**PROGRAM VECTORIZATION FOR REDUCING ENERGY CONSUMPTION IN EMBEDDED SYSTEMS**

By

*Arnaldo J. Cruz-Ayoroa*

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

December, 2014

Approved by:

| | |
|---|---|
| Manuel Jiménez, Ph.D.<br>Chairman, Graduate Committee | Date |
| Rafael Arce-Nazario, Ph.D.<br>Member, Graduate Committee | Date |
| Nayda Santiago, Ph.D.<br>Member, Graduate Committee | Date |
| Pedro Vásquez-Urbano, Ph.D.<br>Graduate Studies Representative | Date |
| Pedro I. Rivera-Vega, Ph.D.<br>Department Chairperson | Date |

Abstract of Thesis Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Master of Science

**PROGRAM VECTORIZATION FOR REDUCING ENERGY CONSUMPTION IN EMBEDDED SYSTEMS**

By

*Arnaldo J. Cruz-Ayoroa*

December 2014

Chair: Dr. Manuel Jiménez
Department: Electrical and Computer Engineering Department

When it comes to software optimization, speedup is the first goal that comes to mind. However, as the integration level of electronic circuits continues its exponential growth, power reduction has also become an important goal. In recent years, the proliferation of mobile devices has also been a driver for reducing energy consumption in embedded systems. Although hardware engineers are already well acquainted with design techniques for low power consumption, software power reduction is still a vastly unexplored topic particularly in the area of compilation; even though, ultimately, software is the main responsible of making efficient use of the hardware.

This work proposes a machine learning driven optimization flow that uses a high-level characterization approach on a program's source code and training with optimization profitability measurements to predict whether to apply a particular optimization if energy consumption is expected to be reduced. The optimization studied in this work is called vectorization, which was found to be not only powerful for reducing execution time, but when applied correctly to also reduce power and energy consumption. Experiments were conducted on an implementation of the popular ARM Cortex-A8. Nevertheless, this methodology is not limited to this particular embedded architecture. A predictor was trained which

decided whether vectorization would have a beneficial or detrimental impact with 74% precision, resulting in an average of 64% decrease in energy consumption and only 5% increase in energy in the case of mis-predictions.

Resumen de tesis presentado a la Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
requerimientos para el grado de Maestría en Ciencias

**VECTORIZACIÓN DE PROGRAMAS PARA REDUCIR EL CONSUMO ENERGÉTICO EN SISTEMAS EMBEBIDOS**

Por

*Arnaldo J. Cruz-Ayoroa*

Diciembre 2014

Consejero: Dr. Manuel Jiménez
Departamento: Ingeniería Eléctrica y Computadoras

Cuando se trata el tema de optimización de software, la primera idea que viene a la mente es la reducción en el tiempo de ejecución. Sin embargo, a medida que el nivel de integración en los circuitos integrados continua un crecimiento exponencial, la reducción de potencia también se convirtió en una meta importante. En años recientes, el consumo energético también ha ganado importancia debido principalmente a la proliferación de dispositivos móviles. Los ingenieros de hardware están familiarizados con técnicas de diseño para bajo consumo de potencia. Sin embargo, este tema ha sido poco explorado en el campo de la compilación, aún cuando el software es finalmente el responsable de hacer uso eficiente del hardware.

Este trabajo propone un sistema de compilación guiado por técnicas de aprendizaje automático que utiliza una caracterización de alto nivel del código fuente de un programa y entrenamiento con medidas de beneficio por optimización, para predecir si aplicar una

optimización en particular resultaría en una reducción en el consumo energético. La optimización estudiada en este trabajo se llama vectorización, la cual encontramos que puede ser efectiva en reducir no solamente el tiempo de ejecución, sino que también cuando es aplicada correctamente puede reducir el consumo de potencia y energía. Los experimentos fueron realizados en una implementación del reconocido ARM Cortex-A8, aunque la metodología no está limitada a esta arquitectura embedida. Logramos entrenar un predictor con la capacidad de decidir si aplicar vectorización tendría un impacto beneficioso o perjudicial con una precisión de 74%, alcanzando una reducción promedio en energía de 64% y solo 5% de incremento en energía en los casos de predicción errónea.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# INTRODUCTION

Since the invention of the integrated circuit (IC), the integration density has been growing exponentially and, up to recent years, this has also been the case for the operational frequency. As a consequence, the power density for a given IC technology has also followed this trend [4]. Power density translates to heat, and sufficiently high heat can degrade the electrical properties of semiconductor materials and shorten the lifetime of an IC. Therefore power consumption has become an important research topic to enable continued advancements in electronics while maintaining the reliability of the devices that use them.

Environmental issues are a more recent motivation for including power as a design constraint. The movement for what has been called *green computing* covers the entire lifecycle of an electrical device and seeks to produce devices that cause less contamination. This includes engineering the hardware and software of embedded systems to efficiently use energy.

Another reason for designing energy efficient systems is the proliferation of battery-operated devices. For some of these devices it is impractical to replace their batteries, examples being electronic surgical implants, sensor networks, and space probes. Mobile communication and entertainment devices are also very popular, and are the economical driving force behind modern computing advancements. Mobile devices are now more popular than ever and have more constraints that general purpose computers. As an example, users want these products to operate on a single charge for as long as possible. Yet advancements in battery technology have not been able to keep up with advancements in mobile devices.

The purpose of this research was to develop a software energy optimization methodology and an implementation that could be used to reduce the energy consumption in embedded systems.

Power optimization can be focused on reducing power fluctuations or on reducing the total energy consumption. Reducing power fluctuations may be needed to prevent damage to the circuit or to comply with standards. However, energy was chosen as the metric of interest in this project because it has broader implications. Reducing energy implies reducing the integral of power in time and doing this will also mitigate the heat related problems that were mentioned previously.

Power management can be implemented in hardware, software, or both. Hardware power management is common in today's electronics. Examples include dynamic frequency scaling and power gating, where unused components can be deactivated or set to sleep mode [4].

This project details the development of a compile-time software solution. Despite the potential savings that could arise from software energy optimization, there are few studies on the subject and there are no standard methodologies for dealing with the problem. A software solution is also more flexible by nature, as it could be applied to existing programs, and would not require hardware modifications. There are also no run-time penalties because program modifications can be performed during compilation. Although hardware should be designed for efficient power management, its design has to be general enough to allow acceptable performance in different types of programs. It is up to the program designer how to efficiently use the hardware resources.

This thesis includes a comprehensive survey on related works that study software-level energy estimation and optimization techniques. We show that vectorization is a powerful optimization technique, that although commonly applied in the context of program speedup, it can also be targeted at power and energy reduction. We integrated this optimization to

a modern compiler approach called machine learning driven compilation. Machine learning algorithms are models that can make use of previous experiences to predict future outcomes.

In this work we trained a machine learning module to instruct the GCC compiler whether to apply vectorization or not, depending on whether it was predicted to have a beneficial or detrimental impact on energy consumption. Experiments were conducted on an implementation of the ARM Cortex-A8, the most popular architecture used in mobile devices today. The performance and assembly of program examples were analyzed to validate the methodology we proposed in this project.

# Chapter 2
# THEORETICAL BACKGROUND

This chapter provides details on the platform chosen for experimentation, the optimization employed to reduce energy consumption, and the optimization driver. Section 2.1 gives an overview of the ARM architecture and power estimates per core module. This serves as motivation for Section 2.2 which describes vectorization, a powerful data parallelization technique with potential for energy reduction. Section 2.3 defines machine learning and how it can be combined with compilers to assist in predicting beneficial code optimizations.

## 2.1 Power consumption in the ARM architecture

ARM has become the most widely used embedded processor architecture [1]. There are seven versions of the ARM architecture; the first three are now obsolete and the latest one is called ARMv7. The ARM architecture is described in the Architecture Reference Manual, sometimes referred to as the ARM ARM. The latest version ARM is divided into three profiles; application (ARMv7-A), real-time (ARMv7-R), and microcontroller (ARMv7-M). Thus the ARM manual is now divided into the ARMv7-AR Architecture Reference Manual and the ARMv7-M Architecture Reference Manual. Each profile is implemented as a family of cores called Cortex and are described in their respective Technical Reference Manual (TRM) [a] .

The purpose of dividing the architecture into profiles is for better accommodating it to different markets. ARMv7-A is intended for high-performance applications such as tablets,

[a] http://infocenter.arm.com/help/index.jsp

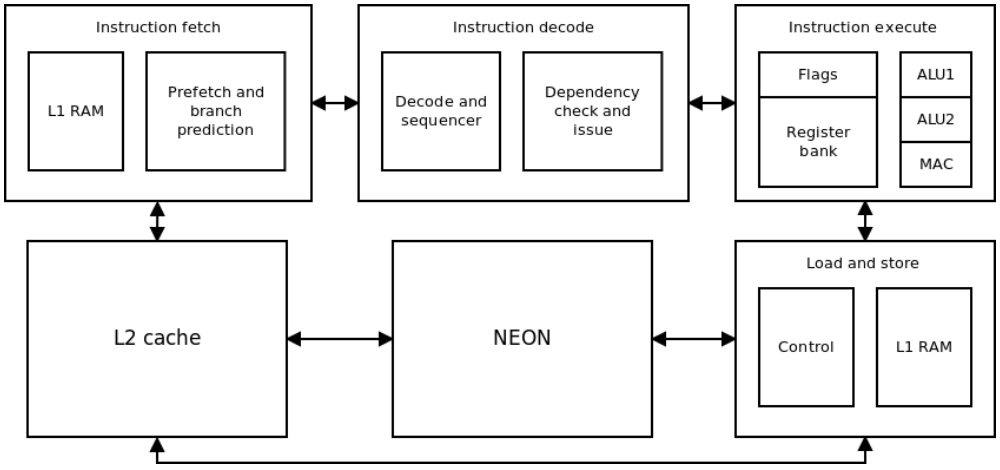



Figure 2.1 : The ARM Cortex-A8 architecture.

smartphones, and mobile gaming consoles that use complex operating systems and multimedia. ARMv7-R and ARMv7-M are intended for deeply embedded industrial applications and low-cost electronics where a simple operating system or no operating system is used. ARMv7-M can be considered a subset of ARMv7-R and the main difference is that the real-time profile operates at a higher frequency while the microcontroller profile is designed for fast interrupt processing.

Figure 2.1 shows a simplified block diagram of the ARM Cortex-A8, the microprocessor architecture of many System-on-Chip (SoC) devices. The main feature that characterizes an ARM as a RISC architecture is a load/store architecture, fixed-width instructions that usually execute in one cycle, and simple addressing modes. The register file is 32-bit wide and 16 words in depth. It implements a Harvard memory model where the data and instructions are stored in different memory caches. There are two Arithmetic Logic Units (ALU) and one Multiply And Accumulate (MAC) pipelines for parallel integer operations. The memory model consists of a single, flat address space of $2^{32}$ bytes and may have up to seven levels of cache. It can be addressed as bytes, 16-bit half-words, or 32-bit words with limited support for doublewords. Instruction access must be aligned and data access can be unaligned. In addition, data can be managed as both little and big endian.

The ARMv7 architecture specifies five instruction sets (IS). Some cores support all of them while others only one. The regular ARM IS consists of 32-bit instructions. The Thumb IS provides 16-bit instructions with a subset of the functionality of the ARM IS. It allows for a trade-off between code density and performance since Thumb instructions can be used to emulate ARM instructions but sometimes at the expense of more execution cycles. The Thumb-2 IS extends Thumb with 32-bit instructions. It provides better code density than Thumb but with performance and functionality similar to ARM. Using the Unified Assembler Language (UAL) both ARM and Thumb-2 can be assembled from a single assembly source. The architecture also supports the execution of Java bytecode through the Jazelle IS. Its successor is called ThumbEE (Execution Environment) and provides hardware acceleration to dynamically generated code.

ARMv7 provides three extensions which are not required in the implementation. The first extension is called Vector Floating-Point (VFP) and adds IEEE 754 single- and double-precision floating-point support. The next two extensions add SIMD support (see Section 2.2). The Advanced SIMD (NEON) extension can handle integer and single-precision floating-point vector operations and is included in the ARM Cortex-A8, as shown in Figure 2.1 . The DSP extension adds a subset of this functionality to the ARMv7-M profile.

Texas Instruments' OMAP3530 SoC makes for an appropriate case study for this work because it includes an ARM Cortex-A8 core with the NEON SIMD instruction processing module, along with other subsystems typical of the SoCs that are used in mobile devices. Figure 2.2 and Table 2.1 show the power consumption distribution per subsystem. These values were obtained from the maximum estimated values of the OMAP3530 power estimation spreadsheet [2] [3].

