profiling- and ISS-based energy estimations against hardware measurements.

8.3.1 Profiling-based estimation performance

ISS execution is often several orders of magnitude slower than hardware. The performance of an ISS is governed by the complexity of the program's underlying algorithms. In contrast, the performance of the introduced profiling is mainly governed by the program size, since retrieving the BB execution counts incurs a negligible execution time overhead when running on the actual platform, as discussed in Section 7.2. The bigger the program the more time will be needed for Stage 1, `LLVM Instrumentation Pass` in the `Profiling` phase and Stage 1, `BB Energy Characterization`, in the `Energy Consumption Estimation` phase, shown in Figure 7.4.

In the case of benchmarks with low algorithmic complexity and function inputs that will trigger a very short simulation time, no significant performance gains will be observed from using profiling over ISS estimation. For example, for the `SFloatAdd` benchmark, with a $O(1)$ complexity, the average profiling estimation performance observed has a negligible gain over the ISS estimation. On the other hand, increasing the complexity of the benchmarks' underlying algorithms and using parameters which trigger longer simulation time, results in the profiler significantly outperforming the simulation. For example, when using 30×30 size matrices for the matrix multiplication benchmark, the profiling estimation achieves a 381 times speedup over the ISS estimation. The benchmark's small size allowed for a fast profiling estimation, but the ISS estimation performance follows the $O(n^3)$ algorithmic complexity of the benchmark.

A further speedup can be achieved for the profiling-based method, when testing the same program with different inputs. If the optimized LLVM IR code obtained is the same across the different inputs, then there is no need to repeat the LLVM IR energy characterization stage. In such cases using the profiling energy estimation has clear performance advantages compared to the ISS-based energy estimation.

8.4 Discussion

8.4.1 SRA-based energy consumption bounding

This thesis investigates the option of using an energy model that was characterized using pseudo-randomly generated data as the basis of the low-level analysis needed by the IPET. The results in Section 8.2 showed that the bounds retrieved by this configuration, energy model used and IPET-based SRA, in some cases underestimate the actual energy consumption. Thus, this approach provides loose upper bounds that are not safe to be used with energy-critical applications. This section investigates further the source of the observed underestimation.

Identifying the exact source of underestimation is a hard task. This is due to the various factors that can affect the estimation's accuracy. Identifying the contribution of each one of these factors to the overall estimation error is impractical, but at least, the effect of each factor on energy consumption estimations can be examined.

Although the loss of accuracy when moving the SRA from the ISA level to the LLVM IR level is only in the range of one percentage point, the investigation will focus on the most accurate ISA-level SRA, to exclude from the investigation any inaccuracies potentially introduced by the high-level analysis introduced in Chapter 6. The maximum underestimation observed across all the 20 benchmarks used for the ISA level SRA-based estimation is in the range of 4% compared to the hardware measurements. In all cases, all the static analysis estimations overestimate the ISS-based estimations. This means that the IPET-based SRA implemented in this thesis is not the source of the observed underestimation, as it successfully provides the worst case path based on the given energy model. Therefore, the investigation is now shifted to the ISS-based estimation.

In the case of the ISS-based estimation, considering that the ISS is a cycle accurate simulator the ISS statistics used for the energy consumption estimation can be excluded from the possible underestimation sources.

The consistency of the hardware energy measurements needs also to be investigated. The same energy measurement setup used to create the initial version of the energy model in [59] is employed in this thesis for both creating the energy model version used in this thesis, in Chapter 5, and for the experimental evaluation of the introduced energy transparency techniques. This ensures that there is no extra noise introduced to the energy estimates by the energy measurement setup used, that was not taken into account when building the energy model.

Reliable hardware measurements are required for the evaluation of the energy estimation techniques. Thus, the next factor to examine is the consistency across different energy measurements taken for the same executable.

Measurements are subject to errors introduced by environmental factors. In particular, temperature and electrical noise can result in variations of the measured energy consumption for multiple runs of the same test. To measure the effect of these factors on the platform used, the `MatMult 4` thread benchmark was executed 100,000 times. This benchmark was selected because it is particularly power intensive and likely to affect the device temperature the most. The variation observed on the hardware was less than 0.7% which is considered negligible and close to the error margin of the hardware measurement equipment. These factors could have a more significant impact on other platforms. Therefore, it is important to examine them when performing energy consumption estimation.

The test harness also introduces a small error by repeatedly calling the benchmark function within a loop. This is necessary to ensure an adequate number of power samples are taken during the test. However, the loop surrounding the call to the benchmark, together with the function call itself, introduce an overhead. This overhead can be significant, especially when the amount of computation in the loop body is low. To mitigate this overhead, the loop was written to be as efficient as possible. Although, this reduced the overhead, a small amount of it still exists in the hardware measurements which contributes a part of the underestimation observed when comparing the energy consumption estimations to the energy measurements. This underestimation is more severe in the case of smaller benchmarks where the number of executed instructions is not significantly larger than the number of instructions introduced by the test harness. Identifying the exact amount of overestimation introduced by the test harness is not feasible with the measurement hardware in place. Further investigation with more fine-grained measurements from an accurate oscilloscope may help.

The next factor to investigate is the energy model's accuracy. An average absolute error of 3% is observed when comparing the ISS-based energy estimations to the hardware energy measurements for the same test cases. A portion of this error is contributed by the statistical approach used during the energy modeling process to capture the cost of instructions not being able to directly measure their energy cost in a loop as explained in Section 4.2. Furthermore, every data-insensitive energy model is expected to introduce an error in the estimates when used with any energy estimation technique, due to the model's inability to capture the data effect. In the case of the ISA data-insensitive energy model used in this thesis, ISS-based energy analysis will provide a single energy estimation regardless of input data values, comparing this to hardware measurements may give a different error for each input data.

To examine this, one of the most data-sensitive benchmarks included in the evaluation is used, `MatMul 4`. The smallest over-estimation, when compared with the hardware measurements, was 5.96% for both the ISA SRA-based and ISS-based energy estimations. This was obtained for matrices that were initialized with randomly generated data. Since the used energy model was characterized using pseudo-randomly generated data, it provides a good fit to the data used for measurement. Thus this result meets expectations of achieving a lower energy consumption estimation error. The maximum over-estimation found was 25%, by initiating both matrices with zero data which minimized the processor's switching activity. By using the same random data in the two matrices, the over-estimation was between the two previous cases, approximately 15%. This is because the processor switching activity is less than in the case of different random data initialized matrices, and more than the case with the zero initialized matrices. Thus, users must be cautious when using SRA with data-sensitive benchmarks.

These findings demonstrate that the closer the data used to characterize the energy model fits the data of the use case, the more accurate the energy consumption estimations. For example, `MatMul` and `fdct` are heavily used in video processing applications with highly correlated data between frames in the video stream. Therefore, a random-data-constructed energy model for these applications is highly possible to result in an overestimation of the energy consumption by both the SRA and profiling. If a worst-case ISA energy model, one that provides the worst-case energy consumption of each instruction, were to be used with the above scenarios, this would result in an even bigger overestimation compared to a random-data-constructed energy model [132].

Furthermore, the above findings lead to new research questions. How can we construct energy models that are fitter for purpose? Secondly, if a data-sensitive energy model were to be constructed, how would this model be composed to be useful for SRA- and profiling-based estimations? These two research questions motivate future work in this area.

As demonstrated, the choice of a pseudo-random data characterized energy model in this thesis, does not provide safe upper bounds when used with an IPET-like SRA. Considering that end-to-end hardware measurements are also inappropriate for capturing safe energy consumption upper-bounds, the IPET-like SRA introduced in this thesis is preferable because it is easier to be deployed and utilized in the daily software development life-cycle than hardware energy measurements. The loose energy bounds retrieved by the introduced SRA are still useful estimates. They can be used as an indication of whether or not an application is likely to exceed an available energy budget and therefore, to compare the retrieved bounds across different implementations of the same algorithm.

In the case that a safe upper bound is needed, then a worst case energy model has to be used with IPET. The attempts in [55, 132] demonstrated that such an approach could lead to significant over-approximation of the worst case energy consumption, up to 26%. This is because the data causing the maximum energy consumption for one instruction may minimize the cost in a subsequent dependent instruction [100]. An over-approximation will lead to an over-provision in the system's energy budget. Such over-provision might be acceptable for deeply embedded applications running on battery but may not be suitable for applications running on energy harvesters. Furthermore, retrieving a worst-case energy model is not a trivial task as it requires end-to-end measurement for the whole input space of each ISA instruction of a processor. This can be impractical for architectures with large ISAs.

To retrieve safe but tighter upper bounds than those that can be retrieved by the use of the IPET and a worst-case energy model, requires the construction of a data-sensitive energy model and SRA. These will take into account the inter-instruction effects, caused by the operand values used for each instruction, and identify the worst case data input. Recent work demonstrated that finding the data that will trigger the worst case energy consumption is an NP-hard problem and that no practical method can approximate (within reasonable time in general) tight energy consumption upper bounds within any level of confidence [89].

All of the above demonstrates that there is no optimum solution for bounding the energy consumption of an application for a particular platform. Instead, from the existing approaches, the one that is most fit for purpose should be used. If safe upper bounds are required, and an over-provision is acceptable on the system's energy budget, then a worst case energy model can be used with IPET. Moreover, this is dependent on the feasibility of constructing such a worst-case energy model for the architecture under consideration. In the case that loose upper-bounds on the energy consumption are acceptable, then an energy model that was characterized using pseudo-randomly generated data gives sufficiently accurate estimations using the IPET-based energy estimation.

An underestimation in the case of the energy consumption estimates retrieved by the profiling technique with the use of a random-data-constructed energy model is not a critical issue. This is because the profiling-based estimation does not intend to provide safe upper bounds, but instead the actual energy consumption in regards to the program's inputs. Nevertheless, improving the accuracy of the profiling-based energy consumption estimations is still desirable as it will allow making energy-aware decisions with a higher degree of confidence. Therefore, as demonstrated earlier in this section, further research needs to be conducted for energy models that are tailored to specific applications.

8.4.2 LLVM IR estimation accuracy

The SRA-based estimation at the LLVM IR level has a deviation in the range of 1% from the ISA SRA, and the profiling-based estimation has an average error deviation of 1.8% from the ISS. This shows that the tuning phase introduced in Section 6.5.3, mitigates the impact of the mapping limitations described in Section 6.5.4, and yields sufficiently accurate results for the architecture and compiler under consideration. Depending on the architecture and the compiler implementation, the tuning phase heuristics can be further improved to achieve the required level of accuracy.

