Power/Energy Estimation and Optimization for Software-Oriented Embedded Systems

by

Mostafa Elsayed Ahmed Ibrahim

Electrical Engineering Department
High Institute of Technology
Benha University

Thesis Submitted to the
Faculty of Engineering at Cairo University, Egypt
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in
Electronics and Electrical Communications Engineering

Faculty of Engineering - Cairo University
Giza, Egypt
November, 2009

Power/Energy Estimation and Optimization for Software-Oriented Embedded Systems

by

Mostafa Elsayed Ahmed Ibrahim

Thesis Submitted to the
Faculty of Engineering at Cairo University, Egypt
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
in
Electronics and Electrical Communications Engineering

Supervised by

Prof. Dr. Serag-Eldin Elsayed Habib

Electronics and Communication Department
Faculty of Engineering - Cairo University

Prof. Dr. Markus Rupp

Institute of Communications and Radio-Frequency Engineering
Vienna University of Technology

Dr. Hossam A. H. Fahmy

Electronics and Communication Department
Faculty of Engineering - Cairo University

Faculty of Engineering - Cairo University
Giza, Egypt
November, 2009

Power/Energy Estimation and Optimization for Software-Oriented Embedded Systems

by

Mostafa Elsayed Ahmed Ibrahim

Thesis Submitted to the
Faculty of Engineering at Cairo University, Egypt
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
in
Electronics and Electrical Communications Engineering

Approved by the Examining Committee:

.....

Prof. Dr. Mohamed Zaki Abd El Mageed, Member

Faculty of Engineering - Al Azhar University

.....

Prof. Dr. Ashraf ElFarghaly Salem, Member

Faculty of Engineering - Ain Shams University

.....

Prof. Dr. Serag-Eldin Elsayed Habib, Thesis advisor

Faculty of Engineering - Cairo University

.....

Prof. Dr. Markus Rupp, Thesis advisor

Faculty for Electrical Engineering and Information Technology
Vienna University of Technology

Faculty of Engineering - Cairo University
Giza, Egypt
November, 2009

ABSTRACT

The importance of power reduction of embedded systems has continuously increased in the past years. Recently, reducing power dissipation and energy consumption of a program have become optimization goals in their own right, no longer considered a side-effect of traditional performance optimizations which mainly target program execution time and/or program size. Nowadays, there is an increasing demand for developing power-optimizing compilers for embedded systems. This thesis is a step towards such important goal.

In this thesis, we develop functional-level power models and investigate several software optimization techniques for embedded-processor systems. As a specific example, we consider the powerful Texas Instruments C6416T DSP processor. We analyze the power consumption contributions of the different functional units of this DSP. We assess the effect of the compiler performance optimizations on the energy and power consumption. Moreover, we explore the impact of two special architectural features of this DSP; namely Software Pipelined Loop (SPLOOP) and the SIMD capabilities, on the energy and power consumption.

We also characterize the application-architecture correlation for our targeted architecture. The PCA multivariate statistical technique is employed to visualize the black box impact of the compiler and the hardware architecture over the software applications. This is achieved with the aid of biplots which is depicted in our analysis in such a way, so that it can show the maximum association between the application and the underlying hardware architecture. Hence, it answers the question whether a given hardware architecture is an appropriate choice for a given software application or not.

The currently-available compiler optimization techniques are handicapped for power optimization due to their partial perspective of the algorithms and due to their limited modifications to the data structures. On the contrary, other software optimization techniques, like source code transformations, can exploit the full knowledge of the algorithm characteristics, with the capability of modifying both data structures and algorithm coding. Furthermore, inter-procedural optimizations are envisioned. Hence, we investigate several loop, data and procedural source code transformations from the power and energy perspectives.

Based on our results and as a step towards a power-aware optimizing compiler, we can recommend the following recommendations for programmers and compiler designers. ***First,***

the programmers, targeting the C6000 DSP family, are strongly recommended to compile and optimize their programs by invoking the optimization level -o3 while disabling the SPLOOP feature (-mu) in conjunction with the utilization of SIMD capabilities via the employment of suitable intrinsic functions.

Second, we recommend the compiler designers to pay more attention to the circular (modulo) and bit reverse addressing schemes which are rarely utilized by the compiler. In addition, they should utilize the power-aware source code transformations.

Third, developers of power simulators need to embed a functional level power consumption model for the target processor in their simulators software.

ACKNOWLEDGEMENTS

As the author of this thesis, I am keenly aware that it represents the fruition of not only my own work, but also the support which other individuals and organizations have lent me over the years, and for which I am profoundly grateful.

First, I would like to thank my advisors, Prof. Serag E.-D Habib, Prof. Markus Rupp, and Dr. Hossam A. H. Fahmy for their support, advice, guidance, and good wishes. They have had a profound influence not only as my PhD advisors, but also on my life. Their availability at all times, dedication towards work and family, professional integrity, and pursuit of perfection helped me becoming a better individual. I am grateful to them for the freedom and flexibility they gave me during the hard PhD years I spent in both Cairo and Vienna.

I would like to express the greatest of gratitude to my parents as well as my brothers and sisters for the extraordinary way they provide me with unfaltering support, encouragement, and love, even when I am far away from them. Finally, I offer special thanks and appreciations to my wife and kids, for their love, understanding, and inexhaustible kindness.

I would like to thank the members (past and current) at the Christian Doppler Laboratory (CD Lab.) for Design Methodology of Signal Processing Algorithms Bastian Knerr, Martin Holzer, and Christoph Angerer who passed on valuable comments on drafts of my paper submissions during all stages of my research.

I would like to thank Dr. Mohammad Bakr for the fruitful discussions and for providing me with some helpful papers during my PhD years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Embedded Systems

An embedded system is a combination of computer hardware and software and sometimes additional parts, either mechanical or electronical designed to perform a dedicated function. The design of an embedded system to perform a dedicated function is in direct contrast to that of the personal computer. It is also comprised of computer hardware and software and mechanical components (disk drives, for example). However, a personal computer is not designed to perform a specific function. Rather, it is able to do many different things. Many people use the term general-purpose computer to make this distinction clear. As shipped, a general-purpose computer is a blank slate; the manufacturer does not know what the customer will do with it. Frequently, an embedded system is a component within some larger system. For example, modern cars and trucks contain many embedded systems. One embedded system controls the anti-lock brakes, another monitors and controls the vehicle's emissions, and a third displays information on the dashboard. Some luxury car manufacturers have even touted the number of processors (often more than 60, including one in each headlight) in advertisements. In most cases, automotive embedded systems are connected by a communications network [1].

In general, "embedded system" is not an exactly defined term, as many systems have some element of programmability. For example, handheld computers share some elements with embedded systems such as the operating systems and microprocessors which power them but are not truly embedded systems, because they allow different applications to be loaded and peripherals to be connected. Embedded systems exhibit certain characteristics that distinguish them from other computing systems. These characteristics are:

- **Single function:** An embedded system usually executes a certain task (or program) repeatedly.

- **Reactive:** Also called Event-Driven, continually reacts to changes in the systems environment. For example, a car's cruise controller must monitor and react to speed and brake sensors.
- **Real-time:** Must compute certain results in certain time without delay.
- **Tightly-constrained:** Because of the nature of embedded systems, their design metrics such as size, speed and power impose tight constraints.

1.1.1 Target Architectures for Embedded Systems

The most typical architectural structures for embedded systems concentrate essentially onto a range of processing units: relevant for software implementations are micro-Controllers (μCs) and Digital Signal Processors (DSPs), or even more specific Application Specific Instruction-Set Processors (ASIPs), typical candidates for hardware implementation are programmable logic and dedicated data-paths. A mixture of these components is either assembled onto a single chip for which the term System-on-Chip (SoC) has prevailed, or is composed by several chips onto a board system.

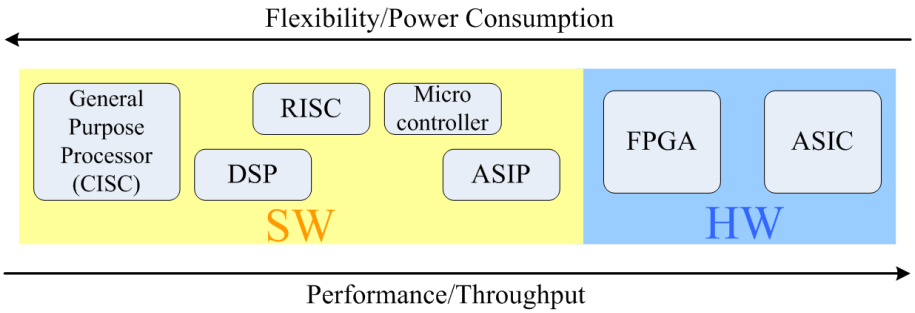

Fig. 1.1: Architectural components and their affiliation to hardware and software.

Figure 1.1 visualizes the common notion of the trade-off between hardware and software architectural components. From the left to the right the complexity of the underlying component is decreasing in terms of instruction set, sophisticated memory access, and pipelining strategies. This is counterbalanced by the increase of the computational speed towards Application Specific Integrated Circuits (ASICs), mostly measured in throughput or number of operations per time unit. The grouping of these processor classes into hardware and software systems has not been clearly defined but is generally understood [2, 3]. In the next paragraphs we present a short overview over the different embedded system architectures.

1.1.1.1 General-Purpose Processors

Although General-Purpose Processors (GPPs) are not considered as viable choices in embedded systems, a short description is given to round up the picture. These processor types are all-rounders on which nearly any application can be executed with a medium performance instead of being optimal for just a single one. Workstations, PCs, servers, and much more are typical candidates for a deployment of these processors. The steep requirements on flexibility and processing speed necessitate very complex circuit structures with superpipelining, branch prediction, hierarchical caching structures, and superscalar scheduling by prefetching instructions. The execution time of a characteristic code block varies therefore, as it is dependent on a number of dynamic effects. For real time systems with strict deadlines on certain parts of the functionality, these processors are normally inappropriate. Another obstacle for the deployment of GPPs in embedded systems is the large power consumption and the tedious and time-consuming interface design for I/O and memory access due to the aforementioned circuit complexity.

1.1.1.2 Digital Signal Processors

Digital Signal Processors (DSPs) are processors dedicated to a specific application domain of digital signal processing, e.g. mobile communication, image processing, audio/video applications. With respect to the general instruction set, they offer very much the same possibilities as general-purpose processors but with less facets and simpler circuitry. Their big advantage is the optimized circuitry for *additional* instructions catering to the specific application domain. Relevant traits for DSPs are amongst:

- Combined multiply-accumulate (MAC) operations.

  In a single instruction cycle a multiply operation of two operands is interlinked with a subsequent accumulation of the result. This instruction has a direct realization in hardware circuitry in a DSP for floating-point or fixed point number formats.
- High jump predictability and zero-overhead loops.

  A humble level of code branching and fixed loop count variable is exploited by special registers, in which start and end address and the loop counter is stored. Every iteration through the loop body triggers the counter's increment or decrement and the subsequent comparison with the end condition, thus not imposing any overhead due to loop

controlling.

- Specialized addressing techniques.
  DSPs provide address generators that are capable to increment or decrement the address pointer by a programmable step width in parallel to the actual instruction processing. Two relevant applications are the circular address scheme, which facilitates filter implementations and bit-reverse address schemes for e.g. Walsh-Hadamard or Fast Fourier Transforms.

Many embedded systems comprise DSPs with fixed-point numeric formats, since a fixed-point arithmetic logic unit (ALU) is much faster than a floating-point ALU given the same chip area. However, the transition towards fixed-point formats additionally complicates the design due to quantization noise, rounding and overflow errors.
Nowadays, C-compilers exist for most of the DSPs on the market, but crucial functions may still be designed in assembler to ensure a better exploitation of the specific architectural features of the DSP. For many applications, as in the image processing domain, time critical code parts that have been manually optimized in assembler can be embedded into C routines.
The digital signal processing domain gained significant attention due to the revolution in mobile communications. Therefore, a large variety of different DSP cores emerged with manyfold innovative architectures [4]: for instance multiple DSP platforms, very large instruction word (VLIW) DSPs, and desktop DSPs.

1.1.1.3 Microcontrollers

As the name suggests a micro controller (μC) is dedicated to control flow dominated applications like protocols that are characterized by a large number of branches, internal states, and boolean logic operations. The data throughput as well as the arithmetic operations do not play a major role. Typically, μCs are used for interrupt handling and support a very fast context switching often seen in protocol state machines. In other words, the current program context is realized completely in the RAM, so that in the case of an interrupt the program address pointer is simply set to a new address.

1.1.1.4 Application Specific Instruction Set Processors

These μPs are even more customized to their specific application domain than DSPs and micro controllers. The key idea is the application-directed generation of a *programmable* device, whose instruction set and data word widths have undergone a fierce optimization towards its purpose. As indicated in Figure 1.1, ASIPs occupy the location with the least flexibility and the highest performance in the software domain.

Since ASIPs are by definition application specific, it is difficult to classify them by their commonalities. Usually, their instruction set includes operator concatenation as MAC operations, or vector arithmetics. Similar to their larger siblings, the DSPs, their circuitry exploits parallelism of address calculation and data operations. On the contrary, ASIPs usually dispense complicated caching schemes and reduce the pin number as far as possible to enable smaller chip sizes. The development of optimizing compilers, debuggers, and linkers for ASIPs has long been subject to intense research. In recent years, a design group from RWTH Aachen developed a mature tool suite for ASIP design called LISA [5, 6], which is now commercially available in the portfolio of CoWare [7].

1.1.1.5 Field Programmable Gate Arrays

Field programmable gate arrays (FPGA) belong in our notion to the hardware domain, although being programmable as the name suggests. A regular arrangement of configurable logic blocks (CLB) is programmable by adjusting the interconnects between them in order to duplicate basic logic gates as AND, OR, XOR, memory or more complex combinatorial functions. The CLBs contain look-up tables, multiplexers, and flip-flops, whose structure usually differs widely to offer high flexibility on a single FPGA. The interconnection network occupies the major portion of the chip area of up to 90%. I/O blocks surround the CLB grid. It is in general distinguished between one time only programming of FPGAs with anti-fuse switches and reconfigurable programming of FPGAs with SRAM switches. In the first case the interconnects and configuration of the multiplexers are burned onto the die to establish a connection (thus *anti*-fuse).

Eventually, these FPGAs resemble ASICs, as their behavior is permanently determined. The configuration of SRAM based FPGAs is accomplished by setting variables in the SRAM units that determine the interconnects and multiplexers. In modern FPGAs at every power up of the FPGA the configuration is loaded from an EEPROM. The development of FPGA cir-

cuitry resembles very much the development of ASICs. Classical hardware design tools are utilized to develop schematics and netlists of integrated circuit elements (gates, flip-flops). The FPGA vendor usually offers integrated tools for the schematics, which automatically transpose the netlist into the configuration data and eventually configures the FPGA.

1.1.2 Embedded Systems Design Metrics

To cope with the rapidly growing complexity of embedded systems, designers must work at higher levels of abstraction [8]. Depending on the abstraction layer, the level of detail used to describe the system, designers can address different concerns. The key is to model the system at each abstraction layer with as little detail as possible and then collect performance metrics that help the development team make sound engineering decisions.
Among the many metrics used to characterize the quality of an embedded system design, we will go through the following metrics:

- Execution-time
- Non-Recurring Engineering (NRE)
- Flexibility
- Time-to-Market
- Power consumption
- Size
- Unit cost
- Maintainability

Execution-time: (as a measure of the embedded system performance) plays an important role in the area of embedded systems and especially hard constrained real-time systems. These systems are typically subject to stringent timing constraints, which often result from the interaction with the surrounding physical environment. It is essential that the computations are completed within their associated time bounds; otherwise severe damages may result, or the system may be unusable. Therefore, a schedulability analysis has to be performed which guarantees that all timing constraints will be met. Schedulability analysis requires the upper bounds for the execution times of all tasks in the system to be known. These bounds must be safe, that is, they may never underestimate the real execution time. Furthermore, they should be tight, that is, the overestimation should be as small as possible. In modern microprocessor architectures, caches, pipelines, and all kinds of speculation are key features for improving (average-case) performance. Unfortunately, they make the analysis of the timing behavior of instructions very difficult, since the execution time of an instruction depends on the execution history. A lack of precision in the predicted timing behavior may lead to a waste of hardware resources, which would have to be invested in order to meet the requirements [9].

Power consumption: has emerged as one of the most important embedded systems design metrics. This is largely due to the proliferation of mobile battery-powered computing devices, the increasing speed and density of CMOS (complementary metal-oxide semiconductor) VLSI (very large-scale integration) circuits, and continuous shrinking of the transistor feature size of deep sub-micron technologies [10].

Power consumption is a major concern for portable or battery-operated devices. Power issues, such as how long the device needs to run and whether the batteries can be recharged, need to be thought out ahead of time. In some systems, replacing a battery in a device can be a big expense. This means the system must be conscious of the amount of power it uses and take appropriate steps to conserve battery life. There are several methods to conserve power in an embedded system, including clock control, power-sensitive processors, low-voltage ICs, and circuit shutdown. Some of these techniques must be addressed by the hardware designer in his selection of the different system ICs. Some power-saving techniques are under software control.

It might seem ideal to select the fastest and most powerful processor available for a particular embedded system. However, one of the tasks of the hardware designer is to use just enough processing power to enable the device to get its job done. This helps reducing the power consumed by the device. The selected processor plays a key role in determining the amount of power an embedded system will consume. In addition, some processors can automatically shut down different execution units when they are not in use [1].

One software technique offered by many embedded processors to conserve power is different operating modes (e.g. run, idle and sleep). These modes allow the software to scale processor power consumption to match the moment-by-moment needs of the application. Operating the processor in different modes can save quite a bit of power. Another power-saving technique that can be controlled by software is to vary processor clock speeds. Some processors accept a fixed-input clock frequency but feature the ability to reduce internal clock speeds by programming clock configuration registers. Software can reduce the clock speed to save power during the execution of noncritical tasks and increase the clock speed when processing demands are high. Substantial power/energy savings can also be achieved through the implementation of adequate dynamic power management policies, for example, tracking instantaneous workloads (or levels of resource utilization) and shutting-down idling/unused resources, so as to reduce leakage power, or slowing down under-utilized re-

sources, so as to decrease dynamic power dissipation [1, 9].

Non-Recurring Engineering (NRE): refers to the one-time cost of researching, developing, designing, and testing a new product. When budgeting for a project, NRE must be considered in order to analyze if a new product will be profitable. Even though a company will pay for NRE on a project only once, NRE can be considerably high and the product will have to sell well enough to produce a return on the initial investment. NRE is unlike production cost, which must be paid continually in order to maintain production.
In a project-type company, large parts (possibly all) of the project represent NRE. In this case the NRE cost are likely be included in the first project's cost. If the company cannot recover this cost, it will have to consider funding part of these from reserves (possibly make a project loss) in the hope that the investment can be recovered from additional profit on future projects [11].
Size: the physical space required by the system, e.g., bytes of memory for software and logic gates or Configurable Logic Blocks (CLB) for hardware.

Flexibility: the ability to change the functionality of the system without incurring heavy NRE cost.

Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost. When comparing technologies by cost, the best option depends on quantity. Let's assume that there are two alternatives technologies for a certain product. The first alternative is technology A; which has a NRE cost of $\$2\,000$ and a unit cost of $\$100$. The second alternative technology B with a NRE cost of $\$30\,000$ and a unit cost of $\$30$. Figure 1.2 illustrates the strong impact of the NRE and the production volume on the final product unit cost [12].
Time-to-Market: the time required to develop a system to the point that it can be released and sold to customers. Growing system complexities, driven by increased IC capacities, requires designers to do more in less time. Figure 1.3 indicates the importance of the time-to-market design metrics from the revenue point of view.

Delays can be costly. Equation (1.1) expresses the percentage of revenue lost. Assume that market rise in Fig. 1.3 is at 45 degree angle and that the product life is $2W$ with a peak market rise at W. Hence (1.2) and (1.3) define the on-time and delayed design entry point for the product. By substituting in (1.1) the final revenue model in (1.4) is obtained. For

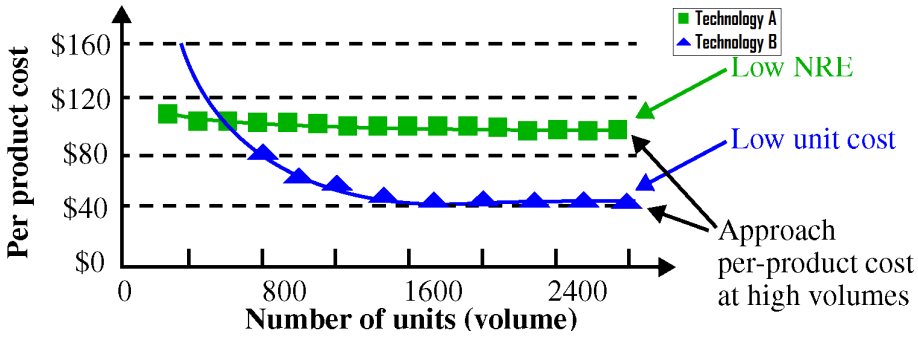

Fig. 1.2: NRE and production volume influence on the product unit cost (reproduced from [12]).

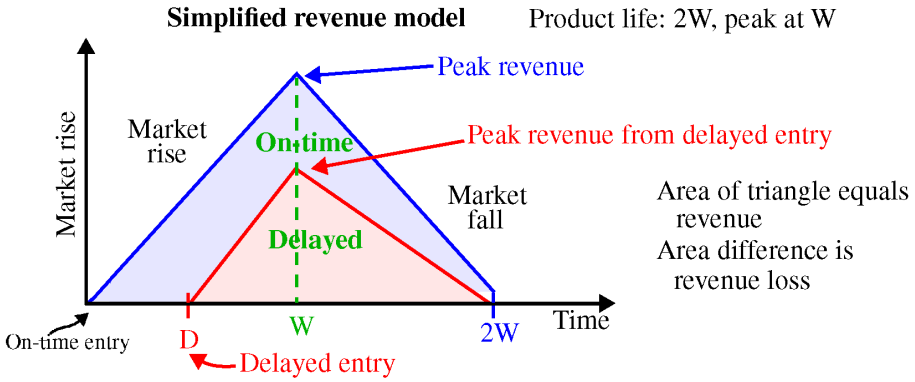

Fig. 1.3: Time-to-Market design metrics impact on the market revenue (reproduced from [12]).

example let $2W = 52$ weeks, delay $D = 10$ weeks hence, by substituting in (1.4) we find out that the percentage revenue lost due to 10 weeks delay in the entry to the market equals 50% [12].

$$\%\ \text{revenue lost} = \frac{\text{revenue}_{\text{On-time}} - \text{revenue}_{\text{Delayed}}}{\text{revenue}_{\text{On-time}}} \times 100 \tag{1.1}$$

$$\text{revenue}_{\text{On-time}} = \frac{1}{2} \times 2W \times W = W^2, \tag{1.2}$$

$$\text{revenue}_{\text{Delayed}} = \frac{1}{2} \times (W - D + W) \times (W - D), \tag{1.3}$$

$$\%\ \text{revenue lost} = \frac{D(3W - D)}{2W^2} \times 100 \tag{1.4}$$

Maintainability: measures the ease and speed with which a system can be restored to oper-

ational status after a failure occurs [12–14].

1.2 Motivation

Embedded systems are rapidly growing in the few recent years. As shown in Fig. 1.4 more than 30% of the car is now in electronics and almost 90% of innovations will be based on electronics [14]. As illustrated in the previous section, the power and energy constraints on embedded systems are becoming increasingly tight as complexity and performance requirements continue to be pushed by the user demand [15].

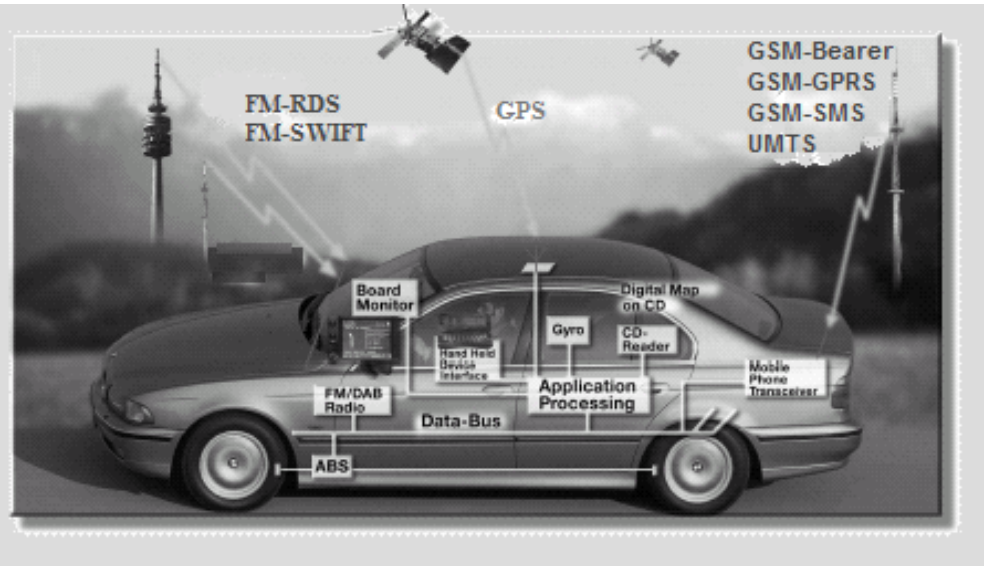

Fig. 1.4: Embedded systems in automotive electronics (reproduced from [14]).

Power density has a direct impact on packaging and cooling cost, and can also affect system reliability, owing to electromigration [16] and hot-electron [17] degradation effects. Thus, the ability to decrease power density, while offering similar performance and functionality, critically enhances the competitiveness of a product. Moreover, for battery operated portable systems, maximizing battery lifetime translates into maximizing duration of service, an objective of paramount importance for this class of products. Power is thus a primary figure of merit in contemporary embedded system design [9].

Integrated circuits in their various manifestations consume some amount of electric power. This power is dissipated both by the action of the switching devices contained in IC (such as transistors) as well as heat due to the resistivity of the electrical circuits. This is a major

consideration in the design of micro-processors and the embedded systems they are utilized in.

Today, digital signal processors (DSPs) are frequently used in embedded systems to permit application specifications in software. Processor speeds have doubled approximately every 18 months as predicted by Moore's law [18]. In order to get an energy-efficient system consisting of processor and compiler, it is necessary to optimize hardware as well as software [19].

The program behavior is difficult to predict due to its heavy dependence on application and run-time conditions [20, 21]. For embedded systems, the application performance can be optimized by utilizing parallel hardware architectures, such as Very-Long Instruction Word (VLIW) architectures [6]. VLIW architectures are a suitable alternative for exploiting Instruction-Level Parallelism (ILP) in programs, that is, for executing more than one basic (primitive) instruction at a time. These processors contain multiple functional units. They fetch from the instruction cache a VLIW containing several primitive instructions, and dispatch the entire VLIW for parallel execution.

These capabilities are exploited by compilers which generate code that has grouped together independent primitive instructions executable in parallel. The processors have a relatively simple control logic because they do not perform any dynamic scheduling nor reordering of operations (as is the case in most contemporary superscalar processors). The instruction set for a VLIW architecture tends to consist of simple instructions (RISC-like). The compiler must assemble many primitive operations into a single instruction word such that the multiple functional units are kept busy, which requires enough ILP in a code sequence to fill the available operation slots.

1.3 Contributions

The main contributions of this dissertation can be summarized in the following folds:

1. First, a complete power and energy characterization of the VLIW fixed-point C6416T DSP is performed.
   - The first step toward this power characterization is the design and implementation of a precise high level software power consumption model for the targeted

processor, while running a software algorithm.

- Next, we prove the validation and precision of our model on many typical algorithms applied in signal and image processing.
- The power consumption estimated by our model, is compared to the physically measured power consumption, achieving a very low average estimation error of 1.65% and a maximum estimation error of only 3.3%

2. Second, a quantitative study is provided wherein we examine the influence of the global optimizations of the C/C++ compiler, of the code composer studio, with respect to the energy and power consumption.
   - we find that enabling general compiler performance optimizations considerably increase the power consumption of the DSP, on average, by 30.35% when the third optimization level (-o3) is invoked.
   - In order to analyze the causes for the power increase we study the effect on some other performance measures, such as:
     - L1D cache misses
     - Memory references
     - Instructions per cycle
     - CPU stall cycles
   - The impact of the special C64x+ architecture feature; namely Software Pipelined Loop (SPLOOP) on the energy usage and power consumption is evaluated.
   - Moreover, the impact of utilizing the targeted architecture Single Instruction Multiple Data (SIMD) capabilities on the energy usage and power consumption is evaluated.
   - Finally, the characterization of the application-architecture correlation for the targeted platform. The Pricipal Component Analysis (PCA) multivariate statistical technique is employed to visualize the black box impact of the compiler and the hardware architecture over the software applications. This is achieved with the aid of biplots which is depicted in our analysis in such a way, so that it can show the maximum association between the application and the underlying hardware architecture. Hence, it answers the question whether a given hardware architecture is an appropriate choice for a given software application or not.
3. Third, since the CCS allows very limited control over the individual optimization tasks embedded within each global optimization levels, we assess the effect of applying source to source code transformations on the power, energy and performance. The

source code transformations that are presented in this work are classified into three major groups: loop, data and procedural transformations.

This thesis is based on the following publications:

- Mostafa E. A. Ibrahim, Markus Rupp, and S. E.-D. Habib. Power consumption model at functional level for VLIW digital signal processor. In proceedings of the conference on Design and Architectures for Signal and Image Processing (DASIP'08), Bruxelle, Belgium, pages 147-152, November 2008.
- Mostafa E. A. Ibrahim, Markus Rupp, and Hossam A. H. Fahmy. Power Estimation Methodology for VLIW Digital Signal Processor. In proceedings of the IEEE conference on Signals, Systems and Computers (SSC'08), Asilomar, CA, US, IEEE, pages 1840-1844, October 2008.
- Mostafa E. A. Ibrahim, Markus Rupp, and S. E.-D. Habib. Compiler-Based Optimizations Impact on Embedded Software Power Consumption. In proceedings of the joint IEEE conference NEWCAS-TAISA'09 Toulouse, France, pages 247-250, June 2009.
- Mostafa E. A. Ibrahim, Markus Rupp, and S. E.-D. Habib. Performance and Power Consumption Trade-offs for a VLIW DSP. In proceedings of the IEEE International Symposium on Signals, Circuits and Systems (ISSCS'09), Iasi, Romania, pages 197-200, July 2009.

- Mostafa E. A. Ibrahim, Markus Rupp, and Hossam A. H. Fahmy. Impact of Code Transformations and SIMD on Embedded Software Power Consumption. In proceedings of the IEEE International Conference on Computer Engineering and Systems (ICCES'09), Cairo, Egypt, December 2009.

1.4 Thesis Outline

The rest of this thesis is structured as follows:

Chapter 2 reviews the evolution and state-of-the-art in processor's power consumption models that rely on the running software. In general two main abstraction levels are surveyed in this chapter. The low-level power modeling and estimation techniques cover the circuit-level, gate-level, Register Transfer (RT)-level and the micro-architecture level. The high-level techniques can be divided into two categories the Instruction Level Power Analysis

(ILPA) and the Functional Level Power Analysis (FLPA). The software and hardware based power saving techniques are surveyed, focusing on the recent attempts to evaluate the impact of different compiler optimizations on the energy and power consumption of the processor. The variety of existing source to source code transformations are analyzed from power and energy perspectives .

Chapter 3 proposes a precise model to estimate the power consumption of the targeted DSP, while running a software algorithm. The modeling is performed at the functional level making this approach distinctly different from other modeling approaches in low level techniques. This means that the power consumption can be identified at an early stage in the design process, enabling the designer to explore different hardware architectures and algorithms. After applying the FLPA, the targeted C6416T architecture is subdivided into six distinct functional blocks (clock tree, instruction management unit, processing unit, internal memory, L1 data cache and L1 program cache). The parameters that affect the power consumption for the identified functional blocks are determined.Typical signal and image processing algorithms and a real time application are used for the purpose of validating the proposed model. The estimated power consumption is compared to the physically measured power consumption

Chapter 4 explores the performance and power trade-offs of the VLIW Texas Instruments C6416T DSP. We assess the effect of the compiler performance optimizations on the energy and power consumption. Moreover, we explore the impact of two special architectural features of this DSP; namely Software Pipelined Loop (SPLOOP) and the SIMD capabilities, on the energy and power consumption. The code binaries utilized in this study were generated with aid of the Texas Instrument C/C++ Compiler that is embedded in the CCS, which allows control over the whole set of optimizations. Finally, we explore the correlation between the software applications and the underlying hardware architecture at which these applications are executed . We employ the Principal Component Analysis (PCA) biplots to visualize the black box impact of compiler and hardware architecture over the software applications.

Chapter 5 assesses the effect of applying source to source code transformations on the power, energy and performance. The source code transformations that are presented in this work are classified into three major groups: data oriented transformations, loop oriented

transformations and finally procedural and inter-procedural transformations. To evaluate the effectiveness of the applied transformations we compile each program, both the original and transformed version, on the target architecture (C6416T DSK). Next, we record the current drawn from the core CPU and hence the consumed power. With the aid of the compiler's profiler we also record the run time and other execution characteristics such as memory references, L1D cache misses and so on. To obtain reliable and precise information, we repeat the whole measuring procedure for each transformation multiple times.

Chapter 6 concludes the thesis commenting on the probable impact of the obtained results. In addition to the summary of the presented unique contributions, a discussion of the possible future directions of research based on this thesis is presented.

Appendix A illustrates an overview of the DSP products of Texas Instrument Inc., the market leader in DSP field, focusing on the architecture of our target DSP C6416T. Moreover this appendix lists the C6416T simulator's performance monitoring events along with their description.

Appendix B shows how the algorithmic parameters, required to estimate the power consumption of the running algorithm, are computed. In addition, it shows the actual computed parameters, the estimated, the measured power consumption for different image and signal processing benchmarks and finally a complete power consumption model at an operating frequency of $1\,000$MHz.

