Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Methodology of extraction of reliable energy data on a basic block level
-
Μεθοδολογία εξαγωγής αξιόπιστων δεδομένων ενέργειας σε επίπεδο basic block

Διπλωματική Εργασία

του

ΜΠΟΥΡΑ ΔΗΜΗΤΡΙΟΥ ΣΤΑΜΑΤΙΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Methodology of extraction of reliable energy data on a basic block level
-
Μεθοδολογία εξαγωγής αξιόπιστων δεδομένων ενέργειας σε επίπεδο basic block

Διπλωματική Εργασία

του

ΜΠΟΥΡΑ ΔΗΜΗΤΡΙΟΥ ΣΤΑΜΑΤΙΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Φεβρουάριος 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δημήτριος Σούντρης Καθηγητής Ε.Μ.Π.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π	Σωτήριος Ξύδης Μεταδιδακτορικος ερευνητης Ε.Μ.Π

Αθήνα, Φεβρουάριος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

(Υπογραφή)

....
Μπούρας Δημήτριος Σταμάτιος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η ενεργειακή κατανάλωση είναι μια αναδυόμενη ανησυχία σε πολλούς κλάδους και τομείς της πληροφορικής, για λόγους ενεργειακού χρηματικού κόστους , απαγωγής θερμότητας , διάρκειας ζωής μπταρίας και περιβαλλοντικών ανησυχιών.

Προηγουμένως, η κατανάλωση ενέργειας σχετιζόταν κυρίως με το υλικό υπό χρήση , ωστόσο το λογισμικό είναι στην πράξη εξίσου σημαντικό με το υλικό πάνω στο οποίο εκτελείται. Ο τελικός στόχος αυτής της διπλωματικής εργασίας είναι να βοηθήσει τους προγραμματιστές και γενικά τους επιστήμονες της πληροφορικής, να καταλάβουν και να σκεφτούν ενεργά για την υλοποίηση “πράσινου λογισμικόυ” στην δουλειά τους, με στόχο την μείωση της κατανάλωσης ενέργειας του λογισμικού τους και την παραγωγή ενεργειακά αποδοτικών προϊόντων.

Βασικό προαπαιτούμενο για την ενεργειακή αποδοτικότητα είναι η εκτίμηση ενέργειας Για να το επιτύχουμε αυτό, ξεκινάμε με την παραγωγή ενός αξιόπιστου dataset ενέργειας, το οποίο στην συνέχεια θα αποτελέσει την βάση για την δημιουργία ενος μοντέλου πρόβλεψης ενέργειας.

Το πρώτο βήμα για την παραγωγή ενός dataset ενέργειας σε επίπεδο basic block είναι η μέτρηση της ενέργειας ενός πολύ μεγαλύτερου κομματιού κώδικα και ο διαμοιρασμός αυτού του συνόλου με δίκαιο τρόπο σε κάθε basic block. Για τους σκοπούς αυτής της διπλωματικής εργασίας, αρχικός κώδικας C χρησιμοποιείται για την δημιουργία εκτελέσιμων, των οποίων η ενέργεια μετριέται μέσω μετρητών ενεργείας της τεχνολογίας Intel RAPL, ενώ ταυτόχρονα αποθηκεύεται το ίχνος εκτέλεσης του εκτελέσιμου. Χρησιμοποιώντας τις υπολογισμένες τιμές ενέργειας και χωρίζοντας το ίχνος σε basic blocks, χρησιμοποιούνται στατιστικές μέθοδοι για να διαμοιραστεί δίκαια η ενεργεία σε όλα τα basic blocks.

Το παραγόμενο dataset είναι αντιπροσωπευτικό όχι μόνο για C κώδικα αλλά και για άλλες γλώσσες προγραμματισμού. Το τελικό dataset αποτελείται απο 3828 μοναδικά basic blocks, τα οποία προκύπτουν απο 24 διαφορετικά benchmarks προγραμμάτων C . Το μέσο σφάλμα για όλο το dataset ανέρχεται στο 2.63%. Αυτά τα αποτελέσματα είναι συγκρίσιμα με το πιο μοντέρνο αυτήν την στιγμή στο χώρο ALEA [1], με την δική μας δουλειά να είναι και open source.

Λέξεις Κλειδιά

LLVM , LLVM pass, intel Perf, intel Rapl, execution trace, energy efficiency, energy dataset, Basic Block, energy prediction, energy overhead

Abstract

Energy consumption is an emerging concern in multiple domains and fields of informatics, due to the monetary energy cost, the heat dissipation, the battery life of devices and environmental concerns.

Formerly, the energy consumption was mainly related to the used hardware, however, the running software is in fact as important as hardware, since it controls the behaviour of the hardware. The ultimate goal of this thesis is to help developers and practitioners understand and actively think about green software design in their work, in order to reduce the energy consumption of their software and deliver energy efficient products.

A prerequisite to energy efficiency though is energy estimation. To achieve this, we start with producing a reliable dataset that will then be used to create a energy predicting model.

The first step to producing an energy dataset on a basic block level is measuring the energy of a larger isolated software process and then distributing the energy on each basic block. For the purposes of this dissertation thesis C source code executables will be used and the energy of the binaries will be measured, through Intel RAPL energy counters, while the assembly execution traces are stored in parallel. Using the calculated values and by splitting the code trace into energy blocks, using statistical methods the energy will be distributed fairly to each basic block.

The produced dataset is actually very representative not only of C code but of any High level programming language and is made of 3828 unique basic blocks which are derived from 24 different benchmarks of C code. The average error for the totality of the dataset is equal to 2.63%. These results are comparable with the state of the art, ALEA [1], with the added benefit of being open source.

Keywords

LLVM , LLVM pass, intel Perf, intel Rapl, execution trace, energy efficiency, energy dataset, Basic Block, energy prediction, energy overhead

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον επιβλέποντα καθηγητή μου κύριο Δημήτριο Σούντρη για την ευκαιρία που μου έδωσε να εκπονήσω την διπλωματική μου εργασία στο εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων.

Επίσης ευχαριστώ ιδιαίτερα τον υποψήφιο Διδάκτορα Χρήστο Λαμπράκο για την καθοδήγησή του, τον χρόνο του, την ακούραστη υπομονή και το μεράκι του. Χωρίς αυτόν η διπλωματική αυτή εργασία δεν θα είχε καταφέρει να επιτύχει το σκοπό της και εγώ δεν θα είχα καταφέρει να αποκτήσω το πτυχίο μου.

Ακόμη, θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική αλλά και υλική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Φεβρουάριος 2023

Μπούρας Δημήτριος Σταμάτιος

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Η διπλωματική αυτή εργασία έλαβε χώρα από τον Οκτώβρη του 2021 μέχρι τον Φεβρουάριο του 2023. Πραγματοποιήθηκε στο χώρου του εργαστηρίου Μικρουπολογιστών και ψηφιακών συστημάτων στην Πολυτεχνειούπολη Ζωγράφου στα νέα κτήρια Ηλεκτρολόγων μηχανικών και μηχανικών υπολογιστών.

Με υπεύθυνο καθηγητή τον Δημήτριο Σουντρη και επιβλέπων τον υποψήιο Διδάκτορα Χρήστο Λαμπράκο αποφασίστηκε να πραγματοποιηθεί η συγκεκριμένη διπλωματική εργασία ώστε να διευρυνθούν τα όρια του χώρου έρευνας πάνω στην ενέργεια εκτέλεσης λογισμικού και να παραχθεί ένα εργαλείο το οποίο θα δώσει την δυνατότητα σε μηχανικούς υπολογιστών να δημιουργουν πιο ενεργειακά αποτελεσματικό λογισμικό.

Προτάθηκαν και δοκιμάστηκαν 3 διαφορετικοί μέθοδοι για την παραγωγή του dataset ενέργειας. Τελικά μόνο η μέθοδος που χρησιμοποιούσε το execution trace του υπό εξέταση προγράμματος ήταν επιτυχής. Βασική για οποιαδήποτε προσπάθεια μας ήταν η τεχνολογία της Intel, RAPL μέσω της οποίας μπορούμε να διαβάζουμε κάποιους counters ενέργειας. Με αφαίρεση της τιμής κάποιου τέτοιου μετρητή σε 2 διαφορετικές χρονικές στιγμές μπορούμε να υπολογίσουμε την ενέργεια που καταναλώθηκε σε αυτό το χρονικό διάστημα. Στην εργασία αυτή χρησιμοποιήθηκε ο μετρητής που υπολογίζει ενέργεια του επεξεργαστή. Επίσης εξίσσου σημαντική ήταν η υποδομή του μεταγλωτιστή LLVM ο οποίος μας δίνει την δυνατότητα να πραγματοποιούμε αυτόματα αλλαγές στο κώδικα μας.

Συγκεκριμένα η πρώτη μέθοδος που δοκιμάσαμε χρησιμοποίησε τα φίλτρα του LLVM για να προσθέσει διαβάσματα ενέργειας πριν από κάθε basic block του υπό εξέταση προγράμματος. Στη συνέχεια με αφαίρεση της ενέργειας πριν από ενα basic block από εκείνη πριν από το επόμενο basic block υπολογίζεται η ενέργεια του αρχικού basic block.

Η μεθοδολογία αυτή όμως αντιμετώπισε μερικά προβλήματα κατά την υλοποίησή της. Αρχικά η τιμή ενέργειας που μετριέται απο τα διαβάσματα RAPL αννανεώνεται με πιο αργό ρυθμό από ότι εκτελόυνται basic blocks. Έτσι βρήκαμε έναν τρόπο να μοιράσουμε την υπολογισμένη ενέργεια για ένα μεγαλύτερο χρονικό διάστημα σε κάθε basic block που εκτελέστηκε σε εκείνο το διάστημα. Η τεχνική αυτή βασίζεται στους κύκλους ρολογιού που χρειάζονται οι εντολές assembly από τις οποίες αποτελείται το κάθε basic block για να εκτελεστούν.

Επιπλέον, το διάβασμα ενέργειας κοστίζει και το ίδιο κάποια ενέργεια η οποία πρέπει να αφαιρεθεί από τις μετρήσεις που κάναμε. Ευτυχώς το κόστος αυτό παραμένει σταθερό και μπορεί να υπολογιστεί και στην συνέχεια να αφαιρεθεί. Τέλος, οι συναρτήσεις που ανοίκουν στις βιβλιοθήκες που συμπεριλαμβάνονται στο πρόγραμμα που εξετάζουμε, δεν λαμβάνονται υπόψη καθώς τα φίλτρα του LLVM λειτουργουν σε στάδιο πριν αυτές προστεθούν στο εκτελέσιμο αρχείο. Αυτό έχει ως αποτέλεσμα μεγάλο κομμάτι του συνολικού κώδικα και της ενέργειας του να χάνεται. Το πρόβλημα αυτό δεν ήταν εύκολο να αντιμετωπιστεί και έτσι οδηγηθήκαμε στην δεύτερη μέθοδο που υλοποιήσαμε.

Η δεύτερη μέθοδος βασίζεται στο lifting εκτελέσιμων σε LLVM IR. Δηλαδή, περιμέναμε πρώτα να συμπεριληφθούν οι συναρτήσεις των βιβλιοθηκών στο εκτελέσιμο, στη συνέχεια μετατρέπαμε το εκτελέσιμο πίσω σε μορφή συμβατή με τα φίλτρα του LLVM και επαναλαμβάναμε την μέθοδο 1. Η λογική αυτής της αντιμετώπισης είναι ορθή αλλά η τεχνολογία του Binary lifting βρίσκεται ακόμα σε αρχικό στάδιο και έτσι τα αρχέια που προέκυψαν μετά απο το πέρασμα από το φίλτρο του LLVM δεν ήταν λειτουργικά.

Η τρίτη μέθοδος μας, βασίστηκε στην αποθήκευση και μελέτη του trace του εκτελέσιμου με την τεχνολογία Intel Perf, αφού έχουν προστεθεί κλήσεις ενέργειας RAPL, και στην συνέχεια ο διαμοιρασμός της ενέργειας δίκαια ανάμεσα στα basic blocks του προγράμματος. Αυτή η μέθοδος ήταν επιτυχής και οδήγησε στην παραγωγή ενός dataset ενέργειας 3828 μοναδικών basic blocks από εκανοντάδες χιλίαδες αρχικές μετρήσεις.

Για να αξιολογήσουμε την δουλειά μας συγκρίναμε το αληθινό ενεργειακό κόστος εκτλέσης ενός προγράμματος μετρημένο με την τεχνολογία Intel Rapl, με το αθροισμα των ενεργειών των basic blocks από τα οποία αποτελείται όπως αυτές έχουν υπολογιστεί στο dataset μας. Το μέσο σφάλμα που προέκυψε ήταν πολύ ενθαρρυντικό, στο 2.64 %, συγκρίσιμο με την πιο καινοτόμα δουλειά που υπάρχει στον χώρο [1]. Σε αντίθεση, όμως, η δική μας μεθοδολογία δεν βασίζεται σε πιθανοτικές προσεγγίσεις, ενώ είναι και open source. Σε σύκριση με άλλες παρόμοιες εργασίες το σφάλμα μας είναι σαφώς μικρότερο και το επίπεδο αφαίρεσης για τις μετρήσεις μας (basic blocks) πολύ πιο λεπτομερές.

Ήδη η εργασία αυτή επεκτείνεται στο να παραχθούν dataset ενέργειας για μνήμη και κάρτες γραφικών, ενώ εκαπιδεύεται και ένα μοντέλο μηχανικής μάθησης πάνω στα δεδομένα μας το οποίο προβλέπει ενέργεια εκτέλεσης λογισμικού σε επίπεδο basic block. Μελλοντικά, άλλες εργασίες μπορούν να στηριχθούν στη δουλεία μας με σκοπό την μείωση της ενέργειας του λογισμικού μέσω χρήσης μηχανισμών μετασχηματισμού κώδικα.

Chapter 1

Introduction

1.1 The need for energy efficient systems

Modern times and lifestyle created numerous needs and usages, especially in the digital sector. Among others, one could easily mention connected devices, wearables, Internet of Things, smartphones, tablets, etc[3]. At the same time, many existing services have migrated, at least partially and sometime even totally, their activities to the Internet: online retailers, banking, advertising, video and music consumption and even public services.

All these new activities have increased the overall environmental footprint of the Information and Communication Technology (ICT) sector, which is estimated to be responsible for approximately up to 4% of the greenhouse gas (GHG) emissions worldwide in 2022 with an alarming growth rate [4]. This is more than double to both the aviation and the shipping industry , as well as more than half of the GHG emissions attributed to commercial buildings [5]. This is an disturbing trend , especially when taking into account the projected growth of the ICT sector, a trend that foreshadows a consequent growth to the GHG emissions attributed to it. Moreover, heat extraction is a also large issue for both portable and non-portable electronic systems. Finally, in recent time the operating costs for large electronic systems, such as data warehouses, have become a concern.

Recent years have witnessed a proliferation of low-power embedded devices [6] with power ranges of few milliwatts (battery-powered) to microwatts (batteryless), and a plethora of techniques have been produced that yield very successful results at designing energy efficient embedded systmes[7].Furthermore efficient commercial CPUs have become more and more available on the market [8], especially for mobile phones due to their battery needs [9] and industry level powerful processors such as the AMD EPYC series offer great power to energy ratios [10].

The capabilities and size of energy efficient systems continue to improve dramatically; however, improvements in battery density and energy harvesting have failed to mimic Moore's law. The battery energy density is the slowest trend in mobile computing and it does not scale exponentially .[11]

In more detail, the battery capacity has improved very slowly , with a factor of 2 to 4

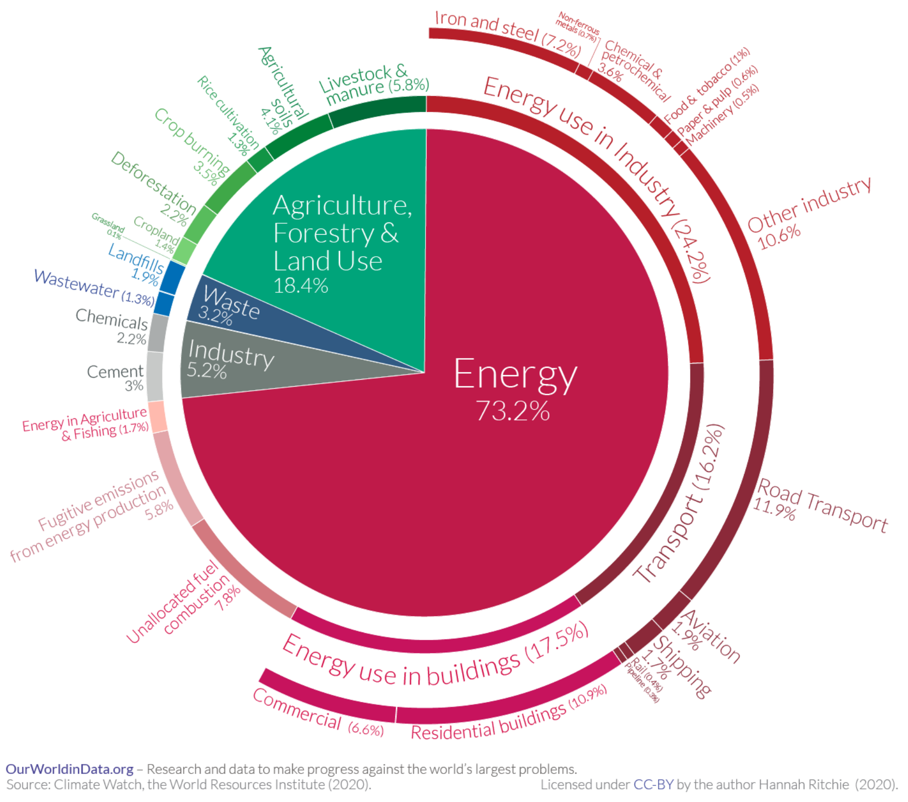


Figure 1.1: *Greenhouse gas emissions per sector.ICT is responsible for up to 4% of the total.*

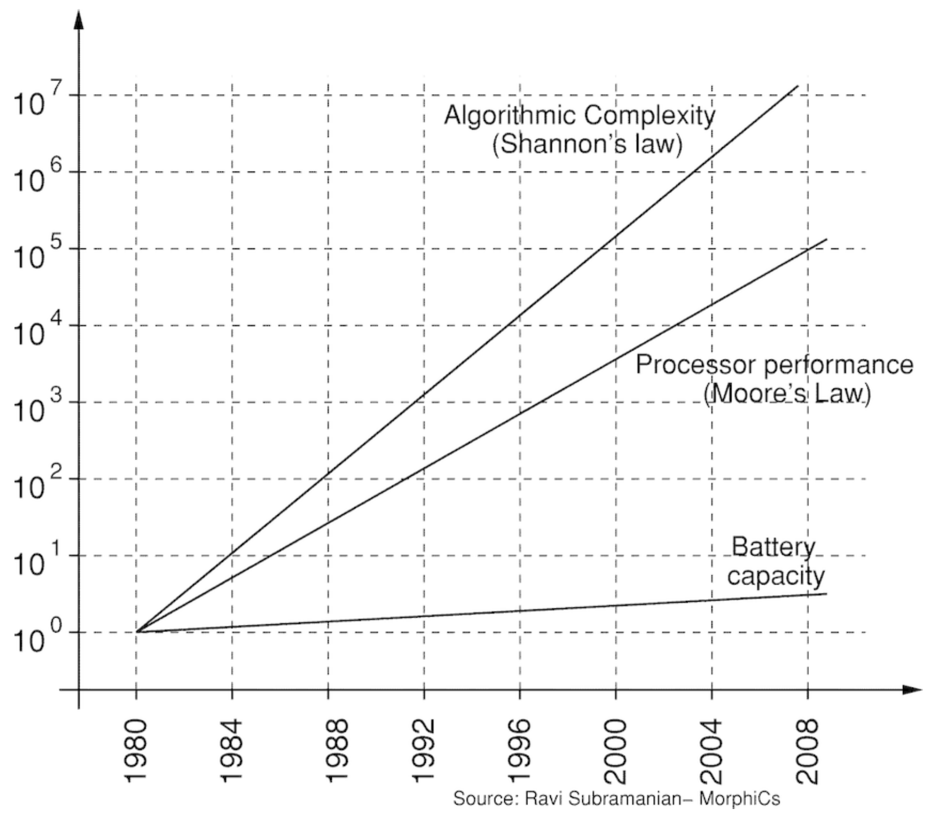


Figure 1.2: *Moore Law compared to battery capacity increase*

over the last 30 years, while the computational demands have drastically increased over the same time frame. There is also a fear that energy efficiency improvements of the last years will not be sustained in the future, as the ”low hanging fruits” have already been harvested, and that the continued increase in computing power might not be offset in the coming years. Thus, energy remains a formidable bottleneck for modern systems , especially embedded systems and mobile devices. For example, circuits of a smart lens can be miniaturized enough to be implanted inside an eye however,its batteries are not made small enough for such implantation [12].