8.5 Conclusion

The IPET-based SRA, introduced in Section 7.1 at both the ISA and the LLVM IR level, was evaluated through 20 representative benchmarks. The results of the ISA-level SRA for parametric benchmarks were fed to regression analysis to extract energy cost functions that are parametric to the benchmarks data inputs. These functions are analogous to the ones that were retrieved in [44, 69, 70] by the use of automatic complexity analysis. The IPET-based SRA approach overall proved to be more powerful than the automatic complexity approach, since it can handle a larger and more complex set of benchmarks than any previous work could by the use of automatic complexity analysis. This is because the IPET does not depend on solving cost relations, which was the main limiting factor in [44, 69, 70].

At the ISA level, the energy estimations retrieved represent loose upper bounds on the benchmark's energy consumption. The maximum overestimation observed when comparing to the actual energy consumption, measured on the hardware, is 5.7%. This is a low overestimation compared to the ones retrieved in [55, 132] that have an observed overestimation of up to 26%, using a similar IPET-like SRA with a worst case energy model. The evaluation also demonstrated that the retrieved bounds in some cases might underestimate the actual worst-case energy consumption, and hence they are considered loose upper bounds rather than safe upper bounds. The maximum underestimation observed when comparing to energy hardware measurements is less than 4%. Further investigation as to the source of this underestimation demonstrated that there are a number of factors that can contribute to this underestimation. These include the overhead introduced to the measured energy consumption of a benchmark from the test harness and the fact that the underlying energy model of the low-level analysis does not account for the data effect on the energy consumption.

Although the energy consumption bounds retrieved by this thesis SRA can not be considered safe, they are still of great value to software developers. This is because the new approach can overcome the problems of the state of the art approaches for bounding the energy consumption of a program or it can complement the current approaches. On the one hand, considering that end-to-end measurements are inappropriate in most cases for retrieving safe energy consumption bounds, the SRA setup of this thesis is preferable because it is easier to deploy and use during the software development life-cycle, and also it can provide fine-grained energy characterization of software. On the other hand, SRA with a worst-case energy model provides safe energy consumption bounds but can lead to significant over-approximations making the retrieved bounds less useful. Moreover, there is no practical way of constructing such worst-case energy models. In this case, this thesis SRA solution can provide useful loose upper bounds. This can be used to estimate whether a program's energy consumption is likely to exceed the available energy budget, and it can still provide valuable guidance to the application programmer, e.g. to compare coding styles or algorithms.

Furthermore, the results demonstrated that the introduced mapping techniques used in this thesis high-level analysis enable SRA at the LLVM IR level with a small accuracy loss in the range of only 1%, compared to SRA at ISA level. SRA-based energy estimation was also applied to a set of multi-threaded programs for the first time to my knowledge. This is a significant step beyond existing work that examines single-threaded programs. As shown in Section 8.2.3, such an analysis can provide significant guidance for time-energy design space exploration between different numbers of threads and cores.

A target-agnostic profiling technique was developed to estimate the actual energy consumption of a program under specific input parameters. The technique is enabled by the LLVM IR energy characterization utilizing the introduced mapping techniques. By design, it ensures that no energy overhead appears in the estimations due to instrumentation code. The experimental evaluation, using 30 representative benchmarks, shows an average absolute error of 3.1% compared to hardware measurements. The technique is more flexible and significantly more efficient than ISS-based estimation. It can also attribute energy consumption to software components, such as BBs, and functions, in contrast to hardware measurements, which can not do that.

The evaluation of the novel target-agnostic mapping techniques, introduced for the high-level analysis and used by both static analysis and profiling, demonstrated that the techniques could enable powerful insights to the LLVM IR optimizer regarding the energy consumption of a program. The accurate energy estimations retrieved at the LLVM IR level can also enable programmers to investigate how optimizations affect their programs' energy consumption, or even help introduce new energy-specific optimizations in future. This is a significant step beyond existing ISA-level energy estimation techniques.

Overall the experimental evaluation demonstrates that the energy transparency techniques introduced in this thesis significantly extend or complement the state of the art of techniques that can expose the energy consumption of software. More specifically, the experimental evaluation demonstrated that the introduced techniques are capable of delivering sufficiently accurate energy estimations at multiple levels of software abstraction. Furthermore, the introduced techniques eliminate the need for expensive simulation or the difficult to deploy hardware energy measurements and can be used for fine-grained energy characterization of software.

Chapter 9

Conclusion and Future Work

"They always say time changes things, but you actually have to change them yourself."
— Andy Warhol, *[134]*

This chapter restates the thesis underlying research questions and summarizes the research carried out to tackle them. The thesis contributions are examined in the light of how they advance the state of the art of energy transparency techniques for deeply embedded processors. A section is devoted on work in progress of porting the introduced techniques to other deeply embedded processors. Finally, through a critical evaluation of the contributions, a number of opportunities are identified for further improvements and extensions to the proposed techniques. These form the thesis future work.

9.1 Thesis achievements

The energy challenge that IoT and many other deeply embedded applications are facing today motivated the research carried out in this thesis. Although hardware innovation had played a significant role in tackling this challenge, by introducing new more energy efficient hardware, this is only a part of the solution. It is well understood that hardware can not solve the problem without the software playing its role [1, 17, 31, 55, 82, 108, 131]. Software has the ultimate control over the hardware. Therefore, designing energy efficient software can enable an optimized usage of the hardware's power saving features. The key prerequisite for developing energy efficient software is the ability to provide feedback on how the various software-related choices affect the energy consumption of the system.

The thesis main underlying question is concerned with how the new concept of energy transparency can be realized to enable energy-aware software development of deeply embedded programs. Chapter 3 provided a thorough review of existing approaches that aim to provide energy consumption estimations at the software level for deeply embedded systems. The literature review helped to understand the limitations of the existing approaches, which limit the emergence of energy-aware development tools and practices. Therefore, the objectives of this thesis are set to further the state of the art of energy transparency techniques to unlock the potentials of energy-aware software development. The objectives presented in Section 1.1 are listed below together with the relevant contributions made by this thesis to address each one of them.

9.1.1 Both the actual energy consumption and bounds are needed

For energy consumption bound estimations a SRA-based estimation technique is introduced, as described in Section 7.1. The analysis is based on the IPET technique, traditionally used for WCET estimation, and on the low-level analysis introduced for the Xcore processor in Chapter 5. The low-level analysis is based on an energy model that was characterized using pseudo-randomly generated data. The experimental evaluation, detailed in Section 8.2, demonstrated that the technique provides energy consumption loose upper bounds, that can not be used in mission-critical applications. However, as explained in Section 8.4, this thesis approach is a significant contribution to the state of the art of techniques that try to provide energy consumption bound estimations. The new approach complements the existing approaches, which use the IPET with an absolute energy model; a data insensitive energy model that provides a worst case energy value for each ISA instruction. Since it is infeasible to retrieve tight worst-case energy consumption bounds [89], depending on the bounds required by a specific application (e.g. safe bounds, loose bounds), the approach that is more fit for purpose should be used.

By applying regression analysis on the SRA-based energy estimations retrieved in this thesis, parametric resource usage equations are extracted. This approach proved at least as powerful as the automatic complexity approach, as explained in Section 8.2.1. The retrieved equations can be utilized in an operating system for energy-aware scheduling of tasks.

Section 7.2 introduces a target-agnostic profiling technique that can estimate the actual energy consumption of a program under specific input parameters. The estimations are retrieved at the LLVM IR level. This is enabled by the LLVM IR energy characterization of the mapping techniques that are introduced in Chapter 6. The profiling-based estimation collects LLVM IR BB execution counts and applies them on the energy characterized LLVM IR to retrieve energy estimations. By design it ensures that no energy overhead appears in the estimations due to instrumentation code. The experimental evaluation, presented in Section 8.3, shows an average absolute error of 3.1% compared to hardware measurements.

The introduced profiling technique can significantly outperform ISS-based estimation, as explained in Section 8.3.1. The high accuracy and performance of the profiler can enable Feedback-

Directed Optimization (FDO) for energy consumption. LLVM IR energy consumption estimations of each compilation can be taken into consideration iteratively in subsequent compilations. Each new compilation will be able to make more energy-aware decisions.

9.1.2 Energy transparency is needed at multiple levels of software abstraction

This thesis energy estimation techniques can provide energy consumption estimations at three levels of software abstraction; the source code level, the compiler's IR level, and the ISA level. These are the three main software abstraction levels that software developers and development tool-chains interact with. Therefore, there is a great potential of influencing the energy consumption of a system through these abstraction levels.

The low-level analysis, introduced in Chapter 5, enables energy estimations at the ISA level. A novel target-agnostic mapping technique is introduced in Chapter 6 to allow energy characterization of the LLVM tool chain intermediate representation, namely the LLVM IR. The technique enabled accurate energy consumption estimations at the LLVM IR level. This is a significant step beyond existing ISA-level energy estimation techniques. The evaluation of the SRA-based estimation, described in Section 8.2, demonstrated that the mapping technique enables SRA at the LLVM IR level with a small accuracy loss in the range of only 1%, compared to SRA at ISA level. The evaluation of the profiling-based estimation, detailed in Section 8.3 demonstrated an average absolute error of 3.1% compared to hardware measurements. Therefore, the mapping technique can give powerful insights to the LLVM IR optimizer regarding execution time and the energy consumption of a program.

As described in Section 6.6, using the mapping information, energy estimates produced at the LLVM IR level can be lifted to the source code level to characterize the energy consumption of basic programming blocks, such as functions and loops.

9.1.3 Fine-grained energy characterization of software is needed

Both the low-level analysis, introduced in Chapter 5, and the high-level analysis, introduced in Chapter 6, provide energy characterization at instruction-level, more specifically at ISA and LLVM IR instructions respectively. This fine-grained energy characterization enables both the SRA and profiling techniques to attribute their energy consumption estimations to the various basic software components of the program under examination, such as CFG BBs and loops. This is difficult to achieve with hardware energy measurements.