Appendix C explains some basic foundations regarding the multivariate statistical technique named Principal Component Analysis (PCA) which is used to characterize the application-architecture correlation.

Appendix D lists the different acronyms utilized in this thesis.

2 RELATED WORK

2.1 Introduction

In this chapter we review the evolution and state-of-the-art in processor's power consumption models that rely on the running software. In general two main abstraction levels are surveyed in this chapter. The low-level power modeling and estimation techniques cover the circuit-level, gate-level, Register Transfer (RT)-level and the micro-architecture level. The high-level techniques can be divided into two categories the Instruction Level Power Analysis (ILPA) and the Functional Level Power Analysis (FLPA). We also survey the software and hardware based power saving techniques, focusing on the recent attempts to evaluate the impact of invoking different compiler optimization levels on the energy and power consumption of the processor. Finally, we analyze the variety of existing source to source code transformations from power and energy perspectives.

2.2 Software Power Consumption Estimation Techniques

This section summarizes the most recent contributions to the problem of power modeling and estimation. Recent approaches to model the power consumption of the software running on a processor can be separated into two main categories:

- Low-Level or Hardware level models.
- High-Level models.

Hardware level models calculate power and energy from detailed electrical descriptions, comprising circuit level, gate level, register transfer (RT) level or system level. High-Level models deal only with instructions and functional units from the software point of view and without electrical knowledge of the underlying architecture [22].

2.2.1 Low-Level Estimation Techniques

The level of detail in the modeling performed by the power simulator influences both the accuracy of estimation as well as the speed of the simulator. In this section we survey the frequently used models at low level. The low level power consumption estimation techniques cover a range of abstractions such as the:

- Circuit/Transistor level.
- Logic gate level.
- RT-level.
- Architectural level.

2.2.1.1 Transistor-Level Estimations

The representation of a microprocessor in terms of transistors and nets is extremely complex and requires to undergo all the steps of the design flow and the layout, routing and parameter extraction inclusive. This is rarely feasible since only few big companies have the know-how and the technology in-house while most of them rely on silicon vendors for the lowest-level steps. Furthermore, a transistor-level view of the system uses components models based on linearized differential equations and works in the continuous time domain. This implies that a simulation of more than one million transistors, even for few clock cycles, requires times that are usually not affordable and anyway not practical for the high-level power characterization [23].

The PowerMil [24] is an early attempt to build a low-level power consumption simulator. PowerMil is a transistor level simulator for simulating the current and power behavior in VLSI circuits. It is capable of simulating detailed current behavior in modern deep sub-micron CMOS circuits, including sophisticated circuitries such as sense-amplifiers, with speed and capacity approaching conventional gate level simulators. For more details about power estimation techniques in VLSI circuits refer to [25, 26].

2.2.1.2 Gate-Level Estimations

Methods to estimate the power consumption based on gate-level descriptions of microprocessors or micro controller cores have been proposed in literature. The main advantage of such methods with respect to transistor-level simulation approaches is that the simulation is event-driven and takes place in a discrete time domain, leading to a considerable reduction of the computational complexity, without a significant loss of accuracy [23].

An example for the gate-level power estimators is the model presented by Chou [27]. Chou et al. present an accurate estimation of signal activity at the internal nodes of sequential logic circuits. The power consumption estimation in Chou et al. is a Monte Carlo based approach that take spatial and temporal correlations of logic signals into consideration.

2.2.1.3 RT-Level Estimations

A design described at RT-level can be seen as a collection of blocks and a network of interconnections. The blocks, sometimes referred to as macros, are adders, registers, multiplexers and so on, while the interconnections are simply nets or group of nets. An assumption underlying the great majority of the approaches presented in literature is that the power properties of a block can be derived from an analysis of the block isolated from a design, under controlled operating conditions. The main factor influencing the power consumption model of a macro is the input statistics [23].

Most of the research in RT-level power estimation is based on empirical methods that measure the power consumption of existing implementations and produce models from those measurements. This is in contrast to approaches that rely on information-theoretic measures of activity to estimate power [28, 29]. Measurement-based approaches for estimating the power consumption of datapath functional units can be divided into two sub-categories. The first technique, introduced by Powel and Chau [30], is a fixed-activity micro-modeling strategy called the Power Factor Approximation (PFA) method. The power models are parameterized in terms of complexity parameters and a PFA proportional constant. Similar schemes were also proposed by Kumar et al. [31] and Liu and Svensson [32]. This approach assumes that the inputs do not affect the switching activity of a hardware block. To remedy this problem, activity-sensitive empirical power models were developed. These schemes are

based on predictable input signal statistics; an example is the method proposed by Landman and Rabaey [33]. Although the individual models built in this way are relatively accurate (a 10% - 15% error rate), overall accuracy may be negatively affected due to incorrect input statistics or the inability to correctly model the interaction.

The second empirical technique, transition-sensitive power models, is based on input transitions rather than input statistics. The method, proposed by Mehta, Irwin, and Owens [34], assumes a power model is provided for each functional unit- a table containing the power consumed for each input transition. Closely related input transitions and power patterns can be concentrated in to clusters, thereby reducing the size of the table. Other researchers have also proposed similar macro-model based power estimation approaches [35, 36].

2.2.1.4 Architectural-Level Estimations

Recently, various architectural power simulators have been designed that employ a combination of lower level of abstraction power consumption models. These simulators derive power estimates from the analysis of circuit activity induced by the application programmes during each cycle and from detailed capacitive models for the components activated. A key distinction between these different simulators is in the degree of estimation accuracy and estimation speed. For example, the *SimplePower* power simulator [37] employs a transition-sensitive power model for the datapath functional unit. The *SimplePower* core accesses a table containing the switch capacitance for each input transition of the functional unit exercised.

The use of a transition-sensitive approach has both design challenges as well as performance concerns during simulation. *The first concern* is that the construction of these tables is time consuming. Unfortunately, the size of this table grows exponentially with the size of the inputs. The table construction problem can be addressed by partitioning and clustering mechanisms. Further, not all tables grow exponentially with the number of inputs. For example, consider a bit-independent functional unit such as a pipeline register where the operation of each bit slice does not depend on the values of other bit slices. In this case, the only switch capacitance table needed is a small table for a one-bit slice. The total power consumed by the module can be calculated by summing the power consumed by each bit transition.

A second concern with employing transition-sensitive models is the performance cost of the table lookup for each component access in a cycle. In order to overcome this cost, simulators such as *SoftWatt* [38] and *Wattch* [39] use a simple fixed-activity model for the functional unit. These simulators only track the number of accesses to a specific component and utilize an average capacity value to estimate the power consumed. Even the same simulator can employ different types of power models for different components. For example, *SimplePower* estimates the power consumed in the memories utilizing analytical models [40]. In contrast to the datapath components that utilize a transition-sensitive approach, these models estimate the power consumed per access and do not accommodate the power differences found in sequences of accesses.

One of the most widely used micro-architectural power simulators is *Wattch* [39]. *Wattch* is a power simulator for superscalar, out-of-order, processors. It has been developed with aid of the infrastructure offered by *SimpleScaler* [41]. *SimpleScaler* performs fast, flexible, and accurate simulation of modern processors that implement a derivative of the MIPS-IV architecture and support superscalar, out-of-order, execution. The power estimation engine of *Wattch* is based on the *SimpleScaler* architecture, but in addition, it supports detailed cycle-accurate information for all models, including datapath elements, memory and Content Addressable Memory(CAM) arrays, control logic, and clock distribution network. *Wattch* uses activity-driven, parameterizable power models, and it displayed an accuracy better than 10% when tested on three different architectures. Another approach to evaluate energy estimates at the architectural-level exploits the correlation between performance and energy metrics. These techniques [42, 43] use performance counters present in many current processors architectures to provide runtime energy estimates [44].

While providing excellent accuracy; low-level power estimation methodologies are slow and impractical for analyzing the power consumption at an early design stage. Moreover, these methodologies require the availability of lower level circuit details or a complete Hardware Description Language (HDL) design of the targeted processor, which is not available for most of commercial off-the-shelf processors.

2.2.2 High-Level Estimation Techniques

Recently, the demand increased for high level power estimation simulators that allow an early design space exploration from the power consumption perspective. The existing high-level power estimation models can be classified into two main categories, Instruction Level Power Analysis (ILPA) and Functional Level Power Analysis (FLPA).

2.2.2.1 Instruction Level Power Analysis

An instruction level power model for individual processors was first proposed by V. Tiwari et al. [45]. By measuring the current drawn by the processor as it repeatedly executes distinct instructions or distinct instruction sequences, it is possible to obtain most of the information that is required to evaluate the power consumption of a program for the processor under test. V. Tiwari et al. modeled the power consumption of the Intel DX486 processor. Power is modeled as a base cost for each instruction plus the inter-instruction overheads that depend on neighboring instructions. The base cost of an instruction can be considered as the cost associated with the basic processing needed to execute the instruction. However, when sequences of instructions are considered, certain inter-instruction effects come into play, which are not reflected in the cost computed solely from base cost. These effects can be summarized as:

- Circuit state: switching activity depends on the current inputs and previous circuit state. In other words the difference between the bit pattern of two successive instructions.
- Resource Constraints: Resource constraints in the CPU can lead to stalls e.g. pipeline stalls and write buffer stalls.
- Cache Misses: Another inter-instruction effect is the effect of cache misses. The instruction timings listed in manuals give the cycle count assuming a cache hit. For a cache miss, a certain cycle penalty has to be added to the instruction execution time.

An experimental method is proposed by V. Tiwari et al. to empirically determine the base and the inter-instructions overhead cost. In this experimental method, several programs containing an infinite loop consisting of several instances of the given instruction or instruction

sequences are used. The average current drawn by the processor core during the execution of this loop is measured by a standard off-the-shelf, dual-slope integrating digital multi-meter as shown in Fig. 2.1.

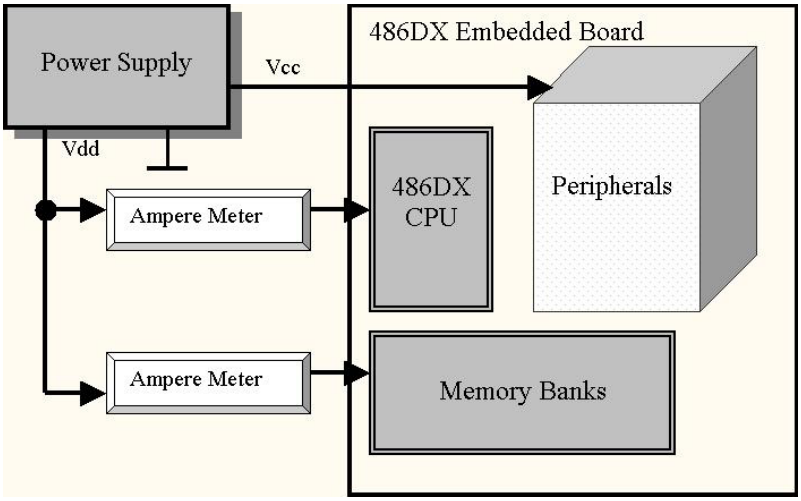

Fig. 2.1: Experimental Setup used for current measurements in [45].

Much more accurate measuring environments have been proposed to precisely monitor the instantaneous current drawn by the processor instead of the average current. One of these approaches has used a high-performance current mirror, based on bipolar junction transistors as current sensing circuit as shown in Fig. 2.2. The power profiler in Niloladies et al. [46] receives as input the trace file of executed assembly instructions, generated by an appropriate processor simulator, and estimates the base and inter-instruction energy cost of the executed program taking into account the energy sensitive factors as well as the effect of pipeline stalls and flushes. The main disadvantage of this approach is the current measuring complexity. [47].

Another approach, to reduce the spatial complexity of instruction-level power models, is presented in [48]. Therein, inter-instruction effects have been measured by considering only the additional energy consumption observed when a generic instruction is executed after a No-Operation (NOP) instruction.

An attempt to modify the original ILPA to create an instruction level power model with a gate level simulator is carried out by Sama et al. [49]. In this approach, the power cost values were obtained through a power simulator rather than actual measurement; thus modeling is possible at design time and can be part of micro-architecture and/or instruction set architecture exploration. More researchers attempted to enhance the original Tiwari ILPA power

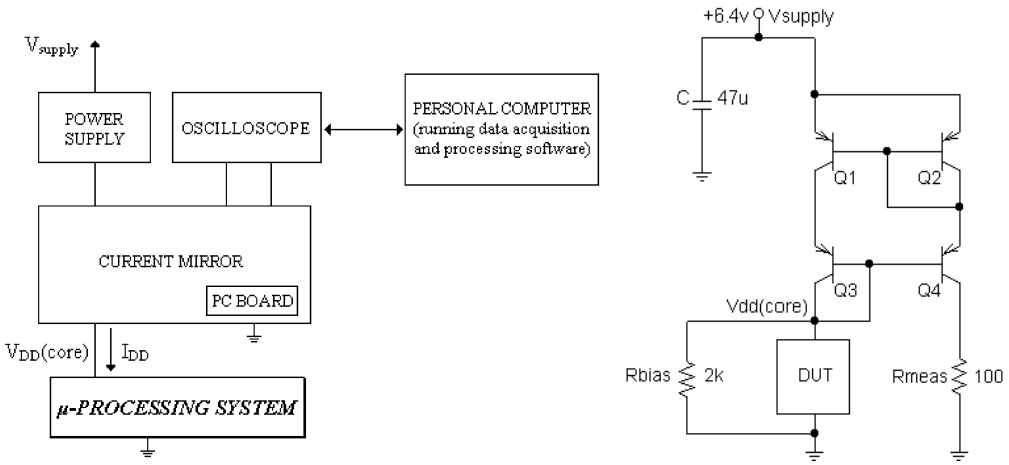

Fig. 2.2: (a) Experimental Setup for current measurement, (b) The simple current mirror. DUT is the Device Under Test [46].

consumption modeling technique as in [50–52].

The ILPA based methods have some drawbacks, one of these drawbacks is that the number of current measurements is directly related to the number of instructions in the Instruction Set Architecture (ISA), and also the number of parallel instructions composing the very long instruction in the VLIW processor. The problem of instruction level power characterization of K-issue VLIW processor is $O(N^{2K})$ where N is the number of instructions in the ISA and K is number of parallel instructions composing the VLIW [53]. Also they do not provide any insight on the instantaneous causes of power consumption within the processor core, which is seen as a black-box model. Moreover, the effect of varying data (as well as address) is ignored in the ILPA models, though this effect can be accounted by an additive factor [54].

2.2.2.2 Function Level Power Analysis

FLPA was first introduced by J. Laurent et al. in [55]. Figure 2.3 illustrates the process of estimating the power consumption with aid of the FLPA technique. The basic idea behind the FLPA is the distinction of the processor architecture into functional blocks like Processing Unit (PU), Instruction Management Unit (IMU), internal memory and others [55]. First, a functional analysis of these blocks is performed to specify and then discard the non-consuming blocks (those with negligible impact on the power consumption). The second step is to figure out the parameters that affect the power consumption of each of the power consuming blocks. For instance, the IMU is affected by the instructions dispatching rate

which in turn is related to the degree of parallelism. In addition to these parameters, there are some parameters that affect the power consumption of all functional blocks in the same manner such as operating frequency and word length of input data [56]. The functional level power modeling approach is applicable to all types of processor architectures. Furthermore, FLPA-modeling can be applied to a processor with moderate effort and no detailed knowledge of the processors architecture is necessary [57].

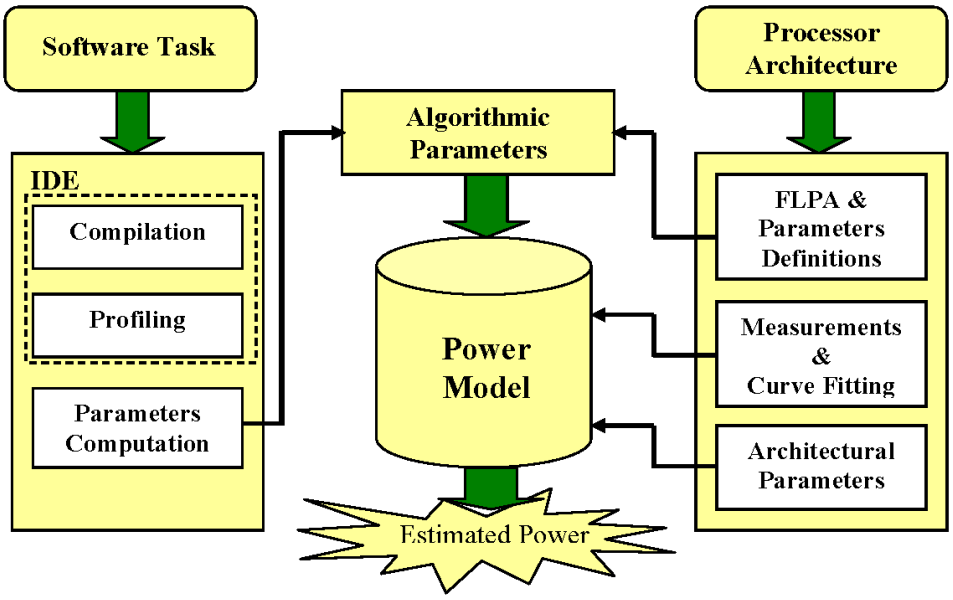

Fig. 2.3: Functional level power estimation general methodology.

2.3 Power Saving Techniques: Overview

Low power design is a complex endeavor requiring a broad range of strategies from floor planning on silicon substrate to the design of application software. In Table 2.1, we enlisted several strategies for achieving power efficiency in an power-conscious system design. In this section, we review some of these strategies.

2.3.1 Manufacturing Level Power Saving

There are three major sources for the power dissipation in digital CMOS circuits, these sources are summarized in (2.1) [58].

$$\mathrm{P} = K \cdot C_L \cdot V_{dd}^2 \cdot F + I_{SC} \cdot V_{dd} + I_{Leakage} \cdot V_{dd}, \tag{2.1}$$

Tab. 2.1: Power saving techniques for embedded systems.

Power Saving Technique	Abstract Level
Manufacturing level power saving	Low level
Processor level power saving	Intermediate level
Dynamic voltage and frequency scaling (DVFS)	Intermediate level
Battery aware power saving	High level
Compiler level power saving	High level

where C_L is the output capacitance load, V_{dd} is the supply voltage, K is the transition activity factor which is defined as the average number of times the circuit makes a power consuming transition in a single clock cycle (this term is defined in [59] as the probability that a power consuming transition occurs), and F is the operating clock frequency. The short circuit current pulse is expressed by the term I_{SC} which is generated when both n-CMOS and p-CMOS transistors are briefly turned on during the output switching, and $I_{Leakage}$ is the leakage current.

It is expected that employing low-power electronics can achieve significant power saving. Half of this power reduction will come from architecture changes and management of switching activity. The other half of this power reduction will come from the utilization of advanced materials technology to allow reduction of V_{dd} to 1V or even below, while also reducing C_L [58, 59].

2.3.2 Processor Level Power Saving

One software technique offered by many embedded processors to conserve power is different operating modes. These modes allow the software to scale processor power consumption to match the moment-by-moment needs of the application [1]. For example:

- Run-mode: the processor core runs at its normal frequency, this is the normal or default operating mode.
- Idle-mode: the processor core is not clocked, but the other peripheral components operate as normal.

- Sleep-mode: this is the lowest power state for the processor.

The embedded system designer needs to determine when the system is not doing anything and how to wake up the processor when it needs to operate, and the designer need to know what events will wake up the system. For example, in an embedded system that sends some data across a network every few minutes, it makes sense to be able to shut down the device to conserve power until it is time to send the data. The device must still be able to wake up in case an error condition arises. Therefore, the designer must understand how a peripheral circuit wakes up the processor when the processor needs to operate (including how long it takes the circuit to wake up and whether any re-initialization needs to be done) [1].

2.3.3 Dynamic Voltage and Frequency Scaling

Dynamic power consumption in a processor (general purpose or application specific) can be decreased by reducing two of its key contributors, supply voltage and clock frequency. In fact, since the power dissipated in a CMOS circuit is proportional to the square of the supply voltage, the most effective way to reduce power is to scale down the supply voltage [9]. Dynamic voltage scaling (DVS) [60, 61] refers to runtime change in the supply voltage levels supplied to various components in a system so as to reduce the overall system power dissipation while maintaining a total computation time and/or throughput requirement. Figure 2.4 shows an example DVS architecture.

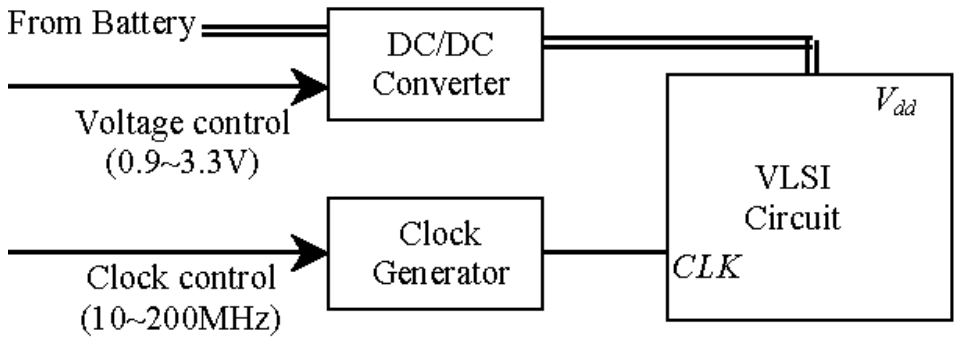

Fig. 2.4: Dynamic voltage scaling example.

Since static voltage scaling cannot deal with variable workload situations in real-time systems, one must be able to change the supply voltages dynamically for different workloads. Meanwhile, one must carefully design the clock generation circuitry because clock rate must

decrease (or increase) with the decrease (or increase) of the supply voltage to make sure that the circuit can work properly [62].

Many contemporary processor families, such as Intels XScale [63], IBMs PowerPC405LP [64], and Transmeta's (Transmeta is acquired by Novafora Inc. since January 2009) Crusoe [65], offer dynamic voltage and frequency scaling features. For example, the Intel 80200 processor, which belongs to the XScale family of processors mentioned earlier, supports a software programmable clock frequency. Specifically, the voltage can be varied from 1.0 to 1.5 V, in small increments, with the frequency varying correspondingly from 200 to 733 MHz, in steps of $33/66$ MHz. Figure 2.5 illustrates the power consumption in a digital circuit with and without Dynamic Voltage and Frequency Scaling (DVFS) [9].

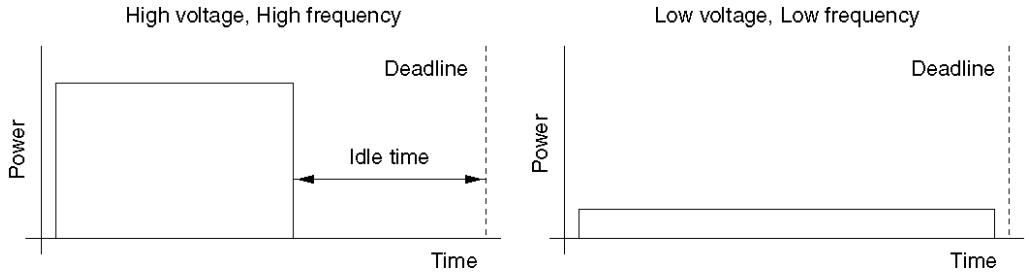

Fig. 2.5: Power consumption without and with dynamic voltage and frequency scaling.

The simplest way to take advantage of the scaling features discussed above is by carefully identifying the smallest supply voltage (and corresponding operating frequency) that guarantee that the target embedded application meets its timing constraints, and run the processor for that fixed setting [9].

Most processors developed for the mobile/portable market already support some form of built-in mechanism for voltage/frequency scaling. Intels SpeedStep technology, for example, detects if the system is currently plugged into a power outlet or running on a battery, and based on that, either runs the processor at the highest voltage/frequency or switches it to a less power hungry mode [9].

2.3.4 Battery Aware Power Saving

Chiasserini and Rao [66] have shown how battery behavior can be exploited to prolong battery life. In particular, they identify the phenomenon of charge recovery that takes place

under pulsed discharge conditions as a mechanism that can be exploited to enhance the capacity of an energy cell. The bursty nature of many data traffic sources suggests that there might be a natural fit between the two.

P. Rong and M. Pedram in [67] address the problem of maximizing the utilization of the battery capacity of the power source for a portable electronic system under a given performance constraint. They propose a new stochastic model of a power-managed battery-powered electronic system, which is based on Continuous-Time Markovian Decision Processes (CT-MDP). In this model, two important characteristics of todays rechargeable battery cells, i.e., the current rate-capacity characteristic and the relaxation-induced recovery are considered.

2.3.5 Compiler Level Power Saving

Compiler design techniques contribute to energy saving in several ways. Kolson et al. [68] address the problem of allocating memory to variables in embedded DSP (digital signal processing) software. The goal is to maximize simultaneous data transfers from different memory banks to registers. In several DSP applications mentioned in [69, 70], two registers are loaded with the required data and an arithmetic operation is performed. Loading two registers with a single double transfer instruction draws a little more current than a move instruction. Both instructions take one clock cycle each. However, energy is saved by employing the double transfer, because the double transfer instruction loads the two registers in one clock cycle, whereas it takes two clock cycles to sequentially load the registers. Instructions with memory operands have much higher energy cost than instructions with register operands [71]. This suggests that energy can be saved by suitably assigning the live variables of a program to registers. But, a processor has only a small number of registers. When the number of simultaneous live variables is larger than the number of available registers, some of the variables must be spilled to memory. Register assignment for loop variables is important because loops are typically executed many times. Algorithms for optimal register assignment to loop variables are presented in [72, 73]. These algorithms can be included in the code generation part of a compiler.

2.3.5.1 Influence of compiler optimizations on power and energy usage: Survey

Recently some attempts to understand the scope of compiler optimizations, from the perspective of power dissipation and energy consumption of programmable processors have been introduced. Tiwari et al. [74] presented an instruction level power model, following the same methodology published in [45], for a Fujistu 3.3v, 40MHz DSP. Moreover, the effect of two architectural features (dual-memory accesses, and packing of instructions into pairs) on the energy consumption has been exposed.

With the help of a cycle-accurate energy simulator (*SimplPower*), a source-to-source code translator, and a number of benchmark codes, Kandemir et al. [75, 76] studied the influence of five high-level compiler optimizations, such as loop unrolling and loop fusion, on energy consumption. Valluri et al. [77] provided an evaluation of some general and specific optimizations in terms of the power/energy consumption of the Alpha processor while running some SpecInt95 and SpecFp95 benchmarks. The processor in their work was simulated by means of Wattch (A frame work for analyzing processor power consumption at architectural-level) [39].

Chakrapani et al. [78] also presented a study into the effect of compiler optimization on the energy usage of an embedded processor. Their work targets an ARM embedded core and they use an RTL level model along with Synopsys Power Compiler to estimate power. Seng et al. [79] revised the effect of the Intel compiler general and specific optimizations, for energy and power consumption, for a Pentium 4 processor running some benchmarks extracted from Spec2000.
Azzemi et al. [80] examined the effect of loop unrolling factor, grafting depth and blocking factor on the energy and performance for the Philips Nexperia media processor "PNX1302". But, they interchangeably use the term energy and power for the same meaning. Hence the improvement in energy is directly related to the performance enhancement. Finally, Casas et al. [81] studied the effect of various compiler optimizations on the energy and power usage of the low power C55 DSP from Texas Instruments. Their work was based on a physical measurement platform for measuring the current drawn by the DSP core. The work does not consider the effect of the compiler optimizations on many performance measures that significantly affect the power and energy usage, such as the memory access and the

instructions per cycle.

2.4 Source to Source Code Transformations

The presence of mixed hardware/software architectures is becoming pervasive in the embedded systems arena, with a growing importance for the software section. Many proposals take into account the environment executing the code (CPU, Memory, Operating System, and so on) as well as the impact of code organization and compiler optimizations on the energy demand of the application as shown in Section 2.3.5.1. Memory optimization techniques focus on reducing the energy related to memory access, exploiting the presence of multilevel memory hierarchy, possibly in conjunction with suitable encodings to reduce the bus switching activity [82].

Other software-oriented proposals focus on instruction scheduling and code generation, possibly minimizing memory access cost [71]. As expected, standard low level compiler optimizations, such as loop unrolling or software pipelining, are also beneficial to energy reduction since they reduce the code execution time. However, there are a number of cross-related effects that cannot be so clearly identified and, in general, are hard to be applied by compilers, unless some suitable source-to-source restructuring of the code is a priori applied. In fact, the optimizations at compile time typically improve performance and occasionally the power consumption, with the main limitations of having a partial perspective of the algorithms and without the possibility of introducing significant modifications to the data structures. On the contrary, source code transformations can exploit full knowledge of the algorithm characteristics, with the capability of modifying both data structures and algorithm coding. Furthermore, inter-procedural optimizations can be envisioned. Another benefit of exploiting restructuring of the source code is related to portability, since the results are normally fairly general to deal with different compilers and architectures, without any intervention on existing compilers [83].

Brandolese et al. [83] stressed the state-of-the-art source to source transformations, to discover and compare their effectiveness from power and energy perspective. The data structure, loop and inter-procedural transformations were investigated with the aid of the GCC compiler. The compiled software codes were then simulated with a framework based on the

SimpleScaler [41]. The simulation framework was configured with a 1-kbyte 2-ways set-associative unified cache.

Ortiz et al. [84] investigated the impact of three different code transformations namely, loop unrolling, function inlining and variable types declaration on the power consumption. They choose three platforms as the target for their work, 8-bit and 16-bit micro-controllers and the 32-bit ARM7TDMI processor. Their results show that loop unrolling has a significant impact on the consumed power in case of the 16-bit and 32-bit processors.

Catthoor et al. [85] showed how source-to-source code transformations play a crucial role in the solution of the data-transfer and storage bottleneck in modern processor architectures. They survey many transformations that are mainely aiming to enhance the data locality and reuse.

Kulkarni et al. [86] were interested in improving the software controlled cache utilization, so as to achieve lower power requirements for multi-media and signal processing applications. Their methodology took into account many program parameters like the locality of data, size of data structures, access structures of large array variables, regularity of loop nests and the size and type of cache with the objective of improving the cache performance for lower power. The targeted platform for their research were the embedded multi-media and DSP processors. In the same way McKinley et al. [87] investigated the impact of loop transformations on the data locality.

Benini et al. [88] proposed three new schemes for code compression, based on the concepts of static (utilizing the static representation of the executable) and dynamic (utilizing program execution traces) entropy and compare them with a state-of-the-art compression scheme, IBMs CodePack [89]. Compression of executable code in embedded microprocessor systems, used in the past mainly to reduce the memory footprint of embedded software, is gaining interest for the potential reduction in memory bus traffic and power consumption. Their proposed schemes are competitive with CodePack for static footprint compression and achieved superior results for bus traffic and energy reduction. Another interesting outcome of their work was that static compression is not directly related to bus traffic reduction; yet there is a trade-off between static compression and dynamic compression, i.e., traffic reduction.

Yang et al. [90] studied the impact of loop optimizations in terms of performance and power trade-offs, with the aid of the Delaware Power-Aware Compilation Testbed (Del-PACT):an

integrated framework consisting of a modern industry-strength compiler infrastructure and a state-of-the-art micro-architecture -level power analysis platform. Both low-level loop optimizations at code generation (back-end) phase, such as loop unrolling and software pipelining, and high-level loop optimizations at program analysis and transformation phase (frontend), such as loop permutation and tiling, are studied.

2.5 Conclusions

In this chapter we review the evolution and state-of-the-art in processor's power consumption modeling and estimation methodologies that rely on the running software. In general two main abstraction levels are surveyed in this chapter. The low-level power modeling and estimation techniques cover the circuit-level, gate-level, Register Transfer (RT)-level and the micro-architecture level.

The high-level techniques can be divided into two categories, the Instruction Level Power Analysis (ILPA) and the Functional Level Power Analysis (FLPA). This survey leads us to the appropriate power estimation technique for VLIW processors.

Second, we go through different abstraction level software and hardware based power saving strategies, with a special focus on the recent attempts to evaluate the impact of different C/C++ compiler optimizations on the power consumption and the energy usage of a programmable processor in embedded systems.

Finally, we outline the variety of existing research efforts that investigate the effect of applying source to source code transformations on the energy and power consumption.

3. PRECISE POWER CONSUMPTION MODEL

3.1 Introduction

The importance of power constraints during the design of embedded systems has continuously increased in the past years, due to technological trends toward high-level integration and increasing operating frequencies, combined with the growing demand of portable systems [91]. Because of the small size and the mobility requirement of portable systems, they are powered by batteries of low rating. In order to avoid frequent recharging and/or replacement of the batteries, there is significant interest in low-energy system design.

In recent years, reducing power dissipation and energy consumption of a program have become optimization goals in their own right, no longer considered a side-effect of traditional performance optimizations which mainly try to reduce program execution time. Power and energy optimizations can be implemented in hardware through circuit design, and by the compiler through compile-time analysis, code reshaping, and supplying information to the operating system [92].

Due to the processing regularity of multimedia and DSP applications, statically scheduled processors such as VLIW processors are a viable option over dynamically scheduled processors, such as state-of-the-art superscalar GPPs. VLIW processors rely on software to identify parallelism and assemble wide instruction packets to issue multiple instructions per cycle. Though energy is actually consumed by the hardware, energy consumption can be reduced, apart from utilizing low-energy electronics, by suitably manipulating the software systems. This is because the hardware activities are controlled through the software.

Let a program X run for T seconds to achieve its goal, VCC be the supply voltage of the system, and I be the average current in Amperes drawn from the power source for T seconds. Consequently, T can be rewritten as $T = N \times \tau$ where N is the number of clock cycles and τ is the clock period. The power consumed by running X is given by: $P = VCC \times I$. Then, the amount of energy consumed by X to achieve its goal is given by: $E = P \times N \times \tau$ Joules. Since for a given hardware, both VCC and τ are fixed, $E \propto I \times N$. However, at the application level, it is more meaningful to talk about T than N, and therefore, energy is expressed as $E \propto I \times T$. This expression shows the main idea in the design of energy-efficient software that is to reduce both T and I. From the running time (average case) of an algorithm, a measure of T is achieved. However, to compute I, one must consider the average current drawn during the execution of the program.