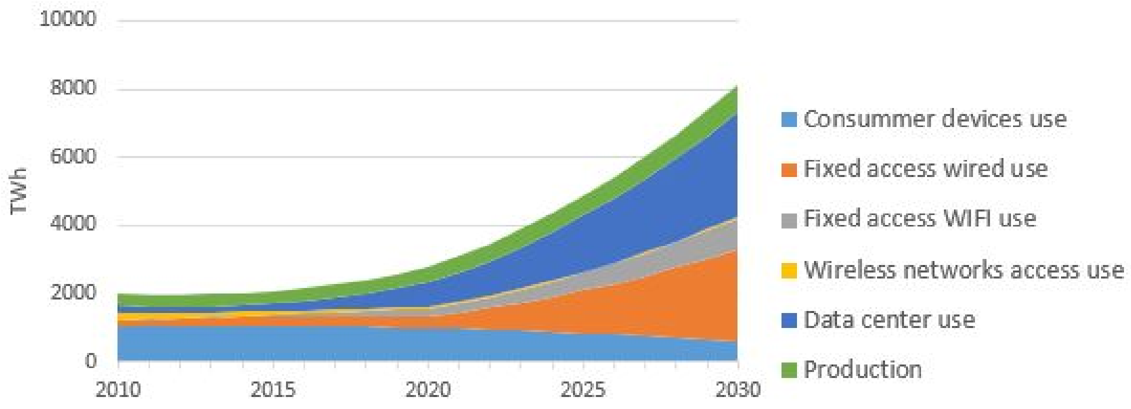


Figure 1.3: *Distribution of energy consumption of ICT sector*

It has become evident that the ability of energy efficient hardware to satisfy the increasing computational needs of the market while keeping the energy and power needs stable or even decreasing them has turned into an uphill battle. Thus the focus of more and more research has turned into the field of energy efficient software . Reducing software energy consumption (SEC) is very important to enhance the energy efficiency of the ICT sector and its CO2 impact. Numerous researches provided studies to reduce SEC .[13, 14, 15]. These studies focused on multiple aspects of the assessment of software energy, such as the accuracy and granularity of the measurements or the different ways of reducing SEC [16], for example.

The main purpose of our thesis is to help developers to produce software that consumes less energy. We believe that promoting GSD (Green Software Design) and SEC considerations should go through an important educational phase, to ensure that the community in general and developers, in particular, are well motivated and aware of the stakes of GSD and their prominent role in reducing SEC and building less consuming software. This is easier said than done, as there is not enough knowledge on how to build ”green” software that can carry developers’ choices for every use case. Small enhancements and insights in this topic are still very welcome to constitute a broad and robust set of knowledge that can be used to reduce SEC.

1.2 Thesis Overview and Contribution

Rather than trying to create hardware with ideal energy proportionality, the thesis of this work is that we can try to approach the energy reduction problem from a software perspective.

The first step towards more energy efficient software production is an understanding of the energy consumption by the software and a mechanism to predict the energy footprint of the code in question. Our work is summarized as the production of a reliable energy dataset on a basic block level, so that a high level of granularity will be achieved, but this title belies the importance and functionality of this project. To be more precise our contribution is not solely a dataset for a specific computer architecture and CPU but rather an energy dataset production mechanism that uses well established and supported open source tools that can be used to create a new dataset for any Intel based computer architecture (the most popular architecture with over 70% of the market share on commercial CPUs). In that way any developer can use our mechanism to produce a reliable energy dataset for his own computer architecture, without any monetary cost and without the need for a high level of knowledge and skills on the field, not to mention the speed of the whole process.

Our dataset and any other dataset produced using our methodology can then be used as a reference for developers in their effort to minimize the energy of their software. Additionally a complimentary work on this thesis will provide a machine learning model architecture with a high degree of accuracy that can be trained on the dataset our methodology produces. So with a simple process as shown on figue 1.4 any developer can create an energy predicting mechanism, custom to his specific computer architecture, which can then serve as the basis for a number of energy reducing techniques such as code transformations, or simply be used as an analysis tool.

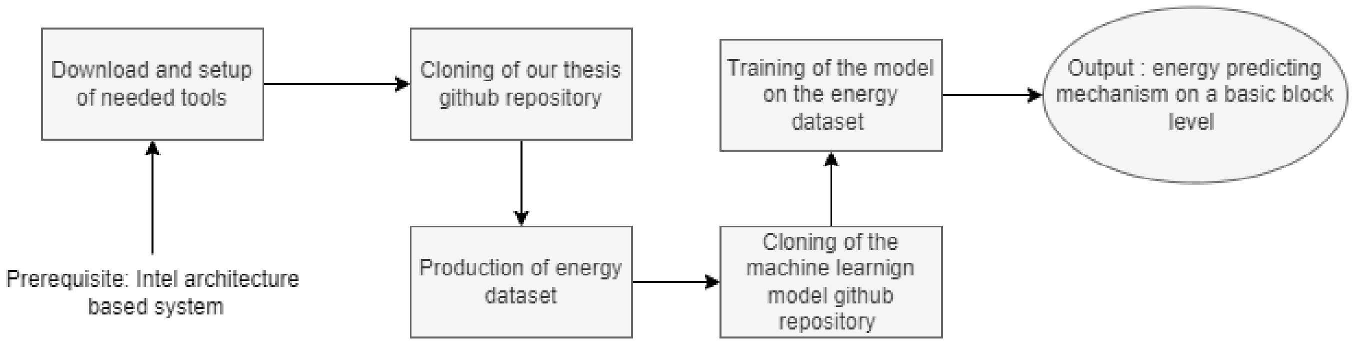

Download and setup of needed tools
Cloning of our thesis github repository
Training of the model on the energy dataset
Output : energy predicting mechanism on a basic block level
Prerequisite: Intel architecture based system
Production of energy dataset
Cloning of the machine learnign model github repository

Figure 1.4: *Process for producing custom energy producing mechanism*

Our reasoning for choosing this specific subject, was the lack of energy predicting mechanisms, and in particular open source and customisable ones. We hope that with this work we provide the ability and the incentive on more developers to produce custom software for their systems with energy efficiency at heart, so that they can benefit from heat and monetary savings, particular during the current energy crisis, while our planet can also benefit from reduced GHG emissions, in particular CO_2.

1.3 Problem Statement

The problem of digital energy consumption is critical. Numerous research results and estimations highlighted the urgent situation that needs to be truly considered. In fact, future ICT infrastructures will increase their energy consumption , even though the cost per computation is projected to decrease. This can be attributed to the increase number of operations per time that future computers will be able to perform. After all, ICT technology will continue to evolve and provide faster, more accurate and overall better machines. But unfortunately , not more energy efficient, not unless our priorities as far as the desired characteristics of computers evolve towards energy efficiency.

Similarly to hardware, software energy consumption plays a crucial role in the global energy consumption of ICT devices. It is however a very complex problem to identify the hotspots of software and reduce its energy consumption. SEC is mostly considered as a young topic. Although more and more research is directed towards this topic, the current situation is still not mature enough to provide concrete solutions for multiple use cases, and guide developers to reduce the energy consumption of the produced software.

Ultimately, the real problem is that software energy consumption is not taken seriously enough by developers and is considered of secondary importance compared to software performance. But this cannot continue to be the case. For things to change, SEC should have similar importance and significance to software performance. What is the reason, though that the priorities of the scientific community and market alike have been established this way? One major reason is that seeking fast software and reducing the execution/response time was always requested, as the results were immediately perceived. On the other hand, the energy resource was not considered to be as critical as execution time for quite a while. Now that the myth of infinite resources is not really arguable, and even with renewable energy sources, developers should put more attention to the energy efficiency of what they are producing.

One major problem next to the lack of knowledge is the bad communication and popularization of the acquired knowledge. In fact, many studies' results are not intended for developers. This implies that no mature open source tool is made available to developers to assist them in reducing or even analysing the energy consumption of their software as it is the case for performance . In this thesis, we try to tackle some of the previous problems, by providing the developers the tool to analyze their software's energy and in this way put them in the position to be more mindful of its energy consumption.

1.4 Objective

Our main goal is to help developers to reduce the energy consumption of the developed software. To approach this, we constitute a set of sub-objectives that will drive our thesis. Firstly we need to understand what would be the appropriate granularity for energy measurements to be performed. Then we need to find the most fitting tools to create an energy measurement pipeline or more accurately a custom energy dataset production pipeline. In what way can we distribute the energy measured reliably and accurately at the granularity level we previously chose. Concretely, we want to answer the following research questions:

1. What software objects should we correspond to energy (Functions, Basic blocks, high level language commands, assembly commands etc)?
2. What tools should we use and how to connect them together so that we can reach our objective and make it easy for others to replicate our work?
3. Once energy is measured at some level, how to distribute it fairly to the level we chose at question 1?

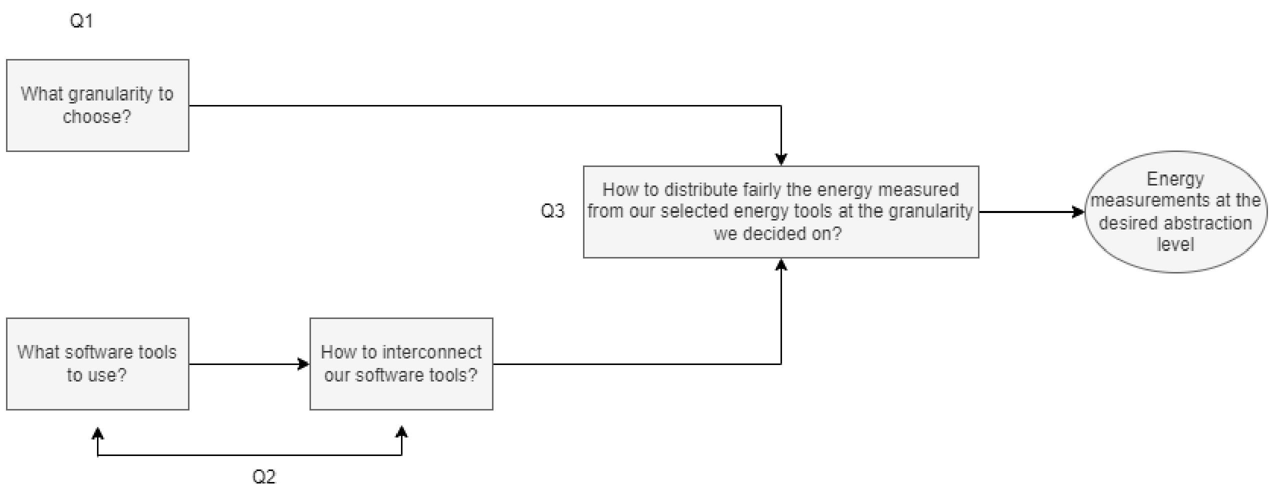

Q1
What granularity to choose?
Q3
How to distribute fairly the energy measured from our selected energy tools at the granularity we decided on?
Energy measurements at the desired abstraction level
What software tools to use?
How to interconnect our software tools?
Q2

Figure 1.5: *Diagram of our objective , split into research questions*

1.5 Organization

The thesis is organized into X chapters, each focusing on a different aspect of the research performed.

- Chapter 2 introduces the tools that have been chosen, answers why they are ideal for our project, how they will be utilised and offers a guide on how to use them.
- Chapter 3 focuses on the first approach we implemented which was based on making consecutive RAPL measurements between basic block executions.
- Chapter 4 explains our second approach that involved binary lifting of statically linked executables, to alleviate some of the problems that our first approach introduced.

- Chapter 5 contains our final and successful implementation that focused on execution tracing and maintained the working parts of our first approach , then used statistical methods to correspond basic blocks to energy.
- Chapter 6 summarizes our results, evaluates them and compares them with similar works.
- Chapter 7 focuses on the continuation of this work, that is already currently under way and proposes further expansions of this thesis.

Chapter 2

Tools and Technologies

2.1 LLVM

2.1.1 What is LLVM?

LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.[17]

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and ”idle-time” optimizations. Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM (or which do not directly use LLVM but can generate compiled programs as LLVM IR) include: ActionScript, Ada, C#,Common Lisp, PicoLisp, Crystal, CUDA, D, Delphi, Dylan, Forth, Fortran, Free Basic, Free Pascal, Graphical G, Halide, Haskell, Java bytecode, Julia, Kotlin, Lua, Objective-C, OpenCL, PostgreSQL's SQL and PLpgSQL, Ruby, Rust, Scala, Swift, XC, Xojo and Zig.

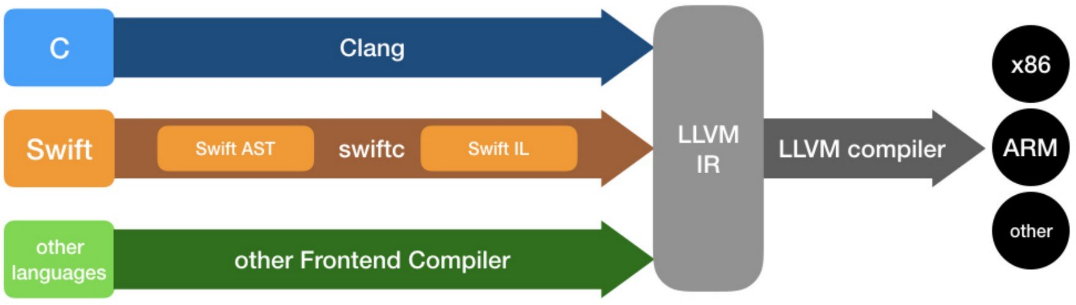


Figure 2.1: *The LLVM project's Architecture*

The name LLVM was originally an initialism for Low Level Virtual Machine. However, the LLVM project evolved into an umbrella project that has little relationship to what most current developers think of as a virtual machine , and since 2011 LLVM is ”officially no longer an acronym” , but a brand that applies to the LLVM umbrella project.The project encompasses the LLVM intermediate representation (IR), the LLVM debugger, the LLVM implementation of the C++ Standard Library .

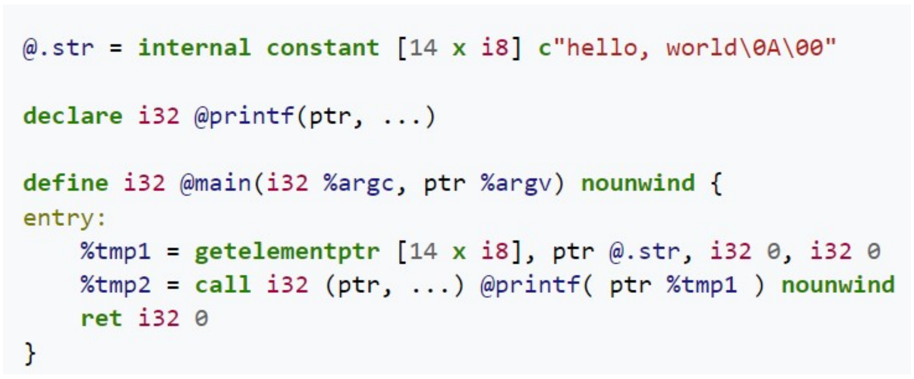
LLVM can provide the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. LLVM can accept the IR from the GNU Compiler Collection (GCC) toolchain, allowing it to be used with a wide array of existing compiler front-ends written for that project. LLVM can also generate relocatable machine code at compile-time or link-time or even binary machine code at run-time.

LLVM supports a language-independent instruction set and type system. Each instruction is in static single assignment form (SSA), meaning that each variable (called a typed register) is assigned once and then frozen. This helps simplify the analysis of dependencies among variables. LLVM allows code to be compiled statically, as it is under the traditional GCC system, or left for late-compiling from the IR to machine code via just-in-time compilation (JIT), similar to Java. The type system consists of basic types such as integer or floating-point numbers and five derived types: pointers, arrays, vectors, structures, and functions. A type construct in a concrete language can be represented by combining these basic types in LLVM. For example, a class in C++ can be represented by a mix of structures, functions and arrays of function pointers.

2.1.2 LLVM IR

The core of LLVM is the intermediate representation (IR), a low-level programming language similar to assembly. IR is a strongly typed reduced instruction set computer (RISC) instruction set which abstracts away most details of the target. For example, the calling convention is abstracted through call and ret instructions with explicit arguments. Also, instead of a fixed set of registers, IR uses an infinite set of temporaries of the form %0, %1, etc. LLVM supports three equivalent forms of IR: a human-readable assembly format an in-memory format suitable for frontends, and a dense bitcode format for serializing

The many different conventions used and features provided by different targets mean that LLVM cannot truly produce a target-independent IR and re-target it without breaking some established rules.



```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(ptr, ...)

define i32 @main(i32 %argc, ptr %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], ptr @.str, i32 0, i32 0
    %tmp2 = call i32 (ptr, ...) @printf( ptr %tmp1 ) nounwind
    ret i32 0
}
```

Figure 2.2: *Example LLVM IR of a simple "Hello world!" program*

2.1.3 Clang

The compiler frontend we chose was Clang, which operates in tandem with the LLVM compiler back end and has been a subproject of LLVM 2.6 and later. As with LLVM, it is free and open-source software under the Apache License 2.0 software license.[18]

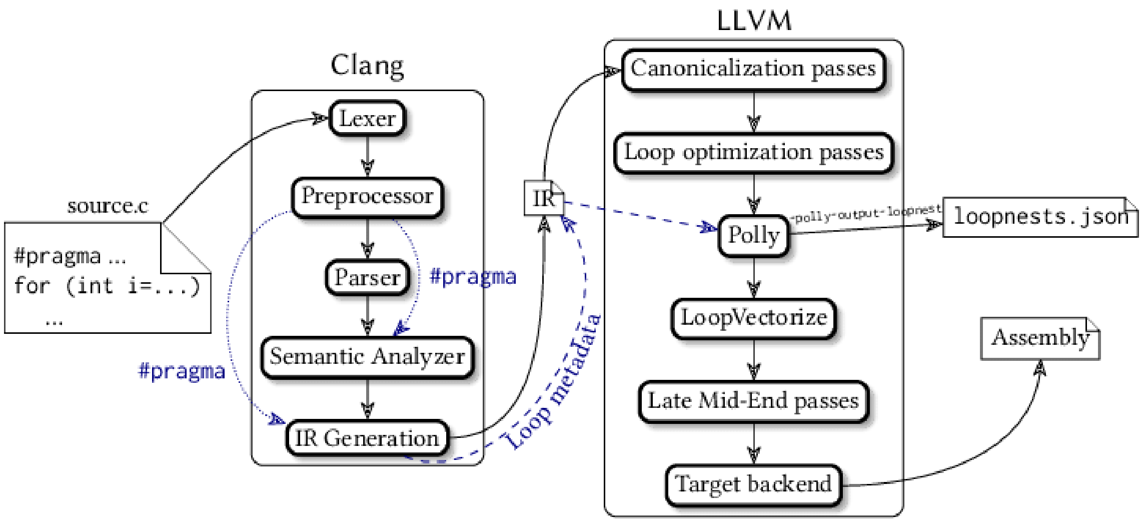


Figure 2.3: *Clang's place in the LLVM architecture*

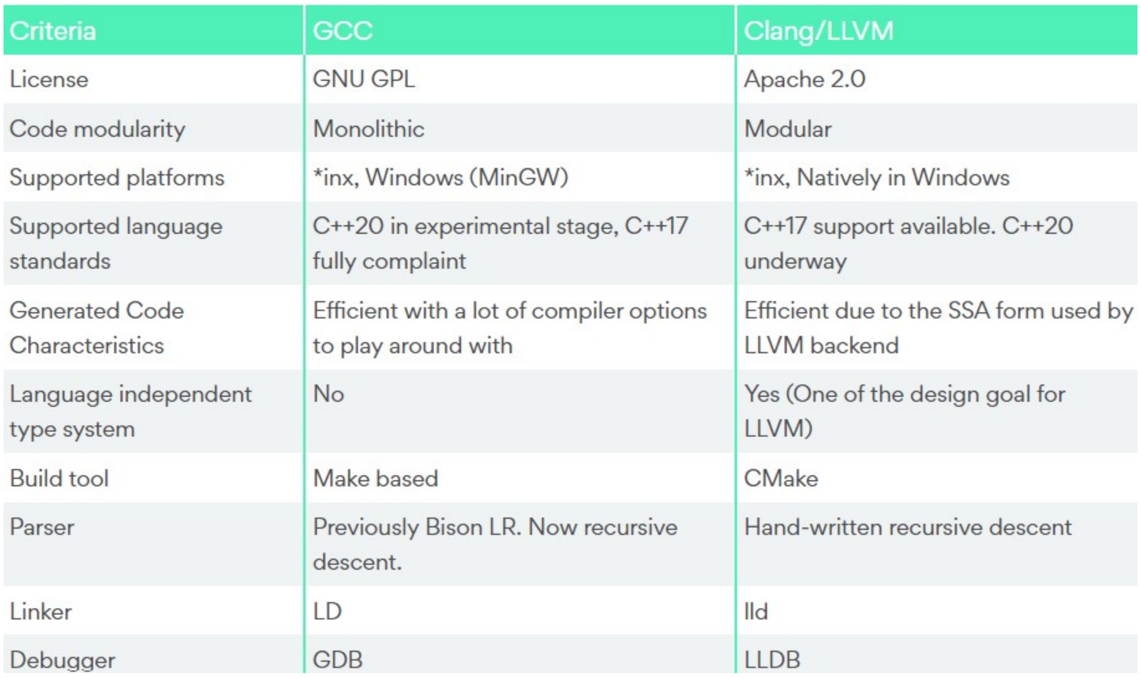
Clang is a compiler front end for the C, C++, Objective-C, and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA and HIP frameworks. It acts as a drop-in replacement for the GNU Compiler Collection (GCC), supporting most of its compilation flags and unofficial language extensions. It includes a static analyzer, and several code analysis tools.[15].

The Clang Compiler has been designed to work just like any other Compiler. Clang works in three different stages. The first stage is the front end that is used for parsing source code. It checks the code for errors and builds a language-specific Abstract Syntax Tree (AST) to work as its input code. The second stage is the optimizer that is used for optimizing the AST that was generated by the frontend. The third and final stage is the back end. This is responsible for generating the final code to be executed by the machine which can depend on the target.

2.1.4 Why LLVM and not GCC?

LLVM and the GNU Compiler Collection (GCC) are both compilers. The difference is that GCC supports a number of programming languages while LLVM isn't a compiler for any given language. LLVM is a framework to generate object code from any kind of source code. While LLVM and GCC both support a wide variety of languages and libraries, they are licensed and developed differently. LLVM libraries are licensed more liberally and GCC has more restrictions for its reuse.