9.1.4 Energy transparency techniques have to be target and programming language agnostic

Both the SRA and profiling techniques introduced are designed to be target agnostic as explained in Sections 7.1 and 7.2 respectively. The only requirement for both techniques is the existence of an ISA energy model. The existence of an energy model is a standard requirement for any method that tries to estimate the energy consumption of a program rather than measure it.

Using an ISA resource model to energy characterize the LLVM IR has significant benefits over a stand alone LLVM IR static energy model. Firstly, the mapping-based approach benefits from the accuracy that ISA models can provide, because the ISA is closer to the hardware than LLVM IR.

Secondly, the dynamic nature of the mapping technique can account for specific architecture behavior, such as the LLVM IR location to which the costs of the FNOPs should be attributed, and compiler specific behavior, such as code transformations. Static IR energy models that are created through statistical approaches are inherently limited in their ability to account for these.

Furthermore, since [127], the seminal approach of constructing ISA energy models, there are several well-defined ISA energy models [19, 51, 109, 110, 122]. These models could now be lifted to the compilers' IR by the introduced mapping technique. Therefore, the approach benefits from a well-understood process by which energy models can be created for deeply embedded systems, where predictability is provided at the ISA level. Execution time models could be lifted into a compiler's IR representation in a similar way.

Energy transparency techniques that are target and programming language agnostic will provide a common framework for energy-aware development. This can be adopted by other deeply embedded platforms. Supporting the same framework for multiple architectures is more practical and cost-effective, rather than maintaining different solutions for each architecture.

9.1.5 The multi-threaded and multi-core case must be considered

As described in Section 7.3, the introduced SRA- and profiling-based energy consumption estimation are also applied to a set of multi-threaded programs; task farms and pipelines. For the SRA, multi-threaded programs were both analyzed at the ISA and at the LLVM IR level. This is a significant step beyond existing work that examines single-thread programs.

9.1.6 Energy transparency techniques must enable design space exploration

When developing multi-threaded applications, energy and time can be interchanged through a variety of different configurations. Such configurations can account for voltage and frequency scaling and number of threads and cores used. Finding the optimal operating point according to the application specifications is a difficult task. Therefore tools are needed that can support design space exploration. The energy transparency techniques developed in this thesis provide the necessary feedback that can enable such design space exploration by both the programmers and the development tool-chains. A demonstration of such resource-aware decisions was presented in Section 8.2.3 for the Biquad filter benchmark. The most energy aware configuration was identified based only on the SRA estimations. In this case, the optimal configuration for energy efficiency can not be found by only examining the execution time of each one of the different configurations.

9.2 Beyond the Xcore processor – work in progress

As discussed in Section 1.1, one of the main objectives of this thesis is to ensure the portability of the proposed techniques to a variety of deeply embedded architectures. The only architecture-specific requirement for transferring the introduced techniques to another architecture is the existence of an ISA energy model. As explained in Section 9.1.4, an energy model is a prerequisite for any energy consumption estimation technique that avoids the need for physical energy measurements. The effort of retrieving an ISA energy model for a deeply embedded processor is outweighed by the fact that the proposed energy-aware framework can enable energy-aware development for a broader spectrum of software developers.

As part of future work, the techniques introduced in this thesis are currently being ported to the ARM Cortex M series processors. A framework that can be used to construct ISA energy models for the Cortex-M series processors was developed. The framework was then used to create an accurate energy model for the Cortex M0 processor [48]. To evaluate the accuracy of the new energy model, energy estimations were retrieved by applying the energy model to instruction statistics collected from an ISS. Figure 9.1 shows the deviation between the hardware measurements and the obtained energy estimations for 47 benchmarks of the BEEBS benchmark suite [99]. The average absolute error is 3%.

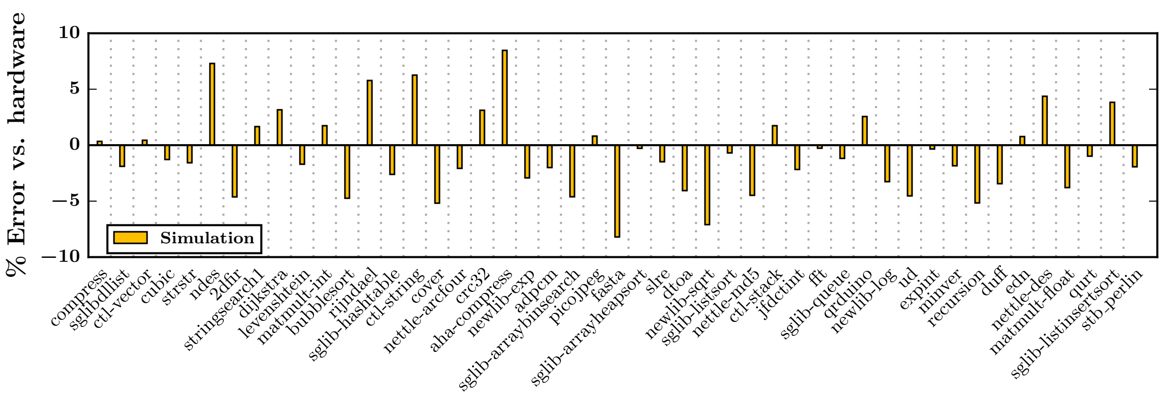


Figure 9.1: Energy model validation for the Cortex M0 processor. The energy model is used with simulation-based execution statistics to obtain energy estimations. The results for each benchmark are compared against hardware measurements (47 benchmarks in total).

The ISA-level SRA-based energy estimation introduced in this thesis has been implemented for the `Cortex M0` processor. This analysis utilizes the new ISA energy model to provide bound estimations for the `Cortex M0` processor. The analysis was evaluated in [54].

Current work in progress ports the profiling technique on the `Cortex M0`. This will allow the evaluation and tuning of the mapping technique for the `Cortex M0`.

9.3 Critical evaluation and future work

The work performed in this thesis sets the ground for a new approach to developing deeply embedded systems; one which will allow the software developer to treat energy consumption as a first class citizen and not as a side effect of improving the execution time of a program. This new approach opens a whole new world of opportunities towards energy-aware software development. The energy transparency enabled by the framework of techniques introduced in this thesis stimulates the research in many directions, which some of them forming future work for this thesis.

Extending the analysis to more complex message passing parallel programs

Future work aims to analyze more complex concurrent programs, such as distinct non-communicating threads, or pipelines of threads with unbalanced workloads. I anticipate that the LLVM IR profiling technique will scale better to these cases than SRA. As the core-count of the analyzed system grows, more in-depth exploration of energy saving techniques such as per-core Dynamic Voltage and Frequency Scaling (DVFS) could be considered both at the model level and SRA or profiling levels.

Utilizing the achieved energy transparency

The focus of this thesis is on enabling energy-transparency during the development of deeply embedded systems. Future work aims to explore the potentials of utilizing the enabled energy transparency to perform resource-aware optimizations. The mapping techniques introduced in this thesis can now be used to lift existing resource models to the compiler's IR level. This allows for a fine-grained characterization at a CFG BB granularity, in regards to energy, execution time and code size. The SRA- and profiling-based estimation methods introduced can also be used to infer bounds and actual usage of the aforementioned resources, based on the program's input parameters. Such resource transparency can enable FDO. By using FDO, optimizations have access to accurate resource information and therefore are less dependent on heuristics. Such optimizations can perform significantly better than heuristic-based optimizations.

As demonstrated in Section 8.2.3, the techniques introduced in this thesis enable for time-energy design space exploration between different numbers of threads and cores. For the examples used, the various configurations (number of threads and cores used, operating voltage and frequency) were manually provided and then SRA was applied to each one of them to identify the most energy saving configuration. Future work can combine the energy estimation techniques introduced in this thesis with auto-parallelization techniques to automate the design space exploration process [68].

This thesis techniques can be extended to support automatic mapping of an application to hardware which supports voltage islands. In [96], a scheme is proposed that enables the compiler to exploit both task and data parallelism and automatically map an application to an embedded multi-processor containing voltage islands. The scheme is based on the workload of each processor, using both hardware parallelism and voltage scaling to reduce energy consumption without increasing the overall execution time. The workload for each processor is calculated based on loop iteration counts and per-instruction cost estimation. Per-core frequency and voltage configurations can be provided to appropriately design Xcore-based systems. Using the introduced profiling technique and LLVM IR energy characterization, accurate workload estimations can be achieved at the LLVM IR level. These could be utilized by the optimization scheme introduced in [96] to enable the LLVM compiler to automatically achieve similar energy consumption savings.

Accounting for non-computation-related energy consumption

This thesis is concerned with the energy-consumption of computation. A typical deeply embedded system may interact with its environment through a number of peripherals. To account for the energy consumption of the whole system, the work performed in this thesis has to be extended to consider the activity of both the processor's I/O operations and the system's peripherals. Such activity is usually controlled by computation. Therefore, one can profile the energy usage of peripherals and I/O operations and include them as part of the energy costs of the computation that triggers them. This will allow the energy estimation techniques introduced to provide system-wide energy estimates. Such an approach is more feasible for systems with predictable behavior.

Accounting for realistic deployment conditions

The ISA energy model introduced in [59] is utilized in both the low-level and the high-level analysis performed in this thesis. For the energy characterization of this model, the platform under investigation was powered by a constant power supply source. A constant power supply is an ideal condition, as there are no significant variations in the power supplied to the processor. This is not an issue in the case that the energy estimations retrieved are used to optimize the energy consumption at development time. In a more realistic scenario, an IoT application could be running on a battery or an energy harvester. To be able to use the energy estimations of this thesis for making real-time decisions, the power profile of the power source for a specific application has to be taken into consideration by both the energy modeling and the energy consumption estimation techniques. This forms future work.

Beyond deeply embedded processors

The techniques introduced in this thesis are focused on deeply embedded architectures that favor predictability over performance. Such architectures are the backbone of IoT applications. Enabling energy-aware software development for such architectures will help to tackle the energy challenge that IoT faces. Some of the introduced techniques, mainly SRA-based estimation, work best with predictable architectures. This is also true for WCET analysis. In fact, static analysis is inherently limited in that sense. Thus, the portability of the profiling-based techniques to more complex processors will be examined in future work.

9.4 Final remarks

Through this thesis, it has been shown that toolchains can be enhanced with new energy transparency techniques to enable energy-aware software development for deeply embedded systems. This is a significant step towards tackling the energy challenge ICT faces in general.