The aim of this chapter is to develop a precise functional level power consumption model. The model will give us a deep insight view for the power characteristics of the targeted processor. This chapter is organized as follow: Section 3.2 addresses the setup for our experiments. In Section 3.3 a detailed description for the proposed model is given, including the different sub-models for the targeted processor functional units. Section 3.4 explains how the proposed model is validated even with signal and image processing benchmarks or real embedded system application. Finally, conclusions are drawn in Section 3.5

3.2 Experimental Setup

The targeted architecture is explored in detail in Appendix A.1. In our setup, the operating frequency ranges from 600MHz to 1200MHz and the DSP core voltage is 1.2V. All measurements are carried out on the TMS320C6416T DSP Starter Kit (DSK) manufactured by Spectrum Digital Inc. There are three power test points on this DSK for DSP I/O current, DSP core current and system current. The C/C++ compiler embedded in the Code Composer Studio (CCS3.1) from Texas Instruments is used for getting the binaries to be loaded to the DSP. The current drawn by the DSP core while running an algorithm is captured by the Agilent 34410A $6\frac{1}{2}$ digit Digital Multi-Meter(DMM). This DMM features very high DC basic accuracy, actually 0.003% of the reading plus 0.003% of the range [93]. As shown in Fig. 3.1 the current is captured in terms of the differential voltage drop across a 0.025Ω

sense resistor inserted, by the DSK manufacturer, in the current path of the DSP core. The input differential voltage drop is divided by 0.025Ω to obtain the current drawn. Several assembly language scenarios have been developed to separately stimulate each of the functional blocks. All the scenarios consist of unbounded loops with a body of more than $1\,000$ instructions, to avoid the effect of branching instructions on the measured current.

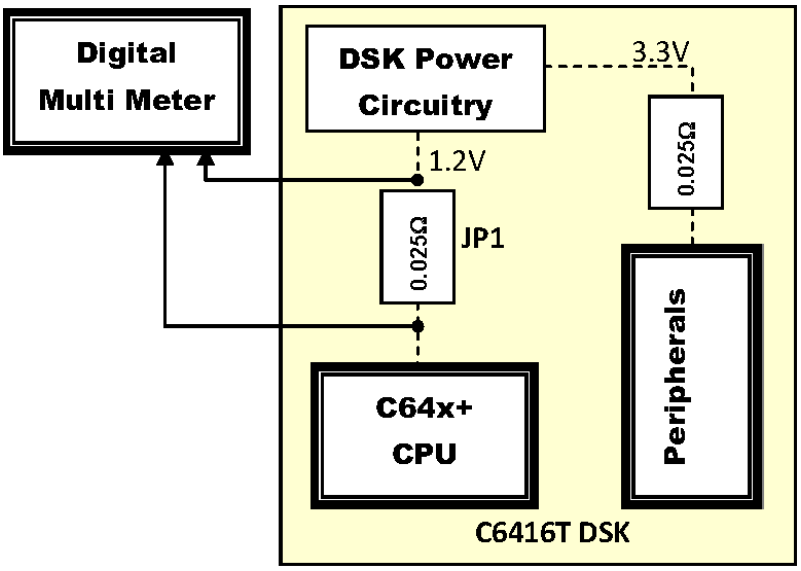

Fig. 3.1: Current Measurement Setup.

3.3 Methodology

The basic idea behind the FLPA is the distinction of the processor architecture into functional blocks like Processing Unit (PU), Instruction Management Unit (IMU), internal memory and others [55]. At first, a functional analysis of these blocks is performed to specify and then discard the non-consuming blocks (those with negligible impact on the power consumption). The second step is to figure out the parameters that affect the power consumption of each of the power consuming blocks. For instance, the IMU is affected by the instructions dispatching rate which in turn is related to the parallelism degree. In addition to these parameters, there are some parameters that affect the power consumption of all functional blocks in the same manner such as operating frequency and word length of input data [56].

By means of simulations or measurements it is possible to find an arithmetic function for each block that determines its power consumption depending on a set of parameters. Hence, to determine the arithmetic function for each functional block, the average supply current

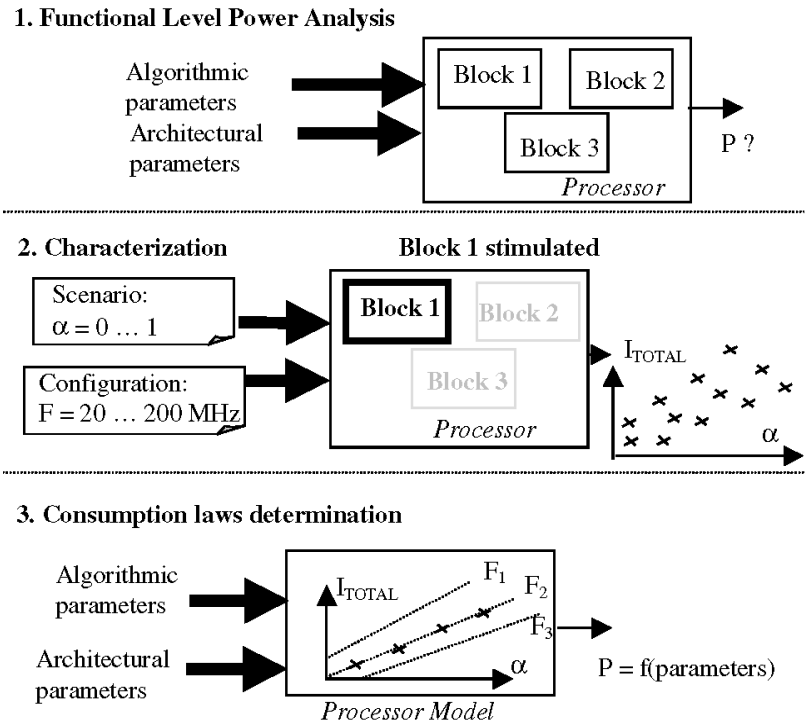

Fig. 3.2: Function level power modeling steps.

of the processor core is measured in relation with the variation of the affecting parameter. These variations are achieved by a set of small programs, called scenarios. Such scenarios are short programs written in assembly language and consisting of unbounded loops with a body of several hundreds of certain instructions that individually invoke each block. The power consumption rules are finally obtained by curve-fitting the measurement values [56].

The parameters that affect the power consumption for each functional block can be extracted from the assembly code generated by the Integrated Development Environment (IDE). Some parameters cannot be extracted directly from the assembly code, such as the execution time and the data cache miss rate. Therefore, the code should be run at least once to obtain these parameters with the aid of the profiler.

After applying the FLPA, the C6416T architecture is subdivided into six distinct functional blocks (clock tree, instruction management unit, processing unit, internal memory, L1 data cache and L1 program cache) as shown in Fig. 3.3. Although the L2 unified memory accesses are considered through the treatment of the L1 data and program cache misses, the L2 cache misses are not handled in our model as the L2 memory size (1-Mbyte) is almost enough for most of the signal processing applications. The parameters that affect the power consumption for the determined functional blocks are also shown in Fig. 3.3. The C6416T

fetches instructions from memory in fixed bundles of eight instructions, known as fetch packets. The instructions are decoded and separated into bundles of parallel-issue instructions known as execute packets.

The dispatching rate α represents the average number of execution packets per fetch packet. The processing rate β stands for the average number of active processing units per cycle. The internal memory read/write access rates ε/λ respectively express the number of memory accesses divided by the number of required clock cycles for executing the code segment under investigation. The data cache miss rate γ corresponds to the number of data cache misses divided by the total memory accesses. Finally, the program cache miss rate δ corresponds to the number of program cache misses divided by the total program cache references. The methodology of computing these algorithmic parameters are presented in Appendix B.2.

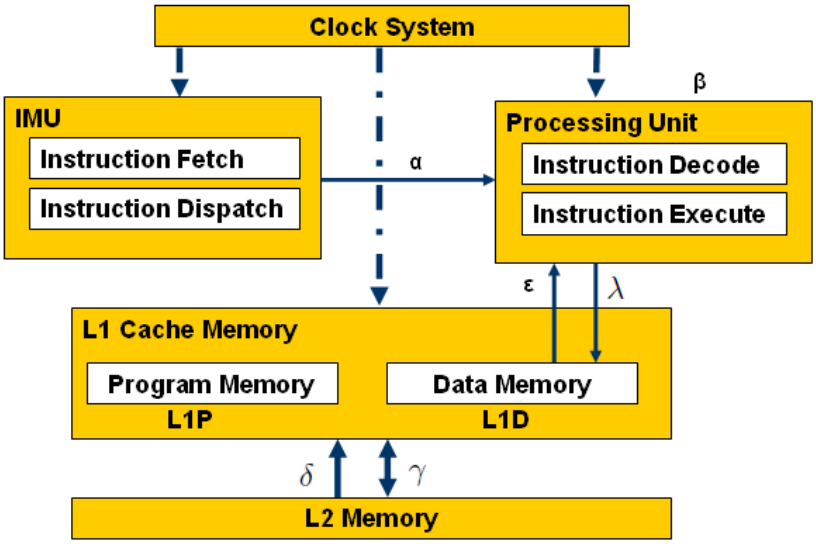

Fig. 3.3: Functional level power analysis for C6416T.

3.3.1 Static and Clock Distribution Power Consumption Sub-Model

The static power consumption of any processor includes the power consumed due to leakage current and the clock distribution network. It is not possible at the functional level analysis to differentiate between those types of power consumption. Hence, Both static and clock distribution power consumption are considered in a single sub-model as the static and clock distribution power consumption model. From now on when we talk about the operating frequency effect on the power consumption, actually the effect of static and clock distribution is meant. First, the effect of the operating frequency on the power consumption is

determined. The operating frequency linearly affects the current drawn by the DSP core and hence, also linearly affects the power consumption of the processor. Figure 3.4 shows the relation between the operating frequency and the current drawn by the DSP core.

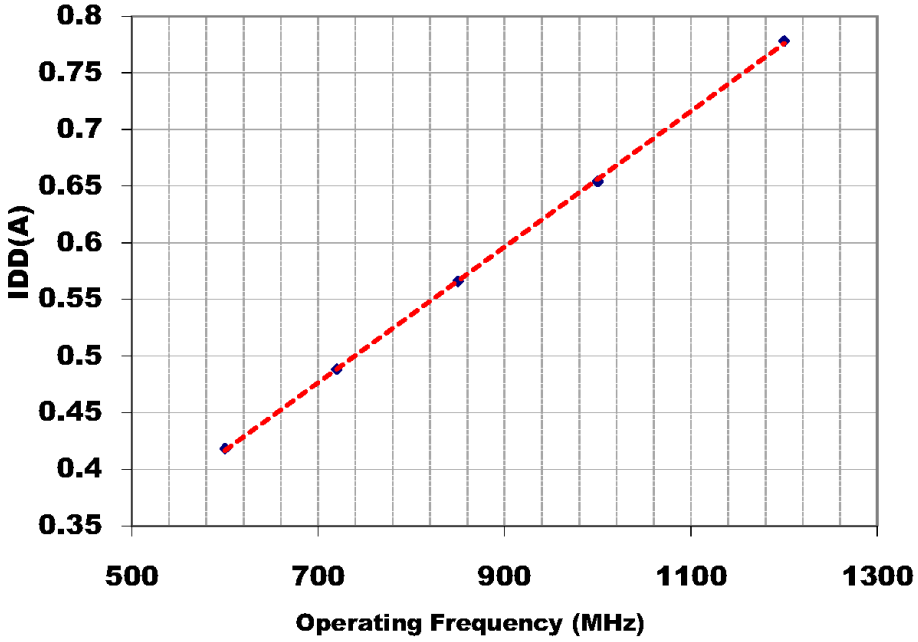

Fig. 3.4: Model function of the C6416T clock tree.

3.3.2 IMU Power Consumption Sub-Model

The IMU unit of the C6416T processor consists of two main sub-units which are the instructions fetching unit and the dispatching unit. The IMU fetches eight instructions per cycle as one fetch packet. The dispatch unit then subdivides this fetch packet into execution packets. Since the C6416T has eight functional units, it is capable of simultaneously executing up to eight instructions. Consequently, the dispatch unit can divide the fetch packet into one (maximum parallelism) to eight (sequential) execution packets. Therefore, it is obvious that the dispatch rate is the parameter that affects the power consumption of the IMU.

Since the NOP instruction does not require any processing unit for its execution, the proposed scenarios to invoke the IMU are composed of an unbounded loop with more than 1 000 No Operations (NOPs). These scenarios vary the dispatch rate (number of fetch packets divided by the number of execution packets) from 0.125 to 1.0. Figure 3.5 shows screen shots of the scenarios to vary the dispatch rate. Figure 3.6 indicates the characteristics of the current drawn by the core processor with a varying dispatch rate when the operating

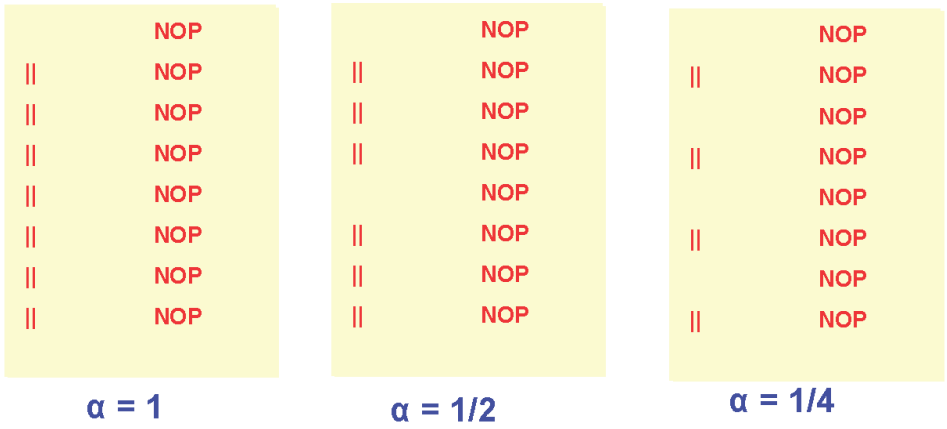

Fig. 3.5: Screen shots of the scenarios for varying α.

frequency is adjusted to $1\,000$MHz. Figure 3.7 indicates that varying α is independent of varying the operating frequency.

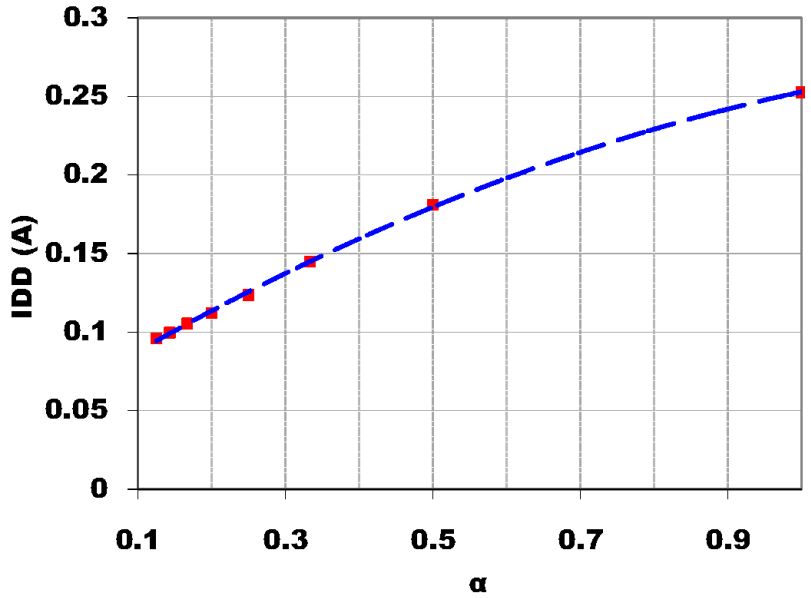

Fig. 3.6: Model function of the C6416T IMU at $F = 1\,000$MHz.

By curve fitting the measurement values in Fig. 3.6 the arithmetical function in (3.1) is obtained.

$$\mathrm{IDD}_{\mathrm{IMU}} = -0.0918\alpha^2 + 0.284\alpha + 0.0603. \tag{3.1}$$

The quality of the fitting process is measured by the value R-squared (R^2): A number from 0 to 1, which is the normalized square of the residuals of the data after the fit. This value expresses what fraction of the variance of the data is explained by the fitted trend line. It reveals how closely the estimated values for the trend line correspond to the actual data. A trend line is most reliable when its R^2 value is at or close to 1.0 [94]. Since the R^2 value

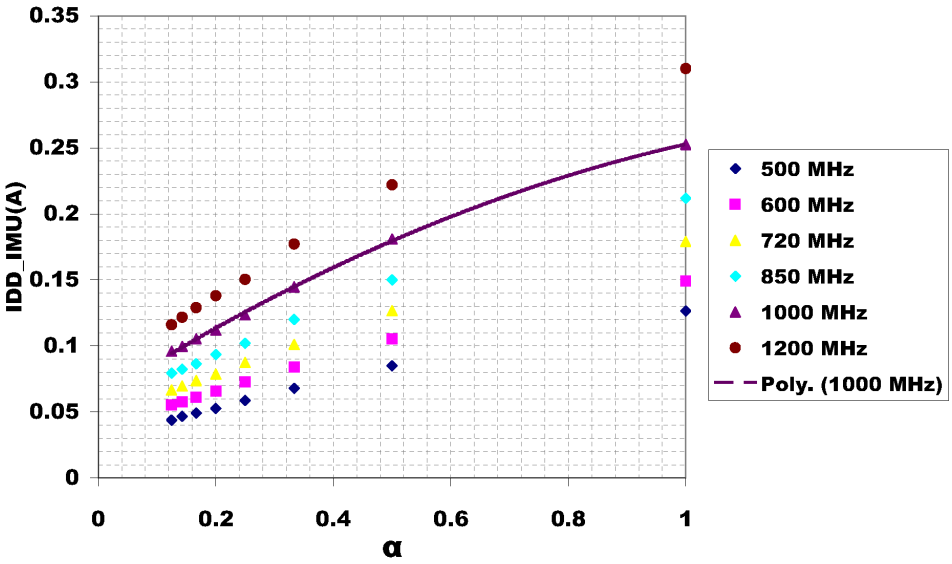

Fig. 3.7: Model function of the C6416T IMU at different frequencies.

for the arithmetic function in (3.1) equals 0.9994 then (3.1) is an excellent fit for the curve values in Fig. 3.6.

The arithmetic function in (3.1) does not consider the effect of pipeline stalls. Many reasons cause the pipeline to stall. For instance, one data cache miss stalls the pipeline for at least six cycles. Hence, the arithmetic function in (3.2) is presented to account for the pipeline stall effect.

$$\mathrm{IDD}_{\mathrm{IMU}} = (-0.0918\alpha^2 + 0.284\alpha + 0.0603)(1 - \mathrm{PSR}), \tag{3.2}$$

where PSR stands for Pipeline Stall Rate which can be expressed as the number of pipeline stall cycles divided by the total cycles required for executing the code segment under investigation.

3.3.3 PU Power Consumption Sub-Model

The data path of the C6416T consists of eight functional units. These functional units can work simultaneously, if the dispatch unit succeeds to compose an execution packet with eight instructions. Unlike the model in [57] that uses the parallelism degree as the affecting parameter for the processing unit model, the fact that the NOP does not require any PU for its execution convinced us that another parameter yields a better description of the PUs.

The new parameter is the processing unit rate which expresses the average number of active processing units per cycle. Figure 3.8 illustrates the difference between the dispatch rate

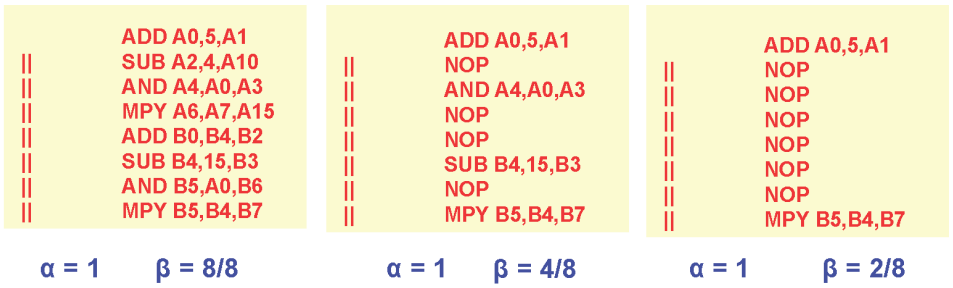

Fig. 3.8: Difference between β and α.

and the processing unit rate. Another important parameter that affects the processing unit power consumption is the word length of the data operands. In the C6416T the word length varies from 8-bits to 32-bits. Thus, in our model 16-bits word length has been chosen to be the typical word length.

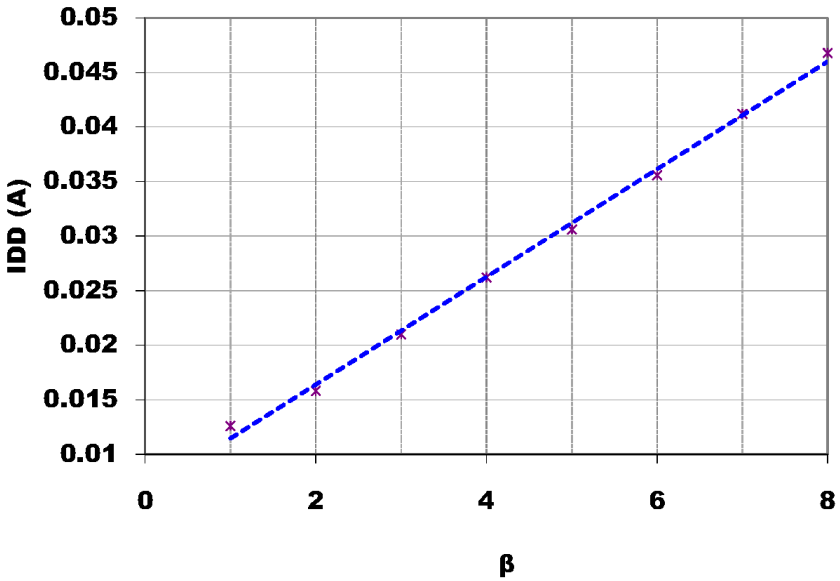

Fig. 3.9: Model function of the C6416T Processing Units at $\alpha = 1$ and $F = 1\,000$MHz.

More than $1\,000$ different instructions compose the scenarios that vary the processing unit rate, that is to account for the inter-instructions effect. The current measured from the DSK is the sum of the clock tree, IMU, and the PU currents. To attain only the current drawn by the PU, the IMU and clock tree currents are subtracted from the measured current.

Figure 3.9 depicts the effect of varying the number of active PU per cycle on the current drawn by the core processor.

$$\mathrm{IDD_{PU}} = (-0.0049\beta + 0.0065)(1 - \mathrm{PSR}). \tag{3.3}$$

The arithmetic function in (3.3) results in an excellent fit for the curve values in Fig. 3.9 with

R^2 value of 0.9982. Compared to other functional units such as clock tree or the IMU, it is clear that the PU does not significantly contribute to the total power consumption of the core processor. It is important to mention that the scenario for invoking the PU does not include any memory instructions. The internal memory operations are handled in a separate scenario.

3.3.4 Internal Memory Power Consumption Sub-Model

As mentioned in the previous Section 3.3.3 the internal memory operations are separately handled. That is because of its distinct execution characteristics. Two categories of memory operations are included in the instruction set of the C6416T DSP load and store. The load instructions represent the read of data from the data cache (if the operand exist in the data cache) to a specific register from the processor's register file. The store instructions represent the write of data into the memory, according to the data cache write policy.

```
   LDHU   .D1  *A25++,A26
|| LDHU   .D2  *B25++,B26
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
```

α = 1/4 ε = 20%

```
   LDHU   .D1  *A25++,A26
|| LDHU   .D2  *B25++,B26
   LDHU   .D1  *A25++,A27
|| LDHU   .D2  *B25++,B27
   LDHU   .D1  *A25++,A28
|| LDHU   .D2  *B25++,B28
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
   NOP
|| NOP
```

α = 1/4 ε = 60%

```
   LDHU   .D1  *A25++,A26
|| LDHU   .D2  *B25++,B26
   LDHU   .D1  *A25++,A27
|| LDHU   .D2  *B25++,B27
   LDHU   .D1  *A25++,A28
|| LDHU   .D2  *B25++,B28
   LDHU   .D1  *A25++,A29
|| LDHU   .D2  *B25++,B29
   LDHU   .D1  *A25++,A30
|| LDHU   .D2  *B25++,B30
   LDHU   .D1  *A25++,A31
|| LDHU   .D2  *B25++,B31
   LDHU   .D1  *A25++,A26
|| LDHU   .D2  *B25++,B26
   LDHU   .D1  *A25++,A27
|| LDHU   .D2  *B25++,B27
   LDHU   .D1  *A25++,A28
|| LDHU   .D2  *B25++,B28
   NOP
|| NOP
```

α = 1/4 ε = 180%

Fig. 3.10: Snapshots of different scenarios for varying ε.

The C64x+ architecture is capable of performing two memory operations per cycle. The affecting parameter for the internal memory sub-model are the memory read access rate ε and the memory write access rate λ. The memory access rate is defined as the number of memory references (read and write) divided by the algorithm execution time.

Figure 3.10 illustrates snapshots of different scenarios to vary the memory read access rate ε from 20% to 180% (as two memory operations can be simultaneously executed). All of those scenarios conducted with the same $\alpha = \frac{1}{4}$. Figure 3.11 shows the measured current

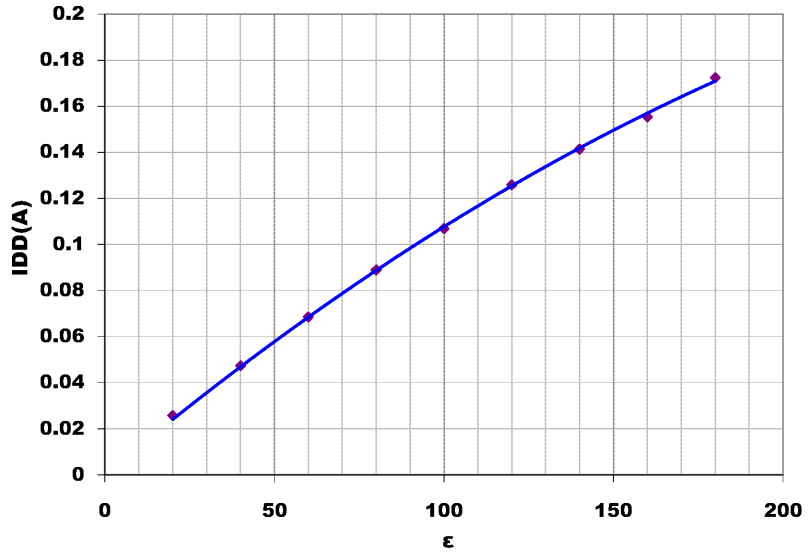

Fig. 3.11: Model function of the C6416T internal memory read at $\alpha = 1$ and $F = 1\,000$MHz.

values for different ε values.

$$\text{IDD}_{\text{Internal\_Memory\_Read}} = (-2 \cdot 10^{-6}\varepsilon^2 + 0.0012\varepsilon)(1 - \text{PSR}). \tag{3.4}$$

The arithmetic function in (3.4) results in an excellent fit for the curve values in Fig. 3.11 with R^2 value of 0.9995. When we tried to fit the values of the curve in Fig. 3.11 with a linear arithmetic function, we hit upon that the resultant R^2 value equals 0.98, equivalent to an error of 2%. As the final model will be the summation of the different functional block sub-models then the final model estimation error will be an accumulation of the sub-models errors. Therefore, we decide to minimize the curve fitting error as much as possible. Hence, we choose the arithmetic function in (3.4) to represent the internal memory read sub-model.

In the same manner (3.5) represents the current drawn from the CPU while running different scenarios that vary the memory write access rate λ. This equation results in a R^2 value of 0.9978.

$$\text{IDD}_{\text{Internal\_Memory\_Write}} = (-10^{-5}\lambda^2 + 0.0049\lambda)(1 - \text{PSR}). \tag{3.5}$$

Hence, Equation (3.6) represent the total internal memory model.

$$\text{IDD}_{\text{Internal\_Memory}} = (-2 \cdot 10^{-6}\varepsilon^2 + 0.0012\varepsilon - 10^{-5}\lambda^2 + 0.0049\lambda)(1 - \text{PSR}). \tag{3.6}$$

3.3.5 L1 Data Cache Power Consumption Sub-Model

The L1 data cache functional block represents the flow of data from the L1 data cache to L2 memory and vice versa. Different scenarios are prepared to stimulate the effect of the data cache miss.

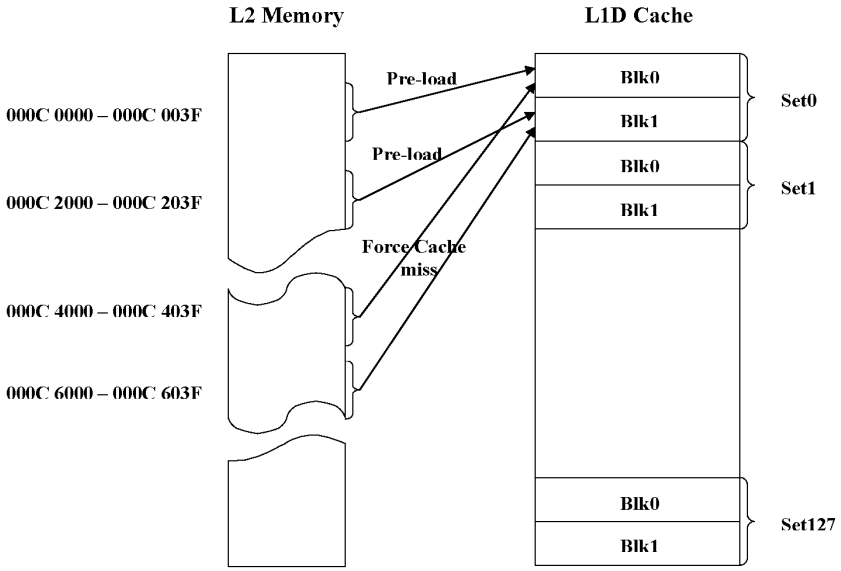

Fig. 3.12: Scenario for forcing a data cache miss.

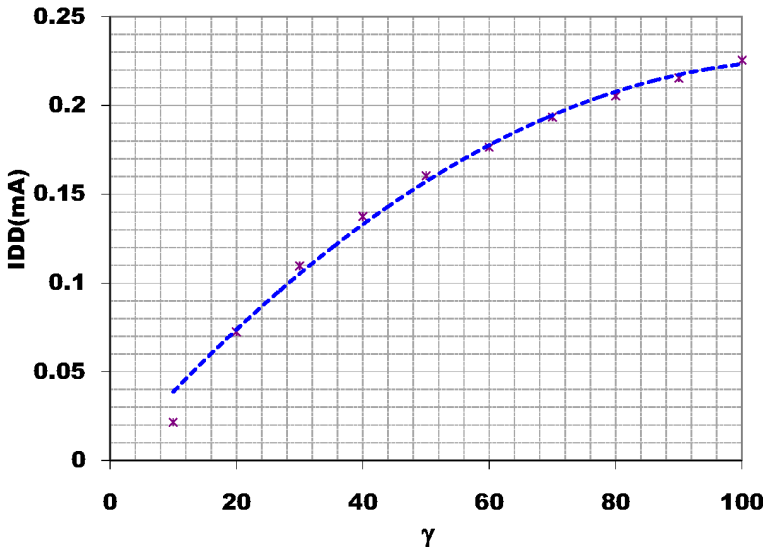

Fig. 3.13: L1D cache miss rate vs. measured CPU current.

The data cache miss rate is used as the affecting parameter for the L1 data cache functional block. Taking into account the fact that the L1D cache is a two-way associative cache, different scenarios that vary the number of data cache misses per fixed number of memory accesses have been developed. In this scenario, a deterministic way for forcing the data cache misses is followed. First, arbitrary data are pre-loaded into both blocks of set 0. Second, data

from L2 memory with addresses that must be mapped into set 0 blocks are loaded to L1D cache. The new data, from L2 memory, addresses are different from those already preloaded to set 0. Hence, a data cache miss occurs as illustrated in 3.12.

Figure 3.13 shows the effect of varying the data cache miss rate on the current drawn by the core processor. The arithmetic function in (3.7) results in an excellent fit for the curve values in Fig. 3.13 with an R^2 value of 0.9909. Although the quadratic term in (3.7) is very small compared to the linear term, it has great impact on the R^2 value. Discarding the quadratic term in (3.7) results in a R^2 value of 0.9272 with an error of 7.28% thus, (3.7) is a very suitable function to represent the L1D cache misses power consumption.

$$\text{IDD}_{\text{L1D}} = (-2 \cdot 10^{-5}\gamma^2 + 0.0041\gamma)(1 - \text{PSR}). \tag{3.7}$$

The arithmetic function in (3.7) differs from the corresponding linear function that was proposed in [57] for the cache functional block. The squared-function yields a better description for the L1 data cache block due to the fact that L1 data cache pipelines the cache misses, to decrease the resulting pipeline stalls. The proposed model in [56] did not separately investigate the effect of data cache misses; instead it is included in the processing unit functional block [95].

3.3.6 L1 Program Cache Power Consumption Sub-Model

With the aid of the profiler of the C6416T device accurate cycle simulator, different scenarios are prepared that arbitrarily vary the program cache miss rate δ. Figure 3.14 shows the effect of varying the program cache miss rate on the current drawn by the core processor. The best arithmetic function that fits the measured values in Fig. 3.14 is obtained as indicated in (3.8) with an R^2 value of 0.9889.

$$\text{IDD}_{\text{L1P}} = (0.0011\delta)(1 - \text{PSR}). \tag{3.8}$$

Table 3.1 shows how the algorithmic parameters of the proposed model are computed with the aid of the C6416T profiler. The C6416T profiler, which is embedded in the CCS3.1, offers many statistics regarding the program under investigation that are utilized in the process of computing our proposed model such as number of execution packets, number of NOP instruction cycles, number of L1D cache misses and so on.