The three components with highest power consumption are the ARM core (43%), the DSP core (16%) and the memory and memory controllers (13%). Thus, code optimizations should focus on making efficient use of these three subsystems. The consumption of

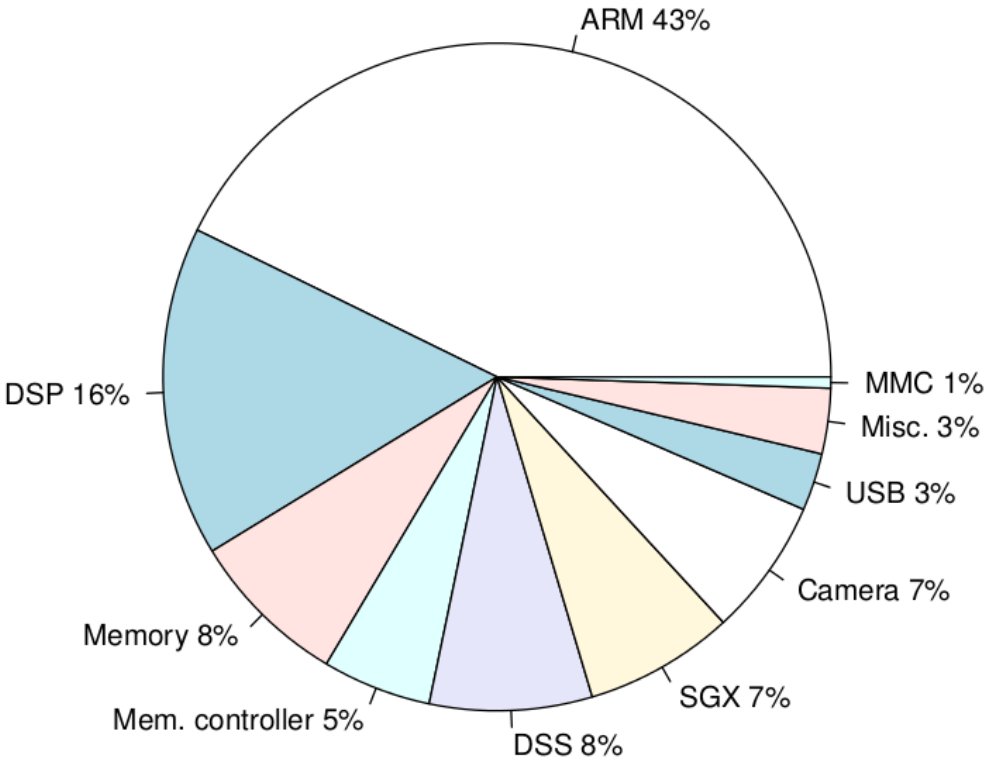



Figure 2.2 : Distribution of the estimated power consumption on the OMAP3530 [2] [3].

| Module | Power (mW) | Description |
|---|---|---|
| ARM | 548.64 | ARM Cortex-A8 with NEON @500MHz |
| DSP | 205.29 | Digital Signal Processor |
| Memory | 99.20 | DDR2 SDRAM @162MHz [3] |
| Memory controller | 67.66 | SDMA (System Direct Memory Access) and SDRC (SDRAM) controllers @166 MHz |
| DSS | 100.95 | Display Sub-System |
| SGX | 93.0 | 2D/3D Graphics Accelerator Engine |
| Camera | 85.46 | Camera and Image Signal Processor |
| USB | 35.71 | Universal Serial Bus |
| Miscellaneous | 40.72 | General purpose timers, general purpose I/O, I2C, SPI, and 1-wire |
| MMC | 6.89 | Multi Media Card controller |

Table 2.1 : Estimated power consumption on the OMAP3530 [2] [3].

the display and storage devices are also be significant, but optimization strategies for the utilization of these devices are beyond the scope of this work.

## 2.2 Single instruction multiple data and vectorization

A vector instruction, or Single Instruction Multiple Data (SIMD), is a type of instruction that operates over fixed-size vectors of data of the same type. This is in contrast to the more commonly used sequential instruction paradigm, which in the context of vector instructions is called Single Instruction Single Data (SISD). SIMD instructions, just like SISD, can perform computation and data transfer operations.

Vector processors have existed for several decades but only recently have they started to become popular in consumer devices. The reason is that modern processors have been designed to operate on scalar data and to rely on instruction-level parallelism to increase their throughput. The result is increasingly more complex architectures. Vector processor architectures take a different approach because they exploit data-level parallelism. Instead of executing multiple instructions in parallel, a vector processor can execute single instructions that operate on vectors of data. This simpler design can yield a much higher throughput in programs that have high data-level parallelism, such as scientific and multimedia applications which usually deal with matrix and streaming data. Some modern processor architectures, however, combine both aspects of parallel execution and parallel data processing. We now examine a generic vector processor architecture as described by Hennessy and Patterson [4].

Vector processors can be classified as memory-memory or vector-register. In memory-memory vector processors all operations are carried in memory while vector-register processors require operands to be loaded from memory into registers before carrying out an operation. Vector-register processors are more common and thus will be assumed in the rest of the discussion. The basic components of a vector processor are similar to a SISD processor, and include vector and scalar register files, the vector functional units, the vector load-store unit, and banked memory.

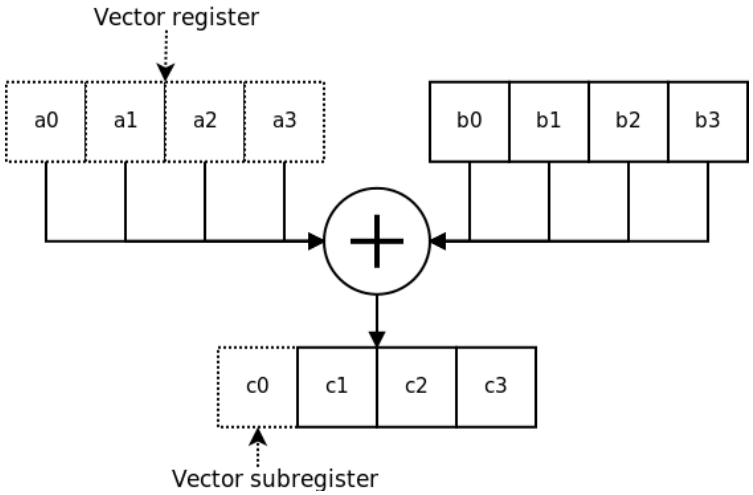



Figure 2.3 : Vector addition with vectorization factor of 4.

Figure 2.3 shows a vector addition operation as an example. There are two input vector registers and an output vector register. Registers in the vector register file are divided into sub-registers. Each sub-register can hold a vector element, and the total number of sub-registers is called the vectorization factor. The vectorization factor in Figure 2.3 is 4. The vector element data type may be fixed or variable, but every element in the sub-register must have the same data type. Element data types may include signed and unsigned integer, floating-point, and polynomial.

Vector functional units are similar to their scalar counterparts. There are functional units for arithmetic and logic, multiply and accumulate, and shift operations. The main difference to scalar processors is that besides providing vertical parallelism through pipelining, they also provide horizontal parallelism in the form of lanes. Each additional lane duplicates a functional unit pipeline to process one more vector element per clock cycle. Since vector operations presuppose that there are no data dependencies between vector elements, lanes can be added to increase the throughput without the risk of data hazards. Vector processors can be single-issue or may issue more than one instruction at a time.

The vector load-store unit handles transfers between memory and the vector register file. Memory is banked and the data is interleaved between banks to be able to retrieve

multiple vector elements at the same time while reducing the memory access latency. Caches may or may not be present. To support access to data structures, memory may be accessed with a non-unit stride. A stride is the number of data units between vector elements. The structure elements may be de-interleaved during loads or interleaved during stores either by software or hardware. Software approaches load the stridden data into vector registers and use instructions to extract the elements to a different set of vector registers. Hardware approaches makes the process transparent to the programmer or compiler but still require more time than unit stride access.

Besides the stride, a second characteristic of memory access is data alignment. Similar to scalar processors, vectors need to be stored and loaded from memory addresses that are multiples of the vectorization factor. Some vector processors may not allow unaligned memory access, and must load the adjacent aligned vectors and reassemble the desired vector in another register. The process of loading adjacent vectors from memory to extract the desired data and work around the unaligned access limitation is called unpacking, while the reverse operation to store an unaligned vector is called packing. Packing and unpacking introduce an overhead because additional instructions have to be added to the program for this purpose.

There are several overheads that can be avoided by using vectorization. A single vector instruction executes the equivalent of many sequential operations, reducing the number of fetched instructions. Replacing a loop by a vector instruction guarantees that there are no loop branch control hazards and no data hazards. In addition, the instructions to compare, update the counter variable, and branch in the original loop can be eliminated. Although no cache may be present, a single load or store instruction transfers a large amount of data from multiple memory banks, reducing the memory access latency. The loaded data has good spatial locality because most, if not all, of the data will be processed as a vector.

There are two ways in which programmers can employ vector instructions. The first is through the use of compiler intrinsics which resemble function calls but are directly

translated by the compiler into a SIMD instruction. A second approach is to let the compiler transform sequential code into vector code, a process known as automatic vectorization. Vectorization is a well known optimization technique which, if used correctly, can speedup loops several-times fold. It can be applied to loops that have no loop-carried dependencies, such as the code in Listing 2.1.

```
float a[110], b[110], c[110];
for (int i = 0; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 2.1: Vectorizable code; no loop-carried dependencies.

In order to vectorize this loop to perform four additions in parallel as shown in Figure 2.3 , the first step is to apply a code transformation called loop unrolling. In loop unrolling, the loop's body is replicated by an unroll factor and the index variable's upper boundary and increment are adjusted to have an equivalent computation. Modern compilers can perform this procedure automatically. As a preparation for vectorization, the loop is unrolled the same number of times as the vectorization factor (VF) as shown in Listing 2.2.

```
float a[110], b[110], c[110];
for (int i = 0; i < 27; i += 4) {
    c[i+0] = a[i+0] + b[i+0];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
for (int i = 108; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 2.2: Unrolled loop for a vectorization factor of 4.