Manufactured hardware has fixed energy saving capabilities. Underutilizing these capabilities and expecting the next generation of more energy efficient hardware to meet an application's energy requirements is not a viable option. Currently, processors' evolution is being threatened by the physical limitations of underlying technologies as well as the dramatically increasing complexity of hardware. Therefore, the responsibility lies on the system engineer to program and configure the selected device in the most energy efficient way for the task at hand. Very few software developers have the required expertise to deliver energy efficient applications. It is our duty to deliver tools and methods that will allow programmers and development toolchains to treat energy consumption as a first-class citizen, by making energy-aware decisions at the software level.

It is time for the software community to take a more active role in delivering more energy efficient systems. Therefore, the quest for energy transparency techniques must continue.

APPENDIX A

Appendix A

An example of output file generated from the proof-of-concept mapping tool in JSON format.

The source code:

```

#include "matmult.h"
#include "samples.h"

int Res[SIZE][SIZE];

#pragma unsafe arrays
void Multiply(matrix A, matrix B, int size)
{
   register int Outer, Inner, Index;
   size--;

   for (Outer = size; Outer >= 0; Outer--)
      for (Inner = size; Inner >= 0; Inner--)
      {
         Res [Outer][Inner] = 0;
         for (Index = size; Index >= 0; Index--)
            Res[Outer][Inner]  += A[Outer][Index] * B[Index][Inner];
      }
}
```

The JSON output file for the above source code, as generated from the proof-of-concept mapping tool:

```
{"program": {
  "functions": [{
    "blocks": [
      {
        "mapping": [
          {
            "ISAMap": [
              "entsp_u6",
              "stwsp_ru6",
              "fnop",
              "stwsp_ru6",
              "stwsp_ru6",
              "fnop",
              "stwsp_ru6",
              "stwsp_ru6",
              "fnop",
              "stwsp_ru6",
              "stwsp_ru6",
              "fnop"
            ],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %0}",
            "LLVMIRInstrEnergy": 1.383389169419272E-8
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{[30 x [30 x i32]]* %1}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %2}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": ["sub_2rus"],
            "LLVMIRIns": "%postdec = add i32 %2",
            "LLVMIRInstrEnergy": 1.1615497919111347E-9
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": ["add_2rus"],
            "LLVMIRIns": "tail call void asm sideeffect \".align 4\\0Anop\"",
            "LLVMIRInstrEnergy": 1.1612550408370088E-9
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "%relopcmp = icmp sgt i32 %postdec",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [
              "ashr_l2rus",
              "brft_ru6",
              "ldc_lru6"
            ],
            "LLVMIRIns": "br i1 %relopcmp",
            "LLVMIRInstrEnergy": 3.500281757001139E-9
          },
          {
            "ISAMap": ["add_2rus"],
          "LLVMIRIns": "%Outer.0 = phi i32 [ %postdec58",
          "LLVMIRInstrEnergy": 1.1612550408370088E-9
        },
      ],
      "name": "Multiply_allocas",
      "lineStarts": 8,
      "lineEnds": 13,
      "energy": 2.0818233324779E-08
    },
    {
      "mapping": [
        {
          "ISAMap": [],
          "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec}",
          "LLVMIRInstrEnergy": 0
        },
        {
          "ISAMap": [],
          "LLVMIRIns": "br label %LoopBody25.preheader",
          "LLVMIRInstrEnergy": 0
        }
      ],
      "name": "Multiply_LoopBody11.preheader",
      "lineStarts": 14,
      "lineEnds": 14,
      "energy": 0
    },
    {
      "mapping": [
        {
          "ISAMap": ["add_2rus"],
          "LLVMIRIns": "%Inner.0 = phi i32 [ %postdec52",
          "LLVMIRInstrEnergy": 1.1612550408370088E-9
        },
        {
          "ISAMap": [
            "mul_l3r",
            "fnop",
            "ldawdp_lru6",
            "fnop",
            "add_3r",
            "ldc_ru6",
            "ldawf_l3r",
            "fnop"
          ],
          "LLVMIRIns": "%subscript16 = getelementptr [30 x [30 x i32]]* @Res",
          "LLVMIRInstrEnergy": 9.31175997771686E-9
        },
        {
          "ISAMap": ["stw_l3r"],
          "LLVMIRIns": "store i32 0",
          "LLVMIRInstrEnergy": 1.2930475445800178E-9
        },
        {
          "ISAMap": [],
          "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec}",
          "LLVMIRInstrEnergy": 0
        },
        {
          "ISAMap": [],
          "LLVMIRIns": "br label %LoopBody25",
          "LLVMIRInstrEnergy": 0
          },
                    {
            "ISAMap": ["add_2rus"],
            "LLVMIRIns": "%Index.0 = phi i32 [ %postdec46",
            "LLVMIRInstrEnergy": 1.1612550408370088E-9
          }
        ],
        "name": "Multiply_LoopBody25.preheader",
        "lineStarts": 16,
        "lineEnds": 18,
        "energy": 1.29E-08
      },
      {
        "mapping": [
          {
            "ISAMap": [],
            "LLVMIRIns": "%\"+=63\" = phi i32 [ %\"+=\"",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": ["add_3r"],
            "LLVMIRIns": "%subscript36 = getelementptr [30 x [30 x i32]]* %0",
            "LLVMIRInstrEnergy": 1.2180335297480592E-9
          },
          {
            "ISAMap": ["ldw_3r"],
            "LLVMIRIns": "%deref37 = load i32* %subscript36",
            "LLVMIRInstrEnergy": 1.2615973407537516E-9
          },
          {
            "ISAMap": [
              "mul_l3r",
              "add_3r"
            ],
            "LLVMIRIns": "%subscript42 = getelementptr [30 x [30 x i32]]* %1",
            "LLVMIRInstrEnergy": 2.465735523613351E-9
          },
          {
            "ISAMap": ["ldw_3r"],
            "LLVMIRIns": "%deref43 = load i32* %subscript42",
            "LLVMIRInstrEnergy": 1.2615973407537516E-9
          },
          {
            "ISAMap": ["mul_l3r"],
            "LLVMIRIns": "%boptmp = mul i32 %deref43",
            "LLVMIRInstrEnergy": 1.2477019938652917E-9
          },
          {
            "ISAMap": ["add_3r"],
            "LLVMIRIns": "%\"+=\" = add i32 %\"+=63\"",
            "LLVMIRInstrEnergy": 1.2180335297480592E-9
          },
          {
            "ISAMap": ["sub_2rus"],
            "LLVMIRIns": "%postdec46 = add i32 %Index.0",
            "LLVMIRInstrEnergy": 1.1615497919111347E-9
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec46}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "%relopcmp48 = icmp sgt i32 %postdec46",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [
              "ashr_l2rus",
              "brbf_ru6"
            ],
            "LLVMIRIns": "br i1 %relopcmp48",
            "LLVMIRInstrEnergy": 2.3414696130664866E-9
          }
        ],
        "lineStarts": 18,
        "name": "Multiply_LoopBody25",
        "lineEnds": 18,
        "energy": 1.22E-08
      },
      {
        "mapping": [
          {
            "ISAMap": [
              "stw_2rus",
              "fnop"
            ],
            "LLVMIRIns": "store i32 %\"+=\"",
            "LLVMIRInstrEnergy": 2.2973862956220446E-9
          },
          {
            "ISAMap": ["sub_2rus"],
            "LLVMIRIns": "%postdec52 = add i32 %Inner.0",
            "LLVMIRInstrEnergy": 1.1615497919111347E-9
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %
               postdec52}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "%relopcmp54 = icmp sgt i32 %postdec52",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [
              "ashr_l2rus",
              "brbf_ru6"
            ],
            "LLVMIRIns": "br i1 %relopcmp54",
            "LLVMIRInstrEnergy": 2.3414696130664866E-9
          }
        ],
        "name": "Multiply_LoopIncrement13",
        "lineStarts": 18,
        "lineEnds": 18,
        "energy": 5.8004057005996655E-9
      },
      {
        "mapping": [
          {
            "ISAMap": ["sub_2rus"],
            "LLVMIRIns": "%postdec58 = add i32 %Outer.0",
            "LLVMIRInstrEnergy": 1.1615497919111347E-9
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "tail call void @llvm.dbg.value(metadata !{i32 %postdec58}",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [],
            "LLVMIRIns": "%relopcmp60 = icmp sgt i32 %postdec58",
            "LLVMIRInstrEnergy": 0
          },
          {
            "ISAMap": [
              "ashr_l2rus",
              "brbf_ru6"
            ],
            "LLVMIRIns": "br i1 %relopcmp60",
            "LLVMIRInstrEnergy": 2.3414696130664866E-9
          }
        ],
        "name": "Multiply_LoopIncrement",
        "lineStarts": 13,
        "lineEnds": 13,
        "energy": 3.5030194049776214E-9
      },
      {
        "mapping": [{
          "ISAMap": [
            "ldwsp_ru6",
            "ldwsp_ru6",
            "fnop",
            "ldwsp_ru6",
            "ldwsp_ru6",
            "fnop",
            "ldwsp_ru6",
            "ldwsp_ru6",
            "fnop",
            "ldwsp_ru6",
            "retsp_u6"
          ],
          "LLVMIRIns": "ret void",
          "LLVMIRInstrEnergy": 1.2719487926090198E-8
        }],
        "energy": 1.2719487926090198E-8
      }
    ],
    "name": "Multiply",
    "lineStarts": 8,
    "lineEnds": 19
  }],
  "name": "matmult.xc",
  "fileLocation": "/home/kakos/kakos/benchmarks/MatMultMultiple/1thread/matmult.xc"
}}
```

Appendix B

SET for Britain 2015 – Poster Competition

The work conducted in this thesis on exposing the software role to the energy consumption of computing systems was shortlisted from hundreds of applicants across the UK to appear in the UK Parliament as part of the Set For Britain 2015 competition. SET for Britain is a poster competition in the House of Commons for early stage or early career researchers. The 50 finalists, including this work, had the opportunity to demonstrate their research to the policy makers of the country, the Members of the Parliament. The poster gained a significant attraction during the exhibition, with some Member's of Parliament following up with more specific questions. The poster presented can be found here [57].