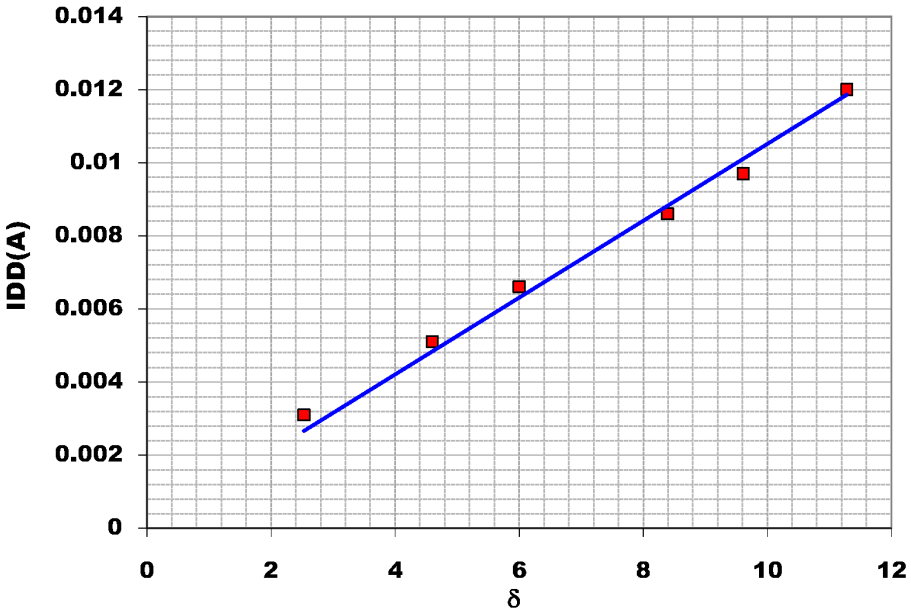

Fig. 3.14: L1P cache miss rate vs measured CPU current.

Tab. 3.1: Algorithmic parameters calculation methodology

Parameter	Computation Methodology
α	No. of fetch packets / No. of execution packets
β	(No. of executed instructions - NOP instructions) / Total code cycles
ε	(No. of L1D read hits / Total code cycles) * 100
λ	(No. of L1D write hits / Total code cycles) * 100
γ	((No. of L1D read misses + No. of L1D write misses) / No. of L1D references) * 100
δ	(No. of L1P misses / No. of L1P references) * 100
PSR	No. of CPU stall cycles / Total code cycles

The complete FLPA power consumption model for the C6416T fixed-point high performance VLIW DSP is shown in Table 3.2. A detailed description of how the parameters in the proposed model are computed as well the complete model with exact constant values at an operating frequency of 1 000MHz are illustrated in Appendix B.

Tab. 3.2: Complete power consumption model for C6416T DSP.

Functional unit	Functional unit power consumption sub-model
Clock Distribution	$\mathrm{P_F} = (\mathrm{a1} \cdot F + \mathrm{a2}) \cdot \mathrm{V_{core}}$
IMU	$\mathrm{P_{IMU}} = (\mathrm{b1} \cdot \alpha^2 + \mathrm{b2} \cdot \alpha + \mathrm{b3})(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
Processing Units	$\mathrm{P_{PU}} = (\mathrm{c1} \cdot \beta + \mathrm{c2})(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
Memory Read	$\mathrm{P_{MemR}} = (\mathrm{d1} \cdot \varepsilon^2 + \mathrm{d2} \cdot \varepsilon)(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
Memory Write	$\mathrm{P_{MemW}} = (\mathrm{e1} \cdot \lambda^2 + \mathrm{e2} \cdot \lambda)(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
L1D Cache	$\mathrm{P_{L1D}} = (\mathrm{g1} \cdot \gamma^2 + \mathrm{g2} \cdot \gamma)(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
L1P Cache	$\mathrm{P_{L1P}} = (\mathrm{h1} \cdot \delta)(1 - \mathrm{PSR}) \cdot F \cdot \mathrm{V_{core}}$
Total Power	$\mathrm{P_T} = \mathrm{P_F} + \mathrm{P_{IMU}} + \mathrm{P_{PU}} + \mathrm{P_{MemR}} + \mathrm{P_{MemW}} + \mathrm{P_{L1D}} + \mathrm{P_{L1P}}$

3.4 Model Validation

3.4.1 Validation with Benchmarks

Some common signal and image processing benchmarks from Texas Instruments libraries are used for demonstration purpose as described in Table 3.3. The input data for all used benchmarks are located in the internal data memory. All the benchmarks are executed in an infinite loop to get a stable reading on the DMM.

First of all, all optimization options which are included in the CCS3.1 are turned off because these optimization options affect the speed or the code size only and are not dedicated to power optimization. The second step is to compile the benchmarks.

The required parameters for the model are calculated either statically from the generated assembly files or with the aid of the CCS3.1 profiler for the parameters that cannot be estimated statically such as the data cache miss rate. For instance, the processing unit rate which is defined as the average number of active processing units per cycle is calculated from the assembly code. The parameter β is the result of dividing the number of processing units (equals the number of instructions excluding the NOP) by the number of cycles per code iteration. Figure 3.15 presents the result of the estimated power consumption versus the measured one for the benchmarks listed in Table 3.3.

Tab. 3.3: Benchmarks used for our experiments.

benchmark	Description
DotP128	Dot product of a vector of 128 16-bit elements
m100	Matrix multiplication for 2 100x100 square matrices
FIR	Computes a real FIR filter, Input data and filter taps are 16-bit
Sobel3x3	Apply Sobel filter of 3x3 window to an image of 8192 pixels
Thresholding	Performs a thresholding operation on an input image of 8192 pixels
Histogram	Takes histogram of an image of 8192, 8-bit pixels
IIR	Performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients
FFT16x16	Performs a mixed radix forwards FFT using a special sequence of coefficients
Correlation3x3	Performs a point by point multiplication of the 3x3 mask with an input image

The absolute average estimation error is 1.65% while the worst is 3.3%. Appendix B.3 illustrates the actual values for the different algorithmic parameters to estimate the power consumption for the different benchmarks. It also presents the estimated and measured power consumption with estimation error for each benchmark.

The results obtained from the previous modeling process are analyzed to figure out the functional unit that is dominantly contributing to the power consumption. Figure 3.16 illustrates the contribution percentages of the different functional blocks of the processor to the power consumption. It is clear that the clock distribution is the largest contributor while the processing unit is the smallest contributor. The clock distribution contribution percentage to the total power consumption is expected to decrease when estimating the power consumption of much more complex algorithms or compiling these algorithms with much aggressive optimization options. This for sure increases the opportunity for more power oriented optimization efforts. However, this processor is not the best choice for battery operated handheld

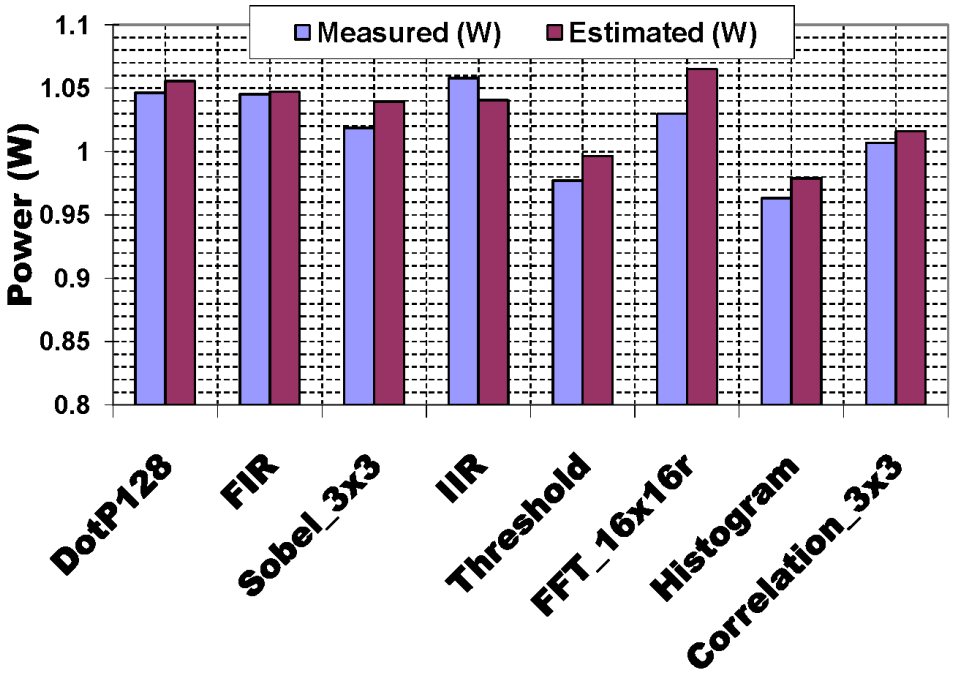

Fig. 3.15: Estimated vs. measured power consumption of the C6416T at $F = 1\,000$MHz.

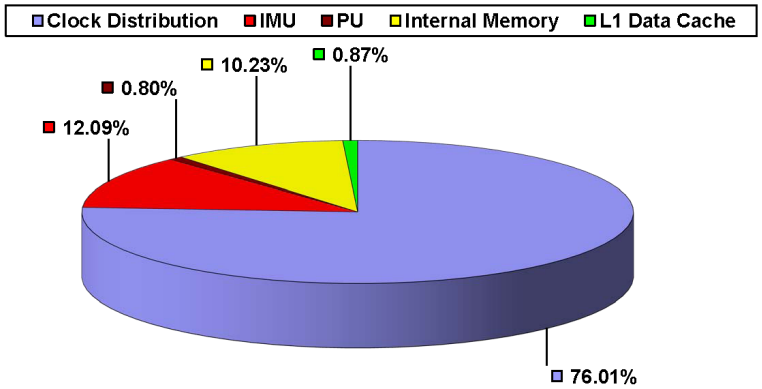

Fig. 3.16: Average functional units contribution to the processor power consumption.

devices.

3.4.2 Validation with a Real Application

The application of complex video processing algorithms with real time constraints requires optimal transformation from the algorithm to the target architecture. For example automatic recognition of bad weeds and their automatic eradication, automatic positioning of transport systems, video quality control of weld seams, and video based 3D scene generation for autonomous in-door vehicles. All those applications have complex video processing in a real time environment in common. Usually, those video processing algorithms are developed in a high level language description like MATLAB or C++ in order to verify their functionality

while performance and power issues are neglected. After this step, the algorithms have to be optimally transformed to a target platform e.g. a PC. Whenever the PC platform does not allow for the requested performance, which is the case in any of the aforementioned projects, the applications have to be thoroughly analyzed in order to overcome the performance gap. Viable choices are typically code transformations to mathematically equivalent expressions that represent a better match to the target architecture, or the migration to a different platform, e.g. DSP boards or DSP-FPGA systems, or even the composition of an optimally tailored target architecture including instruction set extensions and dedicated processing elements (ASIC) for distinct parts of the design. The bad weeds recognition algorithm, serving as a realistic code segment, is shortly surveyed and how it has been mapped onto the targeted DSP board.

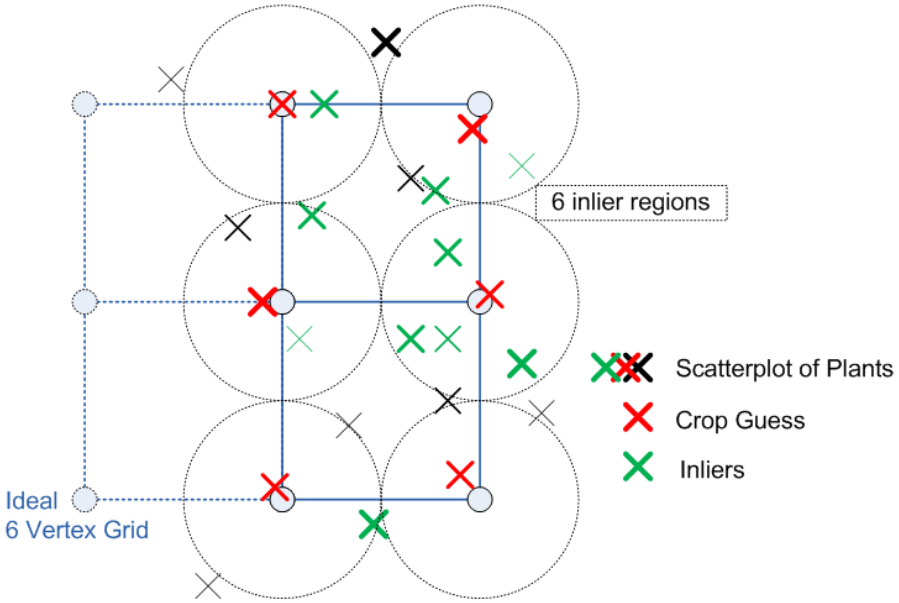

Fig. 3.17: Illustration of the plants scatter-plot.

The bad weeds recognition algorithm is concerned with the automatic recognition and eradication of bad weeds for the agricultural domain. An imaging device scans the ground and transmits data identifying the location and intensity of green spots, i.e. representing either the agricultural crop or bad weeds. Since the crops have been planted on a regular six vertices grid as depicted in Fig. 3.17, the algorithm tries to match the ideal six vertex grid onto the given scatter-plot of candidate spots. The best match found is then supposed to identify the agricultural crop, whereas all other points of the scatter-plot are likely to be bad weeds and are hence eradicated.

The example application is a restricted set exhaustive search algorithm named Elastic Graph Model. Elastic Graph Model is a method for detecting nearly regular located objects in images. It proceeds as follows: any local maximum of the scatter-plot is supposed to represent the left upper node of the ideal six vertex grid. For any of the remaining five ideal grid nodes a subset of local maxima is determined from the remaining local maxima in the scatter-plot. Hence, we obtain for any grid node a set of candidates, the so called inliers, which lie within a certain region around the ideal grid node. In Fig. 3.17 these inliers are highlighted in green. Thus, the number of hypotheses for the elastic graph is any possible combination of inliers for any node. Any of these hypotheses is evaluated by two measures: the external energy, which represents the weight of the six local maxima of the respective hypothesis indicated in Fig. 3.17 by thicker or thinner crosses, and the internal energy, which is a measure that accumulates the squared error of the edge length between two hypothetical nodes (for all seven edges) and the ideal edge length (which is one in the usual case), also weighted by an edge weight (typically also one for all edges).

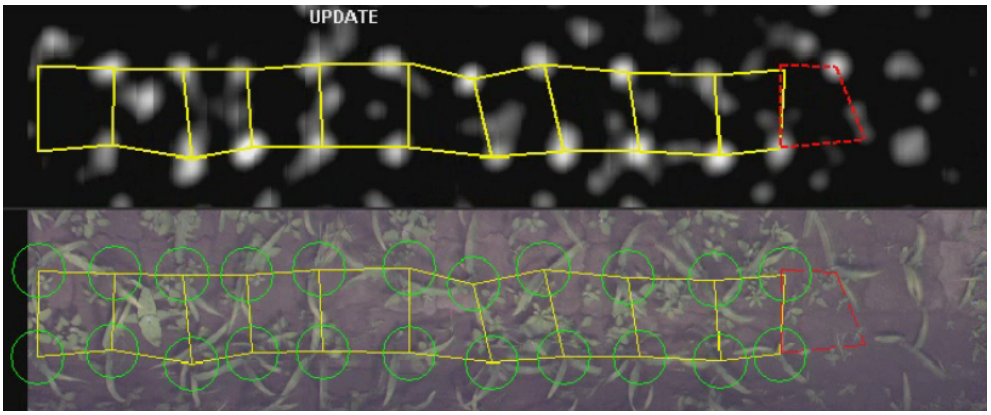

Fig. 3.18: Elastic graph matching algorithm.

This restricted set exhaustive search algorithm is very time-consuming with an exponentially growing algorithmic complexity in the order of the cardinality of the set of scatter-plot vertices and/or the size of the inlier regions as noticed from Fig. 3.18. The task shall be analyzed towards optimization possibilities with respect to the computing platform. Therefore, the algorithm has been migrated to the DSP platform.

This algorithm has been provided by Austrian Research Center (ARC), as a part of the K-Project Embedded Computer Vision (ECV), as C++ program projected within MS Visual Studio .NET. Therefore, the code had to be migrated to the targeted platform, i.e. the DSP C6416T. The following steps summarize the actions taken to perform the migration to the

TI processor:

- The code had been revised and analyzed regarding its basic functionality including debugging and structuring to match our coding requirements.
  - The code included a large portion of dynamic memory allocation that did not match the memory requirements of the DSP platform.
  - Co-operating with the Austrian Research Center (ARC) a modification took place to obtain functionally equivalent code with static memory allocation to ensure portability.
- A suitable linker command file (.cmd) has been prepared for the project.
- The stack and heap memory maps have been adjusted to satisfy the memory requirements of the algorithm.
- The functionality has been tested against the original unmodified code to ensure equivalence.

The power consumption of the Elastic Graph Matching (EGM) is estimated with the aid of our proposed power consumption model described in 3.2. The estimated power consumption equals 1.0498W while the physically measured power consumption equals 1.061W, resulting in an estimation error of 1%. First, we check the impact of increasing the number of allowed inliers. Table 3.4 indicates that the increase in the number of inliers exponentially increases the execution cycles. It is very important to carefully determine the inlier distance.

Tab. 3.4: Impact of increasing number of inliers.

# Local Maxima	Exec. Time (ms)	Code Size(bytes)	Energy (mJ)
6	5.98	3288	6.36
24	90.56	4124	95.85
42	274.01	4972	291.61
60	556.64	5816	590.76

Table 3.5 shows a part from the profiling data for the scatterplot of 60 local maxima from which the number of hypotheses is computed. Each row of Table 3.5 summarizes the state of a certain piece of the code. The row begins by the starting and ending lines of code that

delineate the piece reported. The second column of the table indicates the type of that piece (either function or loop) while the third column gives the number of times this piece is run during the program's execution. Finally, the last row gives the total number of clock cycles spent on that piece of code.

Tab. 3.5: Profiling data for the code with 60 LocalMaxima.

	Symbol Type	Access Count	cycle.total.incl
19-250:main.cpp	Function:main	1	556 637 078
29-35:main.cpp	Loop	15	82
71-238:main.cpp	Loop	60	556 612 470
74-237:main.cpp	Loop	360	556 609 942
90-136:main.cpp	Loop	2 160	554 044 923
93-114:main.cpp	Loop	129 600	547 007 140
148-151:main.cpp	Loop	2 160	12 960
165-232:main.cpp	Loop	360	9 557 712
178-185:main.cpp	Loop	1 440	10 080
190-193:main.cpp	Loop	2 160	129 240
200-212:main.cpp	Loop	2 520	9 369 360

The main function starts at line 19 and ends at 250 (first row of the table). Within the main function there is a loop that starts at line 29 and ends at 35 (second row of the table). There is second loop that starts at line 71 and ends at 238 (third row of the table). This second loop contains another nested loop that starts at line 74 and ends at 237. This last loop contains several other nested loops (remaining rows of the table). The time taken within any loop is the sum of the time taken in its nested loops as well as the remaining instructions that are outside of those nested loops.

The performance monitoring event in this profiling is the cycle.total.incl which indicates that the cycles count for the outer loops include the cycle count for the most inner loops. The profiling results indicate that the most time-consuming part of the code lies in the loop calculating the Euclidean distances. There are two positions in the code that compute the Euclidean distance:

1. When determining the inliers for each node and this loop (93-114:main.cpp most inner loop) consumes 547 007 140 cycle.

2. When evaluating the available hypotheses and this loop (200-212:main.cpp loop) consumes 9 369 360 cycle.

The total number of execution cycles for the whole algorithm code is $556\,637\,078$ cycles. This means that the loop determining the inliers for each node consumes 98% of the execution cycles for the whole algorithm. Hence, more optimization effort should be paid for this code area.

3.5 Conclusions

A precise functional-level model for estimating the power consumption of the commercial off-the-shelf VLIW processor C6416T has been developed. The processor architecture has been divided into several functional blocks specifically, clock tree, instruction management unit, processing unit, internal memory, L1 data cache and L1 program cache.

The parameters that affect the power consumption of each functional block have been determined. Those parameters are divided into two categories: architecture and algorithmic parameters. The architecture parameters are those parameters that affect the power consumption of all the functional blocks such as the operating frequency and the word length. The algorithmic parameters have been computed from the generated assembly code of the IDE. The inter-instructions as well as the pipeline stall effects have been investigated in our proposed model.

We prove the validation and precision of our model on many typical algorithms applied in signal and image processing as well as a real embedded application. The power consumption estimated by our model, compared to the physically measured power consumption, is achieving a very low absolute average estimation error of 1.65% and an absolute maximum estimation error of only 3.3%.

4. COMPILER OPTIMIZATION INFLUENCE ON THE ENERGY AND POWER CONSUMPTION

4.1 Introduction

Given a particular architecture, the programs that run on it will have a significant effect on the energy usage of the processor. The manner in which a program exercises particular parts of the processor will vary the contribution of individual structures to the total energy consumption [79]. For example, if the execution of a particular program generates a significant number of data cache misses, the energy used by the second level cache will increase, as there will be more access to the secondary cache as we present in the previous chapter.
Compilers traditionally are not exposed to the energy details of the processor. Current compiler optimizations are tuned primarily for performance (i.e. execution time) and/or code size. Hence, it is essential to evaluate how these optimization options influence the power and energy consumption within the processor while running a software kernel [77].

In this chapter we evaluate the effects of the global performance optimizations on the energy and power consumption of the C6416T processor. Moreover, we assess the impact of these optimization options on the most important execution characteristics such as the memory references, the L1D cache misses and the Instruction Per Cycle (IPC) as a measure of the instruction level parallelism.

The rest of the chapter is organized as follow: Section 4.2 explores the features of the targeted compiler. The impact of various compiler optimizations on the power and energy is analyzed in Section 4.3 as well as the effect of different execution characteristics on the

performance, power, and energy. In Section 4.4 the effect of two specific C64x+ architectural features namely; Software Pipelined Loop (SPLOOP) and the employment of the Single Instruction Multiple Data (SIMD) on the energy and power consumption is explored. Section 4.5 presents an evaluation to the application-architecture correlation. Finally, conclusions are drawn in Section 4.6.

Tab. 4.1: Features of the global performance optimization options.

Optimizations	Features
-o0	performs control-flow-graph simplification, allocates variables to registers, performs loop rotation, eliminates unused code, simplifies expressions and statements, Expands calls to functions declared inline.
-o1	all -o0 optimizations, plus: performs local copy/constant propagation, removes unused assignments, eliminates local common expressions.
-o2	all -o1 optimizations, plus: performs software pipelining, performs loop optimizations, eliminates global common sub-expressions and unused assignments, converts array references in loops to incremented pointer form, performs loop unrolling.
-o3	all -o2 optimizations, plus: removes all functions that are never called, simplifies functions with return values that are never used, inline calls to small functions, reorders function declarations so that the attributes of called functions are known when the caller is optimized, propagates arguments into function bodies when all calls pass the same value in the same argument position, identifies file-level variable characteristics.

4.2 Targeted Compiler and Applications

The embedded C/C++ compiler version 6.0.1, in the Code Composer Studio (CCS3.1) from Texas Instruments, is used for generating the software binaries to be loaded to the DSP. The TMS320C6000 C/C++ compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages, and produces assembly language source code for the C6416T device. The compiler supports the 1989 version of the C language.

This compiler features many levels of optimization as shown in Table 4.1 mainly tuned for speed and/or code size. This can be achieved by invoking the {-o0, -o1, -o2, -o3} options for global speed optimization [96].

4.3 Global Performance Optimizations Effects on power and Energy

Figure 4.1 presents the measured power consumption at the four global performance optimization levels for different signal and image processing benchmarks along with their averages. It is obvious from Fig. 4.1 that these optimization options, on average, increase the power consumption.

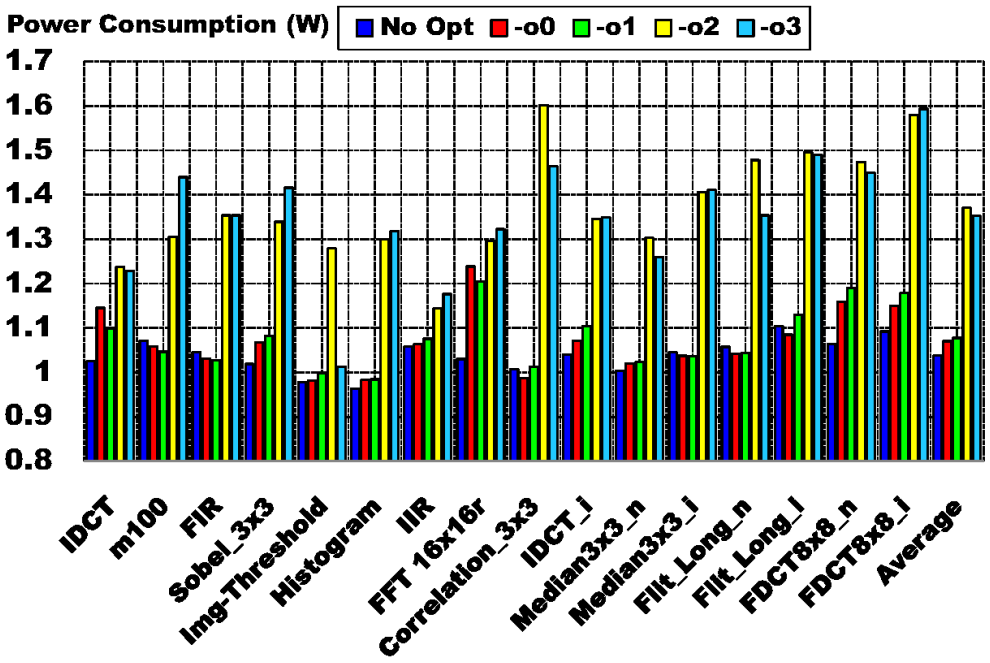

Fig. 4.1: Power consumption of the C6416T while running different benchmarks.

This emphasizes the fact that most aggressive optimizations (although they may lead to minimum execution times) do not necessarily result in the best code from the perspective of power consumption. The highest optimization level -o3 increases the power consumption on average by 30.3% compared to the no optimization option. This percentage reaches 45% for some individual benchmarks, FDCT8x8_i and correlation_3×3. On average, invoking -o2 or -o3 leads to much power consumption than invoking -o0 or -o1. The software pipelining

loop feature is enabled with -o2 and -o3 allowing better instruction parallelization, and as we explain later this has a significant impact on the power consumption.

Although the results in Fig.4.1 demonstrate that invoking the global performance optimizations increases, on average, the power consumption, the energy significantly decreased. Figure 4.2 shows the normalized energy for each of the benchmarks. The normalization is achieved by relating the energy for each of the benchmarks while invoking different optimization levels to the energy when all optimization options are disabled.

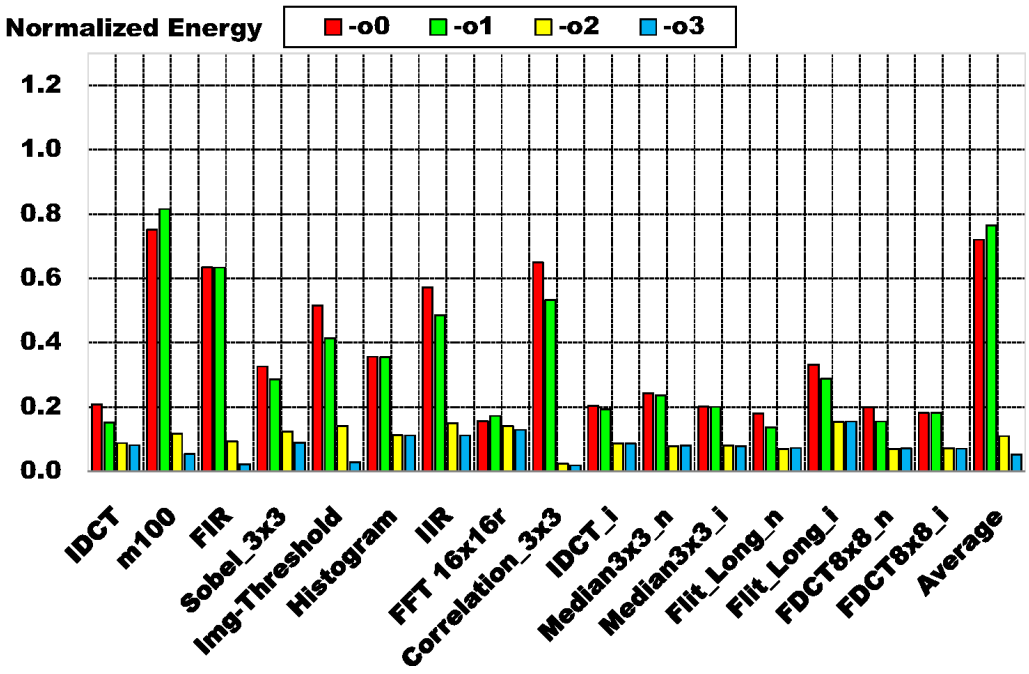

Fig. 4.2: Normalized Energy versus various optimization options.

Figure 4.3 demonstrates that there is a strong correlation between execution time and energy consumption. The most aggressive speed optimization level -o3 reduces the execution time on average by 96.2% compared to the no optimization option. While, it reduces the energy on average by 94.8%. It is obvious that invoking -o2 and -o3 provides significant additional energy saving than when invoking -o0 or -o1. This can be explained by the fact that, at -o2 and -o3 the software loop pipelining is enabled, consequently leading to considerably higher reduction in the execution time and hence in the energy.

Two groups of optimization levels can be identified namely -o0 plus -o1, enhanced mainly by register allocation, and -o2 plus -o3 distinguished by the use of software pipelined loops or in other words hardware (zero-overhead) loops. The no optimization, -o0 and -o1 im-

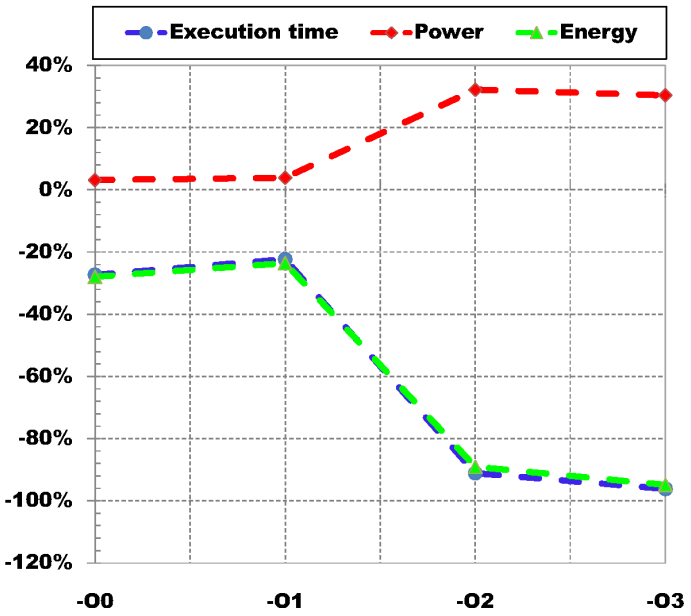

Fig. 4.3: Power, Execution Time and Energy normalized refering to no optimization versus different optimization options.

ply the use of pointer registers. However, in no optimization case, data is pre-fetched from memory prior to the execution of the instruction that needs this data. In case of -o0 and -o1 data is fetched simultaneously with the instruction execution, saving CPU cycles and consequently energy. Therefore, if the programm uses many variables, the power consumption increases in companion with an energy drop. If the programm utilizes few variables it presents similar energy drop with unchanged power consumption, since the registers are loaded less frequently and reused more often.

Optimization levels -o2 and -o3 are defined mainly by the use of hardware loops. Once set up, hardware loops parallelize counter update, comparison and branch operations, thus saving a fixed amount of cycles per loop iteration, mostly by avoiding pipeline stalls. Given that stalls have lower power consumption than normal instruction execution, shortening the programs in this way actually increases power consumption. However, the magnitude of this increase depends on how long the loop kernel is and the instructions within it. To a lesser extent, elimination of global common subexpressions further reduces the cycle count and the power consumption.

4.3.1 Optimizations Effect on Other Execution Characteristics

In order to analyze the previous results we find that it is worth to study the effect of the compiler optimizations on four important execution characteristics: data cache misses, memory references, IPC, and CPU stall cycles. Figure 4.4 illustrates the effect of different optimization levels on the L1D cache misses. The L1D cache misses decreases, on average, by almost 69% when -o3 is invoked. The L1D cache misses require the access of L2D cache/SRAM which in turn provide additional power consumption.

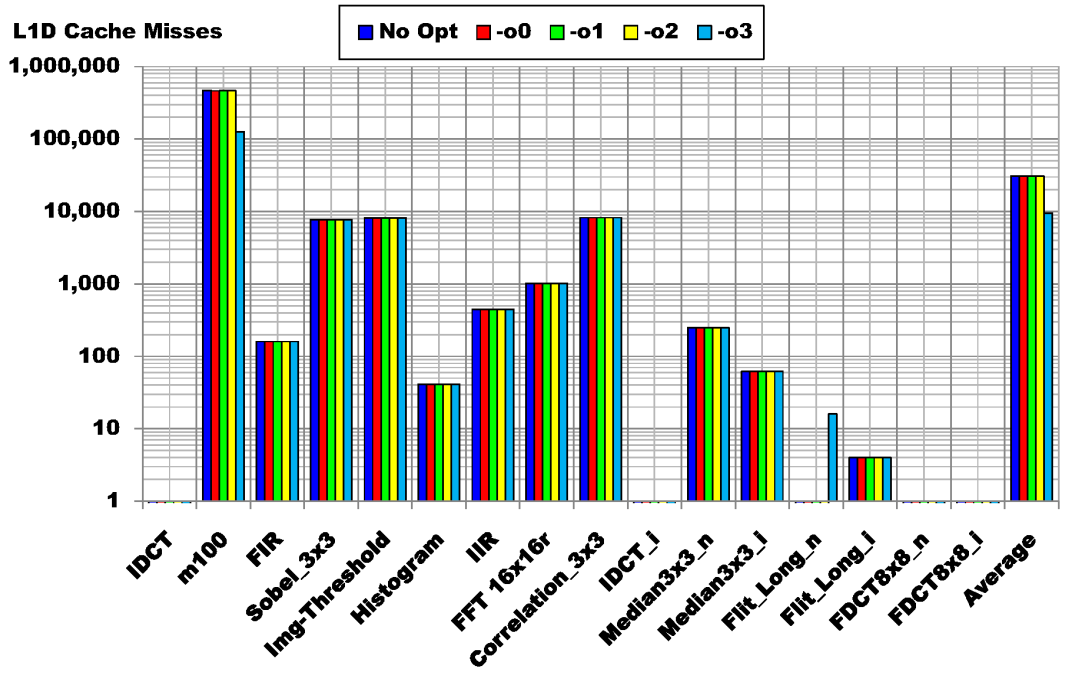

Fig. 4.4: Impact of optimizations on the L1D cache misses.