Note that the upper boundary for the index variable is also divided by the VF and is incremented by VF. Since the iteration range is not a multiple of the VF, a second loop had to be created to perform the remaining operations. This transformation of creating a new loop to perform the first or last iterations of the original loop is called loop peeling. Now the body of the first loop can be vectorized by substituting these SISD operations by a single SIMD operation, called *add4* in Listing 2.3. However, the remaining peeled loop remains as vectorization overhead.

```
float a[110], b[110], c[110];
for (int i = 0; i < 27; i += 4) {
    add4(c, a, b);
}
for (int i = 108; i < 110; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 2.3: Vectorized loop.

The reason vectorization can yield high speedups, as explained in this section, is that operations can be performed in parallel up to the VF and more importantly, memory accesses can be done more efficiently [5]. At the same time, underutilizing the vector registers and performing non-aligned memory access can make the vectorized code perform worse than sequential code. Compiler support is one of the main obstacles to taking advantage of the potential of vector processors. However, modern production compilers have limited support for automatic vectorization [6]. The reason is that compilers currently utilize heuristics and hand-built machine models to estimate optimization profitability which do not capture well enough the complexity of modern systems (in the case of vectorization, see for example the work of Trifunovic et al. [7]). In this work we study a more advanced technique based in machine learning that can aid the compiler in choosing the correct optimization.

## 2.3 Machine learning

Machine learning (ML) is a field in computer science concerned with the development of algorithms that can automatically find structure within data and predict future outcomes. Although it employs statistical techniques, it differs from statistics in its objective. Whereas in statistics the focus is to model and understand the data, ML places emphasis on prediction. That is, the usefulness of a ML model is measured by how well it can predict new

cases. It is important to understand that ML and statistics have different objectives, because there are data manipulation techniques that are used in ML in order to increase prediction precision, or the predictor's usefulness, that are considered inappropriate in statistics. The difference between both fields is eloquently explained by Breiman in his work titled *Statistical Modeling: The Two Cultures* [8]. The ML concepts to be presented in this section are based on Bishop's book titled *Pattern Recognition and Machine Learning* [9].

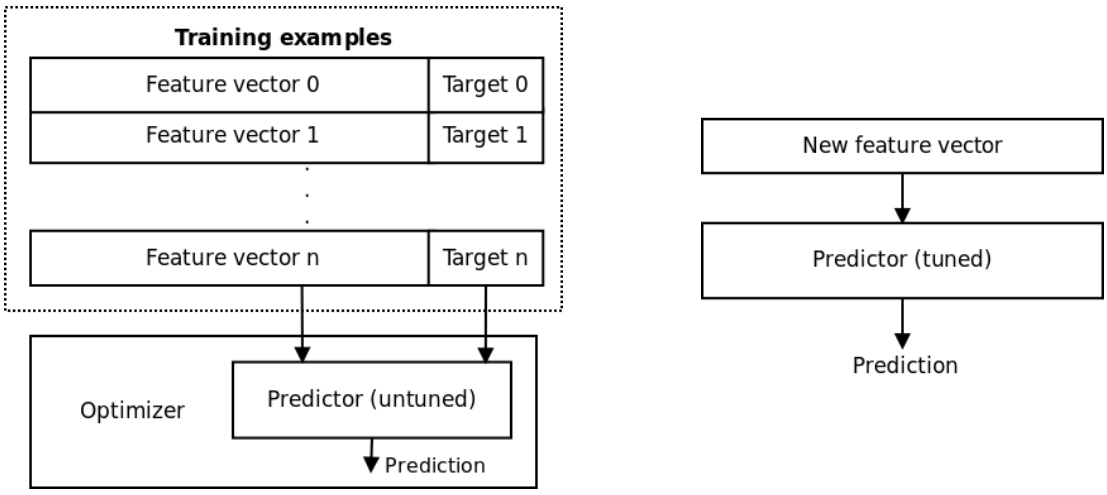
Supervised ML uses generic data models that are tuned based on previous experiences, otherwise known as *training examples*. These models treat the system under study as a black-box and are tuned using only experimental inputs and outputs. This is in contrast to analytical models which are constructed based on an understanding of the system. However, just like analytical models, the inputs and outputs consist of numerical values. In ML terminology, the inputs to the model are called the *feature vector* and the output the *target*. The feature vector is a characterization of the type of object to be recognized. The target is the expected output for a given feature vector. Defining an appropriate feature vector and target is one of the main challenges in applying ML. Experience and understanding of the problem domain is required to be able to define these parameters and to use ML successfully.

ML is subdivided into unsupervised and supervised type of algorithms. In unsupervised learning, we have training examples for which the targets are not known and we want to give some label to each point. A common application is the use of clustering algorithms which label groups of points that are close together. For example, we may want to classify programs in terms of the instruction mix, which is the number of counts per instruction. This could be useful to classify programs as computation or memory bound. The feature vector for each program would be the instructions mix and the unknown target or label would be the cluster to which it belongs. Applying a clustering algorithm would reveal which programs are similar in terms of their instruction mix and allow classification as

computation or memory bound. A well-known example of clustering algorithms is called *k-Means*. The objective of *k-Means* can be described as finding the coordinates of $k$ points such that the Euclidean distance of each point to its neighbor is minimized. At each iteration step the $k$ points are moved independently towards clusters of points until convergence, where each of the $k$ points becomes the centroid of a cluster of training examples. After training, prediction can also be performed by finding the closest centroid to the new feature vector.

In supervised learning, the second type of ML algorithms, we have training examples in which both the feature vectors and targets are known beforehand and we want to predict the target for a new feature vector. There are two types of predictions depending on the ML model being used. Regression models predict continuous values whereas classification models predict discrete values, which is a class to which the new feature vector belongs to. Regression is useful, for example, if we want to predict the execution time of a program. Classification, just as was explained for clustering, could be useful for classifying types of programs. Being a supervised method, the difference to clustering is that the type of program for the training examples is known beforehand and we want to predict the type of program for new feature vectors.

Figure 2.4 shows how a supervised ML model is trained and tested. The objective of the training phase is to create a set of training examples and to use it to tune a ML model to perform predictions. Each learning example is composed of a feature vector and a target value. The task of the optimizer is to input each feature vector to the predictor and to compare the prediction with the corresponding target, which is the expected output. The optimizer will repeat this process while at the same time tuning the model coefficients until the prediction error is minimized. Once the model is tuned it can be used in a testing or prediction phase, where a new feature vector is input to the predictor and a prediction is output. Again, this prediction can be a real value in the case of regression or a discrete value for classification.




(a) Training phase (b) Testing phase

Figure 2.4 : Training and testing phases for a supervised machine learning algorithm.

Having explained supervised learning is performed, we can now discuss two common supervised ML algorithms. Linear regression is the best known of the regression algorithms. The objective of linear regression can be stated as finding a line (or hyperplane in $n$-dimensions) that minimizes the Mean Squared Error (MSE) of the training examples. In other words, during the *training phase* the coefficients of the model are tuned using an optimizer to best align a line to the training examples. The model can now be evaluated during the *prediction phase* for a new feature vector to obtain a real value that lies within the line. In practice, some noise is added during the training phase so that the model does not overfit or is biased towards the training examples. A biased predictor in undesirable because it may not predict well for new feature vectors.

The Support Vector Machine (SVM) is a more advanced supervised ML technique because it can also account for non-linearity in the feature vector space. It was originally designed for classification but can be used for regression as well. In its most basic form it acts as a decision machine, that is, it outputs a single binary value which represents one of two classes. In an n-dimension feature vector space, let us consider a set of data that may be of two classes, 0 or 1. An SVM is a classifier, or a function that takes as input a feature

vector and outputs a predicted class as shown in Formula 2.1.

$$\mathbb{R}^n \longmapsto \{0, 1\} \tag{2.1}$$

To do so, it determines the position of a new feature vector relatively to a hyperplane that separates one class of data from another. This hyperplane is known as the decision boundary. More precisely, the optimization algorithm tunes the coefficients of the decision boundary to maximize the margin or distance from the boundary to a selected group of points called the support vectors. In the case where the data is not linearly separable, the feature vectors can be projected to a different feature space. This is known as the *kernel trick* and there are certain functions, such as Gaussian, that can be used for this purpose.

Another important aspect about ML is how to evaluate a predictor's performance during the testing phase. In this work we utilize the concepts of precision and recall [9]. Precision is defined as the ratio of the number of times the prediction matched the target to the total number of targets (or predictions). Recall is defined as the ratio of the number of times the prediction for one of the outcomes matched the target over the total number of that particular outcome. For example, assume that we are predicting whether to vectorize a program or not. The precision would be the number of times *should vectorize* or *should not vectorize* were predicted correctly divided by the total number of targets. And the recall for *should vectorize* would be the number of times *should vectorize* was predicted correctly divided by the total number times *should vectorize* was a target. Recall would also be computed the same way for the *should not vectorize* outcome.

A naive testing approach would train a model with the training examples and test it with the same data, but this would of course produce biased results. For practical purposes we would like to evaluate whether the predictor is capable of predicting correctly for new inputs. Prior to the training phase, precision and recall evaluation is performed through a process called *n-fold cross-validation*. Cross-validation refers to partitioning the training examples into a training set and a test set, which are used respectively to train the model and

test its precision and recall. When n-folds are used, the training examples are partitioned into n chunks of data. Each chunk is used in progression as the test set, while the remaining *n - 1* chunks are are used as the training set. This process gives a better sense of the predictor performance than a single cross-validation because each point is evaluated in both training and testing phases.

The last topic to be discussed in this section is how ML can be used to drive a compiler optimization process. Figure 2.5 shows one possible configuration in which ML is performed as a module separate from the compiler. The block labeled *machine learning* contains one or more predictors that are used to inform the compiler as to which optimizations to apply and with which parameters, according to the feature vector extracted from the input source code. This we call the *optimization scenario*. The ML component will not take over the compiler entirely but will assist in the optimization process. Since the predictors treat the compiler as a black box, any internal transformations that the compiler performs will be captured during the training phase. The optimization scenario can be communicated to the compiler through command line flags, code annotations, and special programming interfaces provided by compilers such as GCC or LLVM.

An initial set of training examples is generated to train the predictor. This can be done using benchmark source codes which are representative of the types of programs that will be compiled during the testing phase. For each benchmark, a feature vector is extracted, the program is compiled, and its performance is measured. The predictors can then be trained off-line with the training examples for a particular compiler and target system. Thus when the predictors are trained, all the system complexity is hidden within tuning coefficients.

On the testing phase, the compiler transforms and outputs the executable based on the ML module predicted optimization scenario. When the program is executed, the feature vector and target performance can be fed-back into the training examples. In this way, the machine learning module can perform continuous tuning of the models to predict for

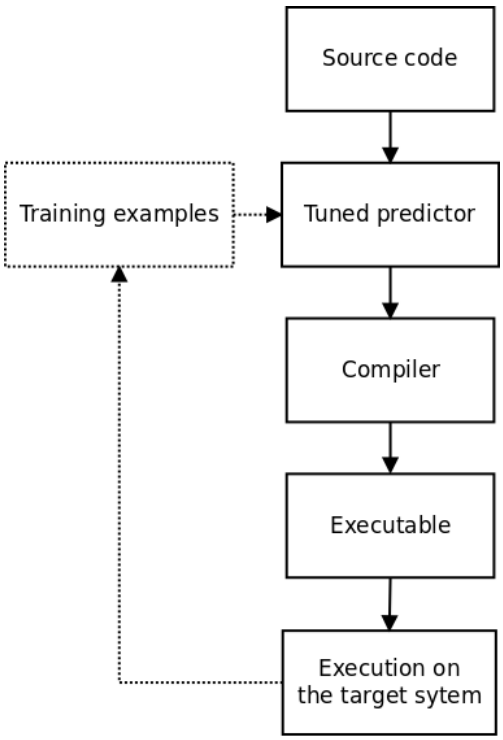



Figure 2.5 : Machine learning driven compilation system.

new types of programs. This last point, however, is an open research topic and was not considered as part of this work [10].

# Chapter 3
# PREVIOUS WORK

In Section 3.1 we survey related works that explore program transformations and how they affect power and energy consumption. Section 3.2 discusses works that take a traditional approach at driving the compiler optimization process. Traditional approaches require the laborious effort of developing performance estimators and heuristics to decide which optimization scenario to apply. These works in particular are focused on the estimation of software power and energy consumption. Finally Section 3.3 discusses works that take a more modern approach and use machine learning driven compilation.

## 3.1 Power and energy reduction

The techniques discussed in this section aim at modifying the source code in a way that lowers energy consumption. These modifications can be at the low-level assembly code, at an intermediate code representation, or at the high-level source code. Since the solution space is very large, heuristics are used to find a solution that has a lower consumption. These heuristics are guided by an estimation strategy and may transform the code in a single pass or iteratively.

### 3.1.1 Low-level techniques

The energy consumption of a processor depends on its previous state and the current inputs, therefore one method for lowering the energy consumption is to reschedule, or reorder, a program's assembly instructions. The problem with this method is that finding the optimal schedule is classified as an NP-hard problem. Given $n$ instructions, there are ($n$-1)! / 2 possible combinations. Kyu-Won and Chatterjee proposed a rescheduling method that

formulated the problem as the *Traveling Salesman Problem (TSP)* for the graph representation of the program of interest [11]. Simulated annealing was applied to find an instruction ordering that provided a near-optimal low-power solution. The simulator used in the study was SimpleScalar version 3.0 with an RTL-level power model. Results compared the power consumption when rescheduling was used and when it was not. The maximum energy reduction observed was 29.19% and the minimum 2.68% with a 95% confidence interval. No time results were reported for the heuristics used in the optimization and therefore it is not known if this is a practical approach.

Krishnaswamy and Gupta report that a significant amount of the energy consumption in a processor is due to the memory cache [12]. In the case of ARM processors, the cache can consume up to 40% of the total energy. It is possible to reduce the code size and energy consumption in an ARM processor through the use of a secondary instruction set called Thumb. The original Thumb instruction set consists of 16-bit instructions instead of the regular width of 32-bit. Using these instructions may reduce not only the code size, but also decrease the energy consumption in the instruction cache (I-cache) because two Thumb instructions are usually fetched for each cache access. This reduces the number of cache accesses and misses. The use of Thumb instructions, however, does not produce programs with half of the code size. The reason is that additional instructions may be required to emulate the original ARM instructions, in some cases resulting in a size increase of 9% to 14% [12]. This increase in code size will result in a lower performance, or a higher number of execution cycles, and may produce a higher I-cache energy consumption when compared to an ARM version of the program.

The objective of Krishnaswamy and Gupta's work was to reduce the code size and cache energy consumption of a given ARM program while maintaining good performance. The authors developed and compared a series of heuristics that received as input the ARM assembly program and would replace sections of the code with equivalent Thumb instructions. While the method was successful at reducing the code size and maintaining good

performance, the I-cache energy consumption remained close to that of the original version of the program. Since the publication of this work, updated versions of Thumb and new instruction sets have been added to the ARM architecture. It is possible that applying similar techniques to those developed in this work with these new instructions might result in lower energy consumption.

### 3.1.2 High-level techniques

Without formal experimental methods it is not possible to conclude that an observed result was caused by some experimental factor. The work of Ortiz and Santiago studied the impact of some high-level source code optimizations on the power dissipation of an embedded system and used Design of Experiment (DOE) techniques to avoid this problem [13]. The reason for optimizing the code at a high level was to take advantage of the portability, readability, and maintainability that is provided at this level. In addition, it was of interest to see if power was indeed reduced after compilation and when tested with different architectures.

Source code optimizations can be classified as algorithmic optimizations, loop transformations, and function inlining methods. Three techniques were chosen: loop unrolling, function inlining, and type replacement. Loop unrolling copies the body of the loop several times to reduce the overhead of comparison and branch instructions. Function inlining replaces a function call with the body of the function and thus reduces the call overhead. Type replacement aims to replace a variable's type with another type that tends to consume less energy.

Three platforms were chosen for experimentation; Intel 8051 8-bit CISC, Motorola HC12 16-bit CISC, and ARM7TDMI 32-bit RISC. The benchmarks were selected from MiBench, which is used to test the performance of embedded processors. These benchmarks were profiled to determine the sections of code where most of the execution time was spent. Optimized and unoptimized versions of the benchmarks were run in an infinite loop, the

current was measured from the processor's power line and was multiplied by the supply voltage to obtain the average power consumption.

An experiment was designed to conclude if the changes in power could be attributed to a particular optimization. A *full factorial* experimental design was used, the factors being the optimization technique and benchmark. The response variable was the power consumption for a given platform. The *null hypothesis* was that optimization techniques and benchmarks do not have an impact on the power consumption. With a 95% of certainty, the conclusion reached was that function inlining did not lower the power regardless of the architecture, no technique was shown to have an impact on the power consumption of Intel 8051, loop unrolling was shown to have an impact on the Motorola HC12 and the ARM7TDMI, and type replacement had an impact only on the Motorola HC12. The main limitation of this work is that few optimizations techniques were tested. In addition, although some of the techniques were shown to be the cause of lower power consumption, it was not established if the reduction was actually significant.

There are few tools that analyze embedded code at the algorithmic level to produce an optimized program with respect to power and performance. A few approaches have been found where a developer performs manual optimizations. The inconveniences of this approach are that the developer needs a to have detailed knowledge of the code, that it can take a significant amount of time to perform, and that it is error prone. The work of Peymandoust et al. provided a tool called SymSoft that took a high-level source code and optimized sections of the code to produce a program that consumed less energy and took less number of cycles to execute, while requiring little interventions from the developer [14]. The tool also allowed other optimizations to be used in conjunction.

The first stage of the methodology was to convert floating-point types into integer types according to annotations introduced by the developer. Integer operations may require less energy because some systems, for example, do not have a floating-point unit and instead emulate such operations. The program was then executed in a simulator for the target

architecture to determine which areas of the code should be optimized. These sections had to be composed of either bit manipulations, Boolean algebra, and linear or non-linear functions. The selected sections of code were then transformed into a polynomial formulations that could be manipulated. The resulting polynomials were evaluated against accuracy constraints and manually reformulated if the constraints were not met. Finally the polynomial representations were passed to a symbolic computer algebra software package to be optimized.

The SimSoft tool was tested with the SmartBadge embedded system and an ISO MP3 decoder. The results showed similar improvements to those achieved with manual optimizations. Energy improvements between 9% and 77% were observed. As for the performance, improvements of 20% to 62% were observed, a 7.3 improvement factor. The main limitation of this work was that although one of the motivations was to avoid manual optimizations and having knowledge of the code internals, the user is still required to annotate which floating-point types could be converted to integers and to sometimes reformulate the polynomial representations. Another limitation was that a profiler was needed to identify sections of code to be optimized.

One aspect of vectorization for which few research has been found on the literature is targeting vectorization for energy reduction. This is a topic worth of research because vectorization can have a significant impact on the three most energy consuming components in an SoC that were explained in Section 2.1. The work of Lorenz et al. is one of the few to study the impact of vectorization on energy consumption, particularly on DSP and typical signal processing computational kernels [5]. The purpose of studying the impact of vectorization on energy was to develop a compiler heuristic and associated energy estimator to optimize high-level programs. The results of this work show that on average, vectorization was able to reduce by 76% the number of memory operations. This translates to an average reduction of 76% in execution time and 72% in energy consumption. The energy reduction was achieved in spite of the fact that a SIMD instruction consumes four

to five times more power than the equivalent sequential operations. This can be attributed to a dramatic decrease of memory accesses when using SIMD instructions more than any reduction achieved on the computations. An interesting finding was that when applying vectorization the average power consumption increased by 32.7% because the functional units involved in the execution of SIMD instructions consume more power that those of regular instructions. However, the energy consumption was less because the execution time was greatly reduced in comparison to the power increase.

## 3.2 Power and energy estimation

The estimation of software energy consumption can be divided into two categories: dynamic and static methods. These classifications refer to the way in which the program under test is analyzed during estimation. In a dynamic method the program is executed to analyze its run-time behavior for a given input set. Execution can be performed in the actual hardware under test or through the use of a simulator for the target architecture. In the static method the source code for the program is analyzed and not executed. Heuristics can be used to predict the program's execution behavior and features of the source code can be extracted and used as estimation parameters.

The software energy estimation methods can also be classified as white-box or black-box. This classification refers to the level of detail used when modeling the microprocessor and it will be used to present the previous work on estimation.

### 3.2.1 White-box model

The term *white-box* is used because these models take into consideration the internal structure of the target processor and the operations that are carried out if the program of interest were to be executed. This process usually involves simulation, which can be performed at the transistor, gate, or functional unit (FU) levels. Simulation at a lower level can produce more accurate estimates but requires more time and is more hardware dependent than simulations at a higher level, and vice-versa. A common method used

for simulation is the use of cycle-accurate instruction set simulators (ISS). There are also studies that investigate the use of structural simulators, or simulators that use FU as the basic blocks, due to the smaller simulation time. The problem with these methods is that if the simulation time is too high then it would not be of practical use when many successive estimations are required. In addition, the hardware specifications needed for a white-box approach may not be readily available.

Wu et al. explored how to lower a target processor's temperature by controlling the power consumption at runtime [15]. They decided to use FU estimation because simulators were not accurate nor efficient enough to be used for this purpose. The problem with estimating the energy consumption of FU is that it requires significant manual effort because a program's behavior usually changes as it executes. These behavior intervals are called phases and each phase utilizes the FU set in a different way which is dependent on the input data. The purpose of this work was to automate the process of obtaining an energy estimate for each FU of a given processor architecture. The processor power model is based on Equation 3.1,

$$P = \sum_{i=1}^{nf} AF_i \times P_i + P_{idle}, \tag{3.1}$$

where $P$ is the average power consumption of the processor, $i$ represents an FU, $nf$ is the total number of FU, $AF_i$ is the activity factor of an FU, $P_i$ the active per access power for a FU, and $P_{idle}$ is the average power when the processor is idle.