When it comes to performance differences, GCC has been considered superior in the past. But clang is gaining ground. The most important reason that we chose clang and LLVM though, was that they provided as with the ability to use "passes" to automatically change the code we wanted to measure after the IR had been produced, to strategically add energy measurements.



Criteria	GCC	Clang/LLVM
License	GNU GPL	Apache 2.0
Code modularity	Monolithic	Modular
Supported platforms	*inx, Windows (MinGW)	*inx, Natively in Windows
Supported language standards	C++20 in experimental stage, C++17 fully complaint	C++17 support available. C++20 underway
Generated Code Characteristics	Efficient with a lot of compiler options to play around with	Efficient due to the SSA form used by LLVM backend
Language independent type system	No	Yes (One of the design goal for LLVM)
Build tool	Make based	CMake
Parser	Previously Bison LR. Now recursive descent.	Hand-written recursive descent
Linker	LD	lld
Debugger	GDB	LLDB

Table 2.1: *LLVM and GCC comparison*

2.1.5 LLVM Passes

The LLVM Pass Framework is an important part of the LLVM system, because LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code. They can be used to mutate the IR code (for example add instructions) or compute properties about it (such as number of functions).[19]

All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. Depending on how your pass works, you should inherit from the ModulePass , CallGraphSCCPass, FunctionPass , or LoopPass,or RegionPass classes, which gives the system more information about what your pass does, and how it can be combined with other passes. One of the main features of the LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets (which are indicated by which class they derive from).

2.2 Intel RAPL

2.2.1 Running Average Power Limit

Most modern processors, including Intel processors, provide Running Average Power Limit (RAPL) interfaces for reporting the accumulated energy consumption of various power domains. RAPL provides a set of counters providing energy and power consumption information. RAPL is not an analog power meter, but rather uses a software power model. This software power model estimates energy usage by using hardware performance counters and I/O models. Based on research performed, they match actual power measurements [20].

RAPL also provides a way to set power limits on processor packages and DRAM. This will allow a monitoring and control program to dynamically limit max average power, to match its expected power and cooling budget. In addition, power limits in a rack enable power budgeting across the rack distribution. By dynamically monitoring the feedback of power consumption, power limits can be reassigned based on use and workloads. Because multiple bursts of heavy workloads will eventually cause the ambient temperature to rise, reducing the rate of heat transfer, one uniform power limit can't be enforced. RAPL provides a way to set short term and longer term averaging windows for power limits. These window sizes and power limits can be adjusted dynamically. [21]

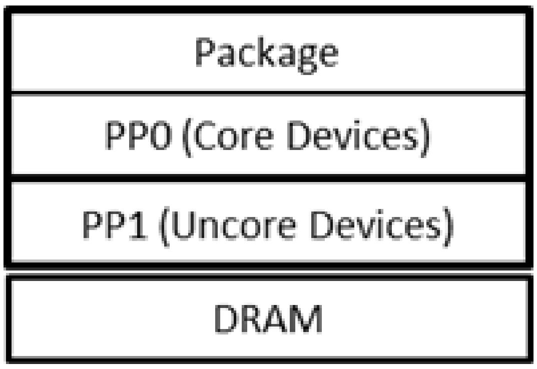
In our thesis RAPL will be used solely as an energy measuring tool and not to set power limitations. The energy measurements we derive from the RAPL counters will be the basis on which the energy dataset will be created.

2.2.2 RAPL Domains

In RAPL, platforms are divided into domains for fine grained reports and control. A RAPL domain is a physically meaningful domain for power management. The specific RAPL domains available in a platform vary across product segments. [22]

Each RAPL domain supports:

- ENERGY_STATUS for power monitoring.
- POWER_LIMIT and TIME_WINDOW for controlling power
- PERF_STATUS for monitoring the performance impact of the power limit
- RAPL_INFO contains information on measurement units, the minimum and maximum power supported by the domain



Package
PP0 (Core Devices)
PP1 (Uncore Devices)
DRAM

Table 2.2: *RAPL Domains*

In this thesis , we have worked with the PP0 counters to create an energy dataset of the energy that is being consumed by the CPU. With minor changes, a corresponding dataset can be created for the remaining 3 domains and we plan to perform those measurements in a future time, and thus create a four-part energy dataset , that could enable developers to perform software energy reductions for specific hardware components (CPU, RAM, GPU et.c)

2.3 Intel Perf

2.3.1 What is Intel Processor Trace?

Intel Processor Trace (Intel PT) [23] is an extension of Intel Architecture that collects information about software execution such as control flow, execution modes and timings and formats it into highly compressed binary packets. Technical details are documented in the Intel 64 and IA-32 Architectures Software Developer Manuals, Chapter 36 Intel Processor Trace[24].

Intel PT is first supported in Intel Core M and 5th generation Intel Core processors that are based on the Intel micro-architecture code name Broadwell. [25]

Trace data is recorded and then it must be decoded which involves walking the object code and matching the trace data packets. For example a TNT packet only tells whether a conditional branch was taken or not taken, so to make use of that packet the decoder must know precisely which instruction was being executed. Decoding is done on-the-fly. The decoder outputs samples in the same format as samples output by perf hardware events, for example as though the ”instructions” or ”branches” events had been recorded. Presently 3 tools support this: perf script, perf report and perf inject.

The main distinguishing feature of Intel PT is that the decoder can determine the exact flow of software execution. Intel PT can be used to understand why and how did software get to a certain point, or behave a certain way. The software does not have to be recompiled, so Intel PT works with debug or release builds, however the executed images are needed - which makes use in JIT-compiled environments, or with self-modified code, a challenge. Also symbols need to be provided to make sense of addresses.

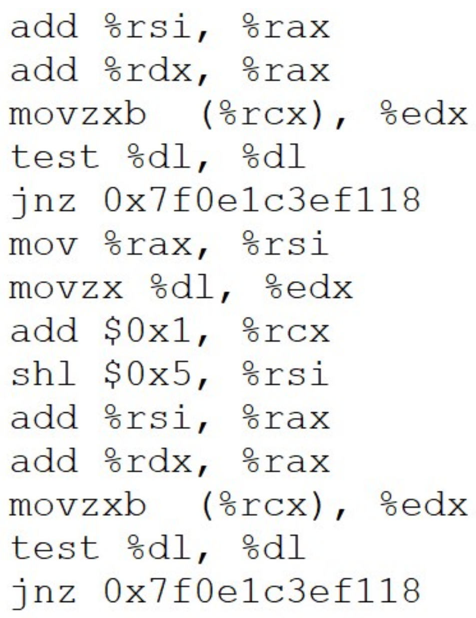
A limitation of Intel PT is that it produces huge amounts of trace data (hundreds of megabytes per second per core) which takes a long time to decode, for example two or three orders of magnitude longer than it took to collect. Another limitation is the performance impact of tracing, something that will vary depending on the use-case and architecture.

2.3.2 How is Intel Perf right for us?

We will use the intel PT technology to get the trace of a binary that we want to measure .To undersatnd how to do that we must first understand the capabilities of the perf record and perf scripts commands.
Perf record : this command runs a command and gathers a performance counter profile

from it, into perf.data - without displaying anything. This file can then be inspected later on, using perf report or script.

Perf script : This command reads the input file and displays the trace recorded. After installing the xed tool , we can dump all instructions in a long trace. That can be fairly slow, but it is the best way to get a complete and detailed trace of the executed binary.



```
add %rsi, %rax
add %rdx, %rax
movzxb  (%rcx), %edx
test %dl, %dl
jnz 0x7f0e1c3ef118
mov %rax, %rsi
movzx %dl, %edx
add $0x1, %rcx
shl $0x5, %rsi
add %rsi, %rax
add %rdx, %rax
movzxb  (%rcx), %edx
test %dl, %dl
jnz 0x7f0e1c3ef118
```

Figure 2.4: *Example of execution trace by the Intel Perf technology*

2.4 System Specifications

For this specific project the computer and specifically the CPU specification are of paramount importance, since those characteristics are a determining factor on the energy consumed. Thus the energy dataset that will be produced in this work will be accurate for computers and CPUs with identical or at least similar specifications. At the following table we will present important specifications concerning the specific computer we used. In that way, any interested developer that wants to use the dataset to measure the energy of his software can compare the specification of his machine to the one we used and if they are similar enough he can safely use the dataset we produced. Of course the more different his system and hardware are the least accurate the predictions will be. For maximum accuracy it is recommended that any interested developer utilizes our open-source tool to create an energy dataset, custom for their computer.

The most important thing that one must take into account is that we use an Intel i7 6th Gen processor so systems equipped with i7 or even i3 or i5 processors particularly of the 6th generation are going to be the suitable for prediction with our dataset.

In general one must keep in mind that if the system whose software energy they want to predict does not include an Intel based CPU they will be unable to produce their own custom dataset since, the RAPL technology required is restricted solely to Intel based systems. The best solution for non intel systems is to use our provided dataset or produce a dataset for a system similar to theirs but with an Intel CPU and use that taking into account that the accuracy of their predictions will be reduced.

Specifications of the computer utilized for this thesis	
Operating System	Linux Ubuntu 20.04.1
Core version	5.15.0-58 -generic
Architecture	x86$_6$4
Processor	Intel Core i7 – 6700
Processor frequency	3.4 GHz
Bridge	Xeon E3-1200 v5/E3-1500 v5/6th Gen
Bus	100 Series/C230 3.0 xHCI Controller
Communication	100 Series/C230 3.0 MEI Controller 1
RAM	32GiB System memory

Table 2.3: *Computer Specifications*

Chapter 3

First approach - unique basic block RAPL Reads

3.1 Initial Idea

3.1.1 Problem statement

As we have discussed in the introduction chapter the purpose of this work is to essentially map energy to basic Blocks of executed code. To do that we must first have a way to measure energy that is being consumed and then have a way to find out which basic block was executed while this energy was measured.

So in essence we perform a process like the one presented in the figure 3.1, where we measure energies at certain times during the execution of a program and we get information about the content (in assembly instructions) and the number of the basic blocks in the executable. But the real problem as stated earlier is to correspond energy to basic block.

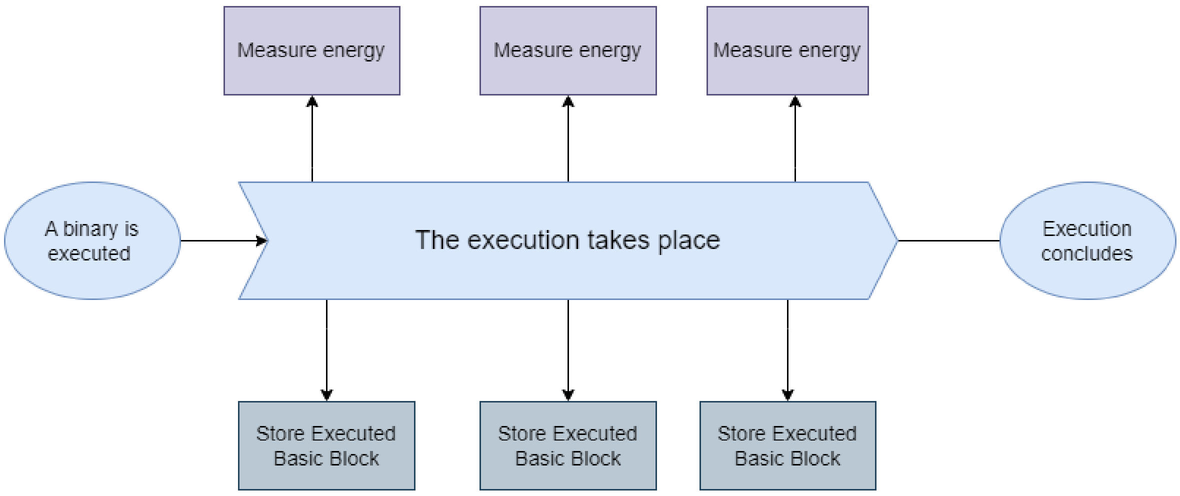

Measure energy
Measure energy
Measure energy
A binary is executed
The execution takes place
Execution concludes
Store Executed Basic Block
Store Executed Basic Block
Store Executed Basic Block

Figure 3.1: *Initial idea of measuring energy and finding basic blocks*

3.1.2 Solution

The solution that we propose is pretty simple and intuitive . The main concept is that we perform an energy read before each basic block that is being executed . Then to measure the energy of that specific basic block, we get the energy of the measurement that happened right before its execution (we call this **energy_before**) and the energy of the measurement that happened right before the execution of the next basic block, meaning right after the execution of the one we want to examine (we call this **energy_after**). The simple subtraction of the **energy_after**r - **energy_before** provides us with the energy that was consumed by the execution of the basic clock we are examining.

$$energy\_of\_BB = energy\_after - energy\_before$$

This process can be very easily understood by the diagram we are presenting later 3.2 as well. As we can see between the execution of each basic block, two RAPL measurements take place. In that way the subtraction of the former from the latter gives us the result we are looking for.

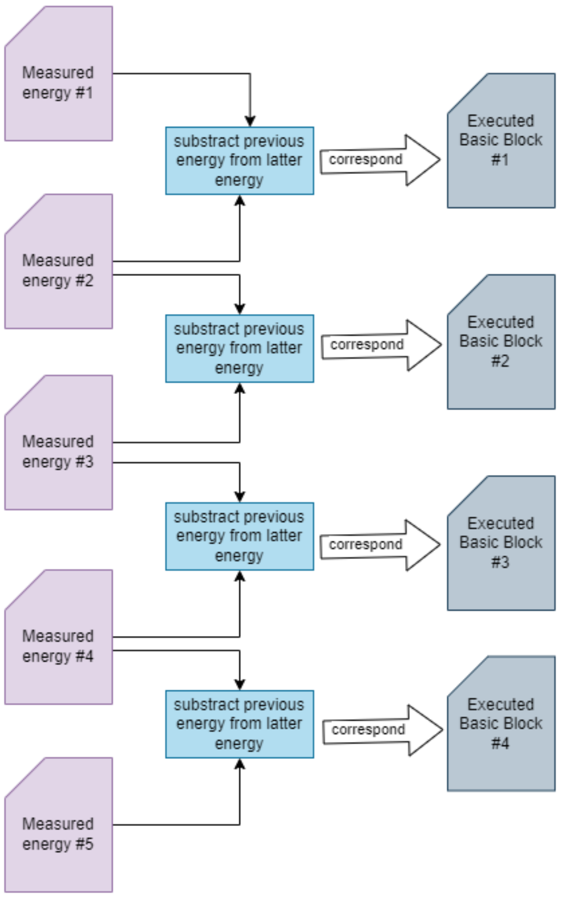


Figure 3.2: *Energy prediction through subtraction of 2 energy measurements*

3.2 Implementation

The implementation can be split into three parts:

- The measurement of the energy
- The splitting of the executable in basic blocks and the acquisition of the assembly instructions that make them up
- The correspondence of each basic block to an energy measurement

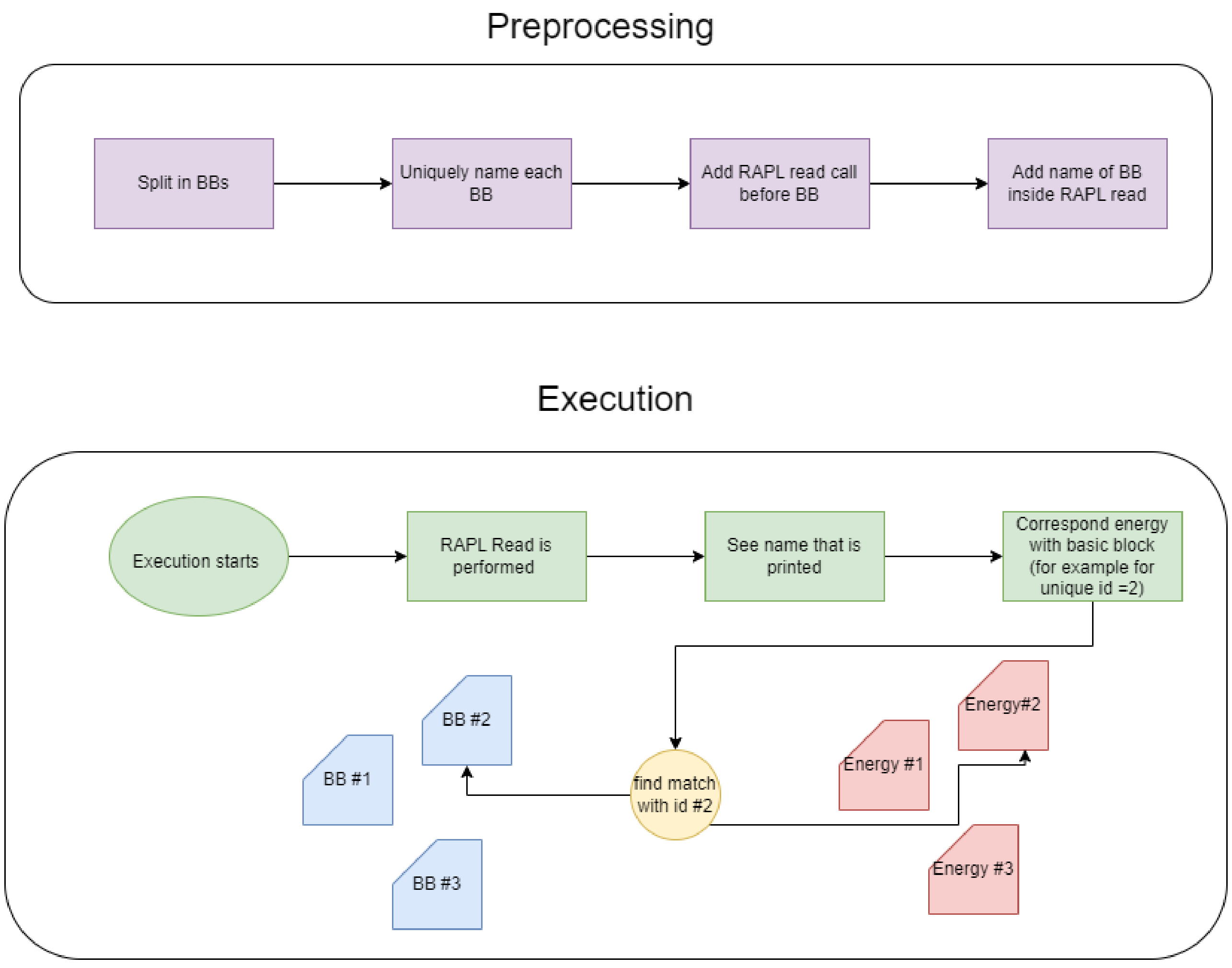


Figure 3.3: *Strategy overview - RAPL read between basic blocks*

3.2.1 RAPL Reads

First lets consider what an energy measurement actually means. As we discussed earlier, the Intel RAPL technology updates an 32-bit register with an energy value at a very low frequency (around a thousand updates each second - one every 1 ms). The value that is stored at this register corresponds to energy units. Those energy units for Skylake generation processors, as is the one we are using correspond to 61 μJ, but for the purposes of this thesis we will simply refer to them as energy units.

So to put it simply, when we talk about performing an energy measurement or otherwise referred to as a RAPL read, we are simply reading the specific register (in this case the register that measures the CPU energy) and getting the last updated value.

This value alone is useless to us. It is nothing more that a number that corresponds to the energy the CPU has consumed since it started counting. But since we have no idea when it started measuring energy we cannot use this value to make any meaningful deduction. This changes, though when a second RAPL read is performed and the energy of the register is read again. Now having 2 energy values at 2 different points in time we can calculate the energy that was consumed from the point the first was measured to the point the second was measured through a substraction. This will be the basis, of the energy measurements we perform in this thesis, the subtraction between the values of an energy register.

3.2.2 LLVM Pass

We will use the LLVM compiler's ability to create Passes that transform and analyse code. In more detail, we want to realise the 3 aspects of the implementations. We saw how an energy measurement can be performed but how can we get the basic blocks and their contents and how can we correspond the Basic Blocks with energy measured. Here is when LLVM passes will be very useful.

Firstly we will create an LLVM analysis pass. This passes purpose will be to split the program into Basic Blocks and will correspond to each basic block a unique identifier so that it is possible for us later to identify the specific Basic Block. Thankfully the LLVM libraries have already built in functions that help make this task possible.

Considering that each function in a program has a unique name we chose the unique identifier of a basic block to be the function name in which it resides plus a number corresponding to the order in which it appears into the function. For exaple the second basic block of the function **hello** will be named **hello_2**.