The CPU stall cycles are decreased by 78% when -o3 is invoked as shown in Fig. 4.5. Several reasons can cause the CPU to stall such as the cache miss, the resource conflicts and the memory bank conflicts. Although one data cache miss causes at least six CPU stall cycles, the C6416T CPU has two features that are expected to decrease the cache miss penalty.
The first feature, the L1D cache of the C6416T DSP pipelines the L1D cache read misses. A single L1D read miss takes six cycles when serviced from L2 SRAM, and eight cycles when serviced from L2 cache. Pipelining of cache misses can hide much of the miss penalty (CPU stall cycles) by overlapping the processing of several cache misses. The miss overhead can be expressed as $(4 + (2 \times M))$ when serviced from L2 SRAM or as $(6 + (2 \times M))$ when serviced from L2 cache where M is the number of cache misses [97].

Therefore, the pipelining of cache misses provides significant reduction in the execution time and consequently the energy but it still has no effect on the power consumption.

The second feature, the write cache miss does not directly stall the CPU because of the use of L1D Write buffer [97]. This also affects the execution time but has no effect on the power consumption especially when the write buffer is not full.

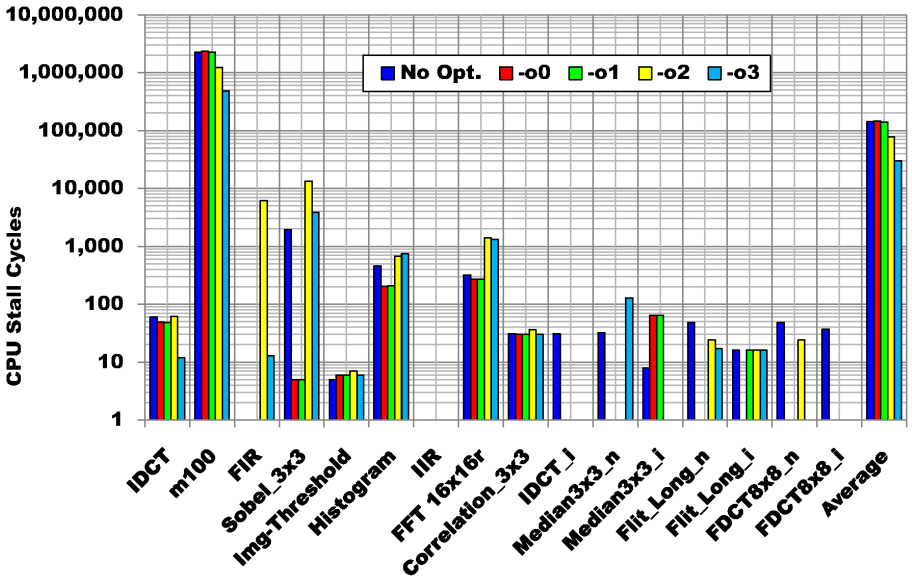

Fig. 4.5: CPU stall cycles versus different optimization options.

Figure 4.6 illustrates that the IPC is increased by about 269% when -o3 is invoked compared to the case when all optimization options are disabled. This surely decreases the execution time and consequently the energy, as more overlapping in the execution of the instructions per cycle will be achieved. But, this results in higher parallelization degree which in turn increases the power consumption.

Figure 4.7 shows the impact of the parallelization on the consumed power as well as the execution time. It is clear from this figure that the compiler occasionally misbehaves for example, invoking -o0 sometimes provides better performance results than invoking -o1. Although the execution time is inversely proportional to the parallelization, the power consumption is directly proportional.

Although, Fig. 4.8 points out that the memory references are decreased by 94% which is expected to save the consumed power, we find that the power is increased. This emphasizes our results in [95,98] that the Instruction Management Unit (IMU), the unit which is responsible for fetching and dispatching instructions, contribution to the total power consumption dominates the memory referencing contribution.

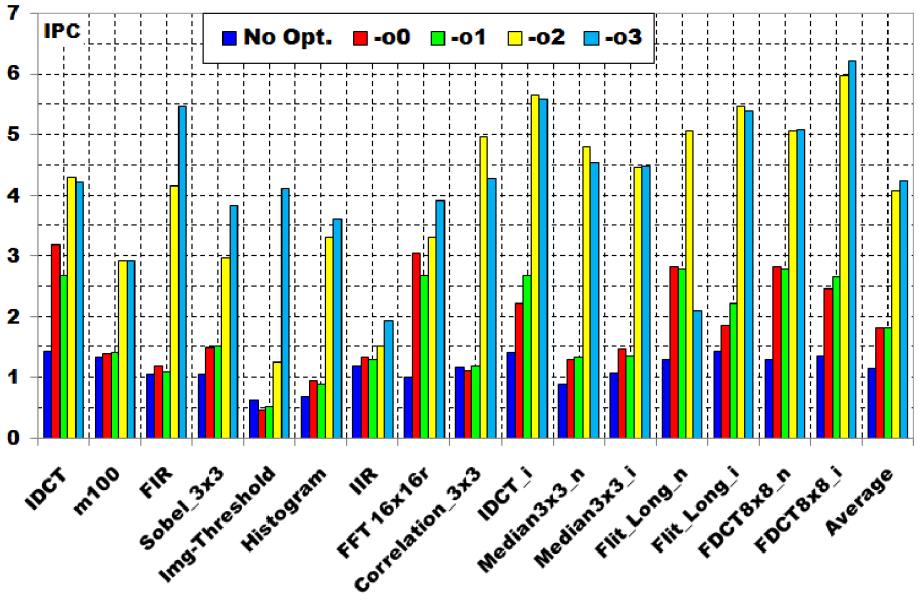

Fig. 4.6: Effect of various optimization options on the instructions per cycle.

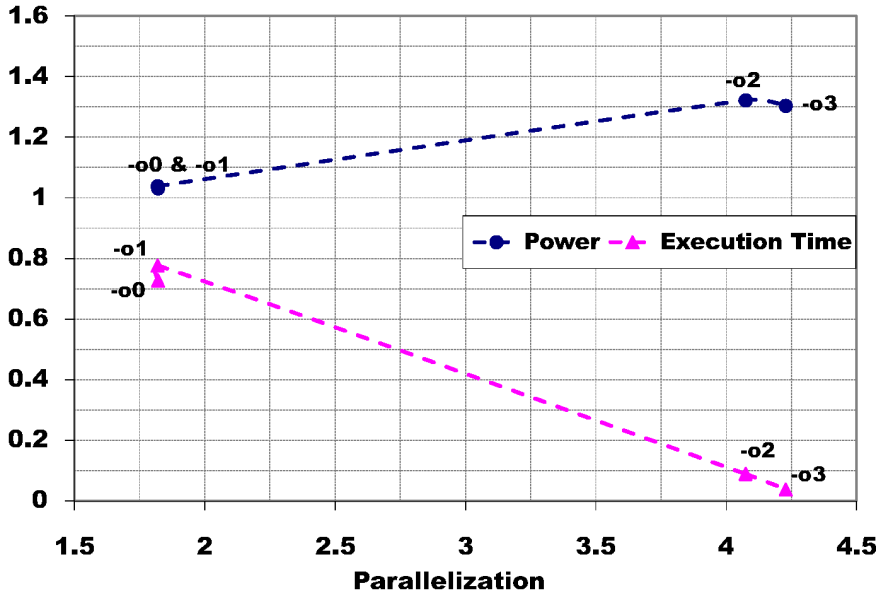

Fig. 4.7: Parallelization impact on the execution time and the power consumption.

Figure 4.9 shows the relation between the memory references in one side and the power and execution time on the other side. The values for memory references, power and execution time are normalized w.r.t. the case of no optimizations. As shown in Fig. 4.9 invoking -o3 saves more memory references than invoking -o2 which explains why the power consumption when -o2 is invoked is slightly higher than when -o3 is invoked.

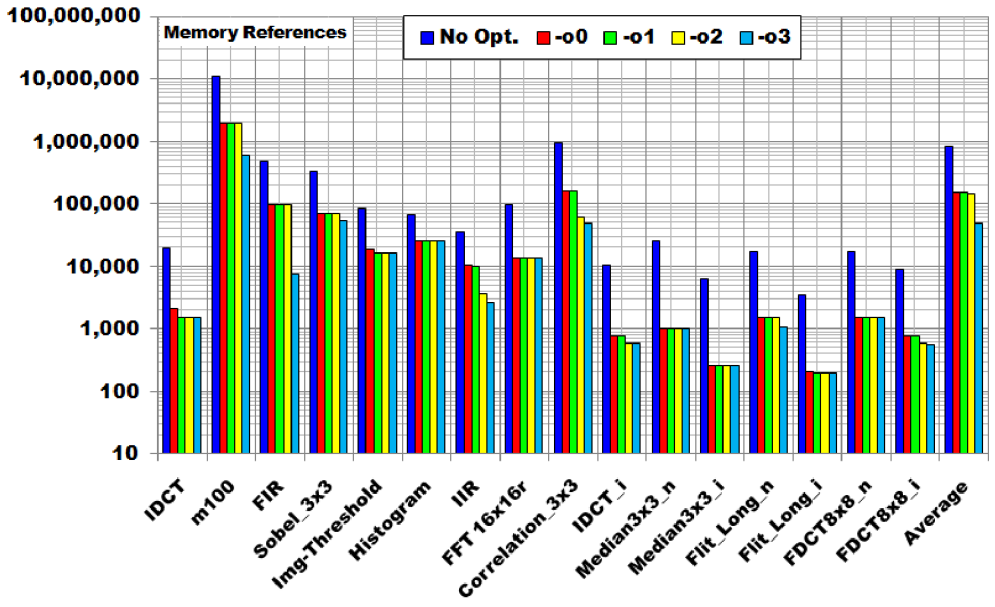

Fig. 4.8: Effect of different optimization options on the Memory accesses.

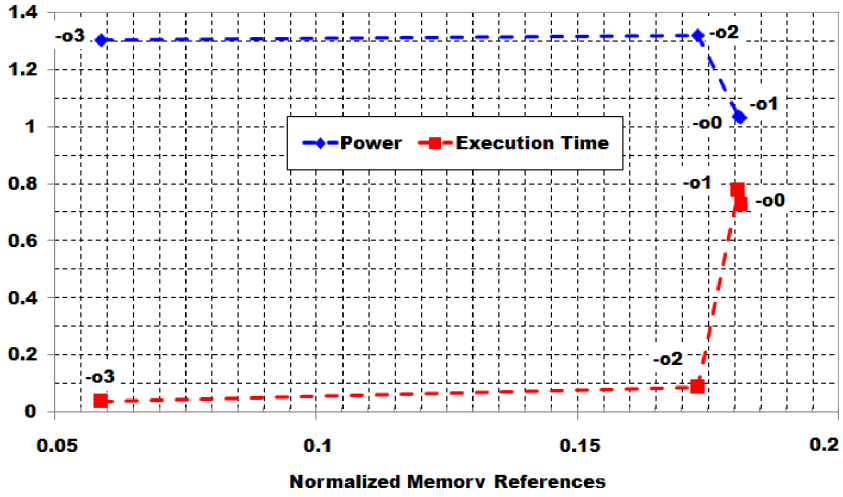

Fig. 4.9: Memory references impact on the power as well as the execution time.

4.4 Specific Architectural and Compiler Features Effects on Power and Energy

4.4.1 Impact of Software Pipelined Loop

Software pipelined loop (SPLOOP), also called hardware Zero-Overhead Loop (ZOL), is a type of instruction scheduling that exploits instruction level parallelism (ILP) across loop iterations. SPLOOP is a specific architectural optimization feature of the C64x+ CPU, the C64x CPU does not support the SPLOOP. This feature allows the CPU to store a single it-

eration of loop in a specialized buffer which contains hardware that will selectively overlay copies of the single iteration in a software pipeline manner to construct an optimized execution of the loop [99]. Modulo scheduling is a form of SPLOOP that initiates loop iterations at a constant rate, called the iteration interval (ii). To construct a modulo scheduled loop, a single loop iteration is divided into a sequence of stages, each with length ii. In the steady state of the execution of the SPLOOP, each of the stages is executing in parallel. The instruction schedule for a modulo scheduled loop has three components:
a prolog, a kernel and an epilog. The kernel is the instruction schedule that executes the pipeline steady state. The prolog and epilog are the instruction schedules that setup and drain the execution of the loop kernel [99].

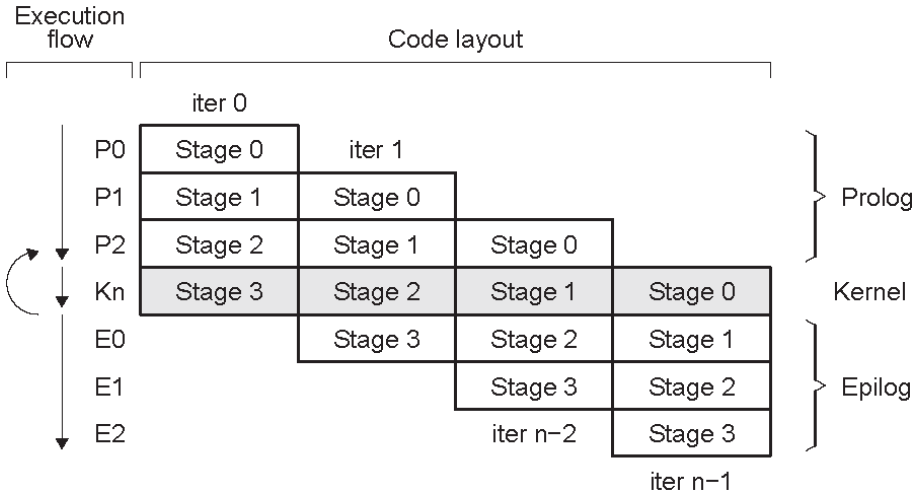

Fig. 4.10: Concept of the SPLOOP.

In Figure 4.10, the steady state has four stages, each from a different iteration, executing in parallel. A single iteration produces a result in the time it takes four stages to complete, but in the steady state of the software pipeline, a result is available every stage (that is, every ii cycles).

In this section we evaluate the impact of the software pipelining on the power and energy consumption. The SPLOOP feature is implicitly enabled with the global optimization options -o2 and -o3. However, we override this by invoking the -mu option which disables only the SPLOOP feature. To distinguish between the case when the software pipelining is enabled or disabled, we utilize the term -o2-mu and -o3-mu to indicate that the SPLOOP feature is disabled.

Table 4.2 summarizes the average power, execution time, and energy for the different signal and image processing benchmarks listed in Table 3.3 when -o2, -o2-mu, -o3, and -o3-mu are

invoked.

Tab. 4.2: Average power, execution time, and energy for the investigated benchmarks.

	Power(W)	Exec.Time(mSec)	Energy(mJ)
-o2	1.371	0.168	0.221
-o2-mu	1.109	0.703	0.766
-o3	1.352	0.072	0.104
-o3-mu	1.117	0.16	0.204

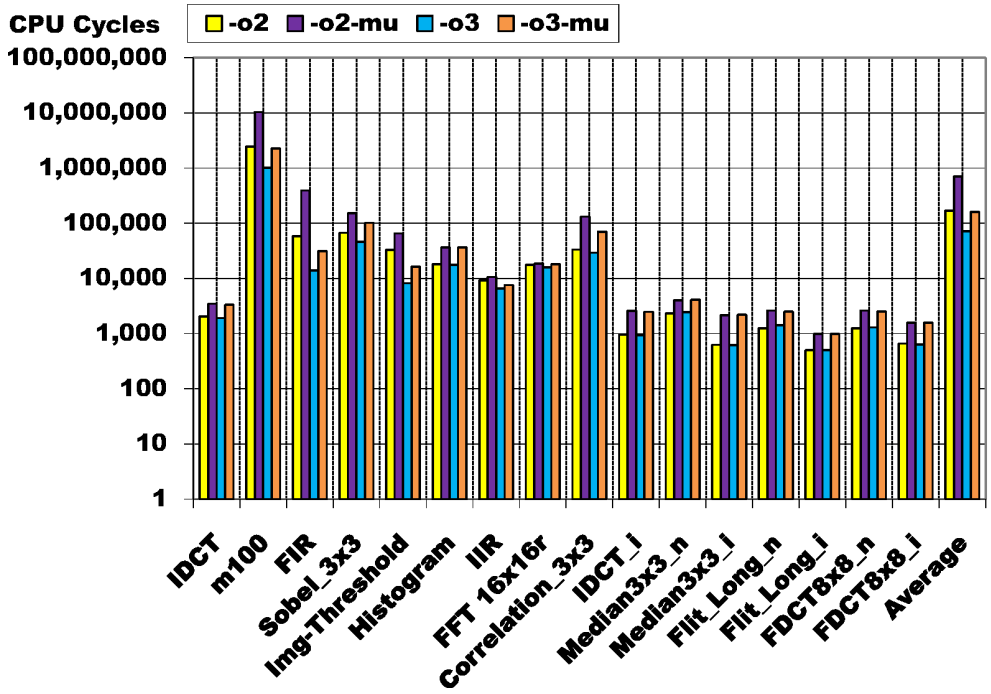

Fig. 4.11: various optimizations versus execution cycles.

Figure 4.11 clearly illustrates the strong impact of the SPLOOP on the execution time. When the -o2-mu and -o3-mu are invoked the execution cycles increase by 317.3% and 120.8% respectively. This increase is relative to the case when -o2 and -o3 are invoked.
It is noticeable that the impact of disabling the SPLOOP on the execution cycles is higher when invoking -o2 than when invoking -o3. Since -o3 include all the individual optimizations that exist in -o2 as well as more performance oriented optimizations that aim to reduce the execution cycles via extra reduction in the memory references.

Despite the increase in the execution cycles when SPLOOP is disabled, the power consumption decreases, on average, by 19.1% and 17.4% when -o2-mu and -o3-mu are invoked respectively. Figure 4.12 shows the power consumption reduction for all the benchmarks

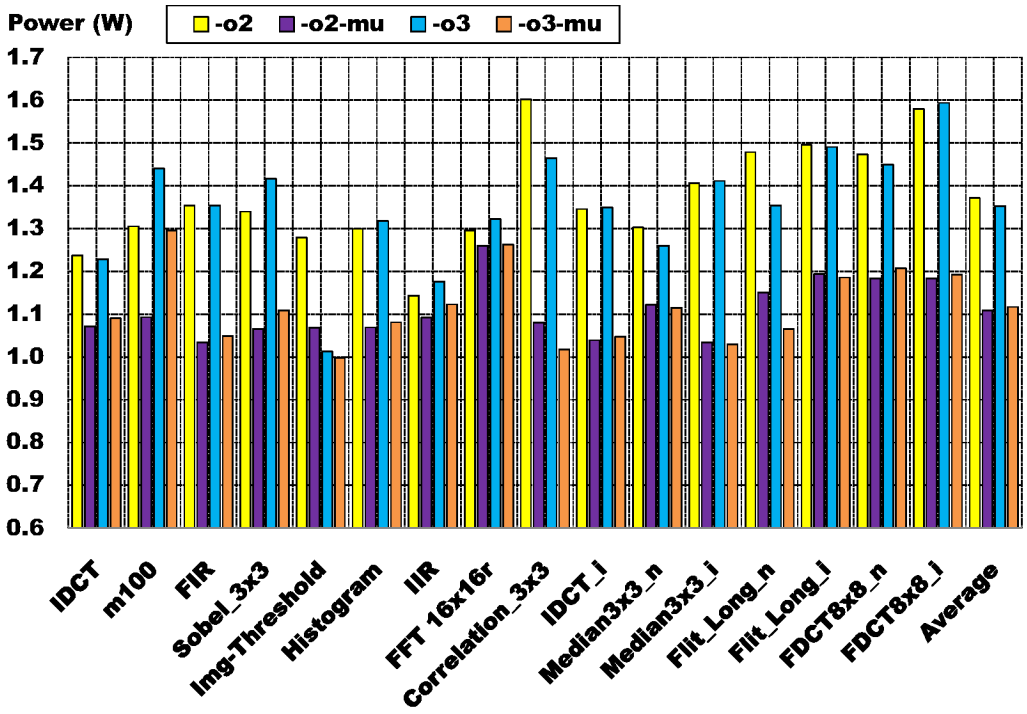

Fig. 4.12: Impact of SPLOOP on the consumed power.

when the SPLOOP feature is disabled. Figure 4.13 shows the effect of the SPLOOP on the energy. It is clear from Fig. 4.13 that the energy increases for all the benchmarks. The increase in the energy when SPLOOP is disabled is directly related to the increase in the execution time.

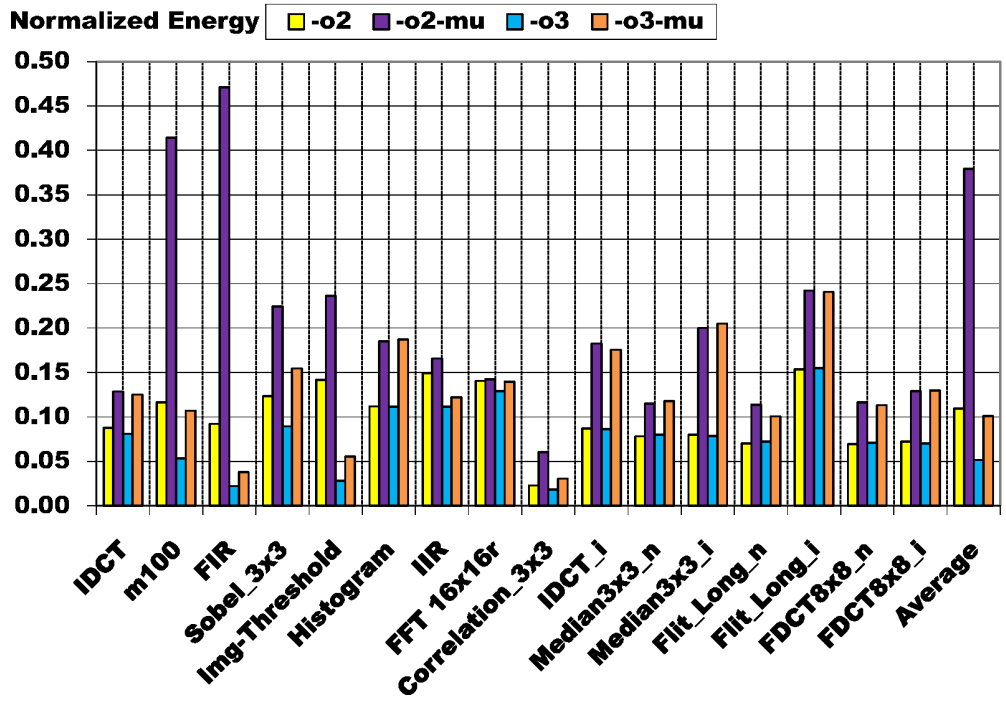

Fig. 4.13: Impact of SPLOOP on the energy usage.

We find that disabling the SPLOOP has no effect on the memory references and the L1D cache miss rate but it significantly affects the IPC.

The IPC decreases, on average, by 55.75% and 48.5% when -o2-mu and -o3-mu are invoked respectively resulting in lower instruction parallelism rate and consequently lead to the pre-mentioned power saving as shown in Fig. 4.14.

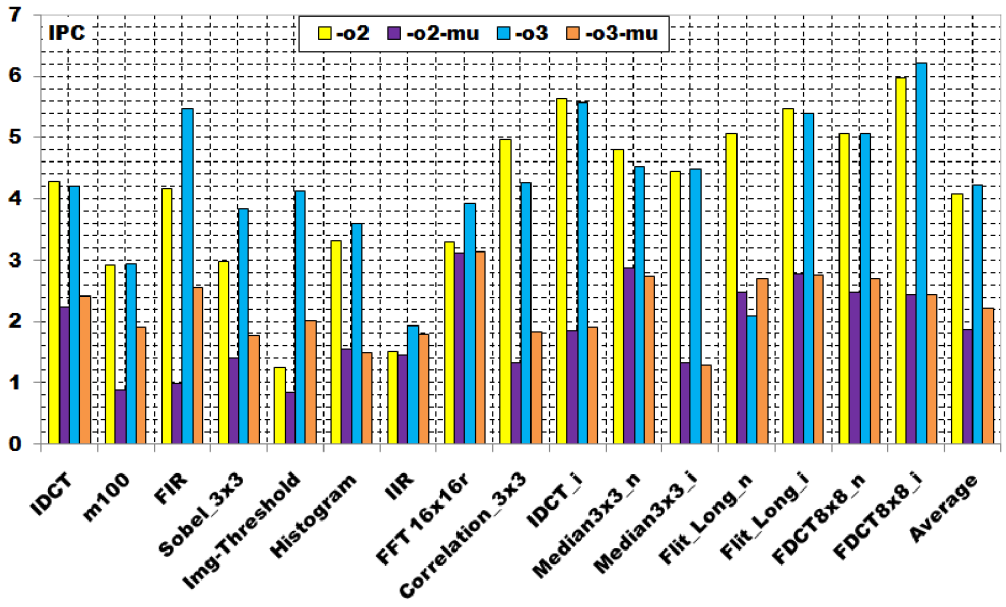

Fig. 4.14: SPLOOP effect on IPC.

The power consumption increases, on average, by 7.67% when -o3-mu is invoked compared to the case when no optimization option is invoked. Hence, the SPLOOP contributes by 70.3% to the total power increase when -o3 is invoked. Therefore, more attention to the design of the specialized hardware for the software pipelining should be paid to compromise the performance and power trade-offs for the C6416T.

Figure 4.15 summarizes our results regarding the effect of the SPLOOP feature on the power consumption. It is pretty clear that invoking -o3-mu can be considered as a trade-off between performance and power consumption.

4.4.2 Impact of SIMD

The C6000 compiler recognizes a number of intrinsic C-functions. Intrinsics allow the programmer to express the meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Most of the intrinsic functions make use of the

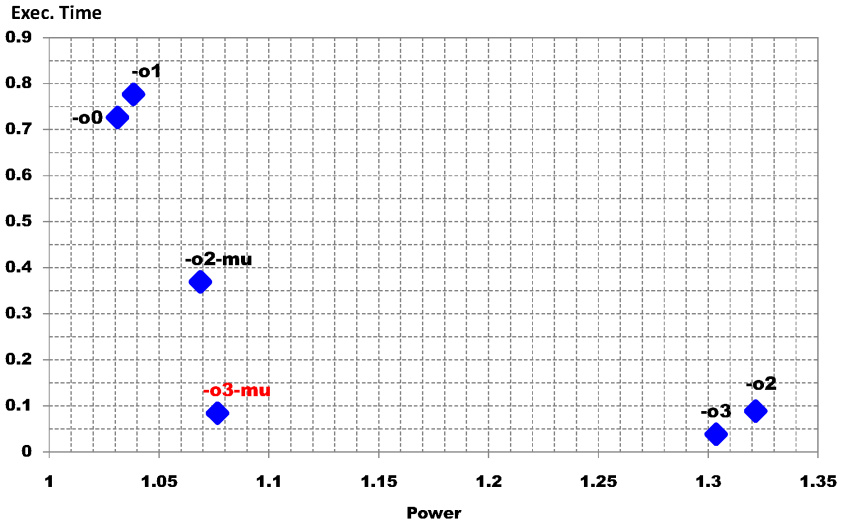

Fig. 4.15: Execution time vs. power consumption with various optimization levels.

SIMD capabilities of the C6416T. Intrinsics are used like functions. The programmer can use C/C++ variables with these intrinsics, just as coping with any normal function. The intrinsics are specified with a leading underscore, and are accessed by calling them as done with a function. For example:

```
int X1, X2, Y;
Y = _sadd(X1, X2)
```

For a complete list of the C6000 and the specific C64x+ intrinsic functions readers are encouraged to look at [96].

In order to assess the effect of utilizing SIMD instructions on the energy and power consumption, with the aid of the Texas Instrument host intrinsics package Ver.0.72 [100], we prepare two versions of the Inverse Discrete Cosine Transform (IDCT) algorithm as a case study. The first version is implemented without utilizing any of the SIMD instructions while the second is implemented with the aid of all possible SIMD instructions as shown in Fig. 4.16. The functionality of both versions are tested and verified to give the same result.

We study the effect of the employing SIMD instructions isolated from the effect of the SPLOOP feature by compiling the two versions with -o2-mu and -o3-mu (-mu disables the SPLOOP feature). It is worth to mention here that invoking -o0 or -o1 do not enable the SPLOOP feature.

Table 4.3 shows the results of applying SIMD instructions when no optimization option is invoked. A significant reduction in the execution cycles, slightly more than 49.5%, is achieved in case of employing the SIMD instructions. This great reduction in the execution cycles is

Original Code

```
#include "idct_8x8_c.h"
#pragma CODE_SECTION(idct_8x8_cn,   ".text:ansi");
void idct_8x8_cn(short *idct_data, unsigned num_idcts)
{
    _nassert((int) idct % 8 == 0);
    _nassert(num_idcts >= 1);
    for (i = 0; i < num_idcts; i++)
    {
        for (j = 0; j < 8; j++)
        {
            F0 = idct[i][0][j];
            F1 = idct[i][1][j];
            F2 = idct[i][2][j];
            F3 = idct[i][3][j];
            F4 = idct[i][4][j];
            F5 = idct[i][5][j];
            F6 = idct[i][6][j];
            F7 = idct[i][7][j];

            P0 = F0;                P1 = F4;
            R1 = F2;                R0 = F6;

            Q1 = (F1*C7 - F7*C1 + 0x8000) >> 16;
            Q0 = (F5*C3 - F3*C5 + 0x8000) >> 16;
            S0 = (F5*C5 + F3*C3 + 0x8000) >> 16;
            S1 = (F1*C1 + F7*C7 + 0x8000) >> 16;

            p0 = ((int)P0 + (int)P1 + 1 ) >> 1;
            p1 = ((int)P0 - (int)P1     ) >> 1;
            r1 = (R1*C6 - R0*C2 + 0x8000) >> 16;
            r0 = (R1*C2 + R0*C6 + 0x8000) >> 16;
```

Code with Intrinsics

```
#include "idct_8x8_i.h"
#pragma CODE_SECTION(idct_8x8_c,   ".text:intrinsic");
void idct_8x8_c(short *idct_data, unsigned num_idcts)
{
    _nassert((unsigned)idct_data % 8 == 0);
    #pragma MUST_ITERATE(4,,4);
    for (i = j0 = j1 = 0; i < num_idcts * 4; i++)
    {
        F00   = _amem4(&_ptr[ 0 + 2*j0]);
        F11   = _amem4(&_ptr[ 8 + 2*j0]);
        F22   = _amem4(&_ptr[16 + 2*j0]);
        F33   = _amem4(&_ptr[24 + 2*j0]);
        F44   = _amem4(&_ptr[32 + 2*j0]);
        F55   = _amem4(&_ptr[40 + 2*j0]);
        F66   = _amem4(&_ptr[48 + 2*j0]);
        F77   = _amem4(&_ptr[56 + 2*j0]);

        if (++j0 == 4) { j0 = 0; i_ptr += 64; }

        F17   = _pack2(F11, F77);
        F53   = _pack2(F55, F33);
        F26   = _pack2(F22, F66);
        F04   = _pack2(F00, F44);

        Q1    = _dotpsu2(F17, C71);
        Q0    = _dotpsu2(F53, C35);
        S0    = _dotpsu2 (F53, C53);
        S1    = _dotpsu2 (F17, C17);

        SOQ0  = _pack2(S0, Q0);
        S1Q1  = _pack2(S1, Q1);
```

Fig. 4.16: An example of the IDCT kernel w/wo SIMD utilization.

Tab. 4.3: SIMD effect when no optimization option is invoked.

	Original	with SIMD	%
Exec. Cycles	28 220	14 224	−49.60
Power (W)	1.025	1.039	1.41
Energy (mJ)	0.0289	0.0148	−48.89
IPC	1.425	1.407	−1.26
CPU Stall Cycles	60	31	−48.33
Memory References	19 821	10 519	−46.93

achieved with less than 1.5% increase in the power consumption. The energy follows the reduction in the execution cycles and hence is reduced by 48.89%.

Tab. 4.4: Impact of SIMD when -o0 optimization options are invoked.

	Original	with SIMD	%
Exec. Cycles	5 244	2 818	−46.26
Power (W)	1.145	1.070	−6.5
Energy (mJ)	0.006	0.00302	−49.75
IPC	3.197	2.229	−30.30
CPU Stall Cycles	49	0	−100
Memory References	2 142	773	−63.91

Table 4.4 shows the results of employing SIMD instructions when invoking -o0 global performance optimization options. The execution time decreases by 46.22% when the SIMD instructions are utilized, which is lower than the decrease in case of no optimization option is

invoked. But, on the other hand, the power consumption decreases by 6.5% which maintains the previous energy saving (when no optimization option is invoked) to 49.75%. The main reason for this power consumption reduction is the decrease in the IPC by 30.3%. Hence, from the power dissipation point of view invoking -o0 with the employment of SIMD outperforms invoking no optimization option without sacrificing the energy saving.

Table 4.5 illustrates the results of applying SIMD instructions when invoking -o1 optimization options. The execution time decreases by 35.43% while the power consumption is imperceptibly increased by 0.5% this is because the IPC is almost the same as in the case when no SIMD is utilized. This leads to an energy saving by 35.15%.