A set of microbenchmarks was created to obtain initial estimate values. The requirements for these programs were to have as few phases as possible and that each one utilized a small and mutually exclusive set of FU. The objective was to assign to each FU a $P_i$ value composed of the dynamic, leakage, and clock powers. The other parameters could be obtained after running the microbenchmarks. $AF_i$ indicates the frequency with which a FU is used and were obtained from the performance counters of each FU. The $P$ and $P_{idle}$ terms are the average power values calculated from the measured current at the processor's power

line multiplied by the supply voltage. The $P_i$ values were obtained by using a least square linear regression on Equation 3.1. The initial estimates were then refined by using the SPEC2K benchmark which had multiple phases. The estimate resulted in a 4.5% average deviation when compared with real measurements. The main limitations of this approach were that the target processor need to have performance counters for each FU and that the microbenchmarks had to be rebuilt for different architectures.

The cache is the component that consumes most of the energy in the processor, after the core. However, there have been few studies that investigate the software energy estimation of cache behavior. The purpose of a work by Chandra and Souray was to provide an operation-level energy model for L1 and L2 caches [16]. Operation-level means that the models relate the independent operations that occur inside a cache to energy values. Operation examples include read or write hits or misses. With such a model, the cache energy cost could be estimated for a given software and processor.

Their cache energy model estimation was compared against the estimate of PowerTheater for the ARM 1136 architecture. There was an average error of 1.7% and the maximum error was less than 4%. The main limitations of this work were that the cache operations and energy values could change between different cache architectures. Furthermore, the specifications for the cache operations could not be available, as was the case of this study, and obtaining this information would require additional effort.

As was previously mentioned, ISS are accurate but slow. Native execution, or execution in the host system, is fast but the collected statistics do not represent the target system. The solution proposed by Muttreja et al. reduced the simulation time by allowing some of the functions to run natively [17]. To decide which functions would run natively, a profiler was used to generate a call graph and simulation time statistics. Based on these statistics, a native function selector chose functions with the objective of minimizing the simulation time under an estimation error constraint.

The results compared hybrid simulations against SimIt-ARM simulations enhanced with the instruction-level energy models from Jouletrack. An average speedup of 70% was achieved. The main limitation of this work was that some information, such as complete resolution of pointer targets, could not be determined at compile-time and this limited the number of functions that could be considered for native execution.

### 3.2.2 Black-box model

A *black-box* approach to estimation takes into consideration the inputs and response of the processor and not the internal details as in the white-box method. This results in a much faster estimation that can be used with a larger number of devices regardless of the internal processor information available. A black-box approach has the disadvantage of a lower estimation accuracy because it involves a higher level of abstraction. However, if the accuracy loss is acceptable, this is the method of choice for estimation and optimization.

This approach usually consists of using an ammeter to measure the current flowing from the power supply to the processor and then multiplying the average current by the source voltage to obtain an average power value for the program that is executing. The process may also involve observing the signals at the input and output ports of the processor.

Tiwari et al. developed the first instruction-level power estimation methodology and model [18]. In general, the methodology consisted of estimating the energy cost for the processor's instruction set, dividing the target code into blocks, assigning an energy value to each block, profiling the code to determine the number of times each block is executed, and calculating the program's energy cost.

To characterize the instruction set, programs in which a single instruction was replicated a number of times were executed inside a loop. The number of replicates had to be large enough to make the contribution of the loop-related instructions negligible, but small enough to not cause cache misses. While each program executed, the processor source current was measured and averaged. An instruction energy base cost table was created by multiplying the average current values of an instruction by the system voltage and then by the particular

instruction's cycle count. The energy cost also varies according to the instructions currently executing in the processor. This is called the processor overhead, and was taken into account by creating a second energy base table with the difference that all pairs of instructions were executed instead of executing the instructions independently.

After characterizing the instruction set, the source code of the program of interest was partitioned into basic blocks. A basic block is a sequence of instructions with single entry and exit points. The base cost for each block was calculated by adding the base cost from the tables for individual and pair of instructions. There were other penalties that were added to the base cost of each basic block, such as effects caused by input data, memory operands, pipeline and write buffer stalls, and cache misses. A traversal of the code was used to determine the number of stall cycles per basic block. A stall cost was determined experimentally for the target architecture and was multiplied by the number of stall cycles of each block and added to its base cost. Experiments were also conducted to determine the average energy cost and average number of cycles per cache miss. These values were multiplied by the cache miss rate of the program as estimated by a cache simulator and were added to the basic blocks' base costs.

The last stage consisted of using a profiler to run the target program and to obtain the number of times each basic block was executed. The energy cost of the basic blocks was multiplied by the number of times each block was run and the values were added to produce the program's energy cost.

The results of the estimation when compared against actual measurements were around 3% of error. The main limitation of this methodology was the strong hardware dependence because of the penalties that have to be added to the basic blocks. In addition, a profiler is needed to determine the program's behavior. Profiling is a slow process and the number of times each basic block executes can vary with the input data.

Modern processors have complex pipelines and prefetch mechanisms. For energy estimation methodologies where current measurement is involved, there needs to be a precise

way of relating the measurement with the instructions being executed. Otherwise, the estimation model would require the addition of adjustment coefficients that take into account the effects due to instruction ordering, pipeline stalls, and cache misses. The objective of the work by Kang et al. was to alleviate this problem by observing what occurred during an inter-prefetch interval (IPI) [19]. An IPI consists of all the operations that occur during the cycles between prefetch operations. With this method the power dissipation could be estimated for each IPI of a program and used to log power over time.

The target architecture chosen for this work was the ARM 1136JF-S. Estimation results were compared against real measurements performed on this device which showed over 96% average accuracy and less than 11% of maximum error. The main limitation of this work is that the program had to be executed in an instrumented processor or possibly in a cycle-accurate simulator to observe the IPI behavior. This method also required access to the processor's prefetch and load-store units.

The work by Tan et al. provided a static analysis method for fast and accurate energy estimation through the use of macromodels [20]. A macromodel is a linear equation used to calculate an energy estimate based on high-level parameters of the target function. The macromodel function is shown in Equation 3.2,

$$\hat{E} = \sum_{j=1}^{p} c_j P_j \tag{3.2}$$

where $\hat{E}$ is the energy estimate, $p$ is the total number of parameters, $P_j$ is the $j$-th macromodel parameter, and $c_j$ is the $j$-th parameter's coefficient.

The first step in creating a macromodel was to select a set of parameters based on an analysis of the high-level source of the target function. The target function was then executed with typical input data vectors in a processor simulator to obtain an energy consumption estimate. The energy estimate given by the simulator was taken as the real energy consumption value, denoted by $E$. Coefficients $c$ were determined by evaluating the equation with $E$ and $P$. Energy estimate $\hat{E}$ was obtained by evaluating the equation for a given

set of $P$. The accuracy of the model was established by comparing the actual (simulated) $E$ and $\hat{E}$.

The results of this work showed an estimation accuracy of 95% and speedups of one to five orders of magnitude when compared with simulation. A limitation of using macromodels was that the accuracy is affected by the input data vectors and therefore these vectors must closely resemble actual data. Another disadvantage is that the target function must be executed in a simulator to be able to compute the coefficients. If the function were to be modified, as would be the case during optimization, then the macromodel could be invalidated.

Finding a subset of parameters that produces an accurate macromodel is a manual process that takes a significant amount of time. For this reason Muttreja et al. proposed an automated system that outputs an accurate macromodel by taking as input the target function and a testbench that passes test data vectors to this function [21].

The results of the system were compared against the SimIt-ARM instruction set simulator (ISS). The estimation error was 0.7% and the estimation time was less than a minute when using the macromodel, compared to more than a day for the ISS. The limitations of this work are the same as those for the approach of Tan et al. [20]. Furthermore, the user may need to add compiler directives to the source code in order to significantly increase the accuracy of the macromodel. In this case the user must have a detailed knowledge of the target program's source code.

The methodology proposed by Brandolese used dynamic estimation to provide the software developer with source code level feedback of energy consumption, execution time, and space costs [22]. An advantage of this method was that estimation did not require the use of an instruction-level simulator nor an instruction-profile model. Instead, the methodology associated kernel instructions, or a set of abstract assembly instructions, to each node of the program's parse tree. The source code was then instrumented, compiled, and executed to obtain execution counts for each node. Finally, the instrumented code

was executed again and the generated profiling information was automatically analyzed to obtain the energy, time, and space estimates.

The authors reported that the instrumented program ran around two times slower than the original code but the method achieved a speedup of 11,000 and an average error of 8.41% for execution time and 8.50% in energy estimation when compared to the estimation performed by an energy-accurate version of the ARMulator 2.10 ISS.

The work by Acevedo et al. addressed the problems posed by dynamic estimation; the requirement of specialized tools such as simulators and profilers, the long estimation times due to code execution, and the data dependence [23]. The proposed solution combined a processor power model and static code analysis in what the authors called static simulation. In this kind of simulation the code was analyzed to estimate its run-time behavior by using techniques such as branching probabilities. This solution was faster than dynamic methods because the code was not actually executed and it also minimized the use of specialized tools and real data. A disadvantage of this technique was that the predictions used during static code analysis resulted in a higher estimation error when compared to dynamic methods. In addition, loop boundaries are sometimes input-dependent and the user had to provide these data. This meant the user needed to have knowledge of the code internals.

An experiment was setup where the alternative hypothesis was that the methodology would estimate power and energy consumption with an error less than 20% when compared against measurements. The hypothesis was tested using a t-test with a confidence level of 95%. The mean error was found to be 14.6% for energy and 17% for power. The upper bounds were 19.2% for energy and 11.7% for power.

### 3.3 Machine learning driven compilation

The literature already contains several works on ML-driven compilation that apply *Support Vector Machine* (SVM) [24, 25], *Nearest Neighbor* (NN) [24, 26], *Artificial Neural Networks* (ANN) [27], and *Logistic Regression* (LR) [28]. They mainly differ in the chosen feature vectors and the types of predictions.

The work from Stephenson and Amarasinghe used classification to determine for each program the unroll factor that yielded the best performance [24]. Their workloads consisted of 2500 loops extracted from several well-known benchmarks targeted at high-performance computing as well as embedded computing. They leveraged both SVM and NN, and managed to correctly predict the best unroll factor with an accuracy of 65% and 62%, respectively. This proved to be far better than their baseline, the Open Research Compiler[a] .

Park et al. proposed using graph mining techniques to directly feed the program's dataflow graph to an SVM as a feature vector and to use speedup regression to predict the best optimization scenario [25]. In contrast, Cavazos et al. used reactions as feature vector [29]. A reaction is the resulting speedup of a program after applying some optimization with respect to the performance of the untransformed program. A limitation of both regression approaches was that they were not scalable in terms of prediction time. Both approaches relied on predicting the speedup for all optimization scenarios in order output the scenario expected to perform best. But this was a combinatorial problem and the optimization scenario space, and thus the prediction time, increased exponentially with the number of optimization techniques.

Reaction methods have the advantage of not depending on the definition and extraction of software characteristics as feature vectors. However, reactions require executing the program at least once during compilation. This does not integrate well with modern compilers which are not iterative. In addition, the behavior of a program will change according to the input data. In the general case, we cannot assume that the input data is available at compile time.

Fursin et al. proposed a collaborative compilation approach to mitigate these issues [10] [30]. This compilation flow relied on the compilation statistics from many users to perform continuous predictor tuning. This meant that the prediction accuracy was increased as

[a] Now called Open64. Online: http://www.open64.net/

new reactions were collected. This method also allowed different data sets to be taken into account, and prediction did not have to be done all at once for a new program as was the case of the regression methods mentioned before. However, new programs had no previous reactions and in this case a default optimization scenario was used. This default scenario was based on the scenario that best performed on average based on the current programs in the database. This is a limitation, since one of the motivations of ML-driven compilation was that no one scenario performs well in the general case.

Kulkarni and Cavazos used ML to tackle the complex problem of the ordering of optimization techniques [27]. This is an important issue, because the order in which optimizations are applied has an impact in performance. They identified optimization scenarios in the training examples using a Markov process. Then they predicted optimization scenarios iteratively, one optimization technique at a time, using Artificial Neural Networks at each step. They applied their method to the just-in-time compiler of a Java virtual machine, and were able to reduce the execution time of compiled programs by up to 20%.

Our work, as well as all the previously detailed ones, aims at improving the quality of the compiler output by improving the performance of the compiled programs. On the other hand, the works from Agakov et al. and Pekhimenko and Brown utilized ML in order to reduce the compilation time, that is, the execution time of the compiler itself [26] [28]. The objective of Agakov et al. was to reduce the time required to find the best optimization sequence to apply to a given program. They adopted a technique based on NN to bias a random and a genetic search algorithm, and managed to reduce the search time by one order of magnitude. On the other hand, Pekhimenko et al. [28] used LR to determine the parameters of optimization techniques inside a fixed optimization scenario, with the aim of leveraging the fast execution time of LR compared to the heuristics implemented into a commercial vendor compiler [28]. They managed to reduce the compilation time by two orders of magnitude while at the same time slightly improving program speedups.

To the best of our knowledge, our work is the first to study ML driven compilation with the objective of energy reduction. It is complementary to the related works because it aims at optimizing energy rather than execution time. Furthermore, most of the related works predicted optimization scenarios with varying results, but we found that predicting a single optimization is challenging by itself. A wiser approach may be to develop methodologies for predicting individual optimizations with high precision before scaling to optimization scenarios. This work is work is a step towards that goal that focuses on vectorization.

# Chapter 4
# PROBLEM STATEMENT AND OBJECTIVES

## 4.1 Problem statement and hypotheses

The main question addressed in this project was: *Can vectorization be targeted at reducing software energy consumption in an embedded system?*

Several subproblems were derived from this question:

- Which estimation and optimization techniques should be used?
- At which stage of a program's lifetime should the optimization be performed? The alternatives are compile time, installation time, when the program is being executed or at idle times when it is not running.
- In the case of compile time optimization, at what code level should estimation and optimization be performed? The alternatives are high, intermediate, and low level.
- How do we assess the effectiveness of the method?
- How can such a system be implemented? The alternatives are as a stand-alone tool or as a module of a tool that already exists.
- What degree of architectural portability can be achieved?

A methodology was followed to answer these questions, and the hypothesis was formulated as:

- Vectorization can be have a beneficial or detrimental impact on energy consumption in embedded systems.
- Machine learning classification is an effective way of predicting from high-level source code when to apply vectorization with the objective of reducing energy consumption.

## 4.2 Objectives

The main objective of this research was to develop a software energy reduction methodology focused on, but not limited to, a particular embedded processor architecture.

The following were specific objectives of the project:

- To identify an optimization technique that can transform an input program into a functionally equivalent version that has a lower energy consumption.
- To develop a prediction component that can predict the energy profitability of the optimization technique for an input program. This component should be retargetable to different compilers and platforms.
- To perform a statistical analysis to determine the effectiveness of the proposed optimization technique and the predictor.

# Chapter 5
# METHODOLOGY

A cognitive map of the research methodology conducted in this project is shown in Figure 5.1 . Each box is a task that was completed to reach the final objective. The project objectives are shown in the boxes with a thicker outline and darker background. The diamond shaped boxes are subtasks that were completed as part of the main task they are under. In the following sections, each task is explained in detail.

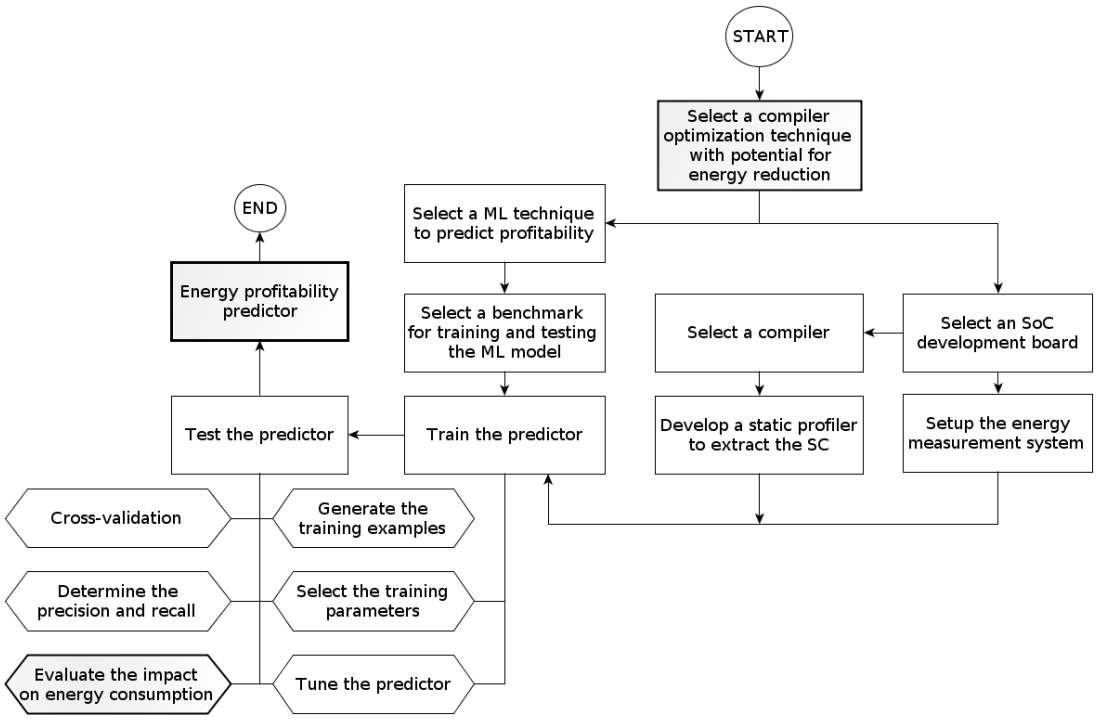



Figure 5.1 : Methodology for reducing the software energy consumption.

## 5.1 Selecting the architecture and optimization technique

The ARM architecture was the obvious choice for this project, as it is the most widely used architecture in mobile devices (Section 2.1). Optimizing software to reduce the energy consumption began by identifying the ARM components that consume the most power (Figure 2.2 and Table 2.1 ). Next, we identified vectorization as a potential optimization because it can dramatically reduce the execution time and allows more efficient memory access patterns through the use of specialized SIMD hardware (Section 2.2).

## 5.2 Development board and energy measurement system setup

The ARM Cortex-A8 is the most common implementation found in mobile devices (Section 2.1). In addition, it includes a NEON co-processor for executing SIMD instructions which has a vectorization factor of 4. The Beagleboard is one of the few development boards that includes a Cortex-A8 core within its SoC. Specifically, we selected the Beagleboard-xM single-board computer which has an DM3730 SoC [a] . The ARM core within this SoC is clocked at 1GHz and the board has 512 MB of DDR RAM. An embedded version of Ubuntu Linux 12.10 was installed to facilitate experimentation.

The development board was instrumented as shown in Figure 5.2 . The system was designed to automatically compile, execute, measure the execution time and the average power values for a vectorized and nonvectorized version of each benchmark program (Section 5.4). Operating system installation was done from the workstation through the UART connection. Subsequent interactions with the board were done through SSH over the Ethernet connection. Interaction with the DS1052E oscilloscope was performed solely from the Beagleboard, and the workstation served only to setup the board and to retrieve all the collected data at the end of the experiment.

[a] http://beagleboard.org/beagleboard-xm

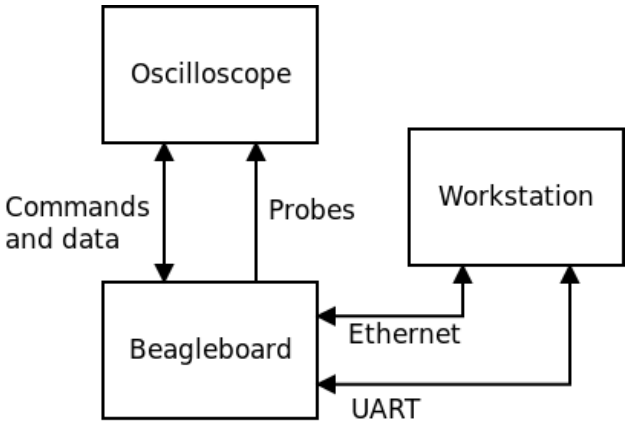



Figure 5.2 : Energy measurement setup.

Instrumentation code was automatically inserted into the programs to count the execution time and to trigger the oscilloscope to record the average power consumption. The programs were compiled within the board using the GCC compiler (Section 5.3). Two probes from the oscilloscope were connected to the J2 jumper on the board to measure the voltage drop over a 0.1Ω resistor. For each execution, 600 voltage samples were taken at the maximum oscilloscope frequency of 50MHz and transfered from the oscilloscope to the Beagleboard for voltage-to-power conversion. Data transfer and post-processing was performed in the Beagleboard rather than the workstation for convenience, but this did not have any impact in the experiment because this was carried after experimentation. This process was repeated 3 times and the values were averaged to obtain an average power consumption and average execution time. With the execution time and average power, the energy consumption was computed for the vectorized and unvectorized versions of each program.

### 5.3 Compiler and static profiler

We selected the GCC compiler because it supports the ARM architecture and NEON instructions, is open source, stable and widely used within production environments. The feature vectors for our benchmark were extracted from the high-level C source codes (Section

2.3, Section 5.4). This required the creation of a static profiler to extract the software features. The LLVM compiler was used because it includes libraries for this purpose.

Table 5.1 lists the compiler options used in GCC version 4.7.2 to activate and deactivate vectorization. In both modes, the optimization level 2 (O2) was used. This allowed the compiler to perform other kinds of transformations other than vectorization. This is analogous to a real-use case, where the ML component assists the compiler in optimizing the input program (Section 2.3). However, we deactivated loop unrolling with the *fno-unroll-loops* compiler option because we were already unrolling the benchmarks manually (Section 5.4).

| Mode | Compiler command line options |
|---|---|
| Vectorize | tree-vectorize tree-slp-vectorize lax-vector-conversions fivopts |
| No vect. | fno-tree-vectorize no-tree-slp-vectorize |
| Both | O2 std=c99 mfpu=neon funsafe-math-operations fno-unroll-loops no-modulo-sched |

Table 5.1 : GCC command line options for vectorization.

### 5.4 Benchmark and software characteristics

A ML predictor had to be trained with representative examples of future inputs. In this work we focused on vectorization, therefore we needed a benchmark that would be representative of vectorization cases. The Test Suite for Vectorizing Compilers (TSVC) was a perfect match because it covered many vectorization scenarios [31]. It is composed of 151 simple loops that are likely to be encountered in real applications. Because the predictor performance tends to improve with the number of training examples, we artificially increased the number of examples by unrolling the original 151 loops by factors from 1 to 20 for a total of 3020 benchmarks. Listing 5.1 shows an example of a TSVC loop called *s431*. In practice, each loop is surrounded by an outermost loop to control the number of times it will execute. This is needed to be able to measure a long enough execution time and enough power samples such that the execution environment noise is reduced.