Secondly we must tackle the problem of inserting an energy measurement before each basic block. LLVM passes come to the rescue once again. The LLVM compiler provides us with the ability to create new instructions and insert them at specific parts of the code. So we will firstly write a function (in the high level language we are using) that performs the act of a RAPL read and then using LLVM we will create an instruction which is a call instruction to the function we just created. We will insert this instruction then

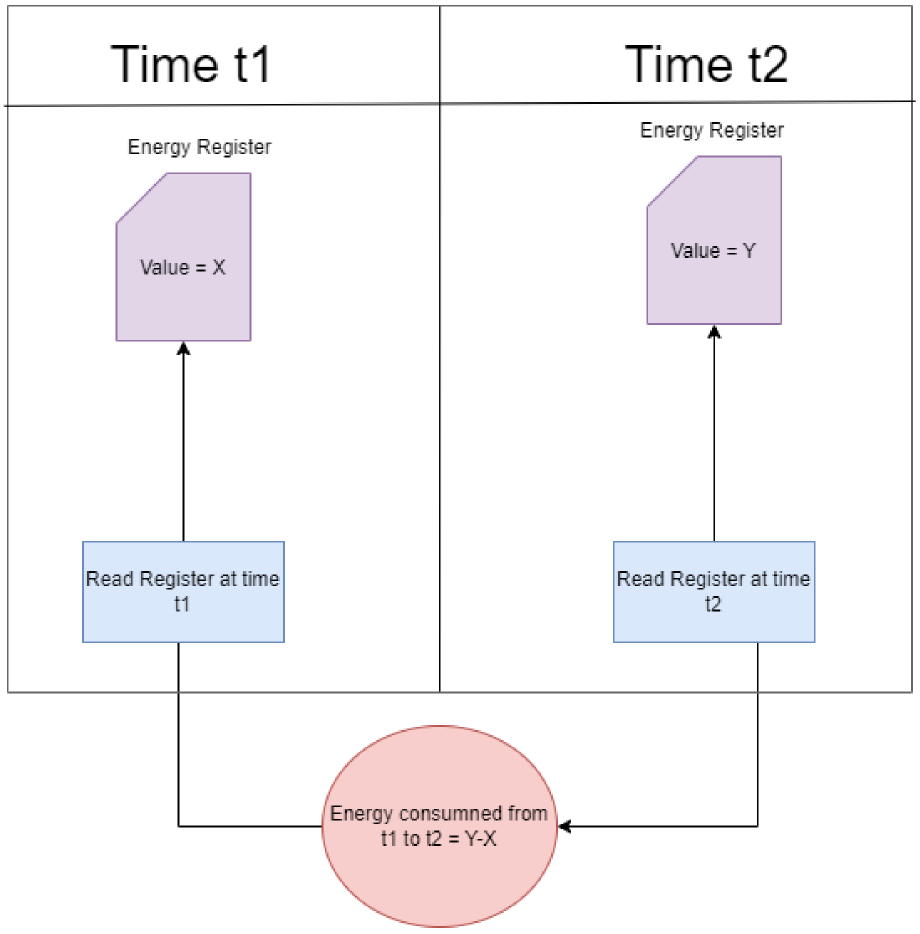


Figure 3.4: *Energy measurement with Intel RAPL visualized*

before the start of everu basic block. In that way before the execution of each basic block the function that measures energy (let's refer to it as **Rapl_read function**) will be called and an energy measurement will be performed. The energy measured will be stored at a file of our choice.

So now every time the executable is run a file with a list of RAPL read energy values will be created. We also have a file with the basic blocks, their IDs and their contents in the form of assembly instructions. But at this point we have no way of corresponding an energy value to a basic block.

To do that we will modify the LLVM pass and the **RAPL_read function**. Specifically we will give as an argument to the **RAPL_read function** the id of the basic block before which it is inserted. And the **RAPL_read function** will be modified to print the name of the Basic block before which it is inserted to the energy value file after it prints the energy. Now we have an energy file with energy values and basic block IDs alternatively. The energy consumed by the execution of a basic block is the energy that appears after its ID minus the energy that appears before its ID.

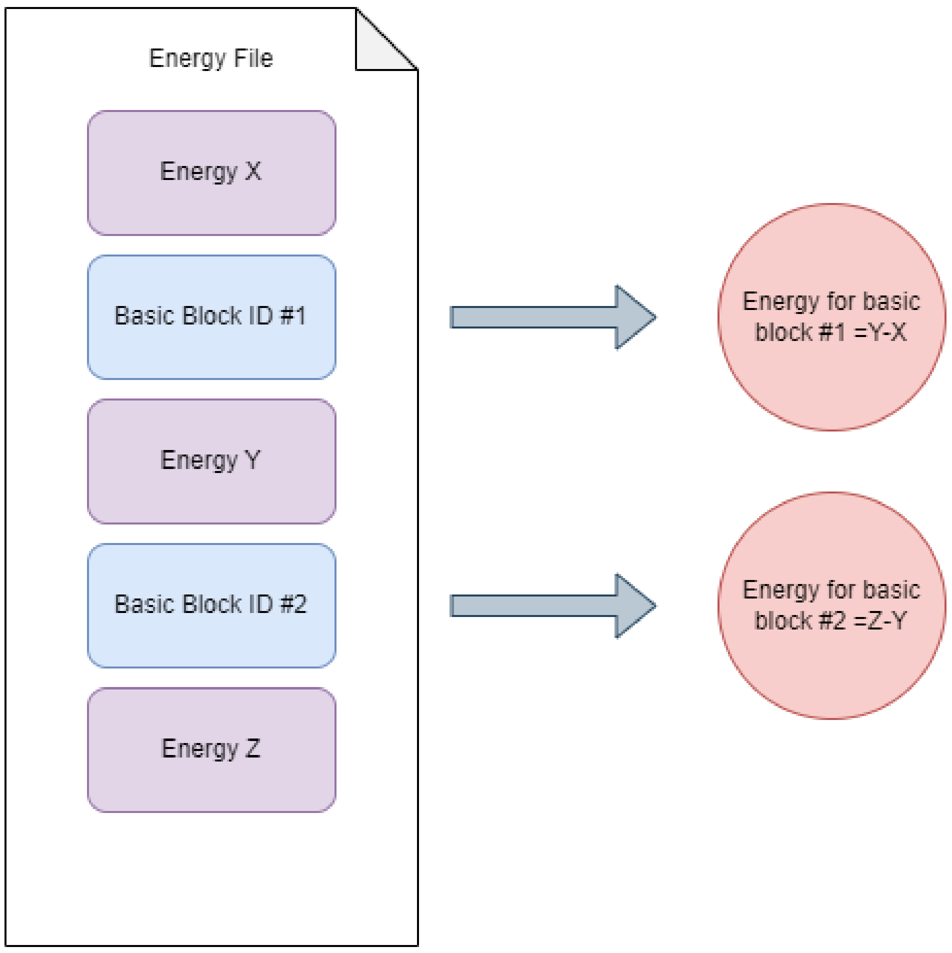


Figure 3.5: *Organization of energy file and correspondence of energy to basic blocks*

3.3 Problems & Redirection of our solution

Although, at first glance the strategy we outlined earlier seems like a viable, simple solution that will allow us to create a complete and reliable energy dataset a closer examination shows that it presents a number of issues that require us to find different approaches to certain parts of our implementation that do not work as expected.

3.3.1 Ineffectiveness of RAPL read granularity

First of all, the whole ides of RAPL reads between basic block execution is based on the idea of energy calculation through energy measurement subtraction that we explained in subsection 3.2.2. Yet, this idea requires that the frequency that the RAPL energy register is updated is higher than the frequency that basic blocks are executed. In more detail, for our strategy to work from the time before the examined basic block is executed, when we read the first energy value, to the time the after the examined basic block is executed and we read the second energy value, the energy register must have updated. Otherwise we will read the same value twice, and the subtraction of the two energies will amount to 0, making the calculated execution energy of the basic block also 0.

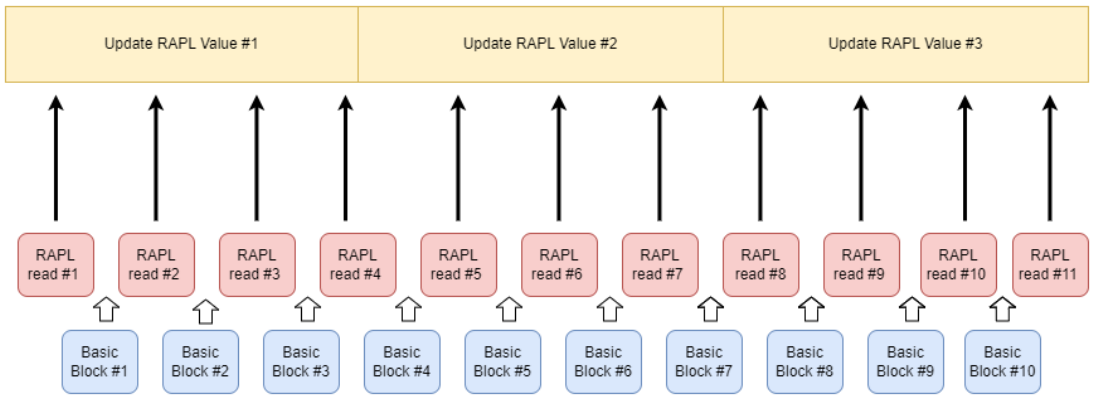


Figure 3.6: *Consecutive RAPL reads between consecutive basic blocks executions measure same energy*

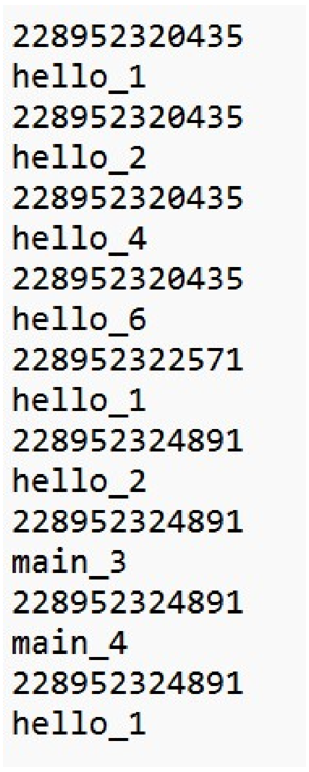
This presents a major problem since the frequency of Basic Blocks at modern CPUs which have a frequency of GHz is much more than 1000 Hz as is the frequency of the RAPL updates. Hence, between two RAPL register updates a number of basic blocks will have been executed.

Thus it is evident that a mechanism to distribute energy of multiple basic blocks to each basic block is necessary. A different approach would be to use a higher level of granularity, meaning that we should not longer measure energy of basic blocks but that of functions ,which are executed at a frequency more similar to that of the RAPL updates. For reasons, we have stated at the Introduction chapter though we believe that the basic block level is the ideal level for an energy dataset and it will be much more helpful for developers. For this reason we decided to follow the first approach and find a strategy to distribute larger sums of energy between multiple basic block executions.

An example run of a test program is presented at the figure 3.7 where we can see that RAPL updates occur after around 3 basic block executions.

The splitting of energies between multiple basic blocks will be based on the idea that energy consumed is analogous to time of execution. For example a basic block that takes X time more at the CPU to execute than another will consume X times more energy as well. This assumption was based on [1] and is fair to be made for a limited number of basic blocks and a short period of execution, not by example for the totality of a high level program. As we can see the tool A.L.E.A that is able to measure basic block energy with very encouraging results used as the basis of how much energy a basic block consumed the time that was needed for it to be executed. Thus if we are able to measure the energy for a number of basic blocks we can then split it fairly between them using as a weight the time that each of them needed to execute.

As we can see on 3.8 there is a number N of basic blocks for which we have a total energy X that we must split between them. We know that each basic block i needed time



```
228952320435
hello_1
228952320435
hello_2
228952320435
hello_4
228952320435
hello_6
228952322571
hello_1
228952324891
hello_2
228952324891
main_3
228952324891
main_4
228952324891
hello_1
```

Figure 3.7: *Example run of test program.*

t_i to execute. So the sum

$$\sum_{i=1}^{N} t_i = T$$

is the total time all these basic blocks needed to execute. Each basic block is hence responsible for energy :

$$\frac{X * t_i}{T}$$

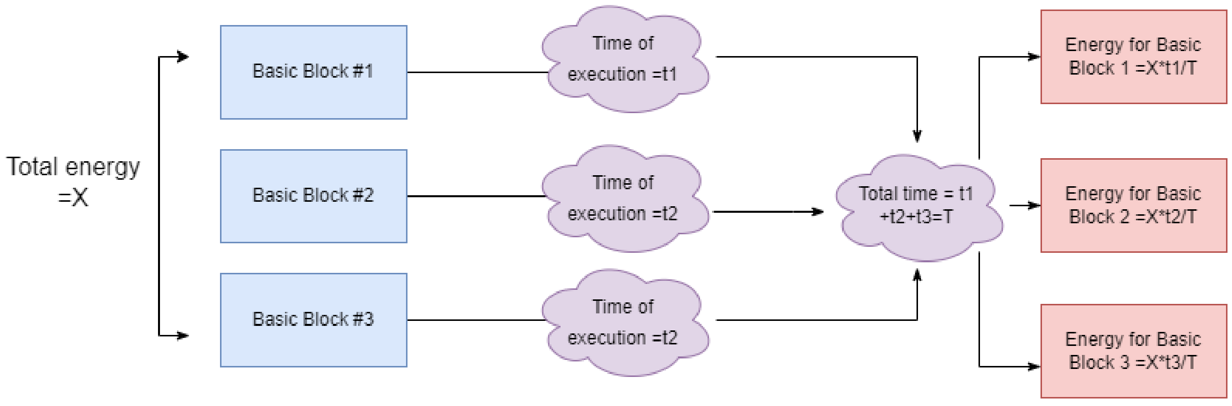


Figure 3.8: *Splitting energy between basic blocks based on execution time*

Now the question arises : How are we going to calculate the execution time of each basic block? And the answer is we can't and we wont. What we will do is allocate total execution time to a series of machine instruction-level execution time data. As we can see in [26] , the execution time of a basic block is analogous to the sum of the execution times of the commands they are made of. And using Intel's measurements for the execution times of the x86 instruction set of the CPU generation we are using we can easily calculate the sum of those times for all the basic block of the examined program [27]. To be precise we won't be using execution times but execution CPU cycles but for simplicity we will refer to it as execution time. To recapitulate, instead of using execution time of a basic block we will use sum of execution times of this basic block's instructions.

So if is there is a number N of basic blocks for which we have a total energy X we must split between them. We know that each basic block i needed is made of M instructions each with execution time w_{iz} . So we will calculate the execution time of basic block i that is made of M instructions as

$$t_i = \sum_{z=1}^{M} w_{iz}$$

. Then the sum

$$\sum_{i=1}^{N} t_i = T$$

is the total weight which is analogous to the time all these basic blocks needed to execute. Each basic block is hence responsible for energy :

$$\frac{X * t_i}{T}$$

3.3.2 Cost of RAPL overshadowing cost of Basic Block

A second problem we noticed early on was also the unaccounted RAPL cost of the RAPL read function as well which could easily overshadow the cost of the basic block execution. To be more precise the RAPL reading process is itself a c function that must be executed and it requires energy to do so. Actually it not even a very simple operation since it requires opening files reading from them and writing to them which sums up to a few dozen assembly instructions. On the other hand, the average size of a basic block is 4 to 5 assembly instructions. As a result in order for us to measure the energy of Y instructions (where Y is the size of the basic block , a number close to 5) we must additionally execute X more instructions and burn energy for them (where X is the number of the assembly instructions in the RAPL read function, a number close to 100). This will lead to the calculated energy to be larger than the actual one by a factor of around 20. This, of course is an unacceptable error and a solution must be found for the energy cost of the RAPL energy measurements themselves to be removed.

This problem was first noticed when we first compared the energies of the execution of an unchanged binary and the execution of a binary that had went through our LLVM pass and RAPL measurements were added 3.10. The discrepancies were enormous and the need for a solution is obvious 3.11.

For the solution, we are lucky that the RAPL read functions remain the same through-

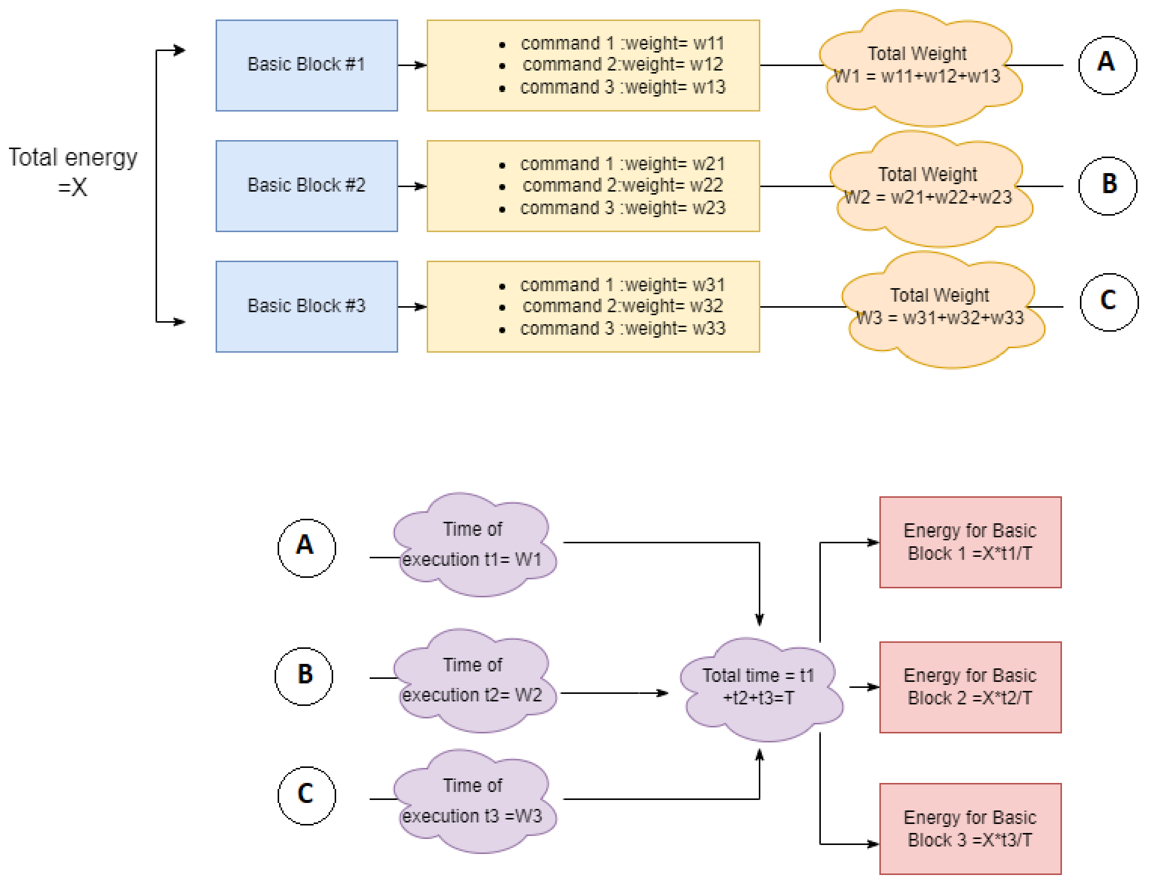


Figure 3.9: *Splitting energy between basic blocks based on weight of assembly commands*

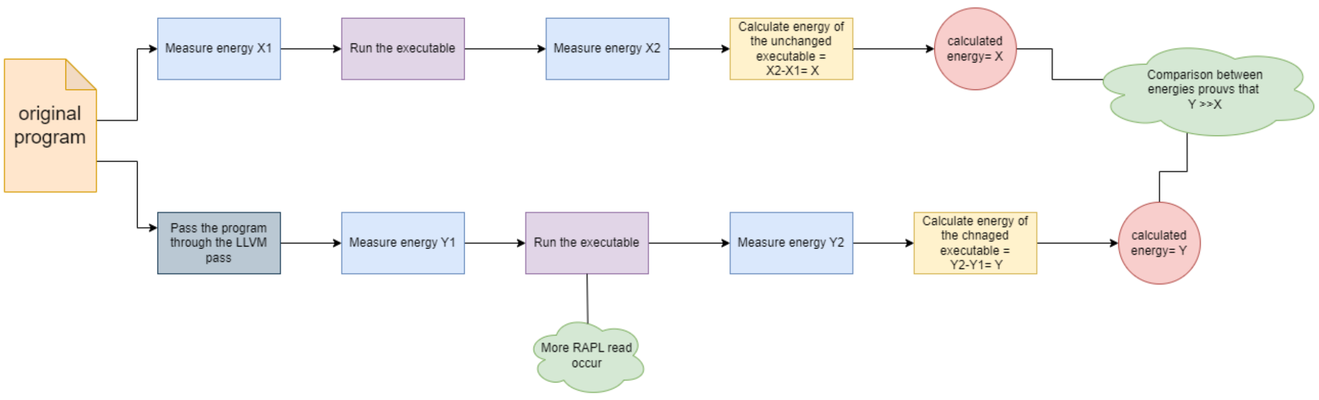
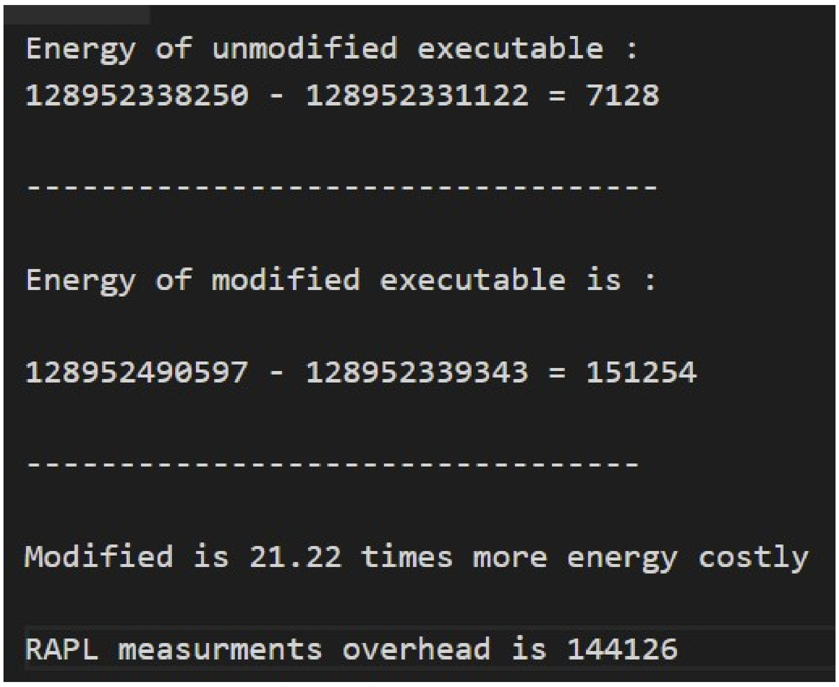

original program
Measure energy X1
Run the executable
Measure energy X2
Calculate energy of the unchanged executable = X2-X1= X
calculated energy= X
Comparison between energies prouvs that Y >>X
Pass the program through the LLVM pass
Measure energy Y1
Run the executable
Measure energy Y2
Calculate energy of the chnaged executable = Y2-Y1= Y
calculated energy= Y
More RAPL read occur

Figure 3.10: *Strategy of experiment to measure effect of extra RAPL read overhead*

out the program, so that subtraction can come to our rescue once more. After all, if we are able to measure the energy cost of a RAPL read function we can simply subtract that overhead from our real readings and find the actual energy measurements.