Tab. 4.5: SIMD influence when -o1 optimization options are invoked.

	Original	with SIMD	%
Exec. Cycles	3 985	2 573	−35.43
Power (W)	1.099	1.104	0.44
Energy (mJ)	0.00438	0.00284	−35.15
IPC	2.678	2.693	0.54
CPU Stall Cycles	48	0	−100
Memory References	1 536	768	−50.0

Comparing the results of utilizing SIMD while invoking -o1 and -o0, we can conclude that utilizing the SIMD while invoking -o0 can be considered as a power aware optimization option. This is of course on the account of longer execution time by almost 10%, if compared to the case of invoking -o1.

It is much valuable to invoke -o2 and the -o3 from the execution speed perspective, but to study the effect of utilizing the SIMD isolated from the effect of the SPLOOP we decide to turn off the SPLOOP, which is implicitly invoked with -o2 or -o3. Hence, Tables 4.6 and 4.7 illustrate the results of utilizing SIMD when -o2-mu and -o3-mu (-mu is used to turn off the SPLOOP feature) are invoked.

Table 4.6 shows that employing SIMD while invoking -o2-mu achieves slightly more than 3% power saving. It also achieves 25.21% enhancement in the execution time. The achieved power saving is mainly caused by the reduction of the IPC by 17.34% while the enhancement in the execution time is related to the significant memory references reduction by more than 62%.

Table 4.7 demonstrates that the employment of SIMD in conjunction with invoking -o3-mu

Tab. 4.6: SIMD Impact when -o2-mu (SPLOOP is disabled) optimization options are invoked.

	Original	with SIMD	%
Exec. Cycles	3 471	2 596	−25.21
Power (W)	1.072	1.039	−3.02
Energy (mJ)	0.00372	0.0027	−27.47
IPC	2.231	1.844	−17.34
CPU Stall Cycles	139	0	−100
Memory References	1 536	578	−62.37

achieves 3.96% power saving while it achieves 25.4% and 28.35% reduction in the execution time and the energy respectively. The achieved power saving is mainly caused by the reduction of the IPC by 20.86% while the enhancement in the execution time is derived by the significant memory references reduction, by more than 62%.

Tab. 4.7: Impact of SIMD when -o3-mu (SPLOOP is disabled) optimization options are invoked.

	Original	with SIMD	%
Exec. Cycles	3 319	2 476	−25.4
Power (W)	1.091	1.048	−3.96
Energy (mJ)	0.00362	0.00259	−28.35
IPC	2.416	1.913	−20.86
CPU Stall Cycles	96	0	−100
Memory References	1 536	576	−62.5

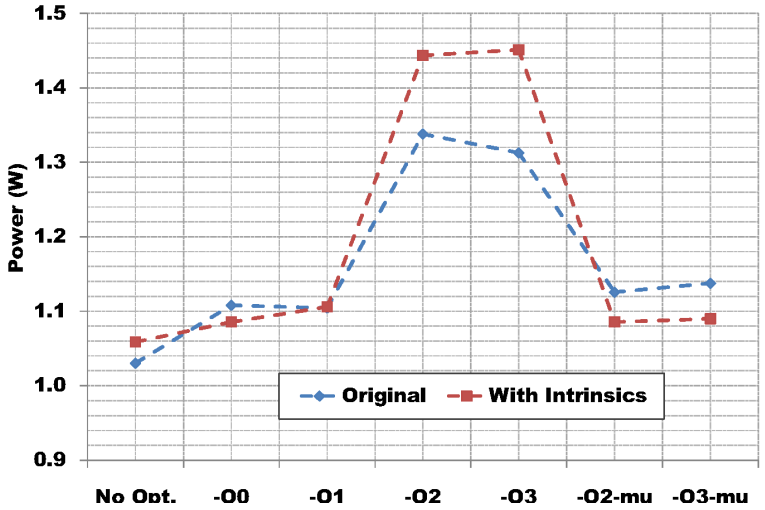

Fig. 4.17: Power consumption w/wo SIMD utilization vs. various optimization options.

To precisely determine the effect of utilizing the SIMD on the power consumption, energy and the execution time we investigate two more case studies, the Discrete Cosine Transform (DCT) and the Median filter with a 3x3 window in the same manner as the investigation of the IDCT. Figure 4.17, 4.18 and 4.19 represent a comparison between the averages of power consumption, energy and the execution cycles of the three case studies with/without SIMD employment against various performance optimization options.

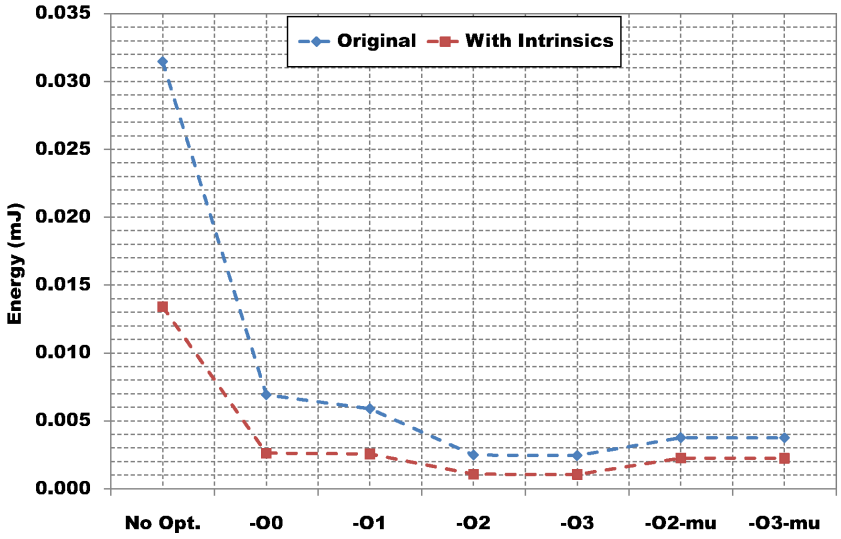

Fig. 4.18: Energy w/wo SIMD utilization vs. various optimization options.

Generally, employing the SIMD significantly enhances the performance and the energy saving. The SPLOOP feature is the main basis for the significant improvement in the performance when -o2 or -o3 is invoked [101]. Hence, by disabling the SPLOOP feature, -o2-mu or -o3-mu, the utilization of SIMD instructions results in a comparable performance enhancement with -o2 or -o3 in addition to the great advantage of, on average, 18.83% and 17% power saving, respectively [102].

Thus, it is pretty clear that rewriting the algorithm to maximally utilize SIMD instructions, while invoking the optimization options -o3-mu, is the best choice from the power consumption and performance perspective. Therefore, it can be considered as a trade-off between the power consumption from one side and the execution time and the energy from the other side.

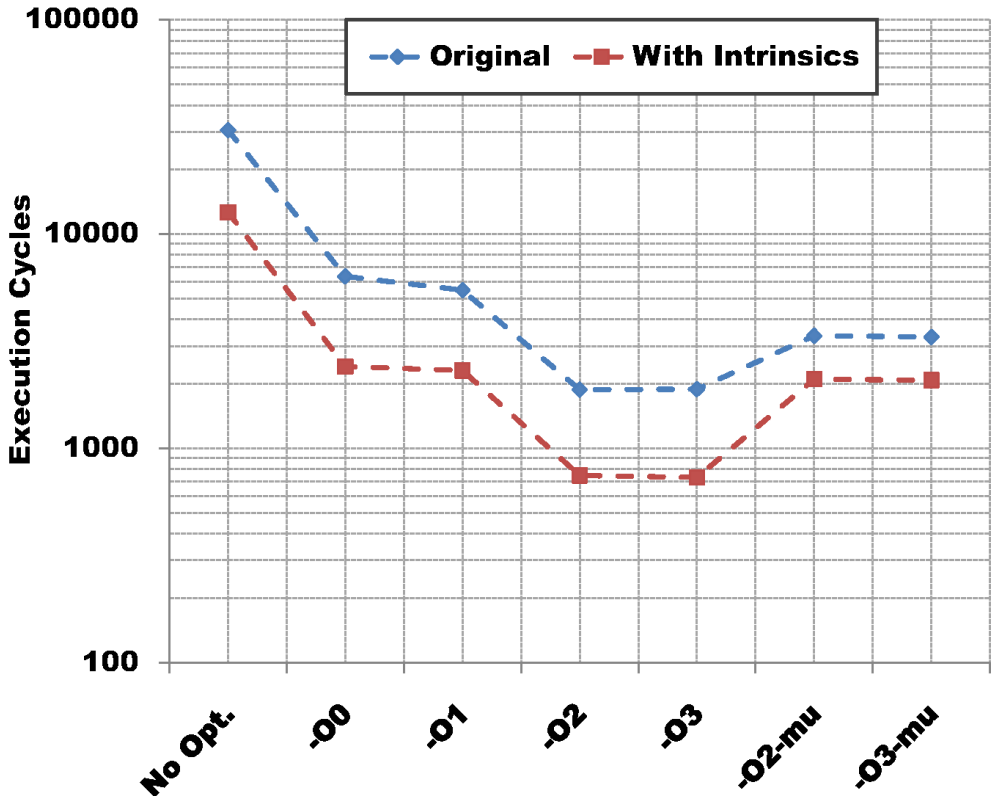

Fig. 4.19: Execution cycles w/wo SIMD utilization vs. various optimization options.

4.5 Characterization of Application-Architecture Correlation

Embedded systems are software running on hardware. An efficient embedded system is that one for which the software application fully utilizes the underlying architecture to deliver optimal energy-cycle performance. The application-architecture correlation is a bidirectional process, matching the algorithmic structure with hardware architecture and vice versa [103]. The objective of this section is to visualize the black box impact of the compiler and the hardware architecture (C6416T) over the software applications. We follow the same methodology utilized in [103]. We prepare 18 applications from image and signal processing benchmarks. The list and description of these applications are presented in Table C.1. In order to characterize the applications at the targeted architecture we choose nine attributes:

- Execution Time (ExecTime)
- Energy (Energy)
- Instructions Per Cycle (IPC)
- L1D Cache Misses (L1DMiss)
- Dispatching Factor (DispFac)
- Power (power)
- Code Size (CodeSize)
- CPU Stall Cycles (stall)
- Memory References (MemRef)

We analyze all the nine attributes data for the 18 applications listed in Table C.1 with the aid of multivariate statistical techniques, in order to determine the application-architecture correlation between these applications and the targeted platform. We utilize box plots, scree plots and Principal Component Analysis (PCA) biplots to explore the correlation between application and underlying hardware architecture, more details about these kind of plots and the PCA are presented in Appendix C.

First, in order to identify the number of necessary principal components, we plot them on a scree plot and a box plot as shown in Fig. 4.20 and Fig. 4.21. The two figures indicate that the first three principal components (PCs) represent more than 90% of the variability in the application profiles. Hence, the first three PCs are sufficient to represent the variability in the application profiles for for the C6416T platforms.

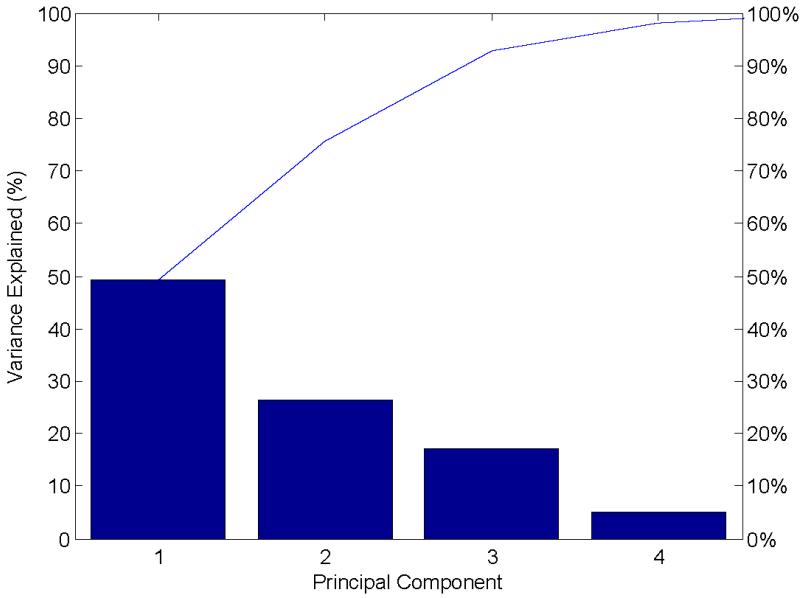

Fig. 4.20: scree plot for the 18 applications at the C6416T using PCA.

Second, the importance of the PCA in reducing the problem dimensionality becomes much more clear with the ability to plot the 18 applications data versus the first two principal components (PCs) as shown in Fig. 4.22. Among the labeled applications A2 (multiplication of two 100x100 matrices) and A12 (Elastic graph matching algorithm) are some of the largest application from execution time, memory references and L1D cache misses perspective. They are definitely different from the remainder of the data, thus they should be considered separately.

Generally PCA is employed to reduce the data dimension. In this section, we focus on the

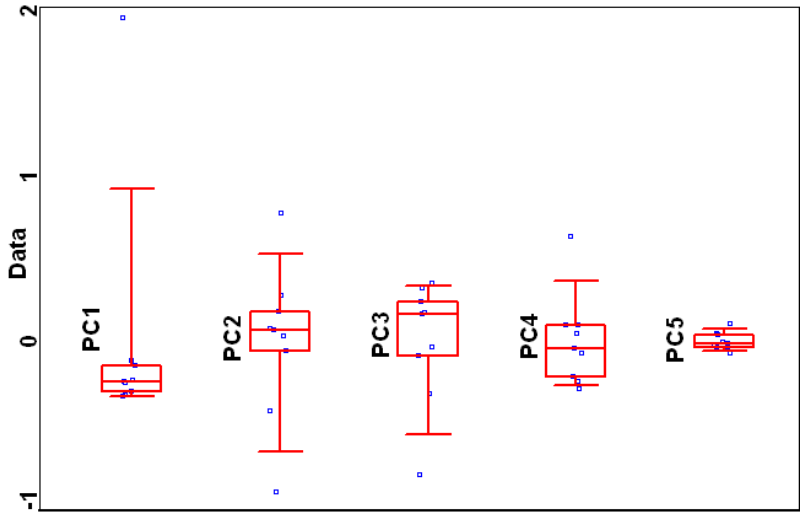

Fig. 4.21: Box plot for the 18 applications at the C6416T using PCA.

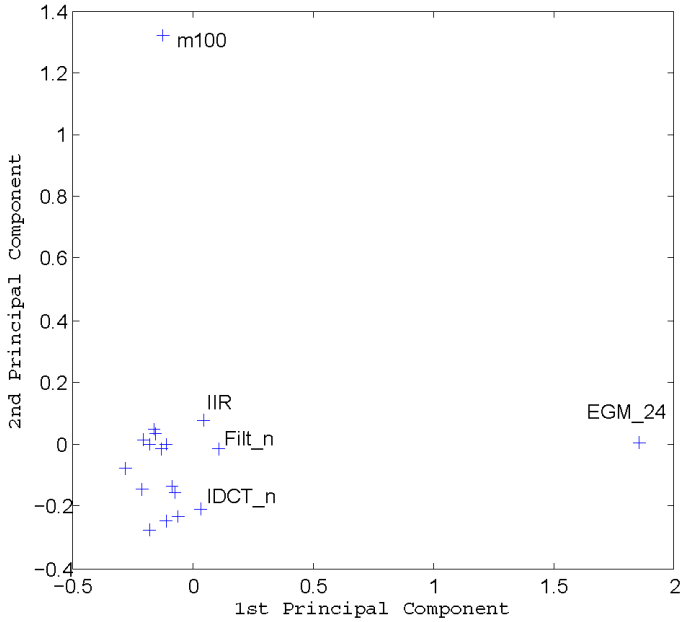

Fig. 4.22: Plot for the 18 applications data vs. the first two PCs.

qualitative analysis of biplots. The biplot helps visualizing both the principal component coefficients for each attribute and the principal component scores for each application in a single plot.

Finally, we utilize the PCA biplot to visualize the black box impact of compiler and hardware architecture over the software applications. We explain first, how we analyze the biplot shown in Fig. 4.23:

- Application names are presented as solid dots.
- Vector lines show the application attributes, they correspond to the nine application attributes.
- The main axes are the first two PCs.

- The biplot is depicted here in such a way, so that it can show the maximum association between the application attributes, PCs and applications.

Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, the first principal component, represented in this biplot by the horizontal axis, has positive coefficients for the attributes ExecTime, Energy, MemRef, CodeSize and Stall, and negative coefficients for the remaining four attributes. That corresponds to vectors directed into the right and left halves of the plot. The second principal component, represented by the vertical axis, has negative coefficients for the attributes CodeSize and IPC, and positive coefficients for the remaining seven attributes. That corresponds to vectors directed into the bottom and down halves of the plot, respectively. This indicates that these components distinguish between applications that have high values for the first set of attributes and low for the second, and applications that have the opposite.

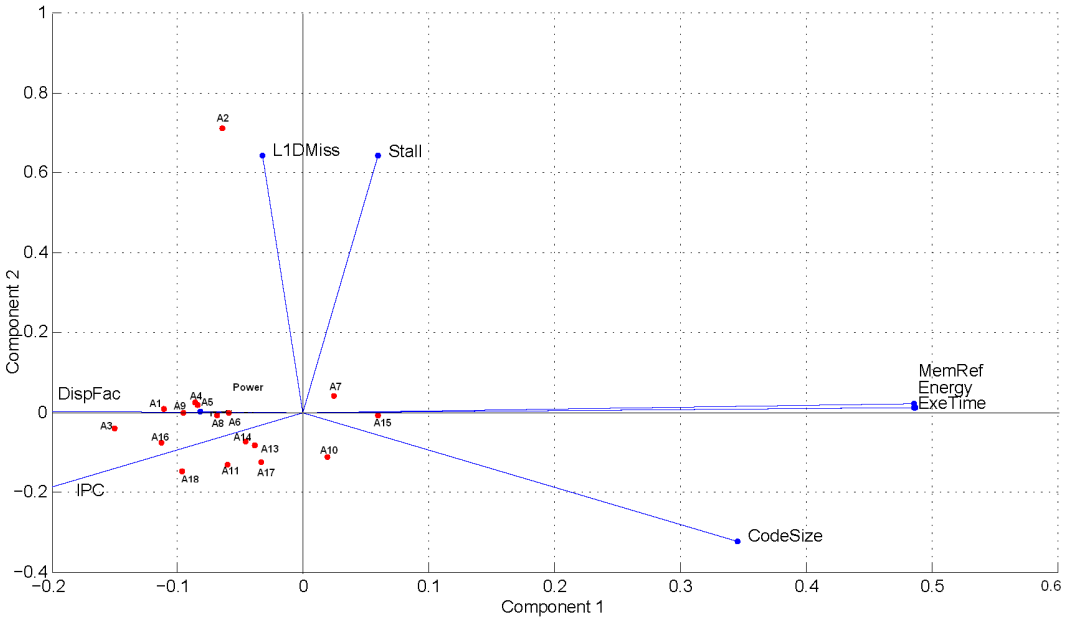

Fig. 4.23: biplot for the 18 applications at the C6416T using PCA.

From Fig. 4.23, it is clear that the majority of the applications are concentrated around the IPC and on the opposite direction of ExecTime, Energy, and MemRef. Which in turn indicates that these applications benefit from the great parallelization capabilities of the C6416T and consequently have small execution time, energy and memory references. The minority of the applications such as A7, A15 and A10 are in the opposite direction that means they relatively consume much more time, energy and have a bigger number of memory references.

On the other hand this is reversed when we consider the power instead of the execution time. Thus, this assures the obtained results in Section 4.3 that the most aggressive optimization level -o3 increases the power consumption, on average, by 30% [104].

4.6 Conclusions

In this chapter we explore the performance and power trade-offs of the targeted architecture. The compiler used to generate the code binaries is the embedded C/C++ compiler Ver.$6.0.1$ in the CCS3.1. We evaluate the effect of invoking the global performance optimization options -o0 to -o3 on the power consumption.
The results show that the most aggressive performance optimization option -o3 reduces the execution time, on average, by 96.2%, while it increases the power consumption by 30.3%. We also find that the energy is significantly decreased, on average, by 94.8%, thanks to the strong correlation between execution time and energy.

To investigate the cause of this power increase we inspected the optimizations effect on some other performance measures, such as the memory references and the data cache misses. Despite the decrease of the memory references by 94%, the IPC increases by 269% and consequently increases the consumed power by 30.3% which emphasizes our results in [95] that the IMU contribution to the total DSP core power consumption dominates the internal memory referencing contribution.

Moreover, we assess the C64x+ architectural feature SPLOOP effect on the power consumption and the performance as well. The results show that the software loop pipelining feature contributes, on average, by 70.3% to the total power consumption increase.
In addition, we investigate the effect of utilizing the targeted architecture SIMD capabilities on the power and energy. The results show that employing the SIMD, in general, has a significant impact on the power consumption, execution time and consequently on the energy. From the power dissipation point of view invoking -o0 with the employment of SIMD can be considered as a power aware optimization option. This of course on the account of longer execution time by almost 10%, if compared to the case of invoking -o1.
In general, invoking -o3-mu (invoking -o3 while disabling the SPLOOP feature), in conjunction with the utilization of SIMD, is a trade-off between execution speed and the power

consumption.

Finally, we characterize the application-architecture correlation for our targeted architecture. The PCA multivariate statistical technique is employed to visualize the black box impact of the compiler and the hardware architecture over the software applications. This is achieved with the aid of biplots which is depicted in our analysis in such a way, so that it can show the maximum association between the application and the underlying hardware architecture. Hence, it answers the question whether a given hardware architecture is an appropriate choice for a given software application or not.

5. IMPACT OF SOURCE CODE TRANSFORMATIONS ON ENERGY AND POWER

5.1 Introduction

Power and energy optimizations can be implemented in hardware through circuit design, and by the compiler through compile-time analysis, code reshaping, and directions to the operating system. While hardware optimizations have been the focus of several studies and are fairly mature, software approaches to optimizing power are relatively new. Progress in understanding the impact of traditional compiler optimizations on the power consumption and developing new power-aware compiler optimizations are important to overall system energy optimization. The optimizations at compile time typically improve performance and rarely the power consumption, as we have explained in Chapter 4, with the main limitations of having a partial perspective of the algorithms and without the possibility of introducing significant modifications to the data structures.
On the contrary, source code transformations can exploit full knowledge of the algorithm characteristics, with the capability of modifying both data structures and algorithm coding; furthermore, inter-procedural optimizations can be envisioned.

In this chapter we present the impact of applying source to source code transformations on the power, energy and performance. The source code transformations that are presented in this chapter are classified into three major groups: loop, data, and procedural transformations.
To evaluate the effectiveness of the applied transformations we compile each program, both

the original and transformed versions, on the target architecture (C6416T DSK). We record the current drawn from the core CPU and hence the consumed power. With the aid of the compiler's profiler we also record the run time and other execution characteristics such as memory references, L1D cache misses and so on. To obtain reliable and precise information, we repeat the whole measuring procedure for each transformation multiple times.

5.2 Loop Oriented Transformations

Among the most important optimizations, in general, are those that operate on loops since the loops are the most time consuming kernels of the code [105]. Loop optimization can be viewed as the application of a sequence of specific loop transformations to the source code, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the result of the program (i.e., be a legal transformation). A transformation is correct, if and only if, it computes the same output values as the original code from the same input values [106].
Evaluating the benefit of a transformation or sequence of transformations can be quite difficult within this approach, as the application of one beneficial transformation may require the prior use of one or more other transformations that, by themselves, would result in reduced performance. Hence, we apply each loop transformation individually on the source code and evaluate its impact on the power as well as the performance with the aid of the target C compiler's profiler.

5.2.1 Loop Reversal

Reversing loop conditions so that they count down instead of up can enhance the speed of loops. Counting down to zero with the decrement operator (i- -) is faster than counting up to a number of iterations with the increment operator (i++). Counting down to zero eliminates the need to a compare instruction and instead the loop is ended with a branch-if-not-equal-zero (BNEZ). The loop reversal transformation is usually used to allow other loop transformations such as the loop interchange or permutation and the loop fusion.
An example of applying the loop reversal transformation is shown in Figure 5.1. In this example the code is composed of two nested loops, the inner loop is reversed to count down

from N-1 to 1.

Original Code

```
for (i = 1; i < N; i++)
{
    for (j = 1; j < N; j++)
    {
        A[i][j] = A[i-1][j+1] + 1;
    }
}
```

Transformed Code

```
for (i = 1; i < N; i++)
{
    for (j = N-1; j > 0; j--)
    {
        A[i][j] = A[i-1][j+1] + 1;
    }
}
```

Fig. 5.1: Loop index reversal transformation.

Table 5.1 shows the impact of applying loop reversal transformation on the execution time, power and energy. Two important facts should be mentioned regarding this transformation. First, this transformation reduces the number of registers in use which is expected to decrease the power consumption. Second, The loop reversal transformation increases the instructions parallelization which has a negative impact on the power consumption. The negative impact of increasing the instructions parallelization on the power consumption compensates the positive effect of reducing the number of registers in use. Thus, the overall power consumption is not enhanced and remains unchanged while the execution time and the energy are enhanced by 3.16%.

But as we mentioned before, the loop reversal still is an important pre-request transformation for some other transformations such as the loop fusion and loop peeling.

Tab. 5.1: Loop reversal transformation effect on energy and power.

	Original	Transformed	%
Exec. Cycles	74 507	72 156	−3.16
Power (W)	0.985	0.985	0.0
Energy (mJ)	0.0734	0.0711	−3.16
IPC	0.905	0.934	3.18

5.2.2 Loop-Based Strength Reduction

Reduction in strength replaces an expression in a loop with one that is equivalent but uses a less expensive operator. Operator strength reduction involves the employment of mathematical identities to replace slow mathematical operations with faster operations. The cost

and benefits will depend highly on the target CPU and sometimes on the surrounding code (depending on availability of other functional units within the CPU).

Figure 5.2 shows an original loop that contains a multiplication operation and a transformed version of the loop where the multiplication is replaced by addition operation.

Original Code

```
for (i = 1; i <= N; i++)
{
    A[i-1] = A[i-1] + c*i;
}
```

Transformed Code

```
T = c;
for (i = 1; i <= N; i++)
{
    A[i-1] = A[i-1] + T;
    T = T + c;
}
```

Fig. 5.2: Loop-based strength reduction transformation.

Table 5.2 represents some examples of the expression strength reduction that replace costly operations such as division and multiplication with less expensive operations such as shifting left or right, subtraction and addition.

Tab. 5.2: Examples of expression strength reduction.

Original	Reduced
$x \times 2$	$x + x$
x^2	$x * x$
$x^{c.5}$	$x^c \times \sqrt{x}$
$i \times 2^c$	$i << c$
$x/8$	$x >> 3$
$x \times 15$	$(x << 4) - x$

Table 5.3 shows the impact of applying the loop-based strength reduction transformation on the power, energy and execution time. Although the power is not enhanced (we cannot consider the decrease of 0.24% as a real enhancement) the execution time is decreased by 14.25% leading to a significant energy saving by almost 14.5%.

Based on our power measurements, we find that the C6416T ISA has no difference between the execution of the ADD and the multiply (MPY) instructions from power consumption perspective. Both the ADD $\&$ MPY instructions with $16-$bit operands consume 941mW at an operating frequency of 850MHz. On the other hand, the MPY instruction takes longer

execution time than the ADD instruction and it is executed only by the .M unit. On the contrary, the ADD instruction can be executed by .L, .S, or the .D functional units. Hence, the loop-based strength reduction allows the compiler to efficiently exploit the processor functional units leading to better instructions parallelization as shown in Table 5.3. Thus, the loop-based strength reduction transformation positively affects the execution time and consequently the energy but keeps the power consumption at the same level.

Tab. 5.3: Loop-based strength reduction transformation impact on power and Energy.

	Original	Transformed	%
Exec. Cycles	3 592	3 080	−14.25
Power (W)	0.994	0.991	−0.24
Energy (mJ)	0.0036	0.0031	−14.46
IPC	1.071	1.125	5.04

5.2.3 Loop Unswitching

This transformation is applied when a loop contains a conditional statement with a loop invariant condition. Thus, the loop unswitching transformation moves the conditional statement outside the loop by duplicating the loop's body inside each branch of the conditional. Hence, this transformation aims to reduce the overhead of unnecessary conditional branches which enables more instructions parallelization that consequently enhance the performance. Figure 5.3 presents an example to demonstrate the idea of the loop unswitching transformation.

Original Code

```
for (i = 0; i < N; i++)
{
    if(a > b)
        C[i] = 0;
    else
        C[i] = 1;
}
```

Transformed Code

```
if(a > b)
{
    for (i = 0; i < N; ++i)
        C[i] = 0;
}
else
{
    for (i = 0; i < N; ++i)
        C[i] = 1;
}
```

Fig. 5.3: Loop unswitching transformation.

Table 5.4 shows the impact of applying the loop unswitching transformation on the power, energy and execution time. Although the number of executed instructions decreases by almost 45% leading to execution time speedup factor of almost two, the power consumption increased by 3.15%. This transformation reduces the number of unnecessary branching instructions that causes the CPU to stall for certain cycles. The main reason for the power increase, due to applying the loop unswitching transformation, is the enhancement of the instructions parallelization by 7.47% in parallel with the significant increase in the L1D cache misses by 44.38%.

Tab. 5.4: Loop unswitching transformation impact on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	29 008	15 008	−48.26
Power (W)	0.973	1.003	3.15
Energy (mJ)	0.0282	0.0151	−46.63
IPC	0.621	0.667	7.47
L1D Cache Misses	676	976	44.38
Executed Instructions	18 008	10 003	−44.40

5.2.4 Loop Permutation

This loop transformation exchanges inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

Figure 5.4 expresses how the loop permutation transformation is applied to two nested loops. Recalling that the C language convention for storing an array in memory is the row-major order (i.e a two dimension array is stored in memory row by row), then the transformed nested loops reduces the stride from stride-N to stride-1.

Definition 1 (stride)**.** The **stride** is the distance in memory between consecutively accessed elements of an array [107].

Reducing the stride from stride-N to stride-1 is expected to enhance the performance and consequently the energy specially when N is too large (in our example the used two dimensional array size is 200×200 with integer data elements).