HiPEAC Collaboration Grant 2015

The author of this thesis was granted a fund by the HiPEAC Collaboration grant scheme [48]. The fund covered a three months visit to work with Embedded Pico Systems (MpicoSys) [90], a Polish Small and Medium Enterprises (SME) located in Gdynia, Poland, to investigate the feasibility of transferring energy consumption analysis techniques developed in this thesis to a real use case beyond those employed in the thesis. The key outcome of this collaboration was the creation of a framework that can be used to construct ISA energy models for the `ARM Cortex M` processor series. Such ISA models can be used by the energy transparency techniques introduced in this thesis. The main components of this framework are an accurate hardware energy measurement system, the extension of an instruction set simulator and the creation of a performance profiler. To evaluate the new framework, a new model was created for the `Cortex M0` processor and the ISA-level SRA presented in this thesis was applied to retrieve energy consumption bounds estimations. The short-time collaboration acted as a preliminary exploitation of the industrial need for the energy transparency techniques, and it will form the basis for exploiting the techniques prospects for commercialization.

ENTRA – Whole-Systems Energy Transparency

The work produced in the scope of this thesis significantly contributed to the ENTRA project, a 3-year research project funded by the EU 7th Framework Programme FET that "aimed to promote energy-aware software development using advanced program analysis and modeling of energy consumption in computer systems" [33]. The project started in October 2012 and concluded in December 2015. The project's final review marked it with excellent progress and stated that "all objectives were achieved and even exceeded expectations". The contributions of this thesis to the project are in the area of energy transparency for deeply embedded systems. The majority of the techniques described in this thesis served as parts of the various project's deliverables. A non-exhaustive list of these contributions is provided below:

- *Deliverable 1.1 – Preliminary Report and Demo on Energy-Aware Tools* [74]: Section 3.2, *Multi-level Mapper Tool*, demonstrates through screenshots the use of a preliminary version of the mapping techniques, which are formalized, implemented and evaluated in this thesis. Section 4.1, *Worst Case Energy Consumption Using Implicit Path Enumeration*, briefly explains the IPET-like SRA technique that are introduced, implemented and evaluated in this thesis.

- *Deliverable 1.2 – Energy-Aware Software Development Methods and Tools* [77]: Section 5.2, *Multi-level mapper tool*, gives a brief overview of the mapping techniques introduced in Chapter 6 and evaluated Chapter 8 of this thesis. Section 5.7, *Implicit path enumeration Energy Consumption Static Analysis (ECSA) applications*, presents some of the preliminary results of the SRA alternative use cases and design space exploration for multi-threaded programs further elaborated in this thesis Sections 8.2.2 and 8.2.3.

- *Deliverable 2.3 – High-Level Energy Models* [30]: Section 2, *Energy modeling at the LLVM IR level*, provides a description of the high-level analysis techniques introduced in Chapter 6 of this thesis. It also provides some preliminary results of evaluating the high-level analysis by the use of the SRA introduced in Section 7.1 of this thesis. The deliverable also has as attachment, attachment D2.3.3, one of the articles that are related to this thesis, namely: *On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs* [41].

- *Deliverable 3.1 – A General Framework for Resource Consumption Analysis and Verification* [73]: A significant part of this deliverable is on how the high-level analysis, introduced in this thesis, is used by the automatic complexity methods developed in the ENTRA project to enable energy consumption estimation at the LLVM IR level. Section 1, discuss the trade-offs of applying energy consumption analysis at various software abstraction levels, as also done in this thesis Section 6.1. Section 4 and 5, are elaborating more on the benefits of applying energy analysis at the LLVM level and how the mapping-techniques work to achieve the high-level analysis and the fine-grained energy characterization of software. This work is also part of this thesis.

- *Deliverable 3.2 – Initial Energy Consumption Analysis (including SW demo)* [75]: Section 2, discusses a preliminary version of the mapping techniques introduced in this thesis.
- *Deliverable 3.3 – Analysis Based Verification and Debugging of Energy, Performance and Precision Properties* [76]: Section 4, discusses the IPET-like SRA introduced in this thesis.

EMC^2 – Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environment

The future work of this thesis is currently taken forward under the EMC^2, an ARTEMIS Joint Undertaking project in the Innovation Pilot Programme “Computing platforms for embedded systems (AIPP5), with the aim to establish Multi-Core technology in all relevant Embedded Systems domains.”

Acronyms

CO_2 Carbon Dioxide. 19

BLE Bluetooth Low Energy. 2

CFG Control Flow Graph. 4, 6, 9, 79, 81, 84, 88

CMOS Complementary Metal–Oxide–Semiconductor. 21–23

DSP Digital Signal Processing. 46

DUT Device Under Test. 55, 56

FNOP Fetch No-Operation. 48, 50

ICT Information Communication Technology. vii, 2, 19, 20

IoT Interent of Things. 1–3, 5, 20, 45

IPET Implicit Path Enumeration Technique. xiv, 8, 14, 77–84, 88, 90, 96

IPV6 Internet Protocol Version 6. 1

IR Intermediate Representation. 6, 8, 14, 16

ISA Instruction Set Architecture. viii, xiii–xv, 4, 6–8, 12, 13, 29–31, 33, 36, 38, 40–42, 45, 46, 48, 50, 51, 63–75, 77–80, 82, 84, 85, 88, 89, 91, 95–97, 99, 101, 105

ISS Instruction Set Simulator. xiv, 8, 31, 50, 56, 57, 59, 60, 84, 95, 96, 101–103

LLVM Low Level Virtual Machine. viii, xiii, 8, 14, 16, 65–68, 84

RISC Reduced Instruction Set Computer. 45

SME Small and Medium Enterprises. 12

SRA Static Resource Analysis. 8, 10, 13, 14, 64, 96, 98

WCET Worst Case Execution Time. 5

BIBLIOGRAPHY

[1] *A Conversation with Steve Furber*, Queue, 8 (2010), pp. 1–8, doi:10.1145/1716383.1716385, `http://doi.acm.org/10.1145/1716383.1716385`.

[2] *Make IT Green. Cloud Computing and its Contribution to Climate Change*, 2014, `http://www.greenpeace.org/international/Global/international/planet-2/report/2010/3/make-it-green-cloud-computing.pdf` (accessed 2016-10-02).

[3] *iOS Benchmarks – Multi-Core*, November 2016, `https://browser.primatelabs.com/ios-benchmarks` (accessed 2016-11-05).

[4] *List of iOS devices*, November 2016, `https://en.wikipedia.org/wiki/List_of_iOS_devices` (accessed 2016-11-05).

[5] *The DWARF Debugging Standard*, Oct. 2016, `http://dwarfstd.org/` (accessed 2016-10-02).

[6] K. M. ABRAHAM, *Prospects and Limits of Energy Storage in Batteries*, The Journal of Physical Chemistry Letters, 6 (2015), pp. 830–844, doi:10.1021/jz5026273, `http://dx.doi.org/10.1021/jz5026273`.

[7] E. ALBERT, P. ARENAS, S. GENAIM, AND G. PUEBLA, *Closed-Form Upper Bounds in Static Cost Analysis*, Journal of Automated Reasoning, 46 (2011), pp. 161–203.

[8] F. Z. AMAN KANSAL, *Fine-Grained Energy Profiling for Power-Aware Application Design*, Annapolis, MD, USA, June 2008, Association for Computing Machinery, Inc., `https://www.microsoft.com/en-us/research/publication/fine-grained-energy-profiling-for-power-aware-application-design/`.

[9] ARM, *Cortex-M Series*, 2016, `http://www.arm.com/products/processors/cortex-m/index.php?tab=Resources` (accessed 2016-10-02).

[10] G. BALAKRISHNAN, S. SANKARANARAYANAN, F. IVANČIĆ, O. WEI, AND A. GUPTA, *SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 238–254, doi:10.1007/978-3-540-69166-2_16, `http://dx.doi.org/10.1007/978-3-540-69166-2_16`.

[11] K. BECK, *Extreme Programming Explained: Embrace Change*, An Alan R. Apt Book Series, Addison-Wesley, 2000, https://books.google.co.uk/books?id=G8EL4H4vf7UC.

[12] B. BLACKHAM, M. LIFFITON, AND G. HEISER, *Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets*, in 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2014, pp. 169–178, doi:10.1109/RTAS.2014.6926000.

[13] C. BLACKMORE, O. RAY, AND K. EDER, *A logic programming approach to predict effective compiler settings for embedded software*, Theory and Practice of Logic Programming, 15 (2015), pp. 481–494, doi:10.1017/S1471068415000174, http://journals.cambridge.org/article_S1471068415000174.

[14] BLUETOOTH SIG, *Bluetooth Low Energy*, 2016, https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy (accessed 2016-10-22).

[15] A. BOGLIOLO, L. BENINI, G. MICHELI, AND B. RICC, *Gate-Level Power and Current Simulation of CMOS Integrated Circuits*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 5 (1997), pp. 473–488, doi:http://dx.doi.org/10.1109/43.736184.

[16] R. BOSCH AND M. TRICK, *Integer Programming*, in Search Methodologies, E. Burke and G. Kendall, eds., Springer US, 2005, pp. 69–95, doi:10.1007/0-387-28356-0_3, http://dx.doi.org/10.1007/0-387-28356-0_3.

[17] C. BRANDOLESE, S. CORBETTA, AND W. FORNACIARI, *Software energy estimation based on statistical characterization of intermediate compilation code*, in Low Power Electronics and Design (ISLPED) 2011 International Symposium on, Aug 2011, pp. 333–338, doi:10.1109/ISLPED.2011.5993659.

[18] G. BRAT, J. NAVAS, N. SHI, AND A. VENET, *IKOS: A Framework for Static Analysis Based on Abstract Interpretation*, in Software Engineering and Formal Methods, Springer, 2014, pp. 271–277.

[19] D. BROOKS, V. TIWARI, AND M. MARTONOSI, *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*, ACM SIGARCH Computer Architecture News, 28 (2000), pp. 83–94, doi:10.1145/342001.339657, http://portal.acm.org/citation.cfm?id=339657http://portal.acm.org/citation.cfm?doid=342001.339657.

[20] A. BROWN AND G. WILSON, *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, The Achrictecture of Open Source Applications, CreativeCommons, 2011, http://books.google.co.uk/books?id=CoEytwAACAAJ.