```
int k1 = 1; int k2 = 2; int k = 2*k1-k2; int nl;
for(int i = 0; i <= 31999; i += 1)
    a[i] = a[i+k]+b[i];
```

Listing 5.1: Sample TSVC loop.

A ML feature vector requires a numerical encoding of all inputs. The characterization of the benchmarks consisted of a series of hardware independent, static features tailored to predict vectorization profitability termed Software Characteristics (SC) [32]. These were profiled at the abstract syntax tree and intermediate representation levels of the code using the LLVM compiler framework. Table 5.2 lists each of thse features and the compilation level at which they were extracted; Abstract Syntax Tree (AST) or Intermediate Representation (IR).

| Identifier | Level | Range | Description |
|---|---|---|---|
| AST1 | AST | $\mathbb{N}$ | The increment of the innermost `for` loop |
| AST2 | AST | $\{0, 1\}$ | Will the address of the first access to the array always be aligned with the machine's vectorization factor? |
| AST3 | AST | $\{0, 1\}$ | In array accesses, are the induction variables involved in the last dimension the one of the innermost loop? |
| AST4 | AST | $\mathbb{N}$ | Number of array accesses in the body of the loop |
| AST5 | AST | $\mathbb{N}$ | The number of arrays accessed inside the loop |
| AST6 | AST | $\{0, 1\}$ | Does the benchmark involve any restrict keyword? |
| IR1 | IR | $\mathbb{N}$ | The size of the dataset |
| IR2 | IR | $\mathbb{N}$ | Estimation of the dynamic instruction count |
| IR3 | IR | $\mathbb{N}$ | The depth of the innermost loop |
| IR4 | IR | $\mathbb{N}$ | The estimated trip-count of the innermost loop |
| IR5 | IR | $\mathbb{N}$ | Number of IR statements in the innermost loop |
| IR6 | IR | $\mathbb{N}$ | The number of SSA variables used in the innermost loop |

Table 5.2 : The software characteristics used in the feature vector for the predictor.

## 5.5 Energy profitability predictor

Vectorization profitability was modeled as a binary classification problem (Section 2.3). In other words, we wanted for the predictor to answer whether vectorizing the input program

would be beneficial or detrimental. Figure 5.3 shows the contents of the *machine learning* component of Figure 2.5 as implemented in this work. The inputs to the predictor were the Software Characteristics (SC) of the program to be optimized and the output was one if vectorization was predicted to reduce energy consumption, and zero otherwise.

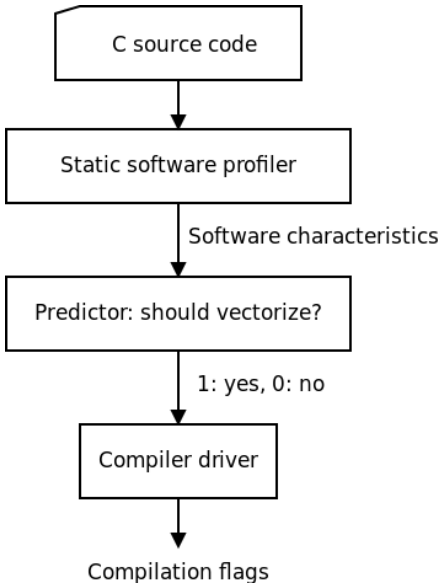



Figure 5.3 : Energy profitability prediction scheme.

The Support Vector Machine (SVM) algorithm described in Section 2.3 was selected for the predictor because it can account for the possibility of nonlinearilty in the relation between the SC and vectorization profitability. We used the *svm* function of the R language package e1071 implementation of SVM with the default training parameters and the Radial Basis Function (RBF) kernel[b] .

To train the predictor, the first step was to generate a training example for each of the benchmark programs. A training example consisted of an SC as feature vector and a target of one if vectorization reduced the energy after execution, or zero if it increased. That is,

[b] http://cran.r-project.org/web/packages/e1071/index.html

during the training phase the predictor was presented with many cases of a characterization of the input program and whether vectorization turned out to be profitable.

After training, we had a dataset in which each record specified the program identifier, execution time, power and energy ratios, the SC and whether vectorization increased the energy or not. The time, power, and energy values were expressed as ratios of the measurement when the program was vectorized divided by the measurement when vectorization was disabled. Thus ratios less than one indicated profitability due to vectorization and ratios greater than one indicated detrimental performance. These data were enough to test the predictor, to assess the impact of vectorization on energy, and to assess how well the predictor could exploit vectorization for reducing energy consumption (Chapter 6).

During the testing phase, 5-fold cross validation was performed over the training examples to obtain a prediction for each example in the set. This fold value was taken as a rule-of-thumb, since typical values are 5 or 10. Now each example had both the target or the real outcome and a predicted outcome. With this information, the quality of the predictor was assessed by computing the precision and recall as defined in Section 2.3.

# Chapter 6
# RESULTS

In this chapter we answer the two main questions of this work. First, is vectorization a suitable optimization technique for reducing energy consumption in embedded systems? And second, is machine learning a suitable technique for predicting vectorization profitability?

In Section 6.1 we answer the first question by examining the collected time, power, and energy measurements after compiling the TSVC benchmark with GCC. We then examine representative programs in order to assess the effect of vectorization on energy consumption. In Section 6.2 we answer the second question by training a predictor and analyzing its effectiveness at reducing energy consumption by making judicious use of vectorization.

It is important to note that both questions are answered using different toolsets. The vectorization impact is analyzed from the point of view of statistics, and accordingly, no data manipulation is performed. The vectorization profitability predictor on the other hand, is trained and analyzed using standard machine learning techniques. As such, the objective is to train a practical predictor that is likely to suggest efficient vectorization while preventing detrimental application of it. Section 2.3 offers more details on the difference of both approaches.

## 6.1 Measurement data analysis

Figure 6.1 shows histograms for the performance distributions of the loop-unrolled TSVC benchmark for the GCC compiler in terms of time, power, and energy ratios (Section 5.4). In this figure, the horizontal axis marks the measured ratios and the vertical axis

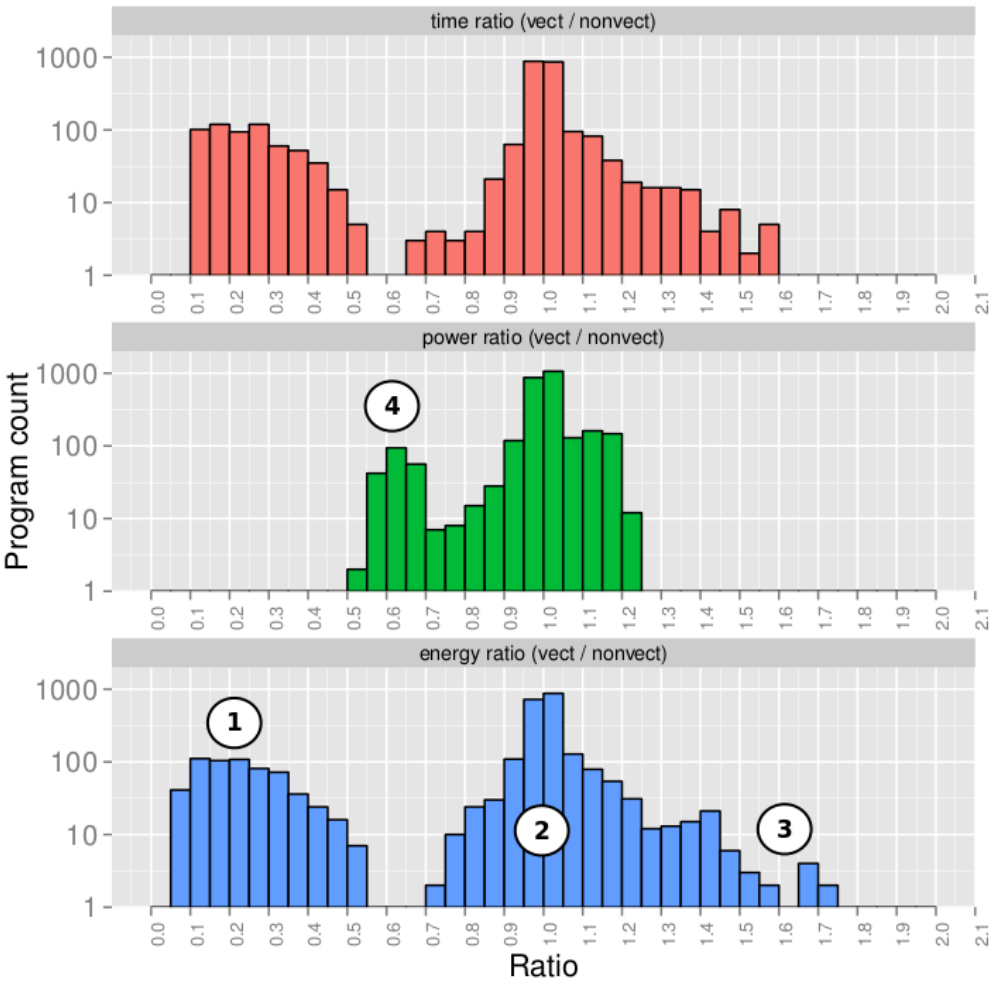



Figure 6.1 : Performance histograms for TSVC compiled with GCC.

shows, in logarithmic scale, the number of programs that fell within a 0.05 ratio interval. Henceforth, performance ratios are expressed as the measurement for a particular metric when vectorization is activated divided by the same metric when vectorization is deactivated. Therefore, ratios below unity were considered to be a beneficial application of vectorization (region marked ①), above unity were detrimental (region ③), and when equal or very close to unity, the compiler was unable to transform the code to vectorize it and the performance was neutral (region ②). Table 6.1 lists the ranges selected for each region. The ranges do not cover the whole performance interval, but rather capture the peak and nearby points that characterize the performance region.

| Region | Id | Range |
|---|---|---|
| Beneficial | 1 | 0.1 to 0.3 |
| Neutral | 2 | 0.9 to 1.1 |
| Detrimental | 3 | 1.4 to 1.8 |
| Power peak | 4 | 0.6 to 0.65 |

Table 6.1 : Performance ratio intervals for the regions annotated in Figure 6.1 .

The first noticeable feature in the histograms is that most measurements are concentrated around unity, or the neutral region ②. In the energy histogram there are 1834 programs in the neutral region from a total of 2753 programs, or 67%. In fact, a logarithmic scale was applied to compress this peak and to show other interesting patterns in the performance profile. The reason is that the GCC compiler is unable to transform the code to use vector instructions for most of our programs, even though every loop in TSVC is vectorizable if the right code transformation were applied. These results confirm the research by Maleki et al. [6] which found that 45% to 75% of TSVC failed to be properly vectorized by the tested GNU GCC, Intel ICC, and IBM XLC compilers. The tendency is shown in both the time and power histograms, which support this conclusion using two independent measurements. Programs that were not vectorized by the compiler were of no interest to this study, which aimed at assessing the potential of vectorization and not evaluating the capacity of the compiler at vectorizing programs.

Nevertheless, it is clear that the dominating factors in energy consumption were the changes in time rather than in power. This is confirmed by the similar behavior shown in both the energy and time histograms. Also, most measurements are located at and below unity, indicating that vectorization tends not no be detrimental to the performance. The programs also seem to cluster around 0.25 for both time and energy ratios. If we invert this time ratio, what we have is 4 times speedup. This value corresponds to the NEON's vectorization factor, that is, vector instruction can load, compute, and store 4 elements at once. Therefore, it makes sense that programs that benefit from vectorization group

around this speedup value. Finally, another noticeable feature is a peak in the power ratio distribution at region ④ of Figure 6.1 which will be explained later on in this section.

Now, let us examine representative programs for each of the regions in the histograms to confirm the observations made above and to quantify the impact of vectorization in performance. As a reference, Tables 6.2 and 6.3 show a description of the ARM assembly instructions used in the sample programs to be discussed.

| Inst. | Description |
|---|---|
| ldr | Load register |
| str | Store register |
| add | Add two scalars |
| fld | Load floating-point register |
| fst | Store floating-point register |
| fmac | Floating-point multiply-and-accumulate |
| fcpys | Copy floating-point register |

Table 6.2 : Selected SISD instructions for the ARM core and VFP co-processor.

| Inst. | Description |
|---|---|
| vldr | Load vector |
| vstr | Store vector |
| vadd | Addition of two vectors |
| vmla | Vectorial multiply-and-accumulate |
| vuzp | De-interleave two vectors |

Table 6.3 : Selected SIMD instructions for the NEON vector co-processor.

To understand the effect of activating automatic vectorization in the compiler, it is essential to study the assembly of a program before and after vectorization for each of the performance regions. Since it is not possible to analyze the thousands of programs in the dataset, samples were taken at random and examined to see if their assembly could explain the performance exhibited in the region they were extracted from. Table 6.4 summarizes the time, power, and energy ratios for the sample programs to be presented next.

As a first approach at analyzing the assembly, it was useful to find programs that fell within all performance regions according to the unroll factor. This made it easier to identify which modifications in the code contributed to their performance, in preparation to analyzing other sample programs. In our benchmarks, only the programs named *s231* and *s235* have loop unrolled instances that fall within regions ①, ②, and ③. That is, depending on how many times the innermost loop of these programs is unrolled, they will respond differently to vectorization in a beneficial, neutral, or detrimental way. The program to be

Table 6.4 : Vectorization performance impact per representative program. The metrics are shown as ratios. UF stands for loop unroll factor.

| Program | UF | Region | Time | Power | Energy |
|---|---|---|---|---|---|
| s235 | 17 | 1 | 0.26 | 0.65 | 0.17 |
| s235 | 20 | 2 | 1.13 | 0.96 | 1.09 |
| s235 | 4 | 3 | 1.82 | 0.94 | 1.72 |
| s1115 | 20 | 3 | 1.44 | 1.10 | 1.58 |
| s235 | 5 | 4 | 0.32 | 0.61 | 0.19 |
| s000 | 2 | 4 | 0.11 | 0.62 | 0.07 |
| s000 | 13 | 4 | 0.28 | 0.60 | 0.17 |
| s000 | 16 | 4 | 0.28 | 0.62 | 0.17 |
| vpvts | 10 | 4 | 0.15 | 0.61 | 0.09 |

studied is named *s235*, and the baseline C source code without loop unrolling is shown in Listing 6.1.