Table 5.5 shows the impact of applying the loop permutation transformation on the power, energy and execution time. Loop permutation transformation reduces the IPC by 6.4% which

Original Code

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
        A[j][i] = 1;
}
```

Transformed Code

```
for (j = 0; j < N; j++)
{
    for (i = 0; i < N; i++)
        A[j][i] = 1;
}
```

Fig. 5.4: Loop permutation transformation.

in turn decreases the power consumption by 2.76%. But, the bad news is that it increased the execution time by 33.26% which really was not expected. We investigated the generated assembly code to understand the reasons behind the significant execution time increase. Our investigation shows that the executed instructions increase by 24.73% due to improper compiler handling of the storing and retrieving of the array index. This significant increase in the executed instructions in conjunction with the decrease in the IPC causes the previously mentioned execution time increase.

Tab. 5.5: Impact of loop permutation on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	360 208	480 009	33.26
Power (W)	1.044	1.015	−2.76
Energy (mJ)	0.3761	0.4873	29.58
IPC	0.891	0.834	−6.4
Executed Instructions	321 009	400 410	24.73

5.2.5 Loop Peeling

This transformation, also called Loop Splitting, attempts to eliminate or reduce the loop dependencies introduced by the first or last few iterations by splitting these iterations from the loop and perform them outside the loop, thus enabling better instructions parallelization. This transformation also can be used to match the iteration control of adjacent loops allowing the two loops to be fused together as we will see in Section 5.2.7.

Figure 5.5 shows an example of loop peeling transformation. In the original code of this example the first iteration only makes use of the variable $p = 10$, and for all other iterations $p = i - 1$. Therefore, in the transformed code the first iteration is moved outside the loop and the loop iteration control is modified.

Original Code	Transformed Code

`int p = 10;`
`for (i=0; i<N; ++i)`
`{`
`  y[i] = x[i] + x[p];`
`  p = i;`
`}`
	`y[0] = x[0] + x[10];`
	`for (i=1; i<N; ++i)`
	`{`
	`  y[i] = x[i] + x[i-1];`
	`}`

Fig. 5.5: Loop peeling transformation.

Table 5.6 shows the impact of applying the loop peeling transformation on the power, energy and execution time. Because of splitting the first iteration from the loop's body and performing it outside the loop, the memory references decreased by 37.78% maintaining the same number of L1D cache misses. Hence, the execution time and the power consumption are enhanced by 11.5% and 2.78% respectively leading to an energy saving of 13.97%.

Tab. 5.6: Impact of loop peeling transformation on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	2 808	2 485	−11.5
Power (W)	1.034	1.006	−2.78
Energy (mJ)	0.0029	0.0025	−13.97
IPC	0.919	0.962	4.6
Memory References	802	499	−37.78

5.2.6 Loop Fusion

Loop fusion, also called loop jamming, is a type of the loop transformations, which replaces multiple loops with a single one. This transformation aims to reduce the loop overhead (loop index increment or decrement, compare and branch). This transformation also is supposed to enhance the cache and register file utilization. Some other loop transformations, such as loop reversal, loop normalization, or loop peeling, may be applied to make sure that the two loop have the same loop bounds and hence can be fused together.
Figure 5.6 presents an example of two loops that have the same bounds and there are no dependencies between the two loop bodies. Thus, these two loops can legally be fused.

Table 5.7 shows the impact of applying the loop fusion transformation on the power, energy and execution time. As we mentioned before the loop fusion reduces the loop overhead

Original Code

```
for (i = 0; i < N; i++)
{
    A[i] = B[i] + i*d;
    d = B[i+4] + d + A[i]*(i+1);
    A[i] = A[i]*d + B[i+2]*d;
}
for (i = 0; i < N; i++)
{
    f = i*e + C[i];
    e = B[i] + i*f;
    e = e*f + B[i]*e;
    B[i] = (i+2)*e + C[i] - f;
}
```

Transformed Code

```
for (i = 0; i < N; i++)
{
    A[i] = B[i] + i*d;
    d = B[i+4] + d + A[i]*(i+1);
    A[i] = A[i]*d + B[i+2]*d;
    f = i*e + C[i];
    e = B[i] + i*f;
    e = e*f + B[i]*e;
    B[i] = (i+2)*e + C[i] - f;
}
```

Fig. 5.6: Loop fusion transformation.

by a factor of two. Hence, it reduces the registers in use and the executed instructions as well, in the proposed example, by 5.44%. Moreover, the loop fusion reduces the memory references by 14.43% and increase the instructions parallelization represented by the IPC by 3.52%. Thus, the power consumption and the execution time are reduced by 2.3% and 8.66% respectively.

Tab. 5.7: Loop fusion transformation impact on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	5 798	5 296	−8.66
Power (W)	1.042	1.018	−2.3
Energy (mJ)	0.0060	0.0054	−10.76
IPC	1.277	1.322	3.52
Memory References	700	599	−14.43
Executed Instructions	7 405	7 002	−5.44

5.2.7 Loop Peeling and Fusion

As we mentioned in Section 5.2.6, loop fusion may be preceded by one or more other loop transformations to make the loops under investigation legible for fusion. Figure 5.7 shows an example where the two loops are initially not legible for fusion. Hence, loop peeling is first applied to unify the loop bounds (both of the two loops starts from 1 and end at $N - 1$) then loop fusion is applied.

Table 5.8 shows the impact of applying two transformations: the loop peeling and the loop

Original Code

```
int p = 10;
for (i = 0; i < N; ++i)
{
    y[i] = x[i] + x[p];
    p = i;
}
for (i = 1; i < N; ++i)
{
    a[i] = a[i] + c;
}
```

Transformed Code

```
y[0] = x[0] + x[10];
for (i=1; i < N; ++i)
{
    y[i] = x[i] + x[i-1];
    a[i] = a[i] + c;
}
```

Fig. 5.7: Loop peeling and then fusion transformations.

fusion transformations on the power, energy and execution time. Applying both loop peeling and fusion increases the instructions parallelization by 7.22%, while it reduces the memory references and the executed instructions by 31.05% and 17.83% respectively. Although the applied two loop transformations do not significantly enhance the power, it outperforms the individually applied loop transformations from energy perspective. The energy decreases by 23.92% in case of applying loop peeling then fusion, while the energy decreases by 13.97% and 10.76% in case of the individual applying of loop peeling and loop fusion respectively.

Tab. 5.8: Impact of loop peeling then fusion on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	4 793	3 673	−23.37
Power (W)	1.006	0.998	−0.72
Energy (mJ)	0.0048	0.0037	−23.92
IPC	0.833	0.893	7.22
Memory References	1 298	895	−31.05
Executed Instructions	3 993	3 281	−17.83

5.2.8 Loop Normalization and Fusion

Loop normalization, also called loop alignment, converts all loops of a given module into a normal form. In this normal form, the lower bound equals one or zero and the increment in each iteration equals one [107]. Loop normalization may be a pre-request for loop fusion. Figure 5.8 shows an example of the loop normalization followed by loop fusion transformations. First the second loop in the original code is normalized to start from one and hence,

the array index in the body of this loop is modified. Second the two loops are fused together.

Original Code

```
for (i=1; i<N; i++)
{
    x[i] = x[i] + c;
}
for (i=2; i<N+1; i++)
{
    a[i] = x[i-1] + a[i];
}
```

Transformed Code

```
for (i=1; i<N; i++)
{
    x[i] = x[i] + c;
    a[i+1] = x[i] + a[i+1];
}
```

Fig. 5.8: Loop normalization and then fusion transformations.

Table 5.9 shows the impact of applying two transformations: the loop normalization and the loop fusion transformations on the power, energy and execution time. The application of the two transformations greatly affect the parallelization, IPC increases by 51.65%. The memory references and executed instructions are reduced by 20.22% and 28.74% respectively. Derived by the great increase in the IPC, the execution time is significantly reduced by 53% and consequently the energy is reduced by 51.12%. The negative impact of the IPC increase on the power consumption overrides the positive impact of reducing the memory reference and the executed instructions and causes the power consumption to increase by 4%.

Tab. 5.9: Influence of loop normalization then fusion transformations on the energy and power consumption.

	Original	Transformed	%
Exec. Cycles	3 022	1 420	−53.01
Power (W)	0.986	1.026	4.01
Energy (mJ)	0.00298	0.00146	−51.12
IPC	0.619	0.939	51.65
Memory References	445	355	−20.22
Executed Instructions	1 872	1 334	−28.74

5.2.9 Loop Unrolling

Loop unrolling, also known as Loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its size. Loop unrolling

replicates the body of a loop some number of times called the unrolling factor (u) and iterates by step u instead of step one. Loop unrolling improves the performance by reducing the loop overhead, effective exploitation of ILP from different iterations, and improving register and data cache locality [107, 108].

Figure 5.9 illustrates an example of the loop unrolling in which the loop unrolling factor (u) equals eight. The loop performs histogram of an input image of size 8192 pixels. It is pretty clear that the code size significantly increased while the overhead of the loop represented in the number of executed branches is significantly reduced.

Original Code

```
for (i = 0; i < n; i++)
{
    hist[image[i]%256]++;
}
```

Transformed Code

```
for (i = 0; i < n; i+=8)
{
    hist[image[i+0]%256]++;
    hist[image[i+1]%256]++;
    hist[image[i+2]%256]++;
    hist[image[i+3]%256]++;
    hist[image[i+4]%256]++;
    hist[image[i+5]%256]++;
    hist[image[i+6]%256]++;
    hist[image[i+7]%256]++;
}
```

Fig. 5.9: Loop unrolling transformation with unrolling factor of 8.

Table 5.10 shows the impact of applying the loop tiling transformations on the power, energy and execution time. Loop unrolling enhances the instructions parallelism by 13.2% which significantly reduces the execution time by 32.76% and consequently saves the energy by 32.55%. The memory references as well as the executed instructions are reduced by 29.14% and 23.93% respectively, which almost maintains the power consumption at the same level.

Tab. 5.10: Impact of loop unrolling transformation on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	303 220	203 894	−32.76
Power (W)	0.955	0.958	0.31
Energy (mJ)	0.2895	0.1952	−32.55
IPC	0.649	0.734	13.2
Memory References	73 797	52 295	−29.14
Executed Instructions	196 788	149 687	−23.93

5.2.10 Loop Tiling

Loop tiling, also called loop blocking, partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of large arrays into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements. It also can be used to enhance the processor register file [107].
The loop tiling transformation is essential for enhancing the utilization of data cache in dense matrix applications. The loop tiling can be followed by loop permutation for the innermost loops to increase the parallelism, the outmost loops can also be interchanged to enhance locality across tiles.
Figure 5.10, the original code, illustrates the need of loop tiling. The innermost loop access to array B is stride-N, while access to array A is stride-1. Thus loop permutation does not help. Moreover, the original loop iteration space is N by N. The accessed chunk of array B[j][i] is also N by N. When N is too large, in our example the matrix size is 100×100, and the cache size of the machine is too small. The accessed array elements in one loop iteration (for example, i=1, j=1 to N) may cross the cache lines, causing cache misses. By iterating over smaller chunks of the iteration space as shown in the transformed code in Fig. 5.10, the loop efficiently uses the cache line.

Original Code

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
    {
        A[i][j] = B[j][i];
    }
```

Transformed Code

```
for (i=0; i<N; i+=64)
    for (j=0; j<N; j+=64)
        for (ii=i; ii<min(i+64,N); ii++)
            for (jj=j; jj<min(j+64,N); jj++)
            {
                A[ii][jj] = B[jj][ii];
            }
```

Fig. 5.10: Loop tiling transformation.

Table 5.11 shows the impact of applying the loop tiling transformations on the power, energy and execution time. It is pretty clear that the loop tiling transformation enhances the cache locality, as the L1D cache misses decrease by 27.52%. The enhancement of the cache locality directly leads to a power saving by 2.95%. On the other hand, the instructions parallelism is significantly increased by 38.41%, which is expected to significantly enhance the execution time. But the number of executed instructions increases by 47.88%, due to new

inserted loops, which consequently override the enhancement in the parallelism. Hence, the execution time increases by 6.84% leading the energy to increase by 3.69%.
However, loop tilling increases the energy, it is still a good transformation for the cases when the data cache size is small and the array dimensions are extremely large.

Tab. 5.11: Impact of loop tiling transformation on energy and power consumption.

	Original	Transformed	%
Exec. Cycles	182 274	194 745	6.84
Power (W)	1.058	1.027	−2.95
Energy (mJ)	0.1929	0.20	3.69
IPC	0.825	1.141	38.41
L1D Cache Misses	12 226	8 862	−27.52
Executed Instructions	150 306	222 273	47.88

5.3 Data Oriented Transformations

In this section we present some transformations that are mainly concerned with the data structures, access modes and the declaration scope of the data variables or arrays. This kind of transformation aims to maximize the register file exploitation and to reduce the memory and cache accesses.

5.3.1 Array Declaration Sorting

The basic idea is to modify the local array declaration ordering, so that the arrays more frequently accessed are placed on top of the stack; in such a way, the memory locations frequently used are accessed by exploiting direct access mode.
In particular, the arrays are allocated in the stack following the order of declaration and the first array is accessed by offset addressing with constant 0, while the others use non-0 constants [83].
Figure 5.11 shows an example where the array access frequency ordering is C[], B[] and A[]: the declaration order, in the original code A[], B[], and C[], is restructured placing C[] in the first position, B[] in the second one and A[] at the end.

Original Code

```
int A[DIM], B[DIM], C[DIM], i;
    for (i = 5; i < 3500; i+=5)
        C[i] = val;
    for (i = 5; i < 2000; i+=10)
        B[i] = val;
    for (i = 5; i < 1000; i+=10)
        A[i] = val;
}
```

Transformed Code

```
int C[DIM], B[DIM], A[DIM], i;
    for (i = 5; i < 3500; i+=5)
        C[i] = val;
    for (i = 5; i < 2000; i+=10)
        B[i] = val;
    for (i = 5; i < 1000; i+=10)
        A[i] = val;
```

Fig. 5.11: Array declaration sorting transformation.

The array declaration sorting reduces the execution time by 1.95% and consequently saves the energy by 2.19%. The power consumption is almost not affected, hence this transformation is not a power hungry transformation.

5.3.2 Array Elements Scalarization

This transformation introduces a set of temporary variables as a substitute of the more frequently used elements of an array. It allows the compiler to optimize the computation by utilizing the processor registers. Figure 5.12 illustrates an example of the scalarization of array elements. In the transformed code three scalar variables, $t0$, $t1$ and $t2$ are inserted to replace the frequent array referencing.

Table 5.12 shows the impact of applying the array elements scalarization transformation on the power, energy and execution time. The array size in our example is 50 but we also verified the results with array sizes of 500 and 5000. The scalarization of array elements transformation increases the IPC by 9.13% and hence, the execution time is reduced by 5.46%. The insertion of new variables increases the number of executed instructions by 3.17%. The compiler did not properly utilize the processor registers to handle the new inserted scaler variables and thus, the number of memory references increases by 35.3%. Therefore, the power consumption increases by 2.41%.

Original Code

```
int i;
int B[DIM], A[DIM];
for (i = 0; i < N; i++)
{
    B[i] = i;
    A[i] = i;
}
for (i = 2; i < M; i++)
{
    B[i] = (A[i] + B[i])/2;
    if(i % 2 == 0)
    {
        A[i] = A[i-2] + 1;
    }
    else
    {
        A[i] = A[i-1] -1;
    }
}
```

Transformed Code

```
int i;
int B[DIM], A[DIM];
int t2, t1, t0;
for (i = 0; i < N; i++)
{
    B[i] = i;
    A[i] = i;
}
t2 = A[0];
t1 = A[1];
for (i = 2; i < M; i++)
{
    t0 = A[i];
    B[i] = (t0 + B[i])/2;
    if(i % 2 == 0)
    {
        t0 = t2 +1;
    }
    else
    {

        t0 = t2 -1;
    }
    A[i] = t0;
    t2 = t1;
    t1 = t0;
}
```

Fig. 5.12: Array elements scalarization transformation.

Tab. 5.12: Influence of array elements scalarization transformation on the energy and power consumption.

	Original	Transformed	%
Exec. Cycles	3 372	3 188 715	−5.46
Power (W)	0.944	0.967	2.41
Energy (mJ)	0.00318	0.00308	−3.17
IPC	0.738	0.806	9.13
Memory References	660	893	35.3
Executed Instructions	2 490	2 569	3.17

5.4 Procedural and Inter-Procedural Transformations

5.4.1 Procedure Call Preprocessing

This transformation associates with a specific function a proper set of macros that will substitute a function call with either an equivalent but low energy function call or a specific result;

in short, the transformation skips a function call, or reduces its impact, when its actual parameters allow to directly identify either the returned value or another equivalent function. Figure 5.13 illustrates a meaningful example of this transformations where two functions, sqrt() that computes the square root of an integer value and fabs() that computes the absolute value of a floating-point number, are predefined as macros.

Original Code

```
#include <math.h>
main()
{
int i, val;
float r1, r2;
    for( i = 1; i < N; i++ )
    {
        val = i % 5;
        r1 = sqrt(val);
    }
    for( i = -20; i < M; i++ )
    {
    r2 = fabs(val);
    }
}
```

Transformed Code

```
#include <math.h>
#define sqrt(x) ((x==0)?0:(x==1)?1:sqrt(x))
#define fabs(x) ((x>=0)?x:-x)
main()
{
int i, val;
float r1, r2;
    for( i = 1; i < N; i++ )
    {
        val = i % 5;
        r1 = sqrt(val);
    }
    for( i = -20; i < M; i++ )
    {
        r2 = fabs(val);
    }
}
```

Fig. 5.13: Procedure call preprocessing transformation.

Table 5.13 shows the impact of applying the procedure call preprocessing transformations on the power, energy and execution time. Procedure call preprocessing reduces the execution time by 23.74% and consequently saves the energy by 24%. The memory references as well as the executed instructions are reduced by 24.21% and 23.36% respectively, which almost maintains the power consumption at the same level.

Tab. 5.13: Influence of procedure call preprocessing transformations on the energy and power consumption.

	Original	Transformed	%
Exec. Cycles	503 728	384 123	−23.74
Power (W)	1.074	1.070	−0.34
Energy (mJ)	0.5410	0.4112	−24.0
IPC	1.255	1.261	0.5
Memory References	126 187	95 640	−24.21
Executed Instructions	632 224	484 513	−23.36

5.4.2 Procedure Integration

Procedure integration, also called procedure inlining, replaces calls to procedures with copies of their bodies [107]. It can be a very useful optimization, because it changes calls from opaque objects that may have unknown effects on aliased variables and parameters to local code that not only exposes its effects but that can be optimized as part of the calling procedure [105].
Although procedure integration removes the cost of the procedure call and return instructions, these are often small savings. The major savings often come from the additional optimizations that become possible on the integrated procedure body: for example, a constant passed as an argument can often be propagated to all instances of the matching parameter. Moreover, the opportunity to optimize integrated procedure bodies can be especially valuable if it enables loop transformations (refer to Section 5.4) that were originally inhibited by having procedure calls embedded in loops or if it turns a loop that calls a procedure, whose body is itself a loop, into a nested loop [105].

Ordinarily, when a function is invoked, control is transferred to its definition by a branch or call instruction. With procedure integration, control flows directly to the code for the function, without a branch or call instruction. Moreover, the stack frames for the caller and callee are allocated together. Procedure integration may make the generated code slower as well: for instance, by decreasing locality of reference.
Figure 5.14 shows an example of the use of procedure integration. In this example the function pred(int) is integrated in the function f(int).

Table 5.14 shows the impact of applying the procedure integration transformations on the power, energy and execution time. As we mentioned before the procedure integration eliminate the call overhead and hence, reduce the memory references in the proposed example by 12.44%. Moreover, the procedure integration reduces the executed instructions by 41.11% and the IPC by 12.59%. Thus, the power consumption and the execution time are reduced by 3.93% and 32.63% respectively.

Finally, Fig. 5.15 summarizes the results of applying different code transformations on the power, execution time, and energy. In Fig. 5.15 the original code represents the 100% hence, the deviation above or under 100% is related to the applied code transformation.

Original Code

```
main()
{
int i,res[DIM] , val = 7;
    for( i = 0; i < N; i++ )
    {
        res[i] = f(val);
        val += 5;
    }
}

int f(int y)
{
    return pred(y) + pred(0)
           + pred(y+1);
}

int pred(int x)
{
    if (x == 0)
        return 0;
    else
        return x-1;
}
```

Transformed Code

```
main()
{
int i,res[DIM] , val = 7;
    for( i = 0; i < N; i++ )
    {
        res[i] = f(val);
        val += 5;
    }
}

int f(int y)
{
    int temp = 0;
    if (y == 0) temp += 0;
    else temp += y - 1;
    if (0 == 0) temp += 0;
    else temp += 0 - 1;
    if (y+1 == 0) temp += 0;
    else temp += (y+1) - 1;
    return temp;
}
```

Fig. 5.14: Procedure integration transformation.

Tab. 5.14: Influence of procedure integration transformations on the energy and power consumption.

	Original	Transformed	%
Exec. Cycles	3 218	2 168	−32.63
Power (W)	1.039	0.998	−3.93
Energy (mJ)	0.0033	0.0022	−35.27
IPC	0.983	0.859	−12.59
Memory References	804	704	−12.44
Executed Instructions	3 162	1 862	−41.11

The results show that several code transformations have good impact on power consumption, energy and performance such as loop peeling, loop fusion and procedure integration. While other transformations improve the power consumption on the account of the performance such as loop permutation and loop tilling. The results also show that some transformations have no impact on the power consumption but they improve the performance and energy. This type of transformations are not power hungry transformations such as loop reversal, loop strength reduction and array declaration sorting. The last type of the code transformations are those which improve the performance on the account of the power consumption such as loop unswitching, loop normalization then fusion and the scalarization of array ele-

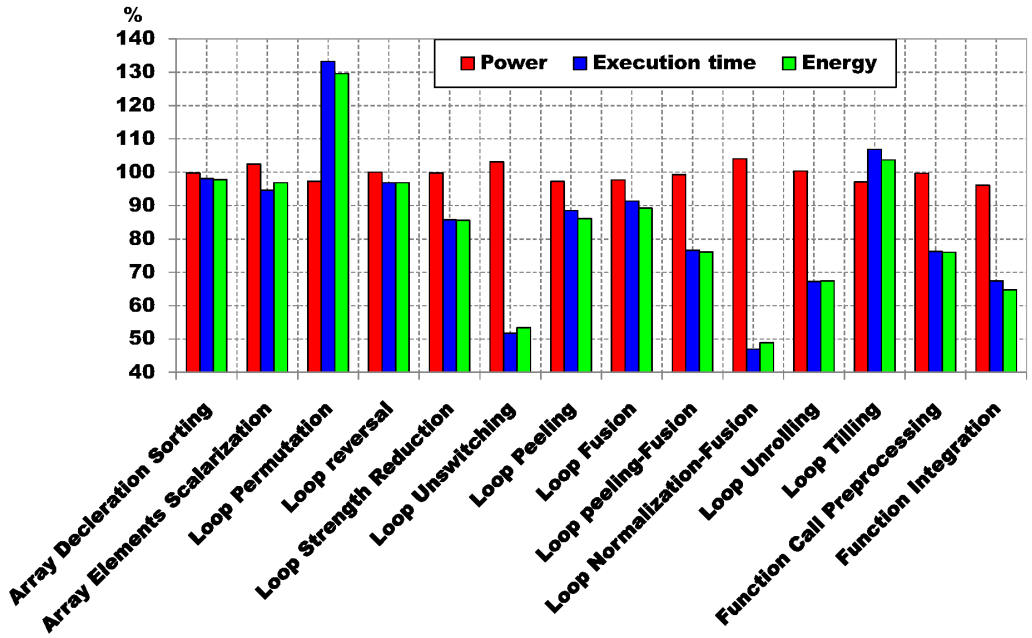

Fig. 5.15: Code transformations impact on power, execution time and energy.

ments.

5.5 Conclusions

The CCS allows very limited control over the individual optimizations embedded within each global optimization level. Thus, in this chapter we assess the effect of applying source to source code transformations on the power, energy and performance. The source code transformations that are presented in this work are classified into three major groups: loop, data, and procedural transformations. To evaluate the effectiveness of the applied transformations we compile each program, both the original and transformed version, on the target architecture (C6416T DSK). Next, we record the current drawn from the core CPU and hence the consumed power. With the aid of the compiler's profiler we also record the run time and other execution characteristics such as memory references, L1D cache misses and so on. To obtain reliable and precise information, we repeat the whole measuring procedure for each transformation multiple times.

The results show that several code transformations have good impact on power consumption, energy and performance such as loop peeling, loop fusion and procedure integration while other transformations improve the power consumption on the account of the performance such as loop permutation (due to the inappropriate compiler handling of the storing

and retrieving of the array index) and loop tilling (due to overhead of extra inserted loops). The results also show that some transformations have no impact on the power consumption but they improve the performance and energy. This type of transformations is not power hungry such as loop reversal, loop strength reduction and array declaration sorting. Other transformations such as loop unswitching, loop normalization then fusion and scalarization of array elements enhance the execution time on the account of the power consumption.

6 CONCLUSIONS

6.1 Summary and Conclusions

The importance of power reduction of embedded systems has continuously increased in the past years. Recently, reducing power dissipation and energy consumption of a program have become optimization goals in their own right, no longer considered as side-effect of traditional performance optimizations which mainly target program execution time and/or program size. Nowadays, there is an increasing demand for developing power-optimizing compilers for embedded systems. This thesis is a step towards such important goal.

In this thesis, we develop functional-level power models and investigate several software optimization techniques for embedded-processor systems. As a specific example, we consider the powerful Texas Instruments C6416T DSP processor. We analyze the power consumption contributions of the different functional units of this DSP. We assess the effect of the compiler performance optimizations on the energy and power consumption. Moreover, we explore the impact of two special architectural features of this DSP; namely Software Pipelined Loop and the SIMD capabilities, on the energy and power consumption.

The currently-available compiler optimization techniques target execution time and rarely improve power consumption. These techniques are handicapped for power optimization due to their partial perspective of the algorithms and due to their limited modifications to the data structures. On the contrary, other software optimization techniques, like source code transformations, can exploit the full knowledge of the algorithm characteristics, with the capability of modifying both data structures and algorithm coding. Furthermore, inter-procedural optimizations are envisioned. Hence, we investigate several loop, data and procedural source code transformations from the power and energy perspectives. This is based on several unique contributions:

- The development of a precise functional-level estimation technique to estimate the

power consumption of the embedded software running on a programmable processor. The commercial off-the-shelf VLIW DSP C6416T from Texas Instruments is utilized as the targeted platform. The inter-instructions as well as the pipeline stall effects have been investigated in our proposed model. The validation and precision of our model have been proven by estimating the power consumption of many typical algorithms applied in signal and image processing as well as a real embedded application. The power consumption estimated by our model, is compared to the physically measured power consumption, achieving a very low absolute average estimation error of 1.6% and an absolute maximum estimation error of only 3.3%.

- The exploration of power and performance trade-offs for the targeted architecture. The compiler used to generate the code binaries is the embedded C/C++ compiler Ver.$6.0.1$ in the CCS3.1. The effect of invoking the global performance optimization options -o0 to -o3 on the power and energy consumption has been evaluated. The results show that the most aggressive performance optimization option -o3 reduces the execution time, on average, by 96.2%, while it increases the power consumption by 30.3%. Due to the perfect correlation between execution time and energy, we find that the energy is significantly decreased, on average, by 94.8%. To investigate the cause of this power increase we inspect the optimizations effect on some other performance measures, such as the memory references and the IPC. Despite the decrease of the memory references by 94%, the IPC increases by 260% and consequently increases the consumed power by 30.3% which emphasizes our results in [95] that the IMU contribution to the total DSP core power consumption dominates the internal memory referencing contribution.

- The evaluation of the SPLOOP, specific C64x+ architectural feature, effect on the energy and power consumption. The results show that the SPLOOP feature contributes, on average, by almost 70.3% to the total power consumption increase when -o3 is invoked.

- The investigation of the impact of utilizing the targeted architecture SIMD capabilities on the power and energy. The results show that employing the SIMD, in general, has a significant impact on the power consumption, execution time and consequently on the energy. Invoking -o3-mu (invoking -o3 while disabling the SPLOOP feature), in

conjunction with the employment of SIMD, is the best choice from the power consumption perspective. Meanwhile, it also can be considered as a trade-off between execution time and the power consumption.

- The characterization of the application-architecture correlation for the targeted platform. The PCA multivariate statistical technique is employed to visualize the black box impact of the compiler and the hardware architecture over the software applications. This is achieved with the aid of biplots which is depicted in our analysis in such a way, so that it can show the maximum association between the application and the underlying hardware architecture. Hence, it answers the question whether a given hardware architecture is an appropriate choice for a given software application or not.

- The assessment of the effect of applying source code transformations on the power, energy and performance. The source code transformations that are presented in this work are classified into three major groups: loop, data, and procedural transformations. The results show that several code transformations have a good impact on power consumption, energy and performance such as loop peeling, loop fusion and procedure integration while other transformations improve the power consumption on the account of the performance such as loop permutation (due to the inappropriate compiler handling of the storing and retrieving of the array index) and loop tilling (due to overhead of extra inserted loops). The results also show that some transformations have no impact on the power consumption but they improve the performance and energy. This category of transformations is not power hungry such as loop reversal, loop strength reduction and array declaration sorting. Other transformations such as loop unswitching, loop normalization then fusion and scalarization of array elements enhance the execution time on the account of the power consumption.

Based on our results and as a step towards a power-aware optimizing compiler, we can recommend the following recommendations for programmers and compiler designers.
First, the programmers, targeting the C6000 DSP family, are strongly recommended to compile and optimize their programs by invoking -o3 while disabling the SPLOOP feature (-mu) in conjunction with the utilization of SIMD capabilities via the employment of suitable intrinsic functions.

Second, we recommend the compiler designers to pay more attention to the circular (modulo) and bit reverse addressing schemes which are rarely utilized by the compiler. In addition, they should utilize the power-aware source code transformations.

Third, developers of power simulators need to embed a functional level power consumption model for the target processor in their simulators software.

6.2 Remarks for Future Work

A number of interesting topics for the future based on the work accomplished in this thesis can be identified:
With respect to the developed power consumption model the methodology can be applied for other processors to build a library. This library can be integrated in a power estimation framework to facilitate an early estimation of the power consumption. In order to obtain higher accuracy regarding the power estimation methodology a combined approach of functional-level and instruction-level power analysis techniques might be applicable.

Developing a tool that statistically analyze the application program codes to automatically compute the required algorithmic parameters for our developed power consumption model.

The qualitative analysis of the source code transformations effect on the power and energy indicates that some transformations are promising from power reduction perspective. Further research regarding the automation of the process of applying these transformations can speed up the optimization process and lead to further more power and energy savings.

BIBLIOGRAPHY

[1] M. Barr and A. Massa, *Programming Embedded Systems: with C and GNU Development Tools*. O'Reilly Media Inc., 2006.

[2] M. Platzner and L. Thiele, "Hardware/Software Codesign," Lectures, 2005, http://www.cs.uni-paderborn.de/fachgebiete/computer-engineering-group/teaching/ss08/hscdvu.html.

[3] G. DeMicheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Co-Design*. San Francisco, CA, USA: Morgan Kaufman Publishers, Academic Press, 2002.

[4] L. Geppert, "High-Flying DSP Architectures," *IEEE Spectrum*, vol. 35, no. 11, pp. 53–56, 1998.

[5] V. Zivojnovic, S. Pees, and H. Meyr, "LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design," in *proceedings of the IEEE Workshop on VLSI Signal Processing*, San Francisco, October 1996.

[6] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Norwell, MA, USA: Kluwer Academic Publishers, December 2002.

[7] CoWare Inc., "Processor Designer," 2005, http://www.coware.com/products/processordesigner.php.

[8] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[9] R. Zurawski, *Embedded Systems Handbook*. Boca Raton, FL, USA: CRC Press, Inc., 2004.

[10] C. Talarico, J. W. Rozenblit, V. Malhotra, and A. Stritter, "A New Framework for Power Estimation of Embedded Systems," *IEEE Computer*, vol. 38, no. 2, pp. 71–78, 2005.

[11] D. Shefer, "Non Recurring Engineering," *http://www.shefer.net/articles.html*, 2005.

[12] J. Plusquellic, C. Kief, and S. Suddarth, "Hardware/Sofware Codesign with FPGAs: Embedded Systems Design," Lectures, October 2008, http://www.ece.unm.edu/faculty/jimp/codesign/.

[13] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning of Hardware-Software Embedded Systems: A Metrics-Based Approach," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 39–56, 1998.

[14] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[15] V. Gutnik and A. P. Chandrakasan, "Embedded Power Supply for Low-Power DSP," *IEEE Transactions of VLSI Systems*, vol. 5, pp. 425–35, 1997.

[16] J. Lloyd, "Electromigration for Designers: An Introduction for the Non-Specialist," http://www.simplex.com/udsm/whitepapers/electromigration1/index.html.

[17] R. Ludeke, "Hot-Electron Effects and Oxide Degradation in MOS Structures Studied with Ballistic Electron Emission Microscopy," *IBM Journal of Research and Development*, vol. 44, no. 4, pp. 517–534, 2000.

[18] G. Moore, "Moore's Law," http://www.intel.com/technology/mooreslaw/.

[19] M. Lorenz, P. Marwedel, T. Dräger, G. Fettweis, and R. Leupers, "Compiler based exploration of DSP energy savings by SIMD operations," in *proceedings of the conference on Asia South Pacific Design Automation ASP-DAC'04*. Piscataway, NJ, USA: IEEE Press, 2004, pp. 838–841.

[20] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *proceedings of the 36th ACM/IEEE conference on Design automation DAC'99*. New York, NY, USA: ACM, 1999, pp. 151–156.

[21] P. S. Diniz, *Adaptive Filtering: Algorithms and Practical Implementation*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[22] C. J. Bleakley, M. Casas-Sanchez, and J. Rizo-Morente, "Software Level Power Consumption Models and Power Saving Techniques for Embedded DSP Processors," *Journal of Low Power Electronics*, vol. 2, no. 2, pp. 281–290, 2006.

[23] C. Brandolese, "A Codesign Approach to Software Power Estimation for Embedded Systems," PhD Disseration, Politecnico di Milano, Institute of Electronics and Information, 2000.

[24] C. X. Huang, B. Zhang, A. Deng, and B. Swirski, "The design and implementation of PowerMill," in *proceedings of the International Symposium on Low Power Design ISLPED'95*. New York, NY, USA: ACM, 1995, pp. 105–110.

[25] F. N. Najm, "A Survey of Power Estimation Techniques in VLSI Circuits," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 446–455, 1994.

[26] S. Gupta and F. N. Najm, "Power macromodeling for high level power estimation," in *proceedings of the 34th annual conference on Design automation DAC'97*. New York, NY, USA: ACM, 1997, pp. 365–370.

[27] T. Chou and K. Roy, "Accurate Estimation of Power Dissipation in CMOS Sequential Circuits," *IEEE Transaction VLSI Systems*, vol. 4, pp. 369–380, September 1996.

[28] D. Marculescu, R. Marculescu, and M. Pedram, "Information theoretic measures of energy consumption at register transfer level," in *proceedings of the International Symposium on Low Power Design ISLPED'95*. New York, NY, USA: ACM, 1995, pp. 81–86.

[29] J. N. Rabaey and M. Pedram, *Low Power Design Methodologies*. The Springer International Series in Engineering and Computer Science, 1996, vol. 336.

[30] S. Powell and E. M. Chau, "Estimating power dissipation of VLSI signal processing chips: the PFA technique," in *VLSI Signal Processing IV*, 1990, pp. 250–259.

[31] N. Kumar, S. Katkoori, L. Rader, and R. Vemuri, "Profile-Driven Behavioral Synthesis for Low-Power VLSI Systems," *IEEE Design and Test*, vol. 12, no. 3, pp. 70–84, 1995.

[32] D. Liu and C. Svensson, "Power Consumption Estimation in CMOS VLSIs Chips," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 6, pp. 663–670, 1994.

[33] P. E. Landman and J. M. Rabaey, "Activity-Sensitive Architectural Power Analysis for the Control Path," in *proceedings of the International Symposium on Low Power Design ISLPED'95*. New York, NY, USA: ACM, 1995, pp. 93–98.

[34] H. Mehta, R. M. Owens, and M. J. Irwin, "Energy Characterization Based on Clustering," in *proceedings of the conference on Design automation DAC'96*. New York, NY, USA: ACM, 1996, pp. 702–707.

[35] Q. Wu, Q. Qiu, M. Pedram, and C.-S. Ding, "Cycle-Accurate Macro-Models for RT-Level Power Analysis," *IEEE Transaction VLSI Systems*, vol. 6, no. 4, pp. 520–528, 1998.

[36] L. Benini, A. Bogliolo, M. Favalli, and G. De Micheli, "Regression Models for Behavioral Power Estimation," *Integrated Computer-Aided Enggineering*, vol. 5, no. 2, pp. 95–106, 1998.

[37] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool," in *proceedings of the 37th conference on Design automation DAC'2000*. New York, NY, USA: ACM, 2000, pp. 340–345.

[38] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," in *proceedings of the 8th International Symposium on High-Performance Computer Architecture HPCA'02*. Washington, DC, USA: IEEE Computer Society, February 2002, pp. 141–151.

[39] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 83–94, 2000.

[40] M. B. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low-Power Caches," in *proceedings of the International Symposium on Low Power Electronics and Design ISLPED'97*. New York, NY, USA: ACM, 1997, pp. 143–148.

[41] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[42] R. Joseph, D. Brooks, and M. Martonosi, "Runtime Power Measurements as a Foundation for Evaluating Power/Performance Tradeoffs," in *proceedings of the Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA'01*, June 2001.