[21] C. BURGUIÈRE, C. ROCHANGE, AND U. P. SABATIER, *History-based schemes and implicit path enumeration*, in In Proceedings of the 6th Workshop on Worst-Case Execution Time Analysis (WCET'06, 2006, pp. 17–22.

[22] A. COLIN AND I. PUAUT, *A modular and retargetable framework for tree-based WCET analysis*, in Proceedings 13th Euromicro Conference on Real-Time Systems, 2001, pp. 37–44, doi:10.1109/EMRTS.2001.933995.

[23] G. CONTRERAS AND M. MARTONOSI, *Power prediction for Intel XScale processors using performance monitoring unit events*, in ISLPED '05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, IEEE, 2005, pp. 221–226, doi:10.1109/LPE.2005.195518, `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1522767`.

[24] A. COSKUN, T. ROSING, K. WHISNANT, AND K. GROSS, *Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 16 (2008), pp. 1127–1140, doi:10.1109/TVLSI.2008.2000726.

[25] P. COUSOT, *Abstract Interpretation Based Formal Methods and Future Challenges*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 138–156.

[26] P. COUSOT AND R. COUSOT, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, 1977, ACM Press, New York, NY, pp. 238–252.

[27] P. COUSOT AND R. COUSOT, *Systematic Design of Program Analysis Frameworks*, in Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '79, New York, NY, USA, 1979, ACM, pp. 269–282, doi:10.1145/567752.567778, `http://doi.acm.org/10.1145/567752.567778`.

[28] Y. CUI, W. ZHANG, AND B. HE, *A Discrete Thermal Controller for Chip-multiprocessors*, in Proceedings of the 2016 Conference on Design, Automation & Test in Europe, DATE '16, San Jose, CA, USA, 2016, EDA Consortium, pp. 67–72.

[29] S. K. DEBRAY, P. LÓPEZ-GARCÍA, M. HERMENEGILDO, AND N.-W. LIN, *Lower Bound Cost Estimation for Logic Programs*, in 1997 International Logic Programming Symposium, MIT Press, Cambridge, MA, October 1997, pp. 291–305.

[30] K. EDER, ed., *Deliverable 2.3: High-Level Energy Models*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), September 2015, `http://entraproject.eu/wp-content/uploads/2016/03/deliv_2.3.pdf`.

[31] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger, and M. Rosendahl, *Entra: Whole-systems energy transparency*, Microprocessors and Microsystems, (2016), pp. –, doi:http://dx.doi.org/10.1016/j.micpro.2016.07.003, `http://www.sciencedirect.com/science/article/pii/S0141933116300862`.

[32] J. Engblom, A. Ermedahl, and F. Stappert, *Comparing Different Worst-Case Execution Time Analysis Methods*, in The Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), November 2000, `http://www.es.mdh.se/publications/837-`.

[33] ENTRA Consortium, *ENTRA: Whole-Systems Energy Transparency. A 3-year research project funded by the EU 7th Framework Programme Future and Emerging Technologies (FET)*, 2016, `www.entraproject.eu` (accessed 2016-10-20).

[34] D. Evans, *The internet of things: How the next evolution of the internet is changing everything*, apr 2011, `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf` (accessed 2016-07-13).

[35] H. Falk and J. Kleinsorge, *Optimal static WCET-aware scratchpad allocation of program code*, in Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, July 2009, pp. 732–737.

[36] J. Flinn and M. Satyanarayanan, *PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications*, in Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications, WMCSA '99, Washington, DC, USA, 1999, IEEE Computer Society, pp. 2–, `http://dl.acm.org/citation.cfm?id=520551.837522`.

[37] Z. Gan, Z. Gu, H. Tan, M. Zhang, and J. Zhang, *Worst-case energy consumption minimization based on interference analysis and bank mapping in multicore systems*, International Journal of Distributed Sensor Networks, 13 (2017), p. 1550147716686969, doi:10.1177/1550147716686969, `http://dx.doi.org/10.1177/1550147716686969`, arXiv:http://dx.doi.org/10.1177/1550147716686969.

[38] K. Georgiou, *Energy Transparency for Deeply Embedded Programs - Benchmarks*, 2016, `https://github.com/kg8280/EnergyTransparency` (accessed 2016-10-03).

[39] K. Georgiou, S. Kerrison, Z. Chamski, and K. Eder, *Energy transparency for deeply embedded programs*, ACM Trans. Archit. Code Optim., 14 (2017), pp. 8:1–8:26, doi:10.1145/3046679, `http://doi.acm.org/10.1145/3046679`.

[40] K. GEORGIOU, S. KERRISON, AND K. EDER, *A Multi-level Worst Case Energy Consumption Static Analysis for Single and Multi-threaded Embedded Programs*, Tech. Report CSTR-14-003, University of Bristol, December 2014.

[41] K. GEORGIOU, S. KERRISON, AND K. EDER, *On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs*, CoRR, abs/1510.07095 (2015), http://arxiv.org/abs/1510.07095.

[42] K. GEORGIOU AND U. LIQAT, *Towards LLVM-Based Energy Consumption Analysis of Programs*, ICT Energy Letters, (2014), pp. 16–17, http://www.nanoenergyletters.com/files/nel/ICT-Energy_Letters_8.pdf.

[43] GRAPHVIZ COMMUNITY, *GraphViz - Graph Visualization Software*, http://www.graphviz.org/ (accessed 2016-09-29).

[44] N. GRECH, K. GEORGIOU, J. PALLISTER, S. KERRISON, J. MORSE, AND K. EDER, *Static analysis of energy consumption for LLVM IR programs*, in Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15, New York, NY, USA, 2015, ACM, doi:10.1145/2764967.2764974.

[45] J. GUSTAFSSON, A. BETTS, A. ERMEDAHL, AND B. LISPER, *The Mälardalen WCET Benchmarks - Past, Present and Future*, in Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, July 2010, http://www.es.mdh.se/publications/1895-.

[46] R. HAMEED, W. QADEER, M. WACHS, O. AZIZI, A. SOLOMATNIKOV, B. C. LEE, S. RICHARDSON, C. KOZYRAKIS, AND M. HOROWITZ, *Understanding Sources of Inefficiency in General-purpose Chips*, in Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, New York, NY, USA, 2010, ACM, pp. 37–47, doi:10.1145/1815961.1815968, http://doi.acm.org/10.1145/1815961.1815968.

[47] M. HERMENEGILDO, G. PUEBLA, F. BUENO, AND P. LÓPEZ-GARCÍA, *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*, Science of Computer Programming, 58 (2005).

[48] HIPEAC, *HiPEAC Collaboration Grants – Towards understanding the limits of Energy Consumption Static Analysis (ECSA) for Reactive Embedded Systems at MpicoSys Embedded PicoSystems*, Sept 2015, https://www.hipeac.net/~kg8280/ (accessed 2016-12-18).

[49] A. HIRATA, H. ONODERA, AND K. TAMARU, *Estimation Of Short-Circuit Power Dissipation And Its Influence On Propagation Delay For Static Cmos Gates*, in Proc. of ISCAS 96, Citeseer, 1996.

[50] S. J. HOLLIS AND S. KERRISON, *Swallow: Building an energy-transparent many-core embedded real-time system*, in 2016 Design, Automation Test in Europe Conference Exhibition (DATE), March 2016, pp. 73–78, `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7459283`.

[51] M. A. IBRAHIM, M. RUPP, AND H. FAHMY, *Power estimation methodology for VLIW Digital Signal Processors*, in 2008 42nd Asilomar Conference on Signals, Systems and Computers, no. 1, IEEE, Oct. 2008, pp. 1840–1844, doi:10.1109/ACSSC.2008.5074746, `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5074746`.

[52] INTEL, *Intel® Edison Compute Module*, `http://www.intel.com/content/www/us/en/do-it-yourself/edison.html` (accessed 2016-10-02).

[53] J. HAUSER, *SoftFloat*, November 2014, `http://www.jhauser.us/arithmetic/SoftFloat.html` (accessed 2016-10-02).

[54] J. MADDOCKS AND K. GEORGIOU AND K. EDER, *Static Resource Analysis Evaluation for the Cortex M0*, `https://github.com/kg8280/SRA_evaluationForCortexM0/blob/master/technical_paper.pdf` (accessed 2017-03-03).

[55] R. JAYASEELAN, T. MITRA, AND X. LI, *Estimating the Worst-Case Energy Consumption of Embedded Software*, in Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE, April 2006, pp. 81–90, doi:10.1109/RTAS.2006.17.

[56] JSON ORGANIZATION, *Introducing JSON*, 2016, `http://www.json.org/` (accessed 2016-09-24).

[57] K. GEORGIOU AND S. KERRISON AND K. EDER, *Cool programs save energy - Greener software can play a significant role in saving the environment*, 2015, `https://goo.gl/0kppgG` (accessed 2015-03-09).

[58] W. KELVIN, *Popular Lectures and Addresses*, no. v. 1 in Nature series, Macmillan & Company, 1889, `https://books.google.co.uk/books?id=7685AAAAMAAJ`.

[59] S. KERRISON AND K. EDER, *Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor*, ACM Transactions on Embedded Computing Systems, 14 (2015), pp. 56:1–56:25, doi:10.1145/2700104, `http://doi.acm.org/10.1145/2700104`.

[60] P. C. KOCHER, J. JAFFE, AND B. JUN, *Differential Power Analysis*, in Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '99, London, UK, UK, 1999, Springer-Verlag, pp. 388–397, `http://dl.acm.org/citation.cfm?id=646764.703989`.

[61] L. Kong and J. Jiang, *A worst-case execution time analysis approach based on independent paths for arm programs*, Wuhan University Journal of Natural Sciences, 17 (2012), pp. 391–399, doi:10.1007/s11859-012-0860-1, `http://dx.doi.org/10.1007/s11859-012-0860-1`.

[62] C. Lattner and D. Patel, *Extensible Metadata in LLVM IR*, Apr 2010, `http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html` (accessed 2016-10-02).

[63] J. H. Lau, *Overview and outlook of throughsilicon via (TSV) and 3D integrations*, Microelectronics International, 28 (2011), pp. 8–22, doi:10.1108/13565361111127304, `http://dx.doi.org/10.1108/13565361111127304`.

[64] Y.-S. Li, S. Malik, and A. Wolfe, *Performance estimation of embedded software with instruction cache modeling*, in Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on, Nov 1995, pp. 380–387, doi:10.1109/ICCAD.1995.480144.