To measure the cost of a RAPL read function there are two possible ways:



```
Energy of unmodified executable :
128952338250 - 128952331122 = 7128

----------------------------------

Energy of modified executable is :

128952490597 - 128952339343 = 151254

---------------------------------

Modified is 21.22 times more energy costly

RAPL measurments overhead is 144126
```

Figure 3.11: *Results of experiment to measure effect of extra RAPL read overhead*

- Measure the energy of a program that simply measures energy X times 3.12.Then the energy of one RAPL function is :

$$\frac{result}{X}$$

- Measure the energy of an unmodified program (unmodified_energy), the energy of the same program if it passes through the LLVM pass (modified_energy), the number of times the RAPL functions is called during the program execution (RAPL_number) 3.13. Then the cost of an individual RAPL function is:

$$\frac{modified\_energy - unmodified\_energy}{RAPL\_number}$$

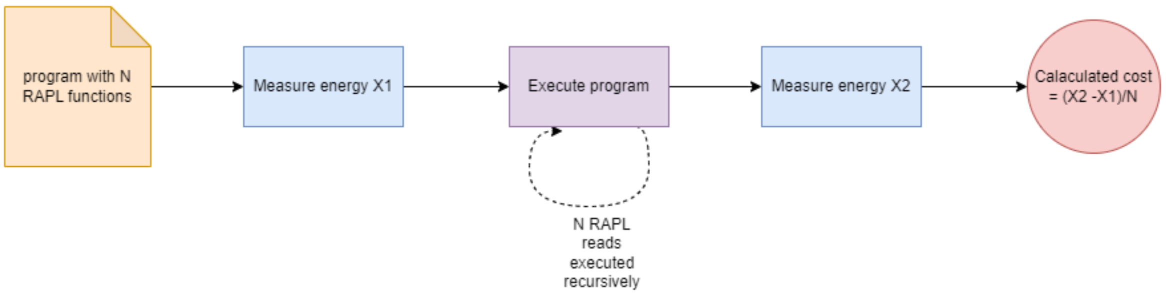

program with N RAPL functions
Measure energy X1
Execute program
N RAPL reads executed recursively
Measure energy X2
Calaculated cost = (X2 -X1)/N

Figure 3.12: *First way for measuring RAPL function cost*

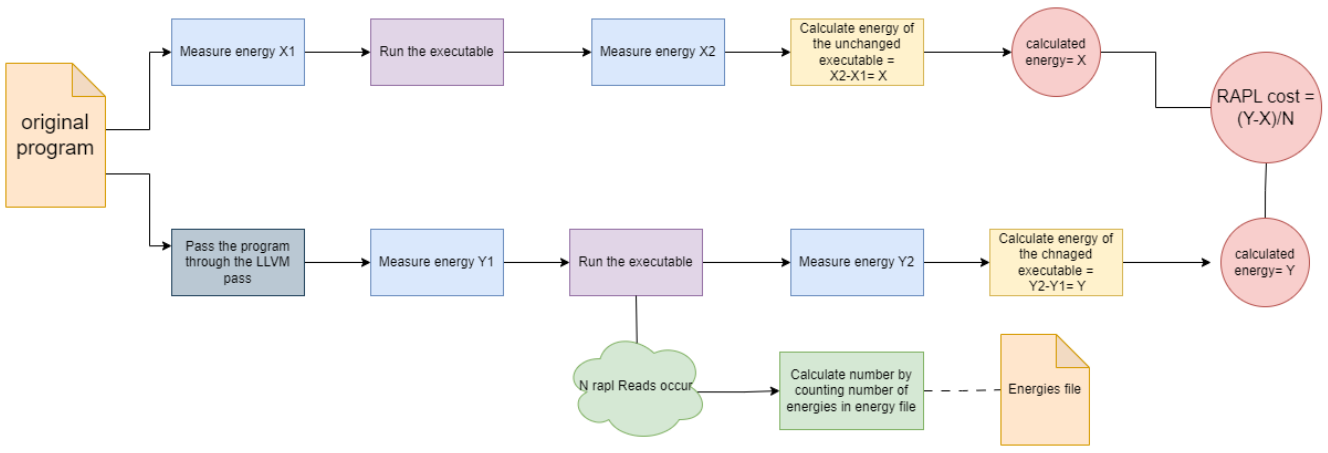


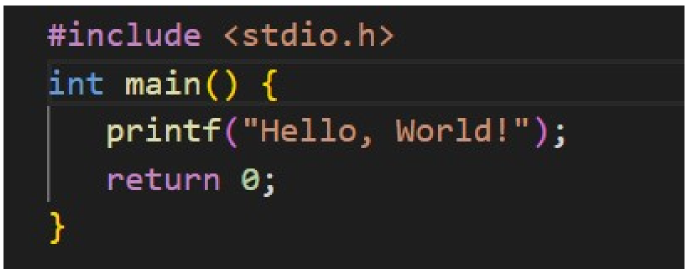
Figure 3.13: *Second way for measuring RAPL function cost*

3.3.3 Lost library functions

The final problem we encounter after we implemented our solution and probably the most important issue we faced was the fact that the LLVM pass can find only the basic blocks inside the source code that are actually written by the developer, not the basic blocks belonging to external functions that are dynamically or statically linked by the compiler at a later stage.

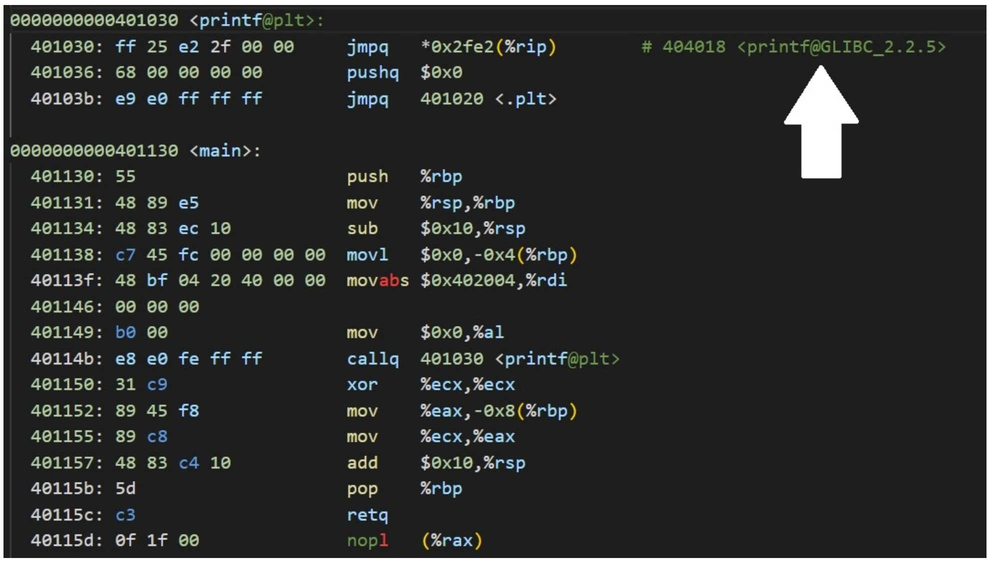
This is a very serious issue since the bulk of the commands that are executed by a program are not written by the developers but are part of the function that are invoked by the programmer, which have already been implemented and are in the libraries the developer includes in his program.

For example let us consider a very simple hello world program . The code we actually write is presented at 3.14. Let's see the assembly translation as well in 3.15. It is obvious that a call to a external function **printf** occurs. The code of this function is not inside this file. It will either be dynamically linked at a runtime or statically at a latter stage in the linking process. As a result the LLVM pass can neither include RAPL reads between each basic block, which is an important issue but not insurmountable, nor know what the contents of those functions are, which is the most serious issue. Without knowing what instructions we are executing it is impossible to create an energy dataset since the dataset must correspond code to energy and even if we calculate the energy we won't know by which instructions it was caused.



```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

Figure 3.14: *C code for simple hello world program*



```
0000000000401030 <printf@plt>:
  401030: ff 25 e2 2f 00 00      jmpq   *0x2fe2(%rip)        # 404018 <printf@GLIBC_2.2.5>
  401036: 68 00 00 00 00         pushq  $0x0
  40103b: e9 e0 ff ff ff         jmpq   401020 <.plt>