```
for(int i = 0; i <= 255; i += 1) {
    a[i] += b[i]*c[i];
    for(int j = 1; j <= 255; j += 1)
        aa[j][i] = aa[j-1][i]+bb[j][i]*a[i];
}
```

Listing 6.1: C source code for the TSVC loop *s235*.

When the innermost loop for program *s235* was unrolled 17 times, vectorization was dramatically beneficial across all metrics and the performance was within region ①. The innermost loop contained an assembly instruction pattern similar to the Listing 6.2, which was repeated 17 times. The impact on performance was a reduction of 74% in execution time, 35% in average power, and 83% in energy consumption. Vectorization had a big impact because memory access could be done efficiently without the need of packing or unpacking instructions (Section 2.2). In addition, although there were SISD additions, the expensive

multiply-and-accumulate instructions were performed with the SIMD version. This sample also shows that vectorization not only can reduce the execution time, but also the average power consumption.

```
add         r1, lr, r3
ldr         sl, [sp, #132]
vmla.f32    q9, q8, q10
vstmia      r5, {d18-d19}
vldmia      r2, {d20-d21}
ldr         r5, [sp, #140]
add         lr, r6, r3
add         r7, sl, r3
adds        r2, r5, r3
ldr         sl, [sp, #124]
```

Listing 6.2: Assembly pattern for *s235* unrolled 17 times.

However, when *s235* was unrolled 20 times, just 3 times more than in the previous example, the performance was located within the neutral region ②. Listing 6.3 contains a similar pattern than when unrolling 17 times, except that SISD rather than SIMD instructions were generated. The different unroll factor caused GCC's automatic vectorization algorithm to fail half-way in applying the optimization, even though extra overhead instructions were generated outside and within the innermost loop. Performance remained close to the unvectorized baseline, with an increase in execution time of 13%, a reduction in power of only 4% and an increase in energy of 9%.