[65] Y.-T. Li and S. Malik, *Performance Analysis of Embedded Software Using Implicit Path Enumeration*, in Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, DAC '95, New York, NY, USA, 1995, ACM, pp. 456–461, doi:10.1145/217474.217570, `http://doi.acm.org/10.1145/217474.217570`.

[66] Y.-T. Li, S. Malik, and A. Wolfe, *Efficient microarchitecture modeling and path analysis for real-time software*, in Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE, Dec 1995, pp. 298–307, doi:10.1109/REAL.1995.495219.

[67] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim, *An accurate worst case timing analysis technique for RISC processors*, in Real-Time Systems Symposium, 1994., Proceedings., Dec 1994, pp. 97–108, doi:10.1109/REAL.1994.342726.

[68] C.-Y. Lin, C.-B. Kuan, W.-L. Shih, and J. K. Lee, *Compilers for low power with design patterns on embedded multicore systems*, Journal of Signal Processing Systems, 80 (2015), pp. 277–293, doi:10.1007/s11265-014-0917-9, `http://dx.doi.org/10.1007/s11265-014-0917-9`.

[69] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, J. P. Gallagher, M. V. Hermenegildo, and K. Eder, *Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR*, Springer International Publishing, Cham, 2016, pp. 81–100, doi:10.1007/9783319465593_5, `http://dx.doi.org/10.1007/978-3-319-46559-3_5`.

[70] U. LIQAT, S. KERRISON, A. SERRANO, K. GEORGIOU, P. LOPEZ-GARCIA, N. GRECH, M. HERMENEGILDO, AND K. EDER, *Energy Consumption Analysis of Programs based on XMOS ISA-level Models*, in Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13), 2014.

[71] B. LISPER, *SWEET – a Tool for WCET Flow Analysis*, in 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, B. Steffen, ed., Springer-Verlag, October 2014, pp. 482–485, `http://www.es.mdh.se/publications/3693-`.

[72] LLVM ORGANIZATION, *The LLVM Compiler Infrastructure*, `http://www.llvm.org/` (accessed 2016-10-02).

[73] P. LÓPEZ-GARCÍA, ed., *Deliverable 3.1: A General Framework for Resource Consumption Analysis and Verification*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013, `http://entraproject.eu/wp-content/uploads/2014/03/deliv_3.1_final.pdf`.

[74] P. LÓPEZ-GARCÍA, ed., *Deliverable 1.1: Preliminary Report and Demo on Energy-Aware Tools*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), December 2014, `http://entraproject.eu/wp-content/uploads/2015/07/deliv_1.1_draft.pdf`.

[75] P. LÓPEZ-GARCÍA, ed., *Initial Energy Consumption Analysis*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), April 2014, `http://entraproject.eu/wp-content/uploads/2014/06/deliv_3.2.pdf`.

[76] P. LÓPEZ-GARCÍA, ed., *Analysis Based Verification and Debugging of Energy, Performance and Precision Properties*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), December 2015, `http://entraproject.eu/wp-content/uploads/2016/03/deliv_3.3.pdf`.

[77] P. LÓPEZ-GARCÍA, ed., *Deliverable 1.2: Energy-Aware Software Development Methods and Tools*, ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), March 2016, `http://entraproject.eu/wp-content/uploads/2016/03/deliv_1.2.pdf`.

[78] LP SOLVE COMMUNITY, *Introduction to lp_solve 5.5.2.5*, `http://lpsolve.sourceforge.net/5.5/` (accessed 2016-09-29).

[79] LP SOLVE COMMUNITY, *LP file format*, `http://lpsolve.sourceforge.net/5.5/lp-format.htm` (accessed 2016-09-27).

[80] K. LU, D. MÜLLER-GRITSCHNEDER, AND U. SCHLICHTMANN, *Hierarchical control flow matching for source-level simulation of embedded software*, in 2012 International Symposium on System on Chip (SoC), Oct 2012, pp. 1–5, doi:10.1109/ISSoC.2012.6376366.

[81] LUCIAN SHIFREN, *Leakage power – it's worse than you think*, 2016, `http://www.eetimes.com/document.asp?doc_id=1264175` (accessed 2016-11-01).

[82] G. LUO, B. GUO, Y. SHEN, H. LIAO, AND L. REN, *Analysis and Optimization of Embedded Software Energy Consumption on the Source Code and Algorithm Level*, in 2009 Fourth International Conference on Embedded and Multimedia Computing, Dec 2009, pp. 1–5, doi:10.1109/EM-COM.2009.5402965.

[83] M. FIELD, *Binary division*, November 2014, `https://github.com/kg8280/EnergyTransparency/blob/master/Benchmarks/radix4Division.zip` (accessed 2014-03-20).

[84] M. P. MILIS, *The Cloud Begins With Coal Big Data, Big Networks, Big Infrastructure, and Big Power – An overview of the electricity used by the global digital ecosystem*, 2013, `http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf?c761ac&c761ac` (accessed 2016-11-04).

[85] A. MARREF, *Predicated Worst-Case Execution-Time Analysis*, PhD thesis, 2009, `https://pdfs.semanticscholar.org/ea06/797322ccf143b92f04a061b7c87c12e3b7a3.pdf`.

[86] D. MAY, *The XMOS XS1 Architecture*, 2009.

[87] T. MEYEROWITZ, A. SANGIOVANNI-VINCENTELLI, M. SAUERMANN, AND D. LANGEN, *Source-level timing annotation and simulation for a heterogeneous multiprocessor*, in 2008 Design, Automation and Test in Europe, March 2008, pp. 276–279, doi:10.1109/DATE.2008.4484897.

[88] E. K. MILLER, *"smart" curve fitting*, IEEE Potentials, 21 (2002), pp. 20–23, doi:10.1109/45.985323.

[89] J. MORSE, S. KERRISON, AND K. EDER, *On the infeasibility of analysing worst-case dynamic energy*, CoRR, abs/1603.02580 (2016), `http://arxiv.org/abs/1603.02580`.

[90] MPICO, *MPICOSYS – EMBEDDED PICO SYSTEMS*, 2016, `https://www.mpicosys.com/` (accessed 2016-09-28).

[91] J. NAVAS, M. MÉNDEZ-LOJO, AND M. HERMENEGILDO, *Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications*, in The Sixth NASA Langley Formal Methods Workshop (LFM 08), April 2008.
Extended Abstract.

[92] J. NAVAS, M. MÉNDEZ-LOJO, AND M. HERMENEGILDO, *User-Definable Resource Usage Bounds Analysis for Java Bytecode*, in Proceedings of BYTECODE, vol. 253 of Electronic Notes in Theoretical Computer Science, Elsevier - North Holland, March 2009, pp. 65–82.

[93] J. NAVAS, E. MERA, P. LÓPEZ-GARCÍA, AND M. HERMENEGILDO, *User-Definable Resource Bounds Analysis for Logic Programs*, in International Conference on Logic Programming (ICLP'07), Lecture Notes in Computer Science, Springer, 2007.

[94] L. ORGANIZATION, *LLVM Command Guide*, 2016, `http://llvm.org/docs/CommandGuide/` (accessed 2016-10-11).

[95] G. OTTOSSON AND M. SJODIN, *Worst-Case Execution Time Analysis for Modern Hardware Architectures*, in In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97), 1997, pp. 47–55.

[96] O. OZTURK, K. MAHMUT, AND C. GUANGYU, *Compiler-Directed Energy Reduction Using Dynamic Voltage Scaling and Voltage Islands for Embedded Systems*, IEEE Trans. Comput., 62 (2013), pp. 268–278, doi:10.1109/TC.2011.229, `http://dx.doi.org/10.1109/TC.2011.229`.

[97] J. PALLISTER, K. EDER, AND S. HOLLIS, *Optimizing the flash-RAM energy trade-off in deeply embedded systems*, in Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on, Feb 2015, pp. 115–124, doi:10.1109/CGO.2015.7054192.

[98] J. PALLISTER, K. EDER, S. J. HOLLIS, AND J. BENNETT, *A High-level Model of Embedded Flash Energy Consumption*, in Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '14, New York, NY, USA, 2014, ACM, pp. 20:1–20:9, doi:10.1145/2656106.2656108, `http://doi.acm.org/10.1145/2656106.2656108`.

[99] J. PALLISTER, S. HOLLIS, AND J. BENNET, *The BEEBS Benchmark Suite*, 2013, `http://www.cs.bris.ac.uk/Research/Micro/beebs.jsp` (accessed 2016-10-03).

[100] J. PALLISTER, S. KERRISON, J. MORSE, AND K. EDER, *Data dependent energy modelling: A worst case perspective*, CoRR, abs/1505.03374 (2015), `http://arxiv.org/abs/1505.03374`.

[101] C. Y. PARK AND A. C. SHAW, *Experiments with a program timing tool based on source-level timing schema*, Computer, 24 (1991), pp. 48–57, doi:10.1109/2.76286.

[102] S. PENOLAZZI, L. BOLOGNINO, AND A. HEMANI, *Energy and Performance Model of a SPARC Leon3 Processor*, in Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on, Aug 2009, pp. 651–656, doi:10.1109/DSD.2009.147.

[103] D. POTOP-BUTUCARU AND I. PUAUT, *Integrated Worst-Case Execution Time Estimation of Multicore Applications*, in 13th International Workshop on Worst-Case Execution Time Analysis, C. Maiza, ed., vol. 30 of OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany, 2013, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 21–31, doi:http://dx.doi.org/10.4230/OASIcs.WCET.2013.21, http://drops.dagstuhl.de/opus/volltexte/2013/4119.

[104] P. PUSCHNER AND A. SCHEDL, *Computing Maximum Task Execution Times - A Graph-Based Approach*, Journal of Real-Time Systems, 13 (1997), pp. 67–91.

[105] G. QU, N. KAWABE, K. USAMI, AND M. POTKONJAK, *Function-level Power Estimation Methodology for Microprocessors*, in Proceedings of the 37th Annual Design Automation Conference, DAC '00, New York, NY, USA, 2000, ACM, pp. 810–813, doi:10.1145/337292.337786, http://doi.acm.org/10.1145/337292.337786.

[106] R. OSBORNE AND S. KERRISON, *AXE (An Xcore Emulator)*, https://github.com/stevekerrison/tool_axe (accessed 2016-09-03).