0000000000401130 <main>:
  401130: 55                     push   %rbp
  401131: 48 89 e5               mov    %rsp,%rbp
  401134: 48 83 ec 10            sub    $0x10,%rsp
  401138: c7 45 fc 00 00 00 00   movl   $0x0,-0x4(%rbp)
  40113f: 48 bf 04 20 40 00 00   movabs $0x402004,%rdi
  401146: 00 00 00
  401149: b0 00                  mov    $0x0,%al
  40114b: e8 e0 fe ff ff         callq  401030 <printf@plt>
  401150: 31 c9                  xor    %ecx,%ecx
  401152: 89 45 f8               mov    %eax,-0x8(%rbp)
  401155: 89 c8                  mov    %ecx,%eax
  401157: 48 83 c4 10            add    $0x10,%rsp
  40115b: 5d                     pop    %rbp
  40115c: c3                     retq
  40115d: 0f 1f 00               nopl   (%rax)
```

Figure 3.15: *Disassembled hello world binary - call to external function*

This problem is not very simple to solve. It caused by far the biggest hurdle in the success of this thesis. Two ideas were finally proposed to solve it :

- Binary lifting of a statically linked executable to LLVM IR
- Execution tracing using the Intel Perf tool

Both were implemented but only one was successful. They are both analysed thoroughly in Chapters 4 and 5.

Chapter 4

Second approach - Static Binary lifting

4.1 Initial Idea

4.1.1 Problem statement

In this chapter we introduce our initial effort to solve the problems that we faced when trying to perform unique basic block energy measurements as discussed in Chapter 3. The most prominent problem was that a large part of the executable's code resided out of our reach. In order to understand the problem better it would be wise to understand what libraries are in coding.

A programming library is a collection of pre-written code that programmers can use to optimize tasks. This collection of reusable code is usually targeted for specific common problems. A library usually includes a few different pre-coded components. Developers use libraries to build apps, tools and websites more efficiently. Each library is designed to provide a solution to a specific feature. Developers will often look up libraries to help with a particular component they want to create quickly or are struggling with. Then, they'll choose the components they want to use all from that one library, so their app is as cohesive as possible.

After all it would be very time and effort inefficient to recreate over and over again functions and in general solutions that are needed very often. For example , when a programmer wants to print something on the computer screen he can simple invoke the **printf** function from the **stdio.h** library and use one line of code instead of taking a couple of hours to program the whole process themselves. This black box approach is used to hide complex and unnecessary information and work from the developer and let them simply focus on their task.

In our case though this proves to be a problem. Since for efficiency, thεsε libraries and their functions are linked with the executable and are called dynamically at runtime or statically before the binaries are produced [28]. But not the whole code is copied . This functions are precompiled elsewhere at the computer memory and simple a call to them is added. Thus it is extremely difficult to acquire access to their source code to see what kind of assembly commands are executed and impossible to use an LLVM pass on them to add energy reads.

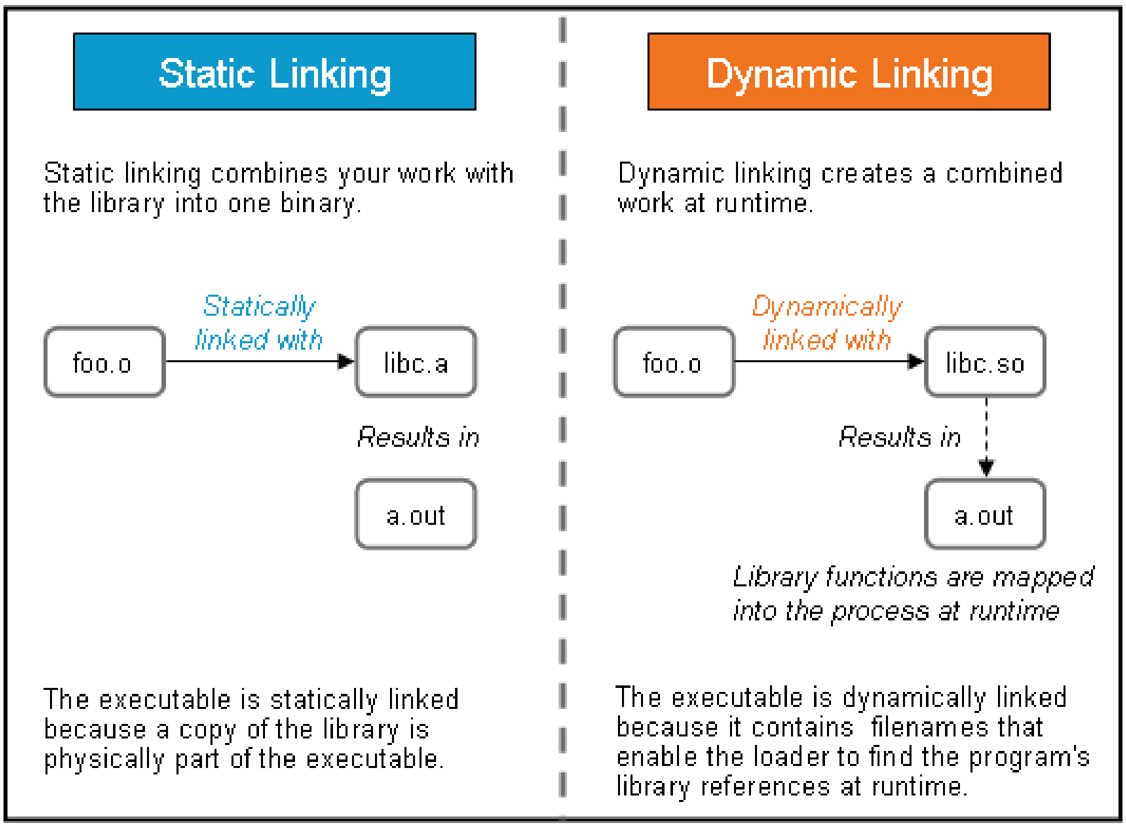

Static Linking
Static linking combines your work with the library into one binary.
foo.o
Statically linked with
libc.a
Results in
a.out
The executable is statically linked because a copy of the library is physically part of the executable.
Dynamic Linking
Dynamic linking creates a combined work at runtime.
foo.o
Dynamically linked with
libc.so
Results in
a.out
Library functions are mapped into the process at runtime
The executable is dynamically linked because it contains filenames that enable the loader to find the program's library references at runtime.

Figure 4.1: *Static vs Dynamic linking [2]*

As a result, we are unable to used the first approach as it was designed in chapter 3 and a way to force the external libraries and functions to come out of hiding and allow us to analyze and transform them, must be found.

4.1.2 Solution

Our solution to those restrictions was to allow the binary to hide all the library functions until it has to compile statically and include them in the binary. At that point, we know for a fact that they are included in that file and all the code that we want to measure is residing in that binary without any calls to other part of our memory. But the binary cannot pass through our LLVM pass. In order for a pass to be able to be applied on a file that file must be in LLVM IR, so either an .ll or an .bc file. So we must transform the binary into an LLVM IR type of file. Here is where binary lifting will be useful to us.

4.2 Implementation

4.2.1 Static compiling and Binary Lifting

Binary lifting is the process of raising machine instructions to higher-level intermediate representations (IR) such as LLVM bitcode. [29] It is in a way, reverse compiling. Instead of taking higher level language and transforming it into an executable it involves

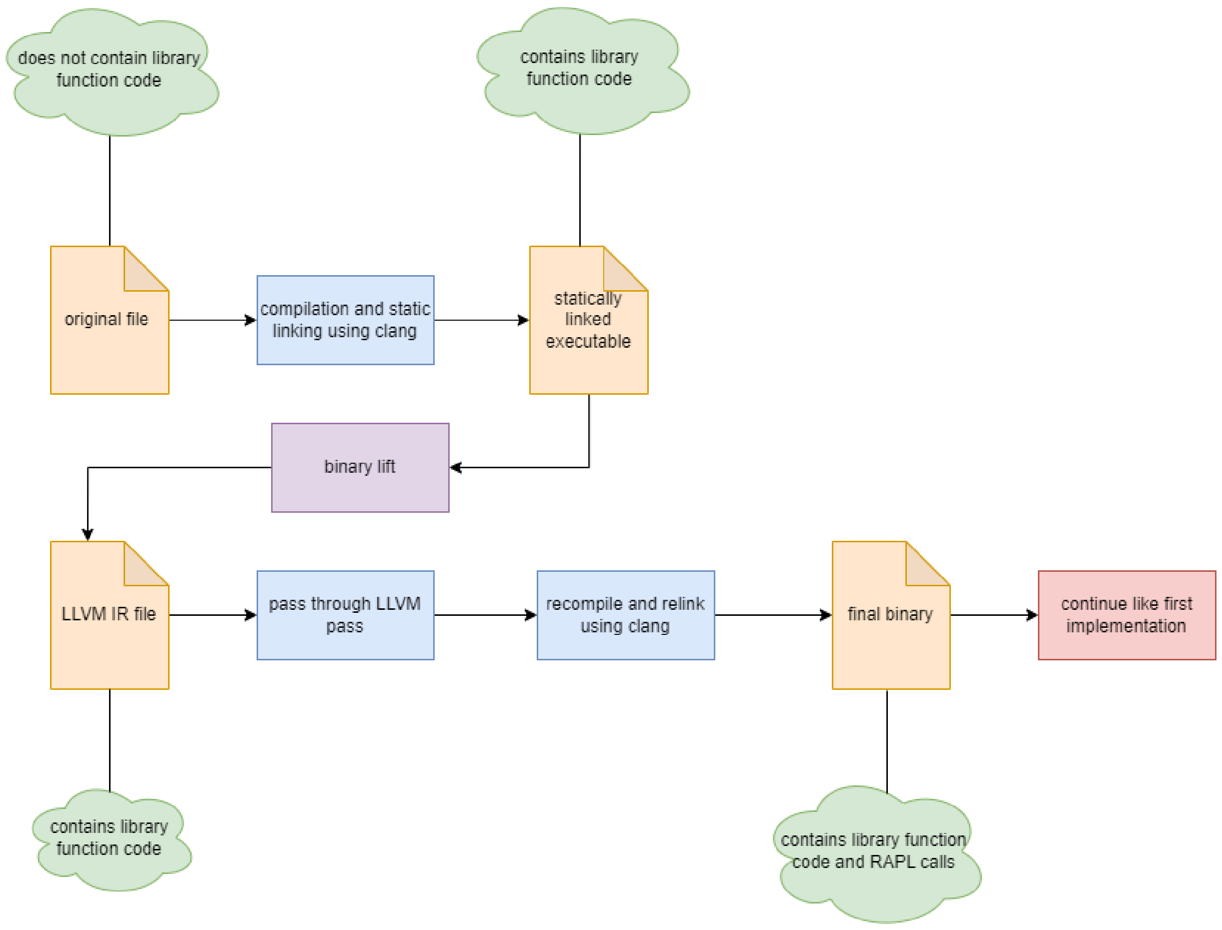

does not contain library function code
contains library function code
original file
compilation and static linking using clang
statically linked executable
binary lift
LLVM IR file
pass through LLVM pass
recompile and relink using clang
final binary
continue like first implementation
contains library function code
contains library function code and RAPL calls

Figure 4.2: *Second strategy - Binary lifting to include the binary functions*

working on an executable to transform it from assembly to a higher level representation. In our case since we are working with LLVM and want LLVM passes to be able to be applied on our code, it would be very beneficial to us to transform the statically linked binary to LLVM IR.

4.2.2 LLVM pass

The LLVM pass will work very similarly to the one described in chapter 4. The purpose is to correspond energy to basic blocks. So a unique ID is created for each basic block and a RAPL function is inserted before each basic block that includes a print of the basic block's unique ID. So after an execution of the final binary we will be left with 2 files. One that has all the basic blocks and their contents (in assembly commands) and one that has energies and basic block id's appearing in pairs. With the subtraction of 2 consecutive energy described in chapter 3 energies and the subtraction of the cost of one RAPL read function which are explained in chapter 3 (section 3.3.2) we deride the final energy cost of the basic block.

If multiple basic blocks end up corresponding to one total energy we will split them using the method described on Chapter 3 section 3.3.1. So in essence, if we manage to lift a statically linked binary and pass it through the LLVM pass then we follow the methodology described n chapter 3 , solving the problems of the Ineffectiveness of RAPL read granularity and that of the extra RAPL read function overhead as described on section 3.3.

With only the problem of binary lifting remaining then , we will mention the binary lifting tools we tried and tested. We experienced varying levels of success but unfortunately although lifting was accomplished we were not able to pass the lifted IR file through the LLVM pass and create a working executable. Nonetheless the validity of this technique remains and if someone in the future is able to lift a binary and pass the result from an LLVM pass they can use our tools and the method described earlier to create a reliable energy dataset. Binary lifting is still at an early stage and maybe the technology that exists at this moment is incompatible with LLVM passes as the resulted IR files are very unstable. Maybe there is some tool that can accomplish this task but we were not able to find it . We mention the tools we used and what we managed to accomplish with each so that anyone that wishes to continue our work can use this as a reference.

4.2.3 Revng

Revng is a static binary translator. Given a input ELF binary for one of the supported architectures (currently i386, x86-64, MIPS, ARM, AArch64 and s390x) it will analyze it and emit an equivalent LLVM IR. To do so, revng employs the QEMU intermediate representation (a series of TCG instructions) and then translates them to LLVM IR.

Using Revng we were able to lift binaries , although not consistently depending on the functions included but we were unable to pass the lifted IR through our pass.

4.2.4 LLVM-mctoll

LLVM-mctoll is capable of raising X86-64 and Arm32 Linux/ELF libraries and executables to LLVM IR. Raising Windows, OS X and C++ binaries needs to be added. At this time X86-64 support is more mature than Arm32. Development and primary testing is being done on Ubuntu 22.04. Testing is also done on Ubuntu 20.04. The tool is expected to build and run on Ubuntu 18.04, 16.04, Ubuntu 17.04, Ubuntu 17.10, CentOS 7.5, Debian 10, Windows 10, and OS X to raise Linux/ELF binaries.

Using LLVM-mctoll we were able to lift binaries if we included the path of all the libraries that the binary used and we were able to pass some of the lifted binaries from our pass, but the resulting executable always failed with a segmentation fault.

4.2.5 McSema

McSema is an executable lifter. It translates ("lifts") executable binaries from native machine code to LLVM bitcode. LLVM bitcode is an intermediate representation form of a program that was originally created for the retargetable LLVM compiler, but which is also very useful for performing program analysis methods that would not be possible to perform on an executable binary directly.

McSema enables analysts to find and retroactively harden binary programs against security bugs, independently validate vendor source code, and generate application tests with high code coverage. McSema isn't just for static analysis. The lifted LLVM bitcode can also be fuzzed with libFuzzer, an LLVM-based instrumented fuzzer that would otherwise require the target source code. The lifted bitcode can even be compiled back into a runnable program! This is a procedure known as static binary rewriting, binary translation, or binary recompilation. McSema supports lifting both Linux (ELF) and Windows (PE) executables, and understands most x86 and amd64 instructions, including integer, X87, MMX, SSE and AVX operations. AARCH64 (ARMv8) instruction support is in active development.

Using McSema we were able to lift binaries consistently and we were able to pass the lifted binaries from our pass. The resulting executable though always failed with a segmentation fault.

4.3 Redirection of our solution

It has become evident that the issue of lost external functions still remains and that binary lifting was not an approach capable to solve this problem. Thus we decided to focus on a new approach that involves execution tracing which will be described in the following chapter, chapter 5.

Chapter 5

Final approach - Runtime tracing

5.1 Initial Idea

5.1.1 Problem statement

Having tried the previous 2 approaches it has become evident that we have created a working pipeline that measures the energy of the basic blocks of a program with one major issue: it does not take account in any way of the functions from the libraries that we have included. As we stated earlier this is unacceptable since the bulk of the programs code and as a result energy consumption is attributed to those library functions, which from now one we will refer as **lost functions**, since we ”lose” the assembly instructions that make them up. For a better understanding of this problem we can refer to the figure 5.1 .

We know that these library functions are added to the executable if we compile it statically but that occurs during the linking stage which happens later than the stage during which our code passes through the LLVM pass and the basic blocks are identified and their contents are stored (in assembly instructions). So we tried to get the linked executable back to a previous stage to pass it through the LLVM pass, through the process of binary lifting to LLVM IR but our efforts were ultimately unsuccessful. Hence , it is necessary for us to find a way to analyze the code after the linking has taken place. This will be our final approach and it will be explained during this chapter.

5.1.2 Solution

5.2 What is Execution tracing? - Intel Perf

In software engineering, tracing involves a specialized use of logging to record information about a program’s execution. This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical-support personnel and by software monitoring tools to diagnose common problems with software.**??**. Tracing is a cross-cutting concern.

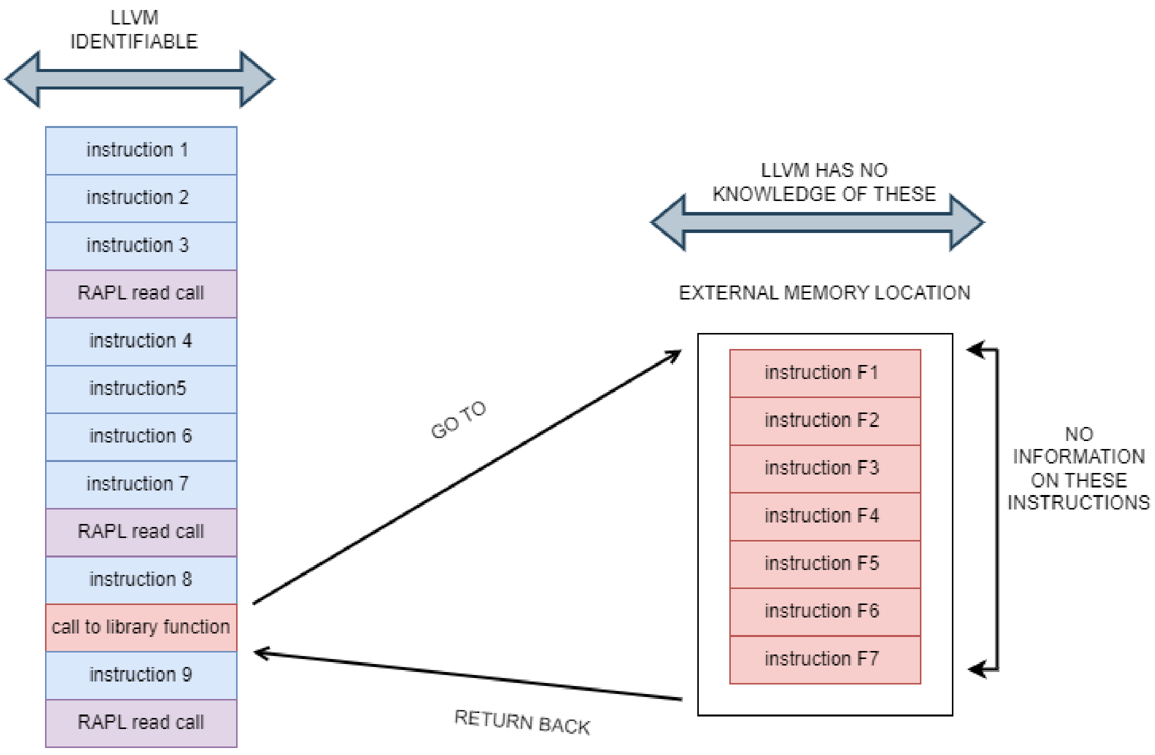


Figure 5.1: *Library functions cannot be analyzed by LLVM*

In our case, the information that we are interested in recording about the executable is all the assembly commands that make it up and that ran during its execution in the correct order. To achieve that we will be using the tool Intel perf that was described in section 2.3.

As a recap, Intel Processor Trace (Intel PT) [23] is an extension of Intel Architecture that collects information about software execution such as control flow, execution modes and timings and formats it into highly compressed binary packets. We will use the intel PT technology to get the trace of a binary that we want to measure. To understand how to do that we must first understand the capabilities of the perf record and perf scripts commands:

- Perf record : this command runs a command and gathers a performance counter profile from it, into perf.data - without displaying anything. This file can then be inspected later on, using perf report or script.

- Perf script : This command reads the input file and displays the trace recorded. After installing the xed tool , we can dump all instructions in a long trace. That can be fairly slow, but it is the best way to get a complete and detailed trace of the executed binary

5.3 LLVM Pass

In this approach the LLVM pass is still very important to our work, but since the basic block code identification, and their corresponding to the energy will be performed at a later stage using Intel perf, those functionalities can now be removed from the LLVM pass. Its sole focus now will be to insert 3 types of RAPL read function calls:

- A RAPL call type A function before the start of every basic block of the program (At this point we have access only to the basic block the developer has written not the one of library function)
- A RAPL call type B function before a call to a library function
- A RAPL call type C function after a call to a library function

In essence, all of this functions are identical and perform the specific task of reading the RAPL energy register and printing its contents to a file. The only difference between them is that each one of them will have a different name type (A , B or C) so that we know when we are reading the execution trace and we see a RAPL function call if the code that follows is a basic block or code from a library function.

5.4 Statistical analysis

So we have reached a point that RAPL reads occur before every basic block of the program and before and after every call to a library function. And we can get a hold of the execution trace of the program through Intel Perf. The challenge now is to tackle the problems we faced during our first approach and also find a way to correspond the code we have stored to energy we have measured.

5.4.1 Command Weights

First of all the problem of RAPL read granularity we faced during the first approach still gives us trouble. We will use the same solution we used there which is based on the splitting of the multiple basic block energy between basic blocks using as a weight the sum of the execution time of the assembly commands that make them up. For more information on this technique and its validity refer to section 3.3.1.

5.4.2 RAPL Read energy removal

Secondly the problem of the RAPL read function energy overhead that was first noticed during our initial approach on chapter 3 will be solved with the exact same way we explain there. For more information on this technique and its validity refer to section 3.3.2.

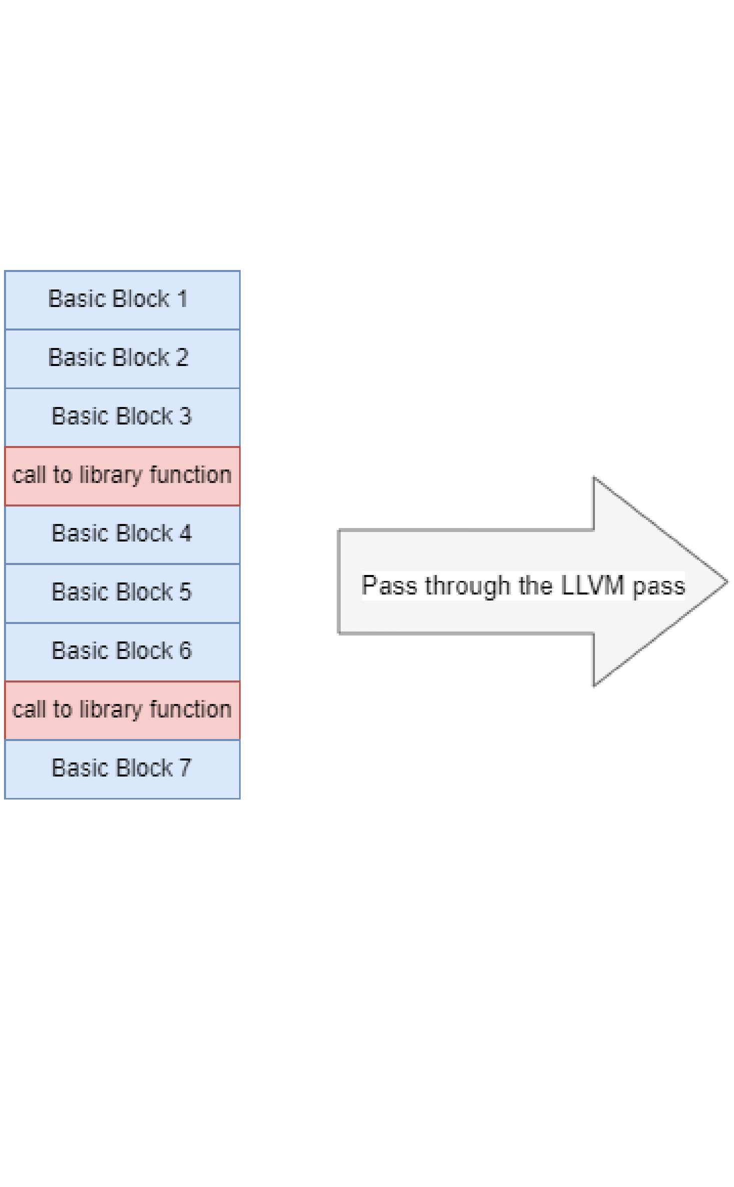
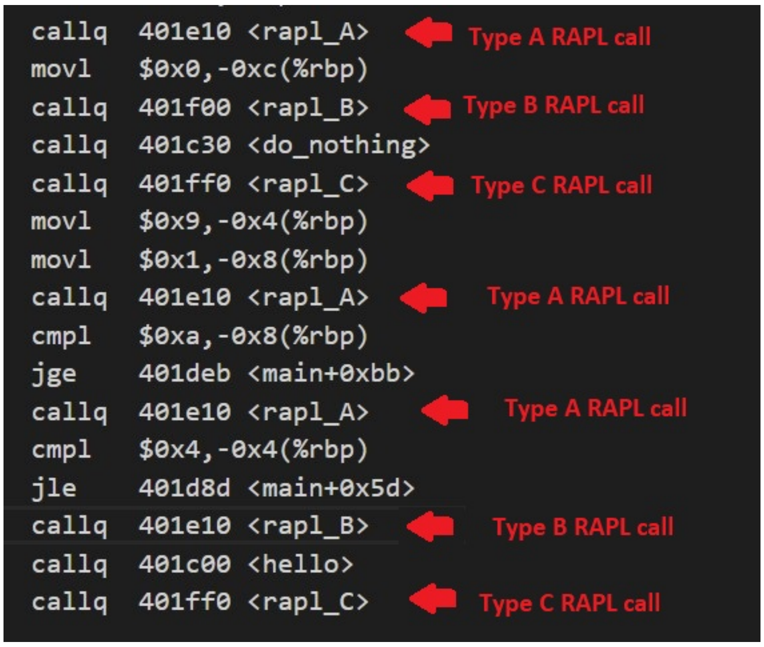


Figure 5.2: *Illustration of how this LLVM pass works*

5.4.3 Unnecessary code removal

In this case, we have access to the assembly code (the execution trace0 that make up our program after we have included the RAPL read functions in our program through the LLVM pass. As a result when we get the execution code it will also include the code for these RAPL read functions that is not important to us and must be removed, since we only want to take into account the actual code of the program we are examining, not anything extra we added.

This is rather simple to do: we will find the code for a RAPL function from an examination of our binary(we call these instructions CODE RAPL). then on our execution trace every time a call to a RAPL function is performed the following instructions will be the