```
add     r2, r0, #16384
add     r3, r5, #15360
flds    s13, [r2, #0]
flds    s14, [r3, #0]
fmacs   s14, s15, s13
add     r3, r0, #17408
flds    s12, [r3, #0]
add     r3, r0, #18432
flds    s13, [r3, #0]
add     r3, r0, #19456
flds    s17, [r3, #0]
add     r3, r0, #21504
flds    s0, [r3, #0]
```

Listing 6.3: Assembly pattern for *s235* unrolled 20 times.

In the third case, program *s235* was unrolled 4 times and as a result the performance was located within the detrimental region ③. The intuition was that this was due to unprofitable vectorization but once more, GCC was unable to vectorize the code. An assembly pattern was generated similar to the last case when the program was unrolled 20 times. However, this time the side effect of a failed vectorization was that the performance dropped significantly to 82% higher execution time, 5% less average power, and 72% increase in energy consumption. Inspecting the assembly did not reveal any distinguishing feature between the unvectorized baseline assembly; the instruction mixes and control flow graphs were similar. Determining how loop unrolling and vectorization interact in the compilation flow is beyond the scope of this work, but still this information is useful because it shows that not all performance loss in our experimental dataset was due to vectorization but to other modifications introduced as part of the compilation process.

Still, it was important to find cases in which vectorization was detrimental. Otherwise, if the instruction overheads produced by GCC on a failed vectorization were to be solved, then the appropriate vectorization strategy to reduce energy consumption would be to always vectorize because the performance was either beneficial or neutral. But plenty of cases were found in which vectorization was the cause of poor performance, and here we examine program *s1115*. When this program was unrolled 20 times and vectorized, its performance was located in the detrimental region ③ with an increase of 44% in execution time, 10% in average power, and 60% in energy consumption. Listing 6.4 shows the instruction pattern that was found in the innermost loop. The program had low performance because SIMD instructions were used to load from memory while addition was done sequentially. Specifically, the *vuzp* instruction was used to de-interleave, or swap the upper two elements of the first vector register with the lower two elements of the second vector register. Although the purpose of this instruction was to load structured objects from memory, this instruction was being used here as an unpacking operation (Section 2.2). One of the swapped registers was then used for memory addressing with the *vldr* instruction, and the retrieved value was added with a SISD instruction. In other words, SIMD instructions were purposelessly used load data for a SISD operation.

```
vuzp.32   q8, q2
vuzp.32   q6, q4
vld1.32   {q2}, [r3]
add       r3, fp, #336
vld1.31   {q4}, [r6]
add       r6, fp, #352
```

Listing 6.4: Assembly pattern for *s1115* unrolled 20 times.

The next cases studied explain the second peak that can be observed at region ④ of the power ratio in Figure 6.1 . This region is interesting because in the execution time

and energy ratios there is no other noticeable peak other than in the neutral region ②. In region ④ there are 94 training example programs, 92 of which also belong to the beneficial region ① or perform even better. There were programs with all unroll factors, therefore this did not seem to be a factor of much influence in their good performance. Five programs were selected from this region for further study. The *s235* program with unroll factor 5 was selected for analysis because this program had already been studied for other unroll factors. The other four programs were selected at random and all the performance results were also included in Table 6.4 .

Program *s235* with unroll factor 5 was successfully vectorized and had a similar instruction mix and performance as when unrolled 17 times. Listing 6.5 shows the C source for the baseline *s000* program without unrolling.