[107] M. ROSENDAHL, *Automatic Complexity Analysis*, in Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89, New York, NY, USA, 1989, ACM, pp. 144–156, doi:10.1145/99370.99381, http://doi.acm.org/10.1145/99370.99381.

[108] K. ROY AND M. JOHNSON, *Software design for low power*, in Low power design in deep submicron electronics, Kluwer Academic Publishers, 1997, ch. 6, pp. 433–460, http://dl.acm.org/citation.cfm?id=265902.

[109] M. SAMI, D. SCIUTO, C. SILVANO, AND V. ZACCARIA, *An instruction-level energy model for embedded VLIW architectures*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21 (2002), pp. 998–1010, doi:10.1109/TCAD.2002.801105, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1028101.

[110] D. SARTA, D. TRIFONE, AND G. ASCIA, *A data dependent approach to instruction level power estimation*, in Low-Power Design, 1999. Proceedings. IEEE Alessandro Volta Memorial Workshop on, Mar 1999, pp. 182–190, doi:10.1109/LPD.1999.750419.

[111] M. SCHELLEKENS, *Unlocking the potential of compositional static average-case analysis*, The Journal of Logic and Algebraic Programming, 79 (2010), pp. 61 –

83, doi:http://dx.doi.org/10.1016/j.jlap.2009.02.006, http://www.sciencedirect.com/science/article/pii/S1567832609000125.

Speical Issue: Logic, Computability and Topology in Computer Science: A New Perspective for Old Disciplines.

[112] S. SCHUBERT, D. KOSTIC, W. ZWAENEPOEL, AND K. G. SHIN, *Profiling Software for Energy Consumption*, in Green Computing and Communications (GreenCom), 2012 IEEE International Conference on, Nov 2012, pp. 515–522, doi:10.1109/GreenCom.2012.86.

[113] Y. SHAO AND D. BROOKS, *Energy characterization and instruction-level energy model of Intel's Xeon Phi processor*, in International Symposium on Low Power Electronics and Design (ISLPED), no. November, IEEE, Sept. 2013, pp. 389–394, doi:10.1109/ISLPED.2013.6629328, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6629328.

[114] W. SHIH, Y.-P. YOU, C.-W. HUANG, AND J. K. LEE, *Compiler Optimization for Reducing Leakage Power in Multithread BSP Programs*, ACM Trans. Des. Autom. Electron. Syst., 20 (2014), pp. 9:1–9:34, doi:10.1145/2668119, http://doi.acm.org/10.1145/2668119.

[115] A. SINHA, N. ICKES, AND A. P. CHANDRAKASAN, *Instruction level and operating system profiling for energy exposed software*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 11 (2003), pp. 1044–1057, doi:10.1109/TVLSI.2003.819569.

[116] J. SOUYRIS, E. L. PAVEC, G. HIMBERT, G. BORIOS, V. JÉGU, AND R. HECKMANN, *Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation*, in 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05), R. Wilhelm, ed., vol. 1 of OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany, 2007, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, doi:http://dx.doi.org/10.4230/OASIcs.WCET.2005.810, http://drops.dagstuhl.de/opus/volltexte/2007/810.

[117] K. SRINIVASAN AND K. CHATHA, *Integer linear programming and heuristic techniques for system-level low power scheduling on multiprocessor architectures under throughput constraints*, Integration, the {VLSI} Journal, 40 (2007), pp. 326 – 354, doi:http://dx.doi.org/10.1016/j.vlsi.2006.01.001, http://www.sciencedirect.com/science/article/pii/S0167926006000253.

[118] A. SRIVASTAVA AND A. EUSTACE, *ATOM: A System for Building Customized Program Analysis Tools*, in Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, New York, NY, USA, 1994, ACM, pp. 196–205, doi:10.1145/178243.178260, http://doi.acm.org/10.1145/178243.178260.

[119] F. Stappert and P. Altenbernd, *Complete worst-case execution time analysis of straight-line hard real-time programs*, Journal of Systems Architecture, 46 (2000), pp. 339 – 355, doi:http://dx.doi.org/10.1016/S1383-7621(99)00010-7, `http://www.sciencedirect.com/science/article/pii/S1383762199000107`.

[120] S. Stattelmann, O. Bringmann, and W. Rosenstiel, *Dominator homomorphism based code matching for source-level simulation of embedded software*, in 2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS), Oct 2011, pp. 305–314, doi:10.1145/2039370.2039417.

[121] S. Stattelmann, O. Bringmann, and W. Rosenstiel, *Fast and accurate source-level simulation of software timing considering complex code optimizations*, in 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), June 2011, pp. 486–491.

[122] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, *An accurate and fine grain instruction-level energy model supporting software optimizations*, in Proc. of PAT-MOS, Citeseer, 2001, doi:10.1.1.115.3528, `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6971&rep=rep1&type=pdf`.

[123] Texas Instruments, *INA219: 26-V, Bidirectional, Zero-Drift, High-Side, I2C Out Current/Power Monitor*, September 2016, `http://www.ti.com/product/INA219` (accessed 2016-09-24).

[124] The GCC team, *GCC, the GNU Compiler Collection*, `https://gcc.gnu.org/` (accessed 2016-09-24).

[125] The Silicon Engine, *1963: Complementary MOS Circuit Configuration is Invented*, 2016, `http://www.computerhistory.org/siliconengine/complementary-mos-circuit-configuration-is-invented/` (accessed 2016-09-28).

[126] H. Theiling and C. Ferdinand, *Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis*, in Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, Dec 1998, pp. 144–153, doi:10.1109/REAL.1998.739739.

[127] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, *Instruction level power analysis and optimization of software*, Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 13 (1996), pp. 223–238, doi:10.1007/BF01130407, `http://www.springerlink.com/index/10.1007/BF01130407`.

[128] J. M. Townley, *Practical programming for static average-case analysis: the MOQA investigation*, PhD thesis, University College Cork, Ireland, 2013.

[129] P. Vasconcelos and K. Hammond, *Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs*, in Proceedings of the Workshop on Implementation of Functional Languages, vol. 3145 of Lecture Notes in Computer Science, Springer-Verlag, September 2003, pp. 86–101.

[130] H. J. M. Veendrick, *Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits*, IEEE Journal of Solid-State Circuits, 19 (1984), pp. 468–473, doi:10.1109/JSSC.1984.1052168.

[131] T. Šimunić, L. Benini, G. De Micheli, and M. Hans, *Source Code Optimization and Profiling of Energy Consumption in Embedded Systems*, in Proceedings of the 13th International Symposium on System Synthesis, ISSS '00, Washington, DC, USA, 2000, IEEE Computer Society, pp. 193–198, doi:10.1145/501790.501831, `http://dx.doi.org/10.1145/501790.501831`.

[132] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, *Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems*, in 2015 27th Euromicro Conference on Real-Time Systems (ECRTS), July 2015, pp. 105–114, doi:10.1109/ECRTS.2015.17.

[133] M. M. Waldrop, *The chips are down for Moore's law*, Nature, 530 (2016), pp. 144–147, doi:10.1038/530144a, `https://doi.org/10.1038%2F530144a`.

[134] A. Warhol, *The Philosophy of Andy Warhol: From A to B and Back Again*, A Harvest book, Harcourt Brace Jovanovich, 1975, `https://books.google.co.uk/books?id=QhbrAAAAMAAJ`.

[135] D. Watt, *Programming XC on XMOS Devices*, XMOS Limited, 2009, `http://books.google.co.uk/books?id=81klKQEACAAJ`.

[136] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, *Measuring Energy and Power with PAPI*, in 2012 41st International Conference on Parallel Processing Workshops, Sept 2012, pp. 262–268, doi:10.1109/ICPPW.2012.39.

[137] B. Wegbreit, *Mechanical Program Analysis*, Commun. ACM, 18 (1975), pp. 528–539.

[138] T. Wei, J. Mao, W. Zou, and Y. Chen, *A new algorithm for identifying loops in decompilation*, in Static Analysis, H. Nielson and G. File, eds., vol. 4634 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 170–183, doi:10.1007/978-3-540-74061-2_11, `http://dx.doi.org/10.1007/978-3-540-74061-2_11`.

[139] R. WILHELM, *Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone*, in Verification, Model Checking, and Abstract Interpretation, B. Steffen and G. Levi, eds., vol. 2937 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 309–322, doi:10.1007/978-3-540-24622-0_25, `http://dx.doi.org/10.1007/978-3-540-24622-0_25`.

[140] R. WILHELM, J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, F. MUELLER, I. PUAUT, P. PUSCHNER, J. STASCHULAT, AND P. STENSTRÖM, *The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools*, ACM Trans. Embed. Comput. Syst., 7 (2008), pp. 36:1–36:53, doi:10.1145/1347375.1347389, `http://doi.acm.org/10.1145/1347375.1347389`.

[141] K. WILKEN, J. LIU, AND M. HEFFERNAN, *Optimal Instruction Scheduling Using Integer Programming*, in Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, New York, NY, USA, 2000, ACM, pp. 121–133, doi:10.1145/349299.349318, `http://doi.acm.org/10.1145/349299.349318`.

[142] XMOS, *Code to perform standard DSP functions, such as Biquads, FIRs, sample rate conversion*, February 2014, `https://github.com/xcore/sc_dsp_filters` (accessed 2017-03-05).

[143] XMOS, *xTimecomposer*, November 2014, `https://www.xmos.com/support/xtools` (accessed 2014-02-21).

[144] XMOS, *Debug with printf in real-time*, 2016, `https://www.xmos.com/support/tools/other?subcategory=&component=14774` (accessed 2016-10-03).

[145] XMOS LTD, *xcore-analog slicekit*, 2016, `https://www.xmos.com/support/boards?product=17554` (accessed 2016-09-24).

[146] W. YE, N. VIJAYKRISHNAN, M. KANDEMIR, AND M. J. IRWIN, *The Design and Use of Simplepower: A Cycle-accurate Energy Estimation Tool*, in Proceedings of the 37th Annual Design Automation Conference, DAC '00, New York, NY, USA, 2000, ACM, pp. 340–345, doi:10.1145/337292.337436, `http://doi.acm.org/10.1145/337292.337436`.

[147] I. YOUNG, *50 years of digital logic and microprocessors at ISSCC*, in Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International, Feb 2003, pp. 29–30, doi:10.1109/ISSCC.2003.1264035.