```
callq  401e10 <rapl_A>
movl   $0x0,-0xc(%rbp)
callq  401f00 <rapl_B>
callq  401c30 <do_nothing>
callq  401ff0 <rapl_C>
movl   $0x9,-0x4(%rbp)
movl   $0x1,-0x8(%rbp)
callq  401e10 <rapl_A>
cmpl   $0xa,-0x8(%rbp)
jge    401deb <main+0xbb>
callq  401e10 <rapl_A>
cmpl   $0x4,-0x4(%rbp)
jle    401d8d <main+0x5d>
callq  401e10 <rapl_B>
callq  401c00 <hello>
callq  401ff0 <rapl_C>
```

Figure 5.3: *Real example of program after it has passed through our LLVM pass*

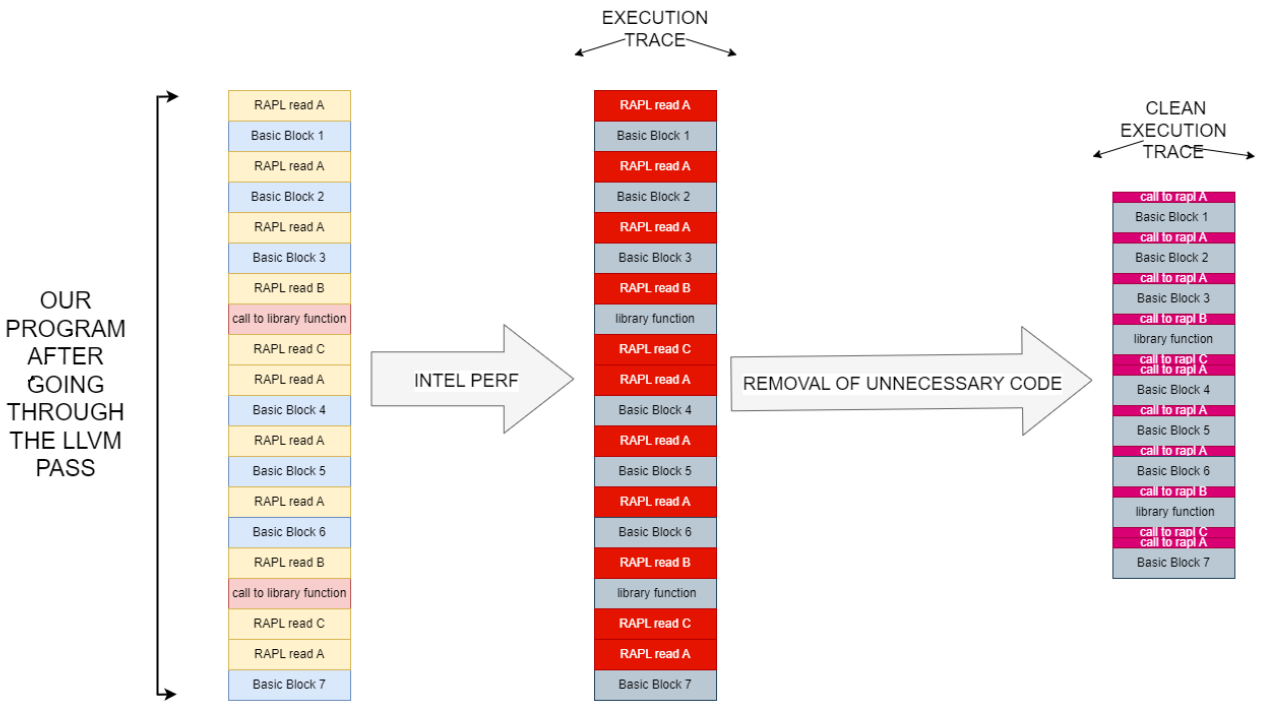


Figure 5.4: *Illustration of the removal of RAPL function code process*

same as CODE RAPL and we will delete them from the execution trace.

5.4.4 Splitting of external functions

Another problem that arises is that while we want our dataset to be in a basic block level, the lirbary functions have not been split into basic blocks and only one RAPL read is performed at their start and one after they conclude. The splitting of the function code into basic blocks is rather easy. We will traverse the code and every time we find a conditional or unconditional branch or a call to another memory address we split the code and a new basic block is created. Now for the energy we have the exact same problem we faced when multiple basic blocks had one total energy. We will solve it with the exact same way by splitting the energy to the newly created basic blocks of the function using as a weight the sum of the execution time of the commands that each basic block is made of. For more detail refer to section 3.3.1.

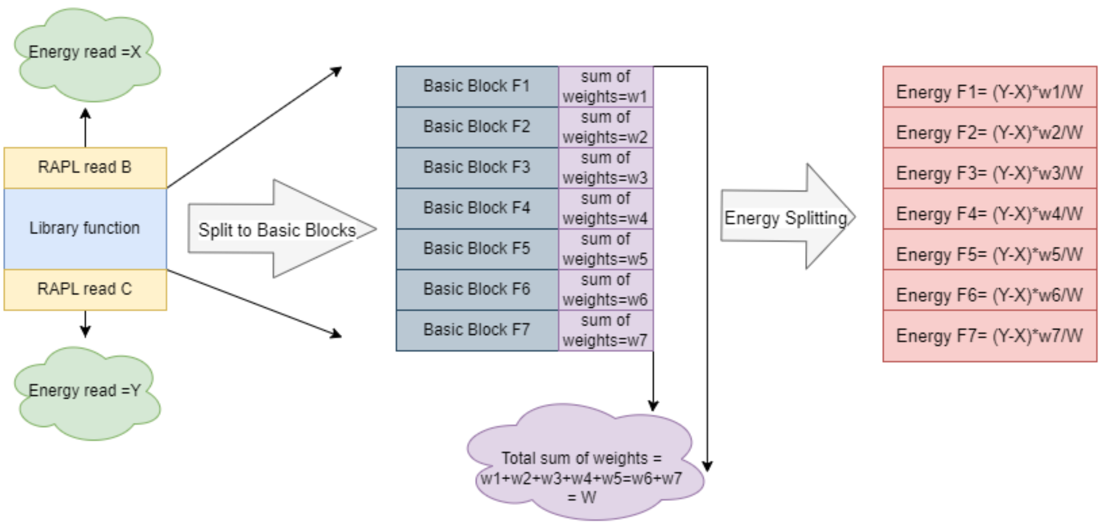


Figure 5.5: *Illustration of the function splitting (code and energy) process*

5.5 Edge cases and problems

5.5.1 Negative energy

A problem we encountered when trying to remove the energy cost of the RAPL read function , was that although on average the cost of a RAPL function is pretty consistent, in some cases the cost of a RAPL read + the basic block executed was less than the standard value of the RAPL read we have calculated meaning that the energy cost of the basic block after the subtraction would have been calculated negative, which is not possible. That happens become the RAPL technology is not perfect and although on average it measures energy consistently some times it may over or under measure it .

To solve this problem , whenever we calculate negative energy we increase the granularity of our reads. In more detail instead of taking into account every RAPL read we ignore a few. So in a case when X RAPL reads happen between X basic blocks and a negative energy is observed for one of them, then we ignore the X-2 internal RAPL reads, taking into consideration only the first and the last RAPL read and split the energy between the X basic blocks using the technique we explained in section 3.3.1. The larger the number X is the less likely we are to be affected by errors of the RAPL technology but it is a trade off with accuracy since the splitting of energy between basic blocks introduces some error to the whole process itself. So usually a relatively small number of basic blocks are bundled together when negative energy is observed. After all in most cases when RAPL performs an underestimation the following energy updates are overestimations so that the total energy remains accurate.

5.5.2 Lost final energy

Another small issue we observed was that we always lost the final basic block of the program. This happens because RAPL reads were inserted before every Basic block and the energy of a basic block is calculated from the subtraction of the energy of the RAPL call before the next basic block and the energy measured by the RAPL call before itself. But the last basic block does not have a basic block after it so it neither has a RAPL call after it. To solve this we insert at the end of each program an empty function call that does nothing and thus introduces no energy overhead so that a final RAPL call will be inserted as well.

5.6 Overall Process

At this point we will summarize the whole process for our third and final approach that we have described in this chapter:

- Firstly we measure the energy of the unmodified program we want to examine with a RAPL read at the start and end of the program. We call this **energy_unmodified**.
- We pass our unmodified program through the LLVM pass.
- We execute the program using Intel perf so that the execution trace is stored.
- At this point the energy file and the execution trace have been created.
- From the first and last RAPL energy we estimate the cost of the modified program which we call **energy_modified**.
- From the subtraction of energy unmodified from the energy modified we get the energy cost of all the RAPL reads , and if we divide that value with the number of RAPL reads we get the individual RAPL cost which we will call **RAPL_cost**.

$$RAPL\_cost = \frac{energy\_modified - energy\_unmodified}{numberofRAPLcalls}$$

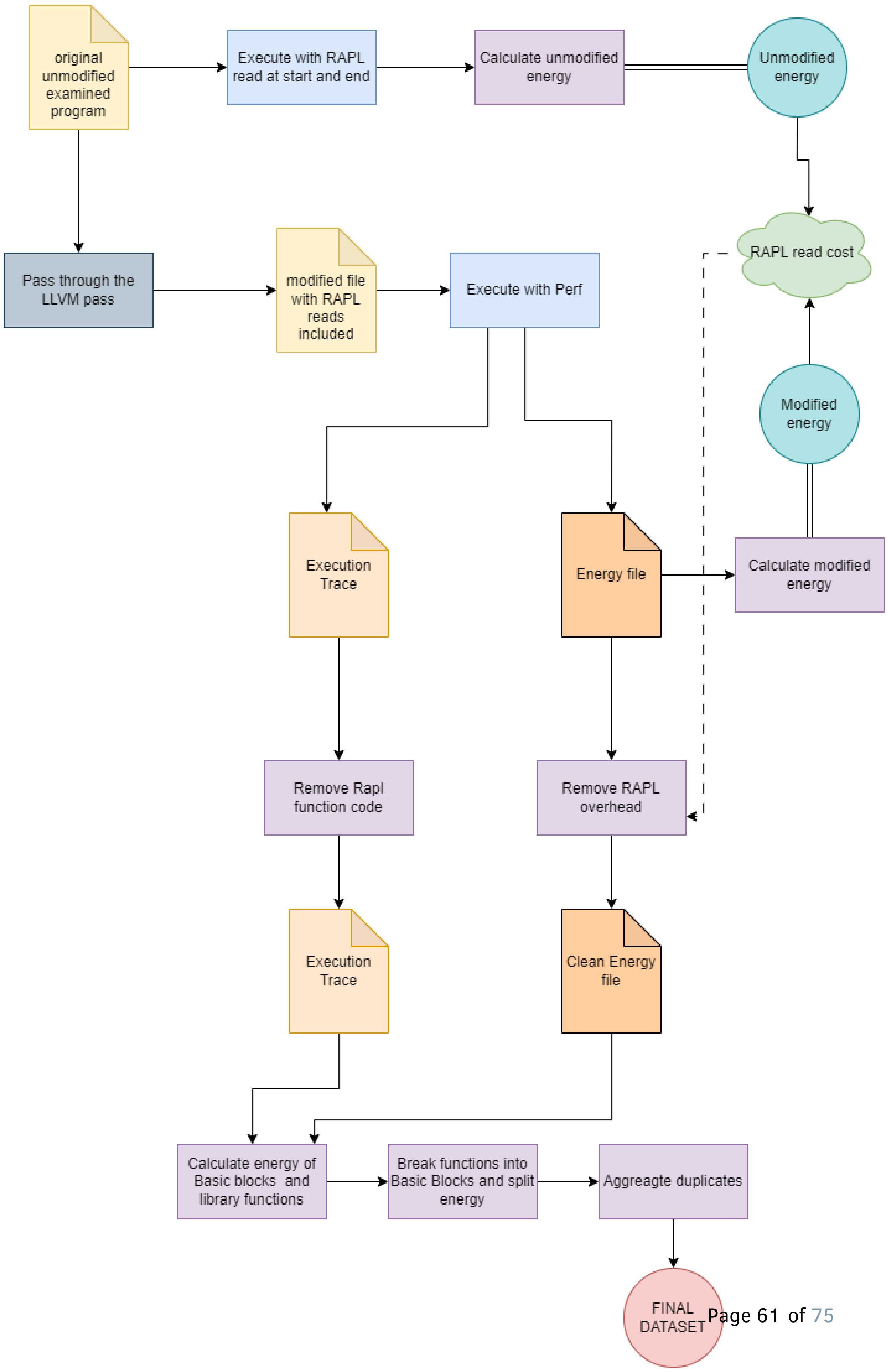
- We traverse the execution trace and remove unnecessary RAPL code and the energy file removing energy overhead **RAPL_cost**.

- We correspond basic block or library functions to energy based on the order with which they appeared. We can differentiate between basic blocks and library functions based on the RAPL call type that preceded them.
- We split energy between basic blocks when we have the problem of RAPL read granularity or that of negative energy.
- We split library functions to basic block and split their energy as well.
- We find average values of duplicate basic blocks.

This whole process was performed with a plethora of scripts as a part of a automatic pipeline that does not need any human assistance. We originally created the main part of the pipeline, meaning the most important scripts that did the bulk of the hard work using python but due to the extremely large size of the execution trace, the analyzing time was extremely long. For that reason we redid the computational heavy scripts in C and the whole process now takes a mater of seconds. To sum up the programming languages that were used were :

- C , for the most important scripts doing the bulk of the work.
- Python , for useful, complementary scripts that were not computationally heavy.
- Bash , for pipeline organization and automation.
- C++ , for the LLVM pass.

We can additionally observe the whole process in the figure 5.6,presented below.


original unmodified examined program
Execute with RAPL read at start and end
Calculate unmodified energy
Unmodified energy
Pass through the LLVM pass
modified file with RAPL reads included
Execute with Perf
RAPL read cost
Modified energy
Execution Trace
Energy file
Calculate modified energy
Remove Rapl function code
Remove RAPL overhead
Execution Trace
Clean Energy file
Calculate energy of Basic blocks and library functions
Break functions into Basic Blocks and split energy
Aggreagte duplicates
FINAL DATASET

Chapter 6

Results and Evaluation

6.1 Total dataset - Distribution of energies

In this section we try to offer a general understanding of the dataset we have created. The dataset is made up of 3828 unique basic blocks . The average number of instructions in each basic block is 5.04 and the average energy value for a basic block is 0.6429 units of energy(61.3 µJ).

Specification of produced dataset	
Number of unique basic blocks	3828
Average length of basic block	5.04 instructions
Average energy of basic block (BB)	0.6429 units of energy
Percentage of BBs with energy > 10	0.16%
Percentage of BBs with energy < 10 and > 5	0.31%
Percentage of BBs with energy < 5 and > 2	2.12%
Percentage of BBs with energy < 2 and > 1	5.24%
Percentage of BBs with energy < 1 and > 0.5	10.81%
Percentage of BBs with energy < 0.5 and > 0	79.85%
Percentage of BBs with energy =0	0.51%

Table 6.1: *Dataset Specifications*

In the figure 6.1 we can see the number of basic blocks that each benchmark contributes to the dataset. Keep in mind though that these numbers include duplicate basic blocks that are executed more than one times. For that reason the number of basic blocks will decrease significantly to a total of 3828. In particular the benchmark variable_name_results is made from more than 100.000 execution basic blocks but it runs

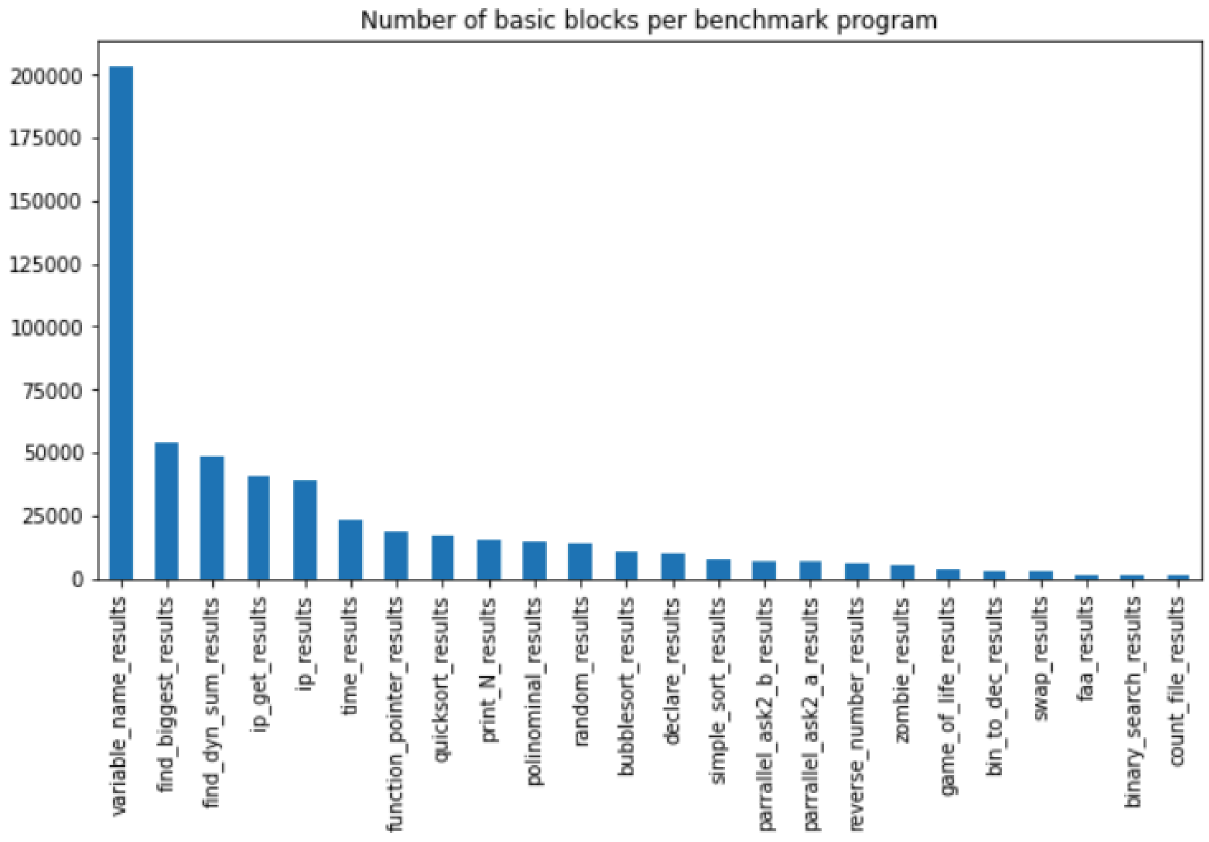


Figure 6.1: *Number of basic blocks per benchmark - duplicates included*

on a large for loop that's why the majority of them are duplicates.

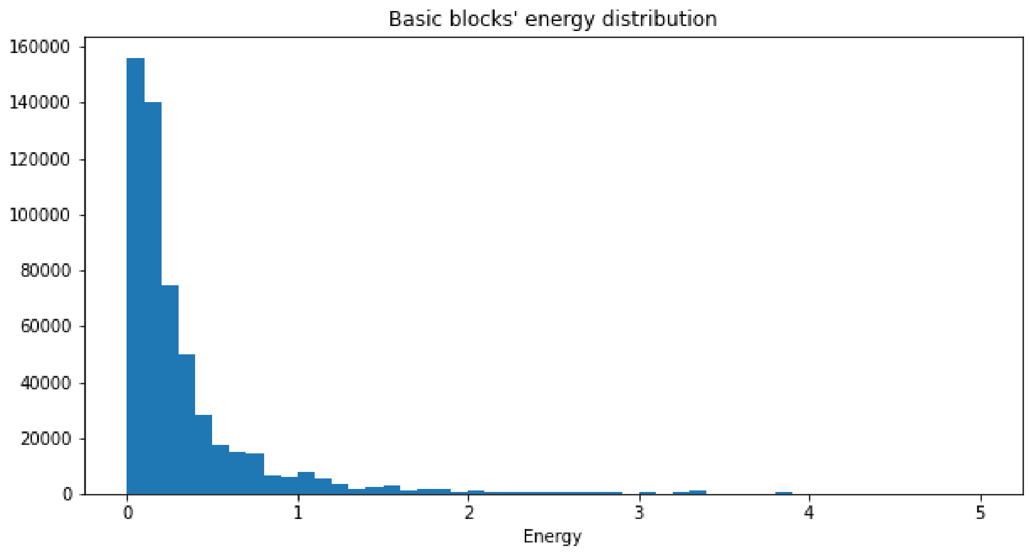


Figure 6.2: *Energy distribution of the total dataset*

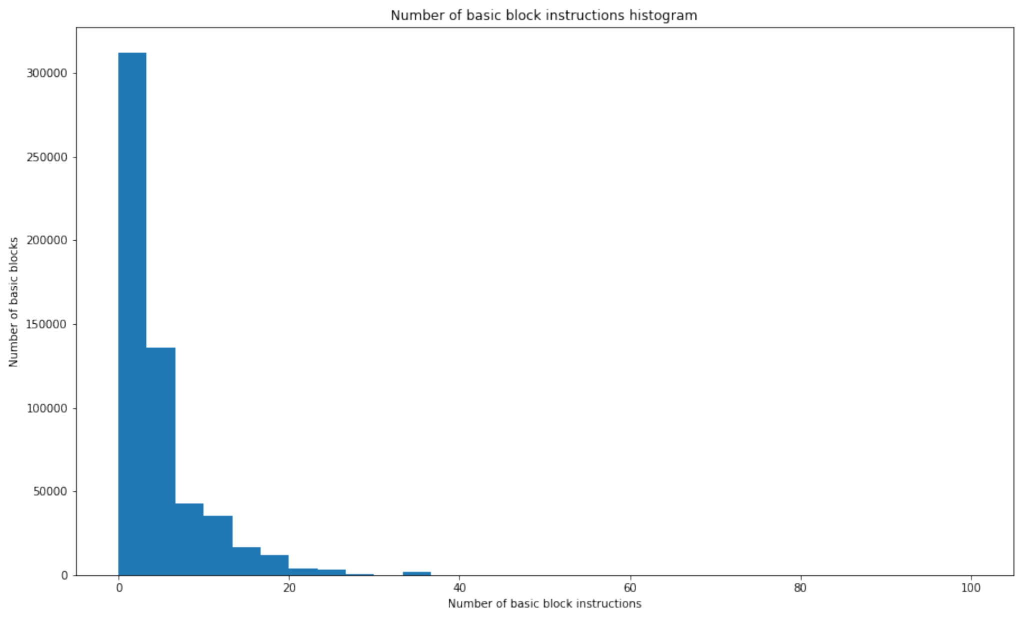


Figure 6.3: *Instructions number in a basic block - distribution of the total dataset*

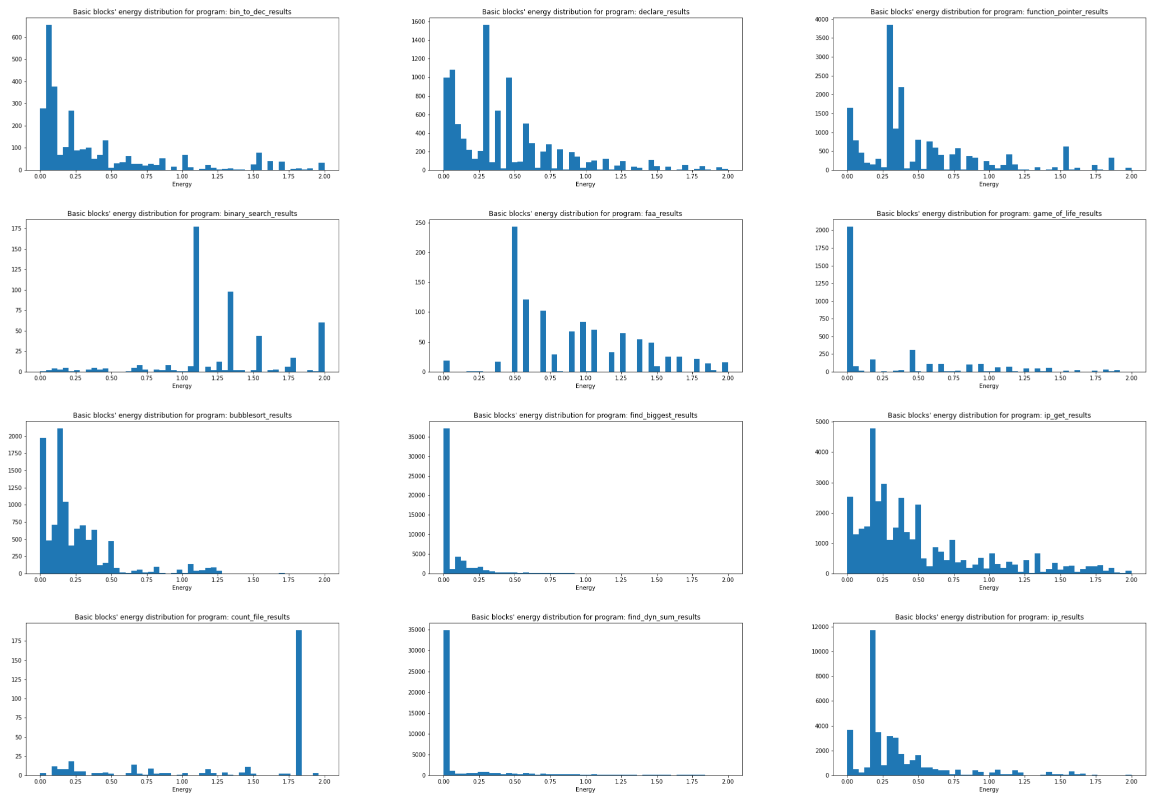


Figure 6.4: *Energy distribution for individual benchmarks part 1*

6.2 Evaluation Method

We have reached the stage that the dataset has been created and we are confident in our methods and the results we have produced. Yet there must be a way to evaluate the

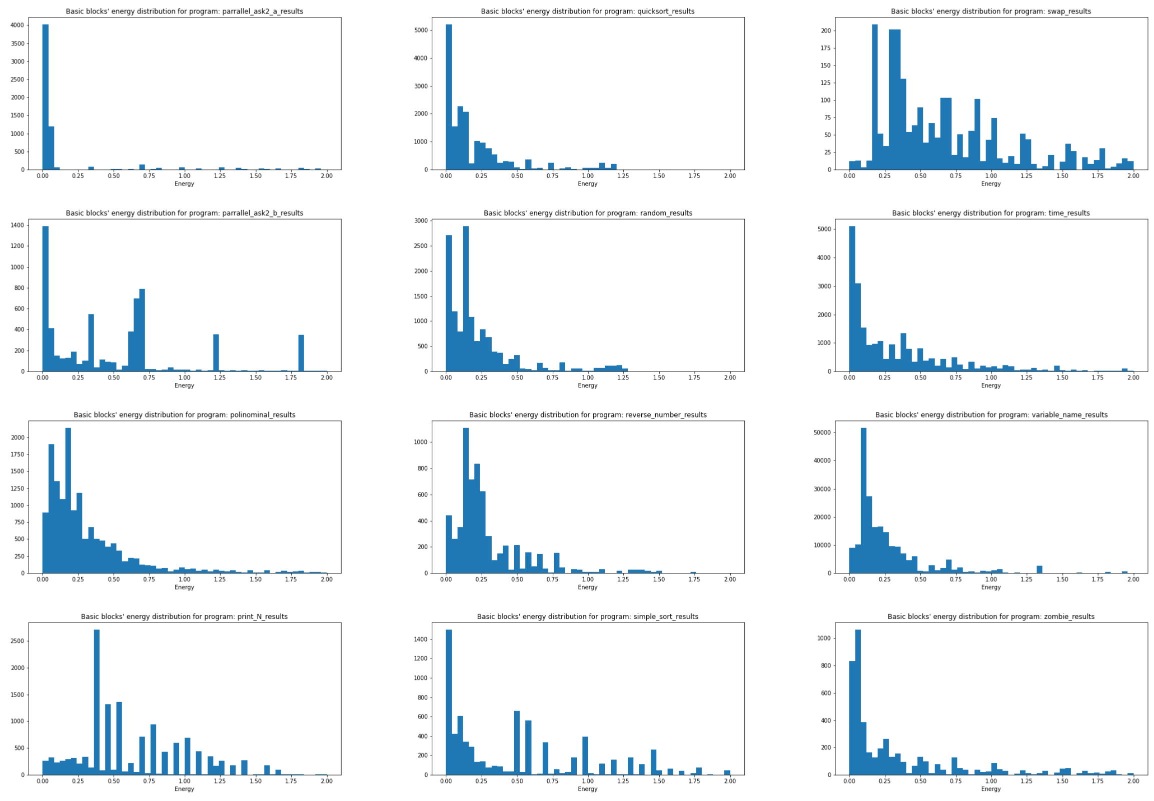


Figure 6.5: *Energy distribution for individual benchmarks part 2*

accuracy of our measurements. Unfortunately there is no other method to get energy measurements for the basic blocks and then compare them to our own measurements. After all if there were such energy measurements available this whole work would have been pointless since the energy dataset would already be in existence.

Thus it is evident that we must find an indirect method to evaluate the accuracy of our methods. This method will be based on the total program execution energy. In more detail lets say that for a program we have created its dataset with the mechanism described in this thesis. Then we know the energy_i of each basic block i of the N total basic blocks that make it up as well as the number of times times_i that each basic is executed during the program. With two RAPL calls, one at the start one at the end of the program we can get the energy cost of the total program and then by running the program a few times (about 100) we can get a safe result about the average energy cost of that program. Theoretically if our measurements are correct then the sum of the energies of the basic blocks we have calculated multiplied by the times each basic block has been executed must equal the average energy cost of the program:

$$average\_energy\_cost = \sum_{i=1}^{N} t_i * energy_i$$

By calculating how different these 2 values are we can find the error of our dataset.

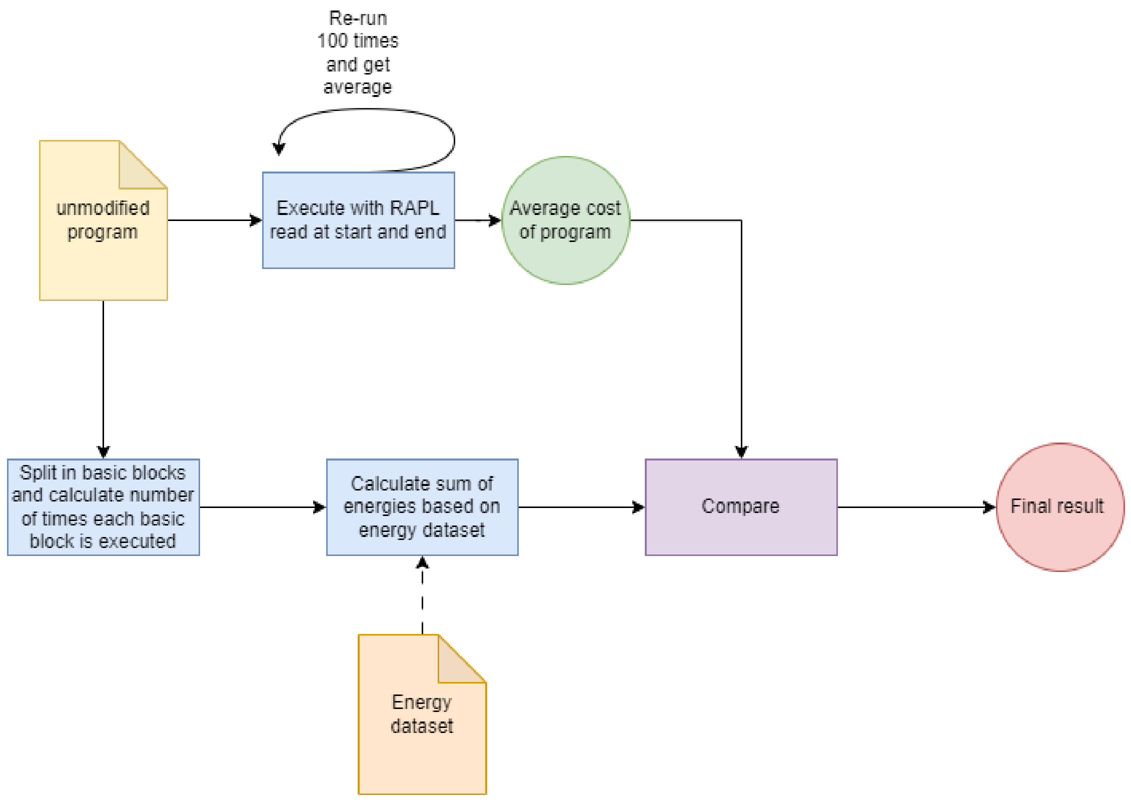

Re-run 100 times and get average
unmodified program
Execute with RAPL read at start and end
Average cost of program
Split in basic blocks and calculate number of times each basic block is executed
Calculate sum of energies based on energy dataset
Compare
Final result
Energy dataset

Figure 6.6: *Evaluation Process*

6.3 Results

The evaluation results will be presented for each benchmark we run and the total average as well. As we can see on figure 6.7 the average error is merely 2.63% which is extremely encouraging for the accuracy of our measurements.

After all the energy of the basic blocks are not really relevant as a single value but more as a total of a program, the sum of which will be the energy that will be consumed. Indeed individual basic block values may not be extremely accurate each time a basic block is executed since this energy is very volatile. But we do not really care about a single execution that may be deviating from the norm. We care about the energy of basic blocks that are executed a large number of times because these are the ones responsible for the majority of the energy our software consumes. And we believe that the average energy cost of these basic blocks after a large number of executions will approach the value that we have calculated in our dataset with high accuracy.

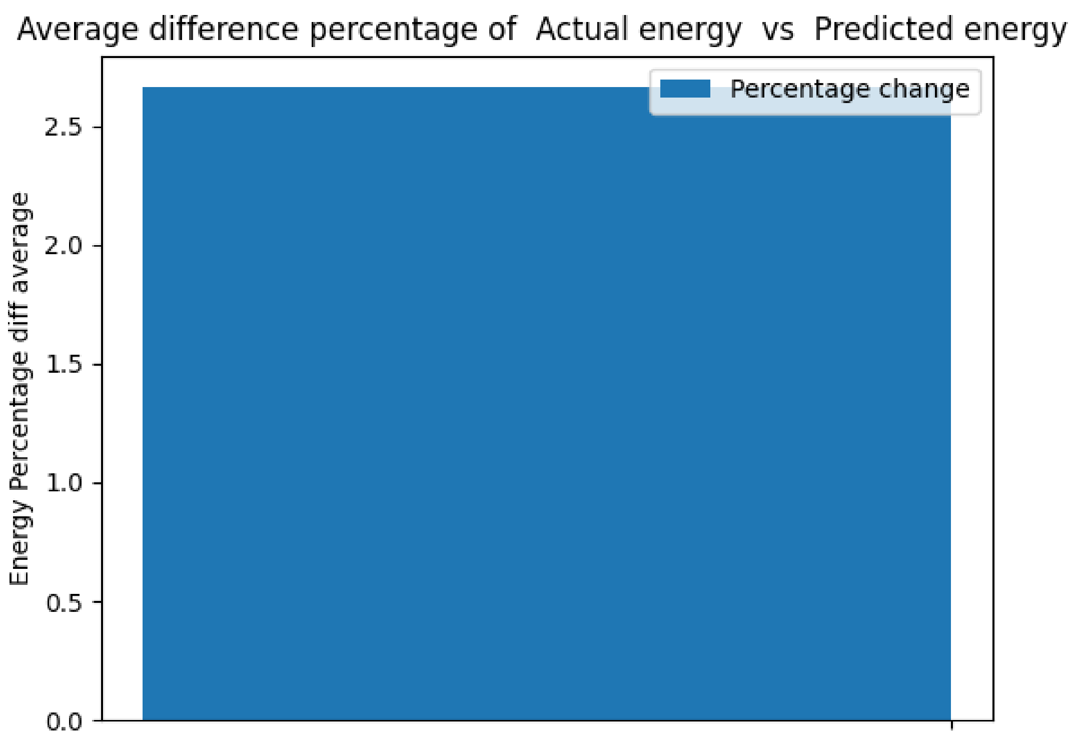


Figure 6.7: *Average error percentage for all dataset*

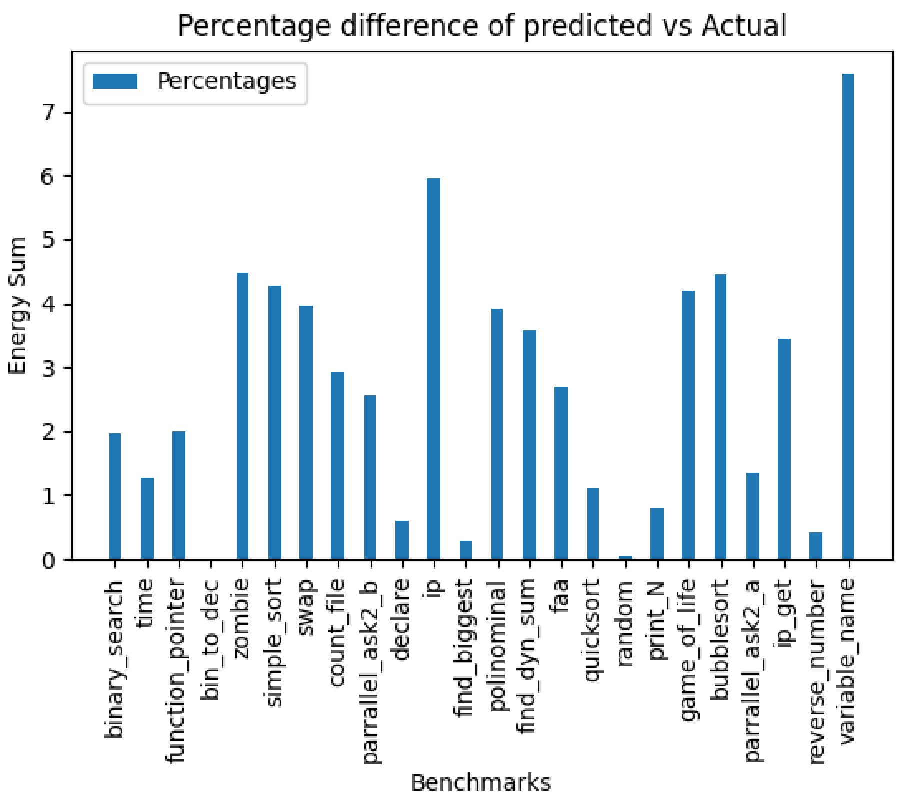


Figure 6.8: *Error percentage for each benchmark*

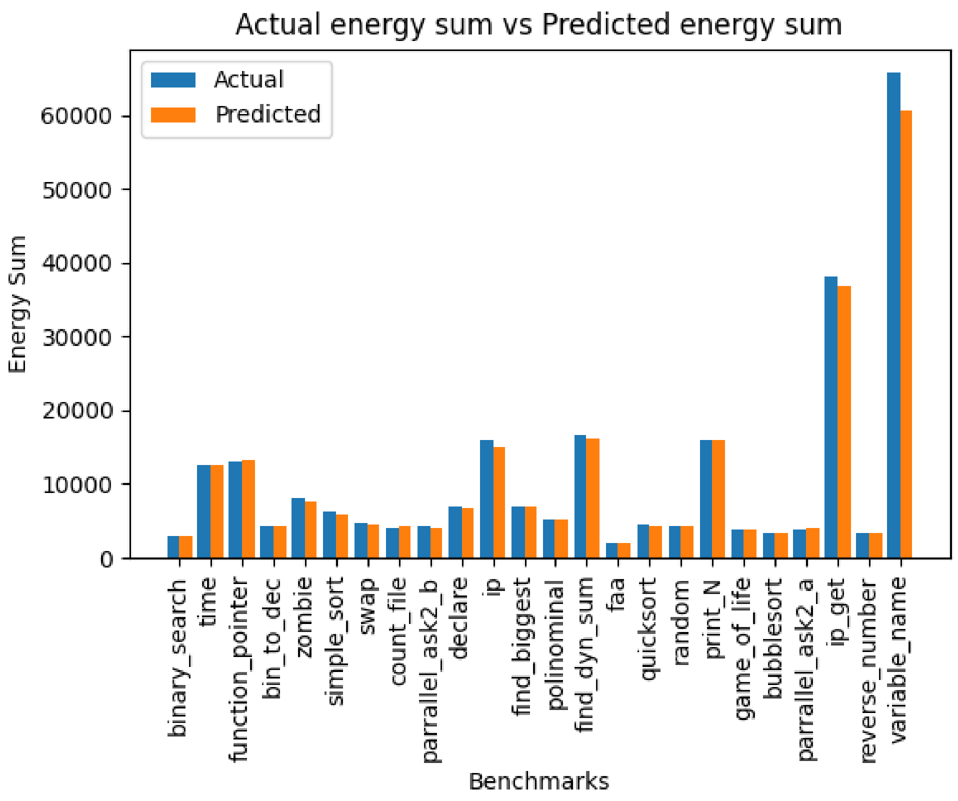


Figure 6.9: *Real vs predicted energy for each benchmark*

6.4 Comparison with related work

6.4.1 ALEA: Fine-grain Energy Profiling with Basic Block Sampling

The most similar project to this thesis is [1], a tool to measure power and energy consumption at the granularity of basic blocks, using a probabilistic approach. The work accomplished similar granularity to ours and comparable results (1.5% to 3.4%). But our tool is not following a probabilistic method but a deterministic approach since all the basic blocks are measured not a sample of them. Additionally our tool is open source and thus available for anyone interested to download and use it .

6.4.2 Tools based on direct power measurement

There has been a number of tools [30], [31], [32] that, based on direct power measurement can accurately measure both component-level and system-wide energy consumption, before and after the system's power supply units. However, the time granularity of the sensors fundamentally limits these tools and as a result the granularity of the measurements is much lower. So code objects such as basic blocks or even function cannot be measure, contrastingly to our work.

6.4.3 Tools that model energy consumption from activity vectors

Tools that model energy consumption from activity vectors can break the granularity barrier of direct energy measurements but suffer from several other shortcomings. Their accuracy is usually limited and highly dependent on architectural variations between platforms and workload patterns[33],[34],[35], These tools have a very extensive time consuming training and benchmarking process that must be repeated per platform and yield errors of more than 4% at the majority of cases.

Chapter 7

Conclusion and Future Work

7.1 What did we accomplish?

So we have reached this point where our energy dataset has been created and its validity and accuracy have been ensured. But is this all we have accomplished? Just an energy dataset that will give very good results but only for our computer or computers that are similar to it? The answer is of course not. Our work is far greater than a single dataset for a single computer. The real accomplishment of this work is not the production of an energy dataset but the production of an energy dataset producing mechanism that is completely open source and free to use.

Any developer can download our tool. Then they have to install the open source LLVM infrastructure and the Intel PT tools that are needed for the execution trace (to do that you need an Intel CPU).Then they can either use our benchmarks or make their own custom benchmarks and execute the tool which will produce their own custom energy dataset tailored to their computer's energy consumption.

In this way, developers can be sure that their dataset is as accurate as possible and in the case they notice any shortages of basic blocks types they can enrich the dataset with their own benchmarks.

We hope that with this work we provide the ability and the incentive to more developers to produce custom software for their systems with energy efficiency at heart, so that they can benefit from heat and monetary savings, in particular during the energy crisis we are currently going through and additionally our planet can benefit from reduced GHG emissions, in particular CO_2.

7.2 Current extensions - Energy Prediction

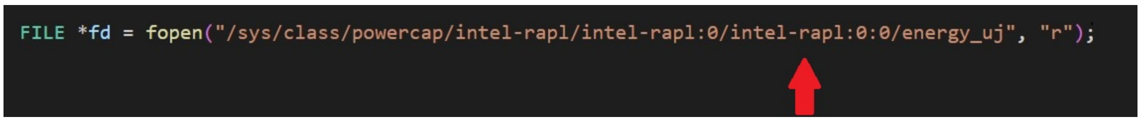
There are already two projects underway that aim to expand this work :

- An alteration of this work to predict memory energy (or even GPU energy), not CPU energy like this thesis.
- A predictive mechanism (neural network) that will be trained on this dataset and provide energy predictions on a basic block level for any program without the need

for execution.

7.2.1 Memory energy prediction

There really is not much to say about this alteration of the original thesis. Everything works in the exact same way, with the only difference being that instead of the RAPL read being performed at the energy register for the CPU they will be performed at the energy register for memory. So the only change necessary is the location of the energy register at the 3 RAPL read function we include to the program with the LLVM pass. The same can be done for GPU energy.