```
for(int i = 0; i <= 31999; i += 1)
    X[i] = Y[i]+1;
```

Listing 6.5: C source code for the TSVC loop *s000*.

When *s000* was unrolled 2 and 13 times, the generated assembly code had about half SIMD and half SISD instructions. There were however no packing instructions present and this should account for the good performance. For an unroll factor 16, however, the generated assembly resembled that of s235 with unrolling factor of 17 where there was almost complete vectorization. The difference was that the *vadd* instruction was used in place of *vmla*, because the innermost statement is adding rather than multiplying.

The last case studied was for the program *vpvts* unrolled 10 times. Listing 6.6 shows the C source for the baseline *vpvts* program without loop unrolling. The instruction mix of the vectorized assembly was similar to program *s235* unrolled 17 times, except there was a much greater number of *vldr* and *vstr* instructions. Still, the performance improvement was even better than for *s235* because most of the *ldr* and *str* in the unvectorized version of the program were replaced by their SIMD counterpart. This highlights the importance of

efficient memory access through SIMD instructions. Performing vectorial memory accesses can significantly improve or hurt performance, beyond the impact of the computational parallelization aspect of SIMD instructions.

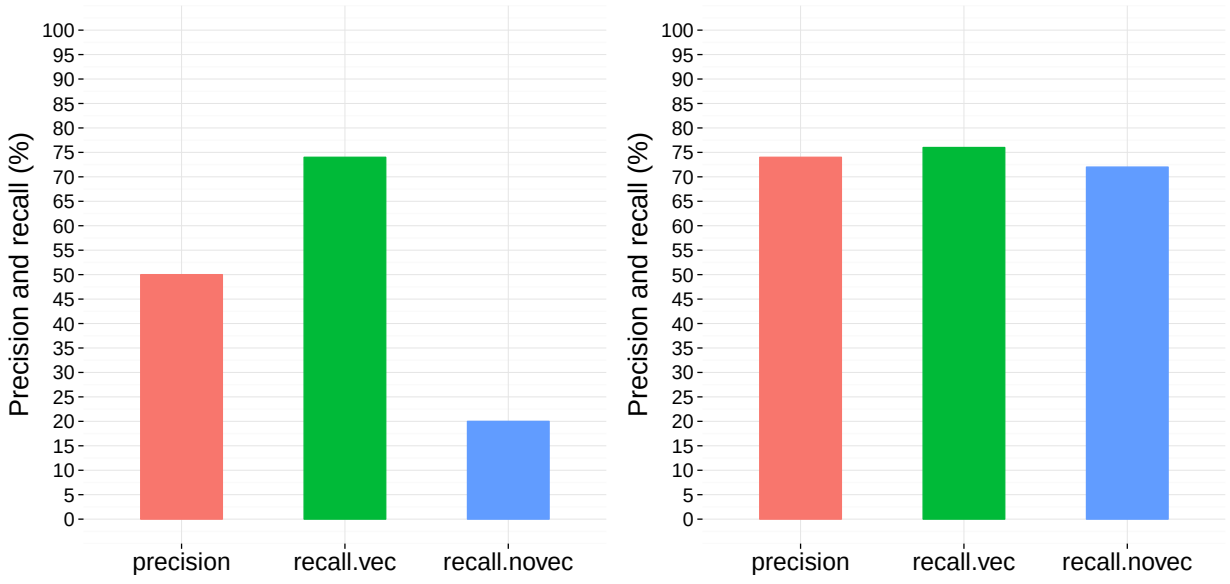
```
for(i = 0; i <= 31999; i += 1)
    a[i] += b[i]*s;
```

Listing 6.6: C source code for the TSVC loop *vpvts*.

In summary, vectorization is a double-edged optimization technique that can have a significant impact on the execution time, power and energy consumption. If performed correctly, it can be a powerful technique for energy reduction. Several cases were presented in which energy was reduced by as much as 90%. At the same time, another example was presented in which vectorization caused around 50% greater energy consumption. Improvements in performance were achieved through not only computational parallelization, but also more efficient memory accesses. This was the case even if a significant number of SISD instructions were not able to be vectorized. On the other hand, detrimental performance was found to be caused by packing instructions and artifacts introduced by the compilation process. This experiment also revealed the limitations of modern compilers in transforming the code for successful vectorization, which confirmed the results of Maleki et al. on TSVC [6].

## 6.2 Vectorization profitability predictor analysis

An SVM machine learning predictor was trained with the extracted software characteristics and the measured energy ratios, and then tested using cross-validation as explained in Section 2.3. Cross-validation required less than 10 seconds to perform for all programs. The predictor performance was evaluated according to its precision and recall (Section 2.3). Precision is the ratio of correct predictions over all predictions. Recall is similar to precision but computed for each class to be predicted. It is defined as the ratio of correct predictions




(a) Untuned predictor.

(b) Tuned predictor.

Figure 6.2 : Predictor's precision and recall.

for that class over all predictions for that class. The objective of these metrics was to estimate how good our predictor was at deciding whether vectorization would be beneficial or not when applied to a given input program.

Figure 6.2 a shows the precision and recall ratios for our first attempt at predicting vectorization profitability. Unfortunately the precision was at 50%, in other words, predictions were being performed at random. Furthermore, the recall of the *should vectorize* class on the second bar was several times larger than the *should not vectorize* recall on the third bar. Our predictor was biased, meaning that it would be much better at predicting one class than the other. A balanced predictor was desired since the impact of mispredicting vectorization can also be high.

In order to improve prediction performance, first we re-examined Figure 6.1 which shows that most points lie around the neutral region ②. These points mostly correspond to programs for which the compiler was unable to apply vectorization. We decided to tune the predictor by methodically removing points within the neutral region, starting at the center of the region and moving outwards until acceptable precision was achieved (Table

6.1 ). When implementing the proposed methodology in a compilation flow, an automated alternative would be to parse the compiler optimization reports to detect those loops in which vectorization failed in order discard them before the training phase.

There are several reasons that justify the predictor tuning. First by intuition, it would be unreasonable to ask a predictor to accurately classify those points that are on the fringe of being labeled as beneficial or detrimental. Furthermore, mis-predicting these points would have the same weight as mis-predicting points that do have a significant impact on energy consumption, thus unnecessarily reducing the predictor's precision. The second reason is that those points in the neutral region are mainly composed of programs which the compiler was unable to vectorize. As previously mentioned, the objective of this work is not to study or predict the compiler's ability at transforming the code to achieve vectorization, but rather on whether vectorization should be applied or not according to the impact on energy.

The last two reasons that justify tuning are related to machine learning techniques. During the training phase, it is standard practice to manipulate the training data in order to improve prediction accuracy. This is true as long as training does not include points to be used in the testing phase, otherwise known as self-validation. But this situation is avoided in this work by using cross-validation. As an example of the data manipulations used, Andrew Ng suggests in his machine learning course to artificially generate training examples by adding noise to already existing training examples to improve hand-writing recognition [a] . Thus, removing the neutral points from the training phase is an acceptable practice in machine learning. The last reason is whether removing these points makes sense during the predictor's testing phase and during precision and recall evaluation. In this research we are looking for a predictor that is accurate at predicting the cases with significant impact. However, the points within the neutral region have an average energy ratio of 1.002. On average, the benefit of having a predictor that is accurate at predicting

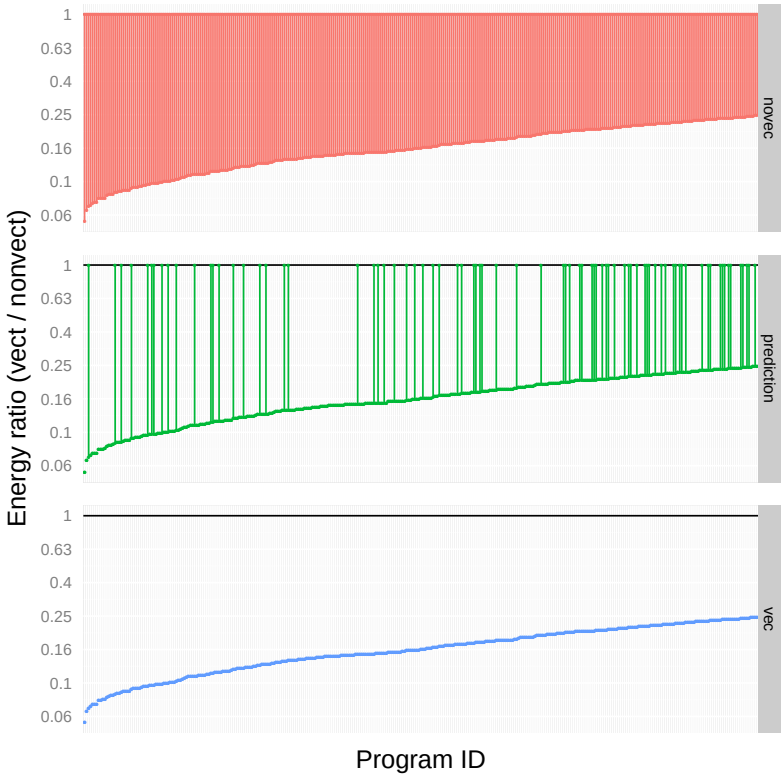
[a] http://class.coursera.org/ml-005/lecture

the neutral cases is negligible. Nevertheless, we found that there is a trade-off between the number of points within the neutral region and the overall predictor precision. That is, minimizing the number of points in this region resulted in higher precision and recall scores for the points that do have significant impact on energy. In this way, tuning reveals the true vectorization profitability prediction capacity. In a real application, the predictor would be inaccurate at predicting for the neutral programs, but the performance of these programs would not vary much in the case of mis-predictions.

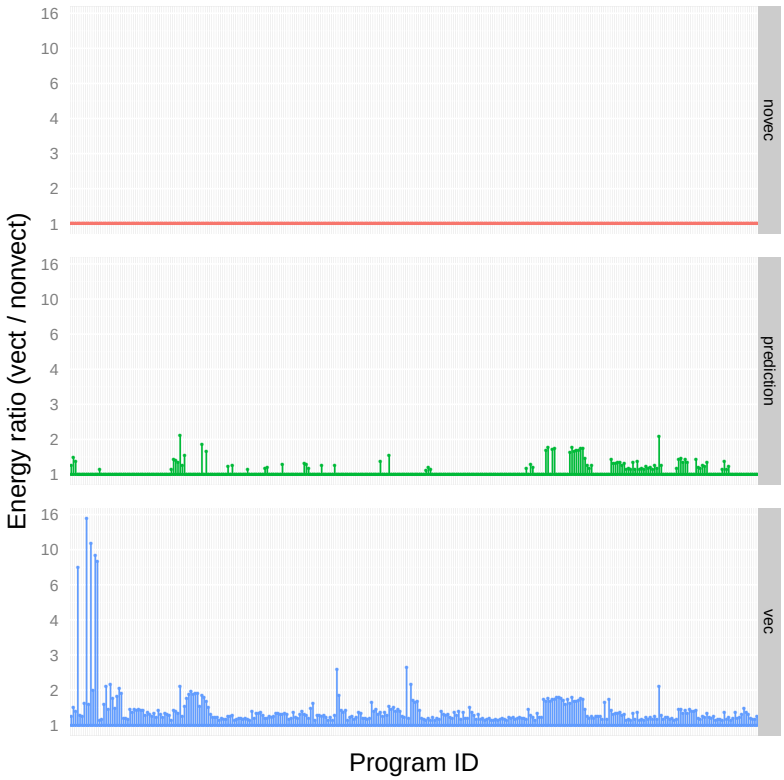
The tuning was performed until either the precision and recall stabilized or we reached the end of the neutral region. Figure 6.3 shows the tuning progression as points were removed and how it affected the precision and recall metrics. The end of the neutral region was reached and the precision and recall curves did not converge. However, by carefully removing these neutral points, the recall for the *should not vectorize* class had a dramatic increase from 20% to 70% while the recall for *should vectorize* remained stable as shown in Figure 6.2 b. As a consequence, the precision also increased from 50% to 74%. We achieved training a vectorization profitability predictor with fairly high precision and balanced recall.


Figure 6.3 : Predictor tuning.

The precision and recall bar plots presented above give an aggregate measure of the overall predictor performance. Next, we analyze the predictor performance on a per program basis to get a better sense of its utility. Figure 6.4 shows the distance in energy ratio to a perfect predictor for three optimization strategies. The vertical axis marks the energy ratios for each remaining program after predictor tuning. There is one point per program arranged in increasing energy ratio. The range of programs is divided in two; Figure 6.4 a corresponds to programs that can benefit from vectorization while Figure 6.4 b corresponds to the programs where vectorization is detrimental. These figures are explained in more detail next.




(a) Beneficial vectorization region.




(b) Detrimental vectorization region.

Figure 6.4 : Energy ratio distance to a perfect predictor.

A perfect predictor would always choose the correct answer of whether to vectorize or not. In other words, the resulting energy ratios of the perfect predictor are either below unity when vectorization is beneficial or at worst unity when it is detrimental. The three optimization strategies are *novec* (vectorization is always deactivated), *prediction* (our predictor), and *vec* (vectorization is always activated). The vertical lines show the distance from applying that strategy to the ratio of the perfect predictor. The better the strategy, the less vertical lines in the plot.

For the *novec* strategy, all beneficial cases were missed but there were no detrimental cases as vectorization was never applied. On the contrary, for the *vec* strategy all beneficial cases are hit but so are the detrimental cases. Therefore no strategy is good by itself in the general case. But when using *prediction*, which is our proposed strategy, the worst detrimental cases are avoided while at the same time hitting 76% of the beneficial vectorization cases.

Statistics for the energy ratios of Figure 6.4 a and Figure 6.4 b are summarized in Table 6.5 and Table 6.6 , respectively. Comparing the results in Table 6.4 a where vectorization is beneficial, the predictor managed to predict correctly for the program with the highest energy reduction of 94% (0.06 ratio). On average the predictor achieved 64% (0.36) reduction in energy, while the perfect predictor and the *vec* strategy reached 84% (0.16). On the other hand, the *novec* strategy missed any benefit from vectorization.

| Strategy | Avg | Sdev | Min | Max |
|---|---|---|---|---|
| Perfect | 0.16 | 0.05 | 0.06 | 0.025 |
| Vec | 0.16 | 0.05 | 0.06 | 0.025 |
| Prediction | 0.36 | 0.36 | 0.06 | 1.00 |

Table 6.5 : Energy ratio statistics for Figure 6.4 a

| Strategy | Avg | Sdev | Min | Max |
|---|---|---|---|---|
| Perfect | 1.00 | 1.00 | 1.00 | 1.00 |
| Vec | 1.33 | 1.20 | 1.06 | 15.13 |
| Prediction | 1.05 | 0.11 | 1.00 | 1.67 |

Table 6.6 : Energy ratio statistics for Figure 6.4 b

Now comparing the results of Table 6.4 b where vectorization is detrimental, on average the predictor only increased energy consumption by 5% (1.05) versus 33% (1.33) increase when the *vec* strategy was applied. And the worst prediction increased energy consumption by 67% (1.67) while the *vec* strategy increased energy consumption by 15 times (15.13). Clearly, using our predictor is the better strategy with 76% probability of choosing vectorization when appropriate while increasing the energy consumption by only 5% on average.

# Chapter 7
# CONCLUSIONS

In this research we addressed the question of whether there is a code optimization technique that can be targeted at reducing software energy consumption in embedded systems. In particular, Texas Instrument's DM3730 SoC which implements an ARM Cortex-A8 was chosen as the target platform due to the prevalence of this architecture in modern portable devices. Vectorization was identified as a powerful optimization technique that is usually studied with regards to program execution time speedups, but that was also found to have a significant impact on power and energy consumption. Then machine learning driven compilation was successfully adopted as an external module that aided the GCC compiler in choosing vectorization when predicted to be beneficial in reducing energy consumption.

It is impractical to develop traditional compiler heuristics that rely on hand-built machine models that can account for the complexity of modern hardware, compilers, and input programs. This is one of the main strengths of our proposed solution, because machine learning-driven compilation inherently encapsulates through a training process the target system and compiler behaviors.

The following contributions can be listed among those of this work to the compiler community.

1. A comprehensive survey on software-level power estimation techniques and ML driven compilation.
2. One of the few studies on how vectorization affects power and energy consumption.
3. A predictor for energy profitability through vectorization with instantaneous prediction time, 74% precision, an average of 64% decrease in energy consumption and only 5%

increase in energy in the case of mis-predictions. This is a compile-time solution that can predict profitability from high-level source code and is adaptable through training to any target system with SIMD instructions and a compiler that supports vectorization flags.

There are several ways in which this work can be expanded. All of them are paths into an open and sought out research goal: a multi-objective ML driven compilation flow that can improve itself with each execution.

1. Avoiding manual predictor tuning. This could be achieved by parsing optimization reports generated by the compiler to eliminate from the training set those programs that failed to be optimized.
2. Increasing the predictor's precision. This would require improving the set of software characteristics so that they better capture the program's dynamic behavior.
3. Predicting combinations of optimizations. This would also entail defining new software characteristics beyond those used for vectorization prediction. This requires using a multi-class rather than binary classification, and opens the way to exploring other types of ML algorithms that can be more suited for this task.
4. Expanding the types of programs that can be predicted for. This would require extracting training example programs from representative benchmarks and to also consider the impact of different input data on the program's behavior. A good lead for this is Berkeley's Seven Dwarfs [a] , a collection of algorithms that represent a broad range of important applications.
5. Considering multi-objective optimizations. For example, optimizing for energy and space at the same time.
6. Using the compiled program's software characteristics and behavior as feedback training data. In this way the ML module would learn continuously from previous experiences, rather than just during an initial training phase.

[a] http://view.eecs.berkeley.edu/wiki/Dwarfs

# Bibliography


[1] Jason Fitzpatrick. An interview with steve furber. *Commun. ACM*, 54(5):34–39, May 2011.

[2] Texas Instruments. Omap3530 power estimation spreadsheet. http://processors.wiki.ti.com/index.php/OMAP3530_Power_Estimation_Spreadsheet.

[3] Texas Instruments. Tms320dm6446/3 power consumption summary. http://www.ti.com/lit/an/spraad6b/spraad6b.pdf.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, 9 2006.

[5] Markus Lorenz, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis, and Rainer Leupers. Compiler based exploration of dsp energy savings by simd operations. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 838–841, Piscataway, NJ, USA, 2004. IEEE Press.

[6] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Leo Breiman. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, 16(3):199–231, 08 2001.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 10 2007.

[10] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7(4):20:1–20:29, December 2010.

[11] Kyu-Won choi and A. Chatterjee. Efficient instruction-level optimization methodology for low-power embedded systems. In *The 14th International Symposium on System Synthesis*, pages 147 – 152, 2001.

[12] Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of arm and thumb instructions. In *LCTES/SCOPES: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 56–64, New York, NY, USA, 2002. ACM.

[13] D.A. Ortiz and N.G. Santiago. Impact of source code optimizations on power consumption of embedded systems. In *Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 133 –136, 2008.

[14] A. Peymandoust, T. Simunic, and G. De Micheli. Low power embedded software optimization using symbolic algebra. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 1052 –1058, 2002.

[15] Wei Wu, Lingling Jin, Jim Yang, Pu Liu, and S.X.-D. Tan. A systematic method for functional unit power estimation in microprocessors. In *Design Automation Conference, 43rd*, pages 554 –557, 2006.

[16] L. Chandra and Sourav Roy. Estimation of energy consumed by software in processor caches. In *IEEE International Symposium on VLSI Design, Automation and Test, VLSI-DAT*, pages 21 –24, 2008.

[17] A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha. Hybrid simulation for embedded software energy estimation. In *Design Automation Conference, 42nd*, pages 23 –26, 2005.

[18] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437 –445, December 1994.

[19] Kyungsu Kang, Jungsoo Kim, Heejun Shim, and Chong-Min Kyung. Software power estimation using ipi(inter-prefetch interval) power model for advanced off-the-shelf processor. In *GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 594–599, New York, NY, USA, 2007.

[20] T. K. Tan, A. K. Raghunathan, G. Lakishminarayana, and N. K. Jha. High-level software energy macro-modeling. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 605–610, New York, NY, USA, 2001.

[21] A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha. Hybrid simulation for embedded software energy estimation. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 23 –26, 2005.

[22] C. Brandolese. Source-level estimation of energy consumption and execution time of embedded software. In *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 115 –123, 2008.

[23] O. Acevedo-Patino, M. Jimenez, and A.J. Cruz-Ayoroa. Static simulation: A method for power and energy estimation in embedded microprocessors. In *53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 41 –44, 2010.

[24] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 196–206, New York, NY, USA, 2012. ACM.

[26] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I Williams. Using machine learning to focus iterative optimization. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 11 pp.–, March 2006.

[27] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.*, 47(10):147–162, October 2012.

[28] G. Pekhimenko and A. Brown. Efficient program compilation through machine learning techniques. 2009. International Symposium on Code Generation and Optimization (CGO).

[29] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 24–34, New York, NY, USA, 2006. ACM.

[30] Grigori Fursin, Yuriy Kashnikov, AbdulWahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, ChristopherK.I. Williams, and Michael OBoyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[31] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Supercomputing '88. [Vol.1]., Proceedings.*, pages 98–105, Nov 1988.

[32] Antoine Trouv, Arnaldo Cruz, Hiroki Fukuyama, Jun Maki, Hadrien Clarke, Kazuaki Murakami, Masaki Arai, Tadashi Nakahira, and Eiji Yamanaka. Using machine learning in order to improve automatic {SIMD} instruction generation. *Procedia Computer Science*, 18(0):1292 – 1301, 2013. 2013 International Conference on Computational Science.