```
FILE *fd = fopen("/sys/class/powercap/intel-rapl/intel-rapl:0/intel-rapl:0:0/energy_uj", "r");
```

Figure 7.1: *Location of the RAPL register address we must change*

7.2.2 Energy predicting neural network

Currently our fellow Electrical and Computer Engineer undergraduate student Theodoros Siozos is concluding a continuation of this work at the Microprocessors and Digital Laboratory of the National Technical University of Athens. He used our dataset to train multiple neural networks with custom and pre-made basic block embeddings which have yielded very encouraging results. Using his mechanism any developer can insert one of their compiled executables and he will get an energy prediction for every basic block that make up their program. Those energy predictions can be the basis for code transformations with energy reduction in mind, which is after all the main goal of our work.

7.3 Future Work - Energy Reduction

As we stated previously the goal is to decrease the energy cost of software. The first step to do that though is to be able to measure the energy that the code consumes. With our work and the predictive mechanism that Mr. Siozos has produced we have accomplished that. Now the next spet should be focused on the utilization of the tools that we have provided so that software energy reduction mechanisms can be designed. A first idea are energy transformation techniques with works like [36],[37] but the sky is the limit since this field of research is just beginning to grow.

Βιβλιογραφία

[1] Lev Mukhanov, Dimitrios S. Nikolopoulos και Bronis R. De Supinski. *ALEA: Fine-Grain Energy Profiling with Basic Block Sampling*. *2015 International Conference on Parallel Architecture and Compilation (PACT)*, σελίδες 87–98, 2015.

[2] Juan David Tuta Botero. *Differences between static and dynamic libraries*.

[3] Zakaria Ournani. *Software eco-design : investigating and reducing the energy consumption of software*. Διδακτορική Διατριβή, Université de Lille, 2021.

[4] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair και Adrian Friday. *The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations*. *Patterns*, 2(9):100340, 2021.

[5] Hannah Ritchie, Max Roser και Pablo Rosado. *CO_2 and Greenhouse Gas Emissions*. *Our World in Data*, 2020. https://ourworldindata.org/co2-and-greenhouse-gas-emissions.

[6] Mastooreh Salajegheh. *Software Techniques to Reduce the Energy Consumption of Low-Power Devices at the Limits of Digital Abstractions*. Διδακτορική Διατριβή, Open Access Dissertation, 2013.

[7] M.T. Schmitz, B.M. Al-Hashimi και P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Springer US, 2006.

[8] Jawad Haj-Yahya, Avi Mendelson, Yosi Ben-asher και Anupam Chattopadhyay. *Energy Efficient High Performance Processors Recent Approaches for Designing Green High Performance Computing*. 2018.

[9] Mahendra PratapSingh και Manoj Kumar. *Evolution of Processor Architecture in Mobile Phones*. *International Journal of Computer Applications*, 90, 2014.

[10] Muhammad Zulhusni. *AMD delivers the latest and world's fastest processors to the modern data center*, 2022.

[11] J.A. Paradiso και T. Starner. *Energy scavenging for mobile and wireless electronics*. *IEEE Pervasive Computing*, 4(1):18–27, 2005.

[12] Thomas Benjamin, Daniel Bailey, Kevin Fu, Ari Juels και Tom O'Hare. *Vulnerabilities in First-Generation RFID-Enabled Credit Cards*. τόμος 4886, σελίδες 2–14, 2007.

[13] Giuseppe Procaccianti. *Energy-Efficient Software*. Διδακτορική Διατριβή, 2015.

[14] José A García-Berná, José L Fernández-Alemán, Juan MCarrillo de Gea, Ambrosio Toval, Javier Mancebo, Coral Calero και Félix García. *Energy efficiency in software: A case study on sustainability in Personal Health Records*, 2021.

[15] Gustavo Pinto και Fernando Castor. *Energy Efficiency: A New Concern for Application Software Developers*. *Commun. ACM*, 60(12):68–75, 2017.

[16] Mastooreh Salajegheh. *Software techniques to reduce the energy consumption of low-power devices at the limits of digital abstractions*. Διδακτορική Διατριβή, University of Massachusetts Amherst, 2013.

[17] *the llvm compiler infrastructure project*.

[18] *Clang: A C language family frontend for LLVM*.

[19] *LLVM's analysis and transform passes*.

[20] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann και Doron Rajwan. *Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge*. *IEEE Micro*, 32(2):20–27, 2012.

[21] *RAPL power API*.

[22] *Reading RAPL energy measurements from Linux*.

[23] *PERF tools support for Intel® Processor Trace*.

[24] *Intel® 64 and IA-32 architectures software developer manuals*.

[25] *Intel Perf linux man page*.

[26] V. Tiwari, S. Malik, A. Wolfe και M.T. C. Lee. *Instruction level power analysis and optimization of software*. *Proceedings of 9th International Conference on VLSI Design*, σελίδες 326–328, 1996.

[27] Agner Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Τεχνική αναφορά με αριθμό, Technical University of Denmark, 2022.

[28] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[29] Alessandro Di Federico, Mathias Payer και Giovanni Agosta. *Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries*. *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, σελίδα 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[30] J. Flinn και M. Satyanarayanan. *PowerScope: a tool for profiling the energy usage of mobile applications*. *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, σελίδες 2–10, 1999.

[31] Liwei Guo, Tiantu Xu, Mengwei Xu, Xuanzhe Liu και Felix Xiaozhu Lin. *Power Sandbox: Power Awareness Redefined*. *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung Ching Chang, Dong Li και Kirk W. Cameron. *PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications*. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.

[33] G. Contreras και M. Martonosi. *Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events*. *2005 International Symposium on Low Power Electronics and Design*, 2005.

[34] C. Isci και M. Martonosi. *Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data*. *36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[35] F. Blagojevic D. S. Nikolopoulos B. R. de Supinski M. Curtis-Maury, A. Shah και M. Schulz. *Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores*. *7th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[36] Eui Young Chung, Luca Benini και Giovanni De Micheli. *Source Code Transformation Based on Software Cost Analysis*. *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, σελίδα 153–158, New York, NY, USA, 2001. Association for Computing Machinery.

[37] Carlo Brandolese, William Fornaciari, Fabio Salice και Donatella Sciuto. *The Impact of Source Code Transformations on Software Power and Energy Consumption*. *Journal of Circuits, Systems, and Computers*, 11:477–502, 2002.