[43] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin, and A. Sivasubramaniam, "VEC: Virtual Energy Counters," in *proceedings of the workshop on Program analysis for software tools and engineering PASTE'01*. New York, NY, USA: ACM, 2001, pp. 28–31.

[44] M. Pedram, *Power Aware Design Methodologies*, J. M. Rabaey, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[45] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software a First Step Towards Software Power Minimization," *IEEE Transaction on VLSI Systems*, pp. 437–445, December 1994.

[46] S. Nikolaidis, N. Kavvadias, P. Neofotistos, K. Kosmatopoulos, T. Laopoulos, and L. Bisdounis, "Instrumentation Set-up for Instruction Level Power Modeling," in *proceedings of the 12th International Workshop on Power and Timing Modeling, Optimization and Simulation PATMOS'02*. London, UK: Springer-Verlag, 2002, pp. 71–80.

[47] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas, "Instruction Level Energy Modeling for Pipelined Processors," *Journal of Embedded Computing*, vol. 1, no. 3, pp. 317–324, 2005.

[48] B. Klass, D. E. Thomas, H. Schmit, and D. F. Nagle, "Modeling Inter-Instruction Energy Effects in a Digital Signal Processor," in *Power Driven Microarchitecture Workshop in conjunction with International Syymposism Computer Architecture*, June 1998.

[49] A. Sama, J. F. M. Theeuwen, and M. Balakrishnan, "Speeding up Power Estimation of Embedded Software," in *proceedings of the International Symposium on Low Power Electronics and Design ISLPED'00*. New York, NY, USA: ACM, 2000, pp. 191–196.

[50] J. T. Russell and M. F. Jacome, "Software Power Estimation and Optimization for High Performance 32-bit Embedded Processors," in *proceedings of the International Conference on Computer Design ICCD'98*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 328–333.

[51] H. Mehta, R. M. Owens, and M. J. Irwin, "Instruction Level Power Profiling," in *proceedings of the International Conference of Acoustics, Speech, and Signal Processing ICASSP'96*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 3326–3329.

[52] V. Steven, R. Gentile, D. R. Kaeli, and G. Olivadoti, "Developing Energy-Aware Strategies for the Blackfin Processor," in *Proceedings of the High Performance Embedded Computing (HPEC'04)*, Septemper 2004.

[53] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "An Instruction-Level Energy Model for Embedded VLIW Architectures," *IEEE Transaction on CAD of Integrated Circuits and Systems*, vol. 21, no. 9, pp. 998–1010, 2002.

[54] M. Balakrishnan, "Low Power Design," Lectures, 2008, http://embedded.cse.iitd.ernet.in/homepage/course/low_power/index.shtml.

[55] J. Laurent, E. Senn, N. Julien, and E. Martin, "High Level Energy Estimation for DSP Systems," in *proceedings International Workshop on Power And Timing Modeling and Optimization and Simulation PATMOS'01*, September 2001, pp. 311–316.

[56] E. Senn, N. Julien, J. Laurent, and E. Martin, "Power Consumption Estimation of a C Program for Data-Intensive Applications," in *proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation PATMOS'02*. London, UK: Springer-Verlag, 2002, pp. 332–341.

[57] M. Schneider, H. Blume, and T. G. Noll, "Power Estimation on Functional Level for Programmable Processors," in *journal of Advances in Radio Science*, vol. 2, May 2005, pp. 215–219.

[58] T. Arslan, A. T. Erdogan, and D. H. Horrocks, "Low Power Design for DSP: Methodologies and Techniques," *Microelectronics Journal*, vol. 27, no. 8, pp. 731–744, 1996.

[59] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low Power CMOS Digital Design," *IEEE Journal of Solid State Circuits*, vol. 27, pp. 473–484, 1992.

[60] T. Pering, T. Burd, and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," in *proceedings of the International Symposium on Low Power Electronics and Design ISLPED'98*. New York, NY, USA: ACM, 1998, pp. 76–81.

[61] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power Optimization of Variable Voltage Core-Based Systems," in *proceedings of the 35th annual conference on Design automation DAC'98*. New York, NY, USA: ACM, 1998, pp. 176–181.

[62] M. Pedram, "Power Optimization and Management in Embedded Systems," in *proceedings of the conference on Asia South Pacific design automation ASP-DAC'01*. New York, NY, USA: ACM, 2001, pp. 239–244.

[63] Intel Corporation, "www.intel.com."

[64] International Business Machines(IBM) Corporation, "www.ibm.com."

[65] Novafora Inc., "www.novafora.com."

[66] C. F. Chiasserini and R. R. Rao, "Pulsed Battery Discharge in Communication Devices," in *proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking MobiCom'99*. New York, NY, USA: ACM, 1999, pp. 88–95.

[67] P. Rong and M. Pedram, "Battery-Aware Power Management Based on Markovian Decision Processes," in *proceedings of the 2002 IEEE/ACM international conference on Computer-aided design(ICCAD'02)*. New York, NY, USA: ACM, 2002, pp. 707–713.

[68] D. J. Kolson, A. Nicolau, N. Dutt, and K. Kennedy, "Optimal Register Assignment to Loops for Embedded Code Generation," *ACM Transaction on Desgin Automation of Electronic Systems*, vol. 1, no. 2, pp. 251–279, 1996.

[69] M. O. Tokhi, M. A. Hossanin, and M. H. Shaheed, *Parallel Computing for Real-Time Signal Processing and Control*, 1st ed. London, UK: Springer-Verlag, March 2003.

[70] T. V. K. Gupta, R. E. Ko, and R. Barua, "Compiler-Directed Customization of ASIP Cores," in *proceedings of the tenth international symposium on Hardware/software codesign CODES'02*. New York, NY, USA: ACM, 2002, pp. 97–102.

[71] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview," 1994, pp. 38–39.

[72] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, vol. 1, no. 3, pp. 65–75, 1984.

[73] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," in *proceedings of the 36th Annual Symposium on Foundations of Computer Science FOCS'95*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–382.

[74] M. T.-C. Lee, M. Fujita, V. Tiwari, and S. Malik, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Transaction VLSI Systems*, vol. 5, no. 1, pp. 123–135, 1997.

[75] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of Compiler Optimizations on System Power," *IEEE Transaction VLSI Systems*, vol. vol. 9, pp. 801–804, 2001.

[76] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler Optimizations for Low Power Systems," *Chapter 10, Robert Graybill and Rami Melhem: Power aware computing*, pp. 191–210, 2002.

[77] M. Valluri and L. John, "Is Compiling for Performance == Compiling for Power?" in *proceedings of the 5th Workshop on Interaction between Compilers and Computer Architectures INTERACT'01*, Monterrey, Mexico, Januarry 2001.

[78] L. N. Chakrapani, "The Emerging Power Crisis in Embedded Processors: What Can a Poor Compiler Do," in *proceedings of the international conference on Compilers, Architecture, and Synthesis for Emdedded SystemsCASES'01*, November 2001.

[79] J. S. Seng and D. M. Tullsen, "The Effect of Compiler Optimizations on Pentium 4 Power Consumption," in *proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures INTERACT'03*. Washington, DC, USA: IEEE Computer Society, February 2003, pp. 51–56.

[80] Z. N. Azeemi and M. Rupp, "Energy-Aware Source-to-Source Transformations for a VLIW DSP Processor," in *proceedings of the 17th ICM'05*, Islamabad, Pakistan, December 2005, pp. 133–138.

[81] M. Casas-Sanchez, J. Rizo-Morente, C. Bleakley, and J. Gonzalez, "Effect of Compiler Optimizations on DSP Processor Power and Energy Consumption," in *proceedings of the conference on Design of Circuits and Integrated Systems DCIS'07*, Seville, Spain, November 2007.

[82] E. Macii, M. Pedram, and F. Somenzi, "High-Level Power Modeling, Estimation, and Optimization," in *proceedings of the 34th annual conference on Design automation (DAC'97)*. New York, NY, USA: ACM, 1997, pp. 504–511.

[83] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The Impact of Source Code Transformations on Software Power and Energy Consumption," *Journal of Circuits, Systems, and Computers*, vol. 11, no. 5, pp. 477–502, 2002.

[84] D. Ortiz and N. Santiago, "Impact of Source Code Optimizations on Power Consumption of Embedded Systems," June 2008, pp. 133–136.

[85] F. Catthoor, K. Danckaert, S. Wuytack, and N. D. Dutt, "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 70–82, 2001.

[86] C. Kulkarni, F. Catthoory, and H. De Man, "Code Transformations for Low Power Caching in Embedded Multimedia Processors," in *proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Sympo-*

sium (IPPS'98). Washington, DC, USA: IEEE Computer Society, 1998, pp. 292–297.

[87] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 424–453, 1996.

[88] L. Benini, F. Menichelli, and M. Olivieri, "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems," *IEEE Trans. Comput.*, vol. 53, no. 4, pp. 467–482, 2004.

[89] M. Game and A. Booker, "CodePackTM: Code Compression for PowerPC Processors," white paper, 1998, http://users.ece.gatech.edu/l̃eehs/CS8803/papers/CodePack_whitepaper.pdf.

[90] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu, "Power and Energy Impact by Loop Transformations," in *proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*, 2001.

[91] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power Estimation of Embedded Systems: A Hardware/Software Codesign Approach," *IEEE Transaction VLSI Systems*, vol. 6, no. 2, pp. 266–275, 1998.

[92] U. Kremer, "Low Power/Energy Compiler Optimizations," in *Chapter 35, Christian Piguet: Low-Power Electronics Design*, 2005.

[93] Agilent Technologies Inc., *Agilent 34410A Digital Multimeter, Datasheet*, October 2007, 5989-3738EN. [Online]. Available: http://www.home.agilent.com/agilent/product.jspx?pn=34410A

[94] N. R. Draper and H. Smith, *Applied Regression Analysis*, 2nd ed., ser. Wiley Series in Probability and Mathematical Statistics. New York, NY: John Wiley and Sons, Inc., 1981.

[95] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy, "Power Estimation Methodology for VLIW Digital Signal Processor," in *proceedings of the conference on Signals, Sys-*

tems and Computers (SSC'08). Asilomar, CA, US: IEEE Signal Processing Society, October 2008, pp. 1840–1844.

[96] Texas Instruments Inc., *TMS320C6416T, Fixed Point Digital Signal Processor, Optimizing Compiler User Guide*, May 2004, SPRU187l. [Online]. Available: www.ti.com

[97] *Texas Instruments Inc., TMS320C6416T,DSP Two-Level Internal Memory Reference Guide*, February 2006, SPRU610C. [Online]. Available: www.ti.com

[98] M. E. A. Ibrahim, M. Rupp, and S. E.-D. Habib, "Power Consumption Model at Functional Level for VLIW Digital Signal Processors," in *proceedings of the conference on Design and Architectures for Signal and Image Processing (DASIP'08)*, Bruxelles, Belgium, November 2008, pp. 147–152.

[99] Texas Instruments Inc., *TMS320C64x/C64x+, DSP CPU and Instruction Set Reference Guide*, July 2007, SPRU732D. [Online]. Available: www.ti.com

[100] *Texas Instruments Inc., C6000 Host Intrinsics*, January 2009. [Online]. Available: www.tiexpressdsp.com

[101] M. E. A. Ibrahim, M. Rupp, and S. E.-D. Habib, "Performance and Power Consumption Trade-offs for a VLIW DSP," in *proceedings of the IEEE International Symposium on Signals, Circuits and systems(ISSCS'09)*. Iasi, Romania: IEEE, July 2009, pp. 197–200.

[102] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy, "Impact of Code Transformations and SIMD on Embedded Software Power Consumption," in *proceedings of the IEEE International Conference on Computer Engineering and Systems (ICCES'09)*, Cairo, Egypt, December 2009.

[103] N. Z. Azeemi, "Energy Aware Frame Work for Mobile Computing," PhD Disseration, Vienna University of Technology, Institute of Communication and Radio Frequency-Engineering, August 2007.

[104] M. E. A. Ibrahim, M. Rupp, and S. E.-D. Habib, "Compiler-Based Optimizations Impact on Embedded Software Power Consumption," in *proceedings of the IEEE joint conference NEWCAS-TAISA09*. Toulouse, France: IEEE, June 2009, pp. 247–250.

[105] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[106] J.-F. Collard, *Reasoning about Program Transformations*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.

[107] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 421–461, 1994.

[108] V. Sarkar, "Optimized Unrolling of Nested Loops," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 545–581, 2001.

[109] *Texas Instruments Inc., C6000 Digital Signal Processors Documentations*. [Online]. Available: www.ti.com

[110] *Texas Instruments Inc., TMS320C6416T, Fixed Point Digital Signal Processor, Datasheet*, November 2003, SPRS226J. [Online]. Available: www.ti.com

[111] P. J. A. Shaw, *Multivariate statistics for the Environmental Sciences*. Wiley, 2009.

[112] T. Cserhati, *Multivariate Methods in Chromatography: A Practical Guide*. Wiley, 2008.

[113] K. R. Gabriel, "The Biplot Graphic Display of Matrices with Application to Principal Component Analysis," *Biometrika Journal of Statistics*, vol. 58, no. 3, pp. 453–567, 1971.

[114] W. Yan and M. S. Kang, *GGE Biplot Analysis: A Graphical Tool for Breeders, Geneticists, and Agronomists*. CRC Press, 2003.

[115] *Mathworks Inc., Matlab 7.7*. [Online]. Available: www.mathworks.com

[116] *ViSta: The Visual Statistics System*. [Online]. Available: www.visualstats.org

APPENDICES

A. C6416T ARCHITECTURE AND PROFILER EVENTS

The DSP market two years ago (2007) was divided into two parts. One, the market leader Texas Instruments Inc., accumulates 65% market share, and the remaining companies among them as the biggest Freescale Semiconductor share the remaining 35% of the market. According

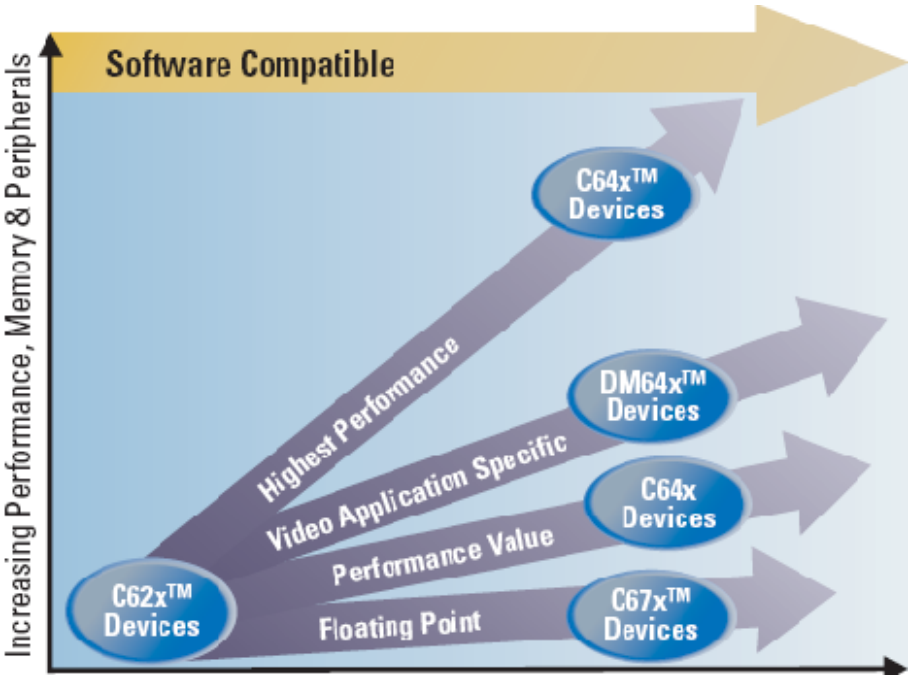

Fig. A.1: C6000 DSP platform roadmap (reproduced from [109]).

to this fact, we restrict our consideration to the market leader Texas Instruments Inc. and give a short overview about their product tree and the price policy in their DSP segment. The C6000 DSP platform, which forms a bundle of high performance DSPs is mainly divided into two categories fixed-point and floating-point DSPs.

Figure A.1 gives an overview about the single representatives of the C6000 family. They are fixed-point DSP architectures, with the exception of the C67xx family which is a floating

point architecture, priced from $7.53 to $320.95 at clock frequencies of up to 2400MHz [109].

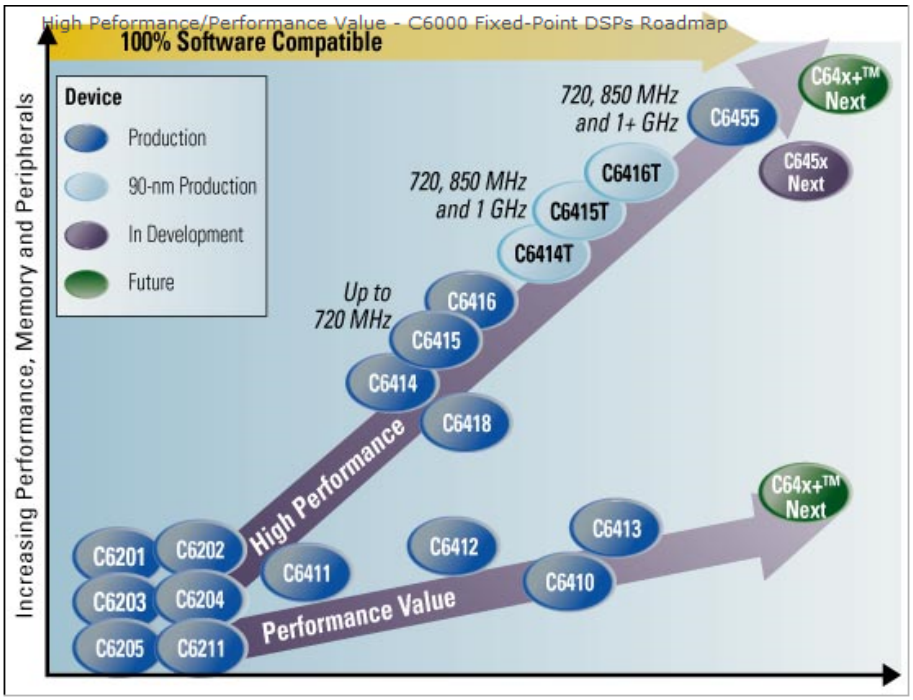

Fig. A.2: C6000 fixed-point DSPs roadmap (reproduced from [109]).

Figure A.2 gives an overview about the high performance C64x Fixed-Point DSPs Roadmap. The C64x^{TM} DSP generation features TIs VelociTI.2^{TM} VLIW architecture extensions that include support for packed data processing and special purpose instructions to accelerate broadband infrastructure and imaging applications. The C64x generation is shipping DSPs with clock speeds available up to 1.2 GHz and can incorporate multiple memory, peripheral and voltage combinations to address a wide range of high performance applications [109].

A.1 Target Architecture

Since the C6000 form a special DSP family which comprises of extreme DSP cores and connected video ports. They typically possess fast memories, wide data busses and parallelism like Very Long Instruction Word (VLIW) and Simple Instruction-Multiple Data(SIMD). Hence, the targeted DSP is the TMS320C6416T (for the rest of the thesis it is referred to as C6416T for brevity).

A block diagram of the C6416T DSP CPU is shown in Fig. A.3. This DSP is considered as

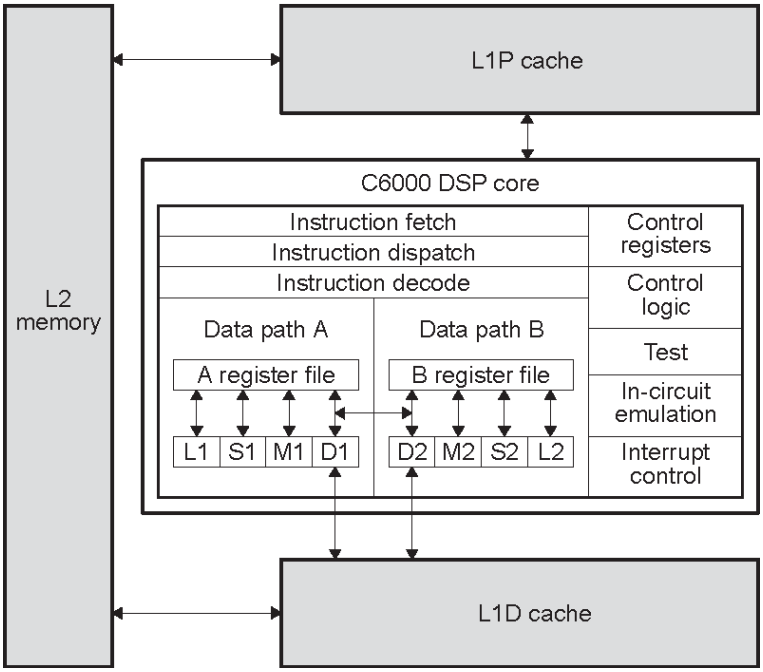

Fig. A.3: C6416 block diagram (reproduced from [110]).

a complex processor architecture since it features the following:

- One of the highest performance fixed-point DSP.
  - Deep pipeline (11 stages).
  - Eight 32-bit instructions/cycle.
  - twenty-eight operations/cycle.
  - Up to 8000 MIPS.
- VLIW TMS320C64x+ DSP Core.
  - Six ALUs (32-/40-bit), each supports single 32-bit, dual 16-bit, or quad 8-bit arithmetic per clock cycle.
  - Two multipliers support four 16×16-bit multiplies (32-bit results) per clock cycle or eight 8×8-bit multiplies (16-bit results) per clock cycle.
  - 6432-bit general purpose registers.
  - Non-aligned load-store architecture.
- Instruction set features.
  - Byte-addressable (8-/16-/32-/64-bit data)

- L1/L2 memory architecture.
  - 16-kbyte L1 two-way set associative data cache, with a 64-byte line size and 128 sets.
  - 16-kbyte direct-mapped L1P program cache, with a 32-byte line size and 512 sets.
  - 1024-kbyte L2 unified mapped RAM/cache, with flexible allocation configurations.

More details about the C6416T DSP can be found in [110].

A.2 C6416T Simulator Performance Monitoring Events

cycle.CPU	Counts cycles consumed by the CPU (including instruction execution, & pipeline stall cycles) This event count includes instruction execution cycle count, cross path stalls and memory bank conflict stalls
cycle.Total	This event count includes instruction execution cycle count, all stalls (including pipeline stalls), memory latency and system effects
L1D.hit.read	CPU read access is a hit in L1D cache
L1D.hit.write	CPU write access is a hit in L1D cache
L1D.hit.summary	Total hits in L1D Cache
L1D.miss.read	CPU read access is a miss in L1D cache
L1D.miss.write	CPU write access is a miss in L1D cache
L1D.mis.summary	Total misses in L1D Cache
L1D.access	All data accesses from CPU to L1D cache (hit and miss accesses)
CPU.stall.mem.L1D	CPU stall cycles due to L1D cache

L1D.stall.write_buf_full	A write buffer exists between the L1D and L2 caches. There can be up to four non-mergeable write misses outstanding in the write buffer without stalling the CPU. If a write miss occurs when the write buffer is full, this event will occur.
L1P.hit	L1P cache hit
L1P.miss.summary	Total L1P cache misses
L1P.access	all program fetches from CPU to L1P
CPU.stall.mem.L1P	CPU stall cycles due to L1P
CPU.discontinuity.branch	This will be reported every time a discontinuity (or jump) occurred in the PC value due to the execution of a branch instruction.
CPU.execute_packet	Number of instruction packets that have been decoded. Events will be reported against the address of the first instruction.
CPU.instruction.executed	Total number of instructions which got executed.
CPU.NOP	Number of No-Operation cycles executed.

B POWER ESTIMATION DETAILS

B.1 Computation of the Model Parameters

Table B.1 shows how the algorithmic parameters, required to estimate the power consumption of the running algorithm, are computed.

Tab. B.1: Algorithmic parameters calculation methodology

Parameter	Computation Methodology
α	No. of fetch packets / No. of execution packets
β	(No. of executed instructions - NOP instructions) / Total code cycles
ε	(No. of L1D read hits / Total code cycles) * 100
λ	(No. of L1D write hits / Total code cycles) * 100
γ	((No. of L1D read misses + No. of L1D write misses) / No. of L1D references) * 100
δ	(No. of L1P misses / No. of L1P references) * 100
PSR	No. of CPU stall cycles / Total code cycles

B.2 Complete Functional-Level Power Consumption Model at 1000MHz

Table B.2 summarizes the whole power consumption model for the C6416T at an operating frequency equals $1\,000$MHz. In the clock distribution sub-model as shown in Table B.2 F is substituted with the operating frequency in MHz.

Tab. B.2: Complete power consumption model for C6416T DSP at $F = 1\,000$MHz.

Functional unit	Functional unit power consumption sub-model
Clock Distribution	$\mathrm{P}_{\mathrm{Clock\_Distribution}} = (0.0006F + 0.0574) \times \mathrm{V}_{\mathrm{core}}$
IMU	$\mathrm{P}_{\mathrm{IMU}} = (-0.0918\alpha^2 + 0.284\alpha + 0.0603)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$
Processing Units	$\mathrm{P}_{\mathrm{PU}} = (-0.0049\beta + 0.0065)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$
Memory Read	$\mathrm{P}_{\mathrm{Mem\_Read}} = (-2 \cdot 10^{-6}\varepsilon^2 + 0.0012\varepsilon)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$
Memory Write	$\mathrm{P}_{\mathrm{Mem\_Write}} = (-10^{-5}\lambda^2 + 0.0049\lambda)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$
L1D Cache	$\mathrm{P}_{\mathrm{L1D}} = (-2 \cdot 10^{-5}\gamma^2 + 0.0041\gamma)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$
L1P Cache	$\mathrm{P}_{\mathrm{L1P}} = (0.0011\delta)(1 - \mathrm{PSR}) \cdot \mathrm{V}_{\mathrm{core}}$

B.3 Power Estimation for Benchmarks

Table B.3 shows the actual computed parameters, the estimated, and the measured power consumption for different image and signal processing benchmarks at an operating frequency of $1\,000$MHz. For computing the average estimation error we used the absolute values of the estimation error values in Table B.3. Hence, the absolute average estimation error for the used benchmarks equals 1.6%.

Tab. B.3: Power Estimation for different benchmarks at $F = 1\,000$MHz

Benchmark	α	β	ε	λ	γ	$1 -$ PSR	Est. Power	Measured Power	Error %
DotP128	0.227	1.496	16.62	5.58	20.09	0.923	1.056	1.0464	0.88
FIR	0.1405	0.996	46.85	11.74	0.033	1.0	1.039	1.037	0.21
Sobel3x3	0.1775	0.936	30.61	12.25	2.33	0.997	1.035	1.019	1.59
Threshold	0.1369	0.487	22.31	2.705	9.75	0.9987	0.9991	0.977	2.19
Histogram	0.1406	0.5504	21.7	8.84	0.061	0.998	0.981	0.9624	1.795
IIR	0.1745	1.079	44.63	9.36	1.24	1.0	1.0413	1.0579	−1.597
FFT 16x16r	0.1528	0.9329	41.08	18.19	1.06	0.998	1.066	1.031	3.27
Correlation_3x3	0.1718	0.8814	31.57	9.516	0.847	1.0	1.0173	1.007	1.012

C MULTIVARIATE STATISTICS

C.1 Principal Component Analysis (PCA)

PCA is used for dimensionality reduction in a data set by retaining those characteristics of the data set that contribute most to its variance, by keeping lower-order principal components (e.g., PC1, PC2, PC3) and ignoring higher-order ones (such as PC4,PC5 and higher). Such low-order components often contain the most important aspects of the data. But this is not necessarily the case, depending on the application. PCA is an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by any projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. PCA is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where a graphical representation is not available, PCA is a powerful tool for analyzing data [111, 112].

C.1.1 Box Plot

The Box, Diamond and Dot plot uses boxes, diamonds and dots to form a schematic of a set of observations. The schematic can give you insight into the shape of the distribution of observations. Some Box, Diamond and Dot plots have several schematics. These side-by-side plots help to see if the distributions have the same average value and the same variation in values. The plot always displays dots. They are located vertically at the value of the observations shown on the vertical scale. The dots are jittered horizontally by a small random amount to avoid overlap. The plot can optionally display boxes and diamonds. Boxes summarize information about the quartiles of the variable distribution. Diamonds summarize information about the moments of the variable distribution. The box plot is a simple

schematic of a variable distribution. The schematic gives information about the shape of the distribution of the observations. The schematic is especially useful for determining if the distribution of observations has a symmetric shape. If the portion of the schematic above the middle horizontal line is a reflection of the part below, then the distribution is symmetric. Otherwise, it is not. In the box plot, the center horizontal line shows the median, the bottom and top edges of the box are at the first and third quartile, and the bottom and top lines are at the 10^{th} and 90^{th} percentile. Thus, half the data are inside the box, half outside. Also, 10% are above the top line and another 10% are below the bottom line. The width of the box is proportional to the total number of observations.

C.1.2 Scree Plot

The Scree plot shows the relative fit of each principal component. It does this by plotting the proportion of the data variance that is fit by each component versus the component number. The plot shows the relative importance of each component in 5.1. Terminologies 71 fitting the data. The numbers beside the points provide information about the fit of each component. The first number is the proportion of the data variance that is accounted for by the component. The second number is the difference in variance from the previous component. The third number is the total proportion of variance accounted for by the component and the preceding components. The Scree plot can be used to aid in the decision about how many components are useful. We use it to make this decision by looking for an elbow (bend) in the curve. If there is one (and there often is not be likely to) then the components following the bend account for relatively little additional variance, and are good candidates to be ignored.

C.1.3 Biplot

The biplot was introduced by Gabriel at (1970) [113]. Biplots are a type of graph used in statistics. A biplot allows information on both samples and variables of a data matrix to be displayed graphically. Samples are displayed as points while variables are displayed either as vectors, linear axes or nonlinear trajectories. In the case of categorical variables, category level points may be used to represent the levels of a categorical variable. A generalized biplot displays information on both continuous and categorical variables [114].

C.1.4 PCA Example

Consider a sample application (reproduced from Matlab help [115]) that uses nine different indices of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each index, higher is better. For example, a higher index for crime means a lower crime rate. The data of this example is organized in the following matrix format:

Name	Size	Bytes	Class
categories	9×14	252	char array
names	329×43	28 294	char array
ratings	329×9	23 688	double array

- **categories**, a string matrix containing the names of the indices.
- **names**, a string matrix containing the 329 city names.
- **ratings**, the data matrix with 329 rows and 9 columns.

The box plot gives a quick impression of the ratings data as shown in Fig. C.1. The boxplot can be obtained by utilizing the following matlab function:

```
boxplot(ratings,'orientation','horizontal','labels',categories)
```

There is substantially more variability in the ratings of the arts and housing than in the ratings of crime and climate. Ordinarily, we need to graph pairs of the original variables, but there are 36 two-variable plots. Thus, we need a way to reduce the dimensionality of the problem, principal components analysis can reduce the number of variables to consider.
Sometimes it makes sense to compute principal components for raw data. This is appropriate when all the variables are in the same units. Standardizing the data is often preferable when the variables are in different units or when the variance of the different columns is substantial (as in this example). Data can standardized by dividing each column by its standard deviation. The principal component analysis can be easily applied to our example data with the aid of many programs such as Matlab 7.7 from Mathworks [115] or the ViSta [116] and so on. The princomp function of the Matlab has four outputs:

```
stdr = std(ratings);
sr = ratings./repmat(stdr,329,1);
[coefs,scores,variances,t2] = princomp(sr);
```

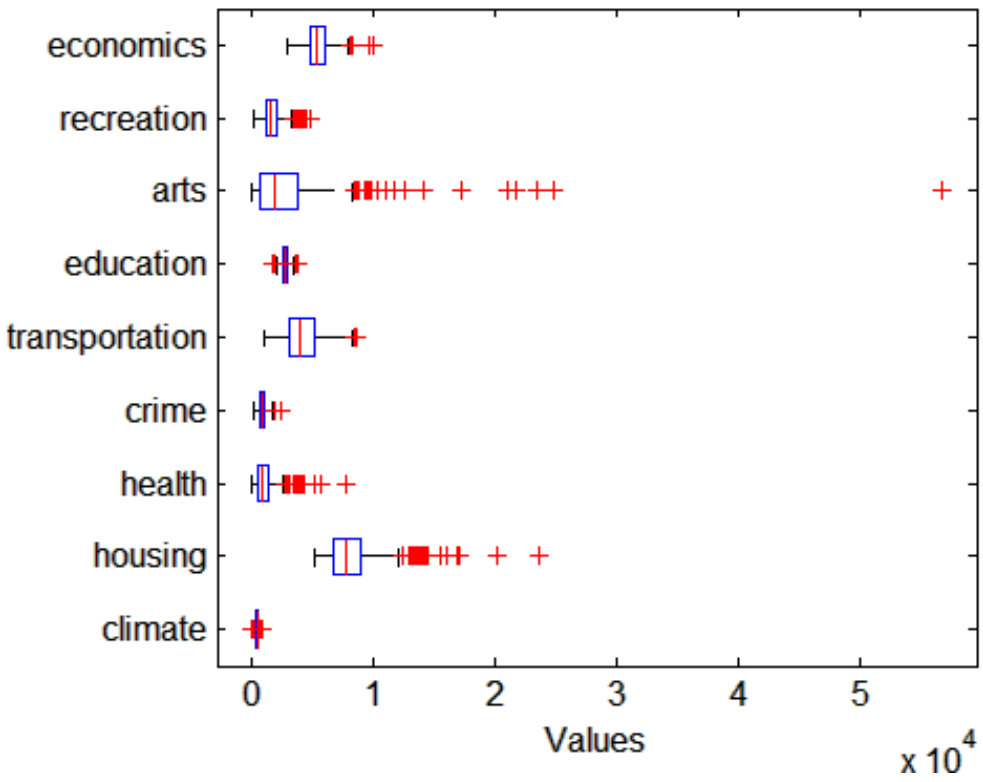

Fig. C.1: Box plot for the data ratings.

- **Coefficients:** The first output of the princomp function, coefs, contains the coefficients of the linear combinations of the original variables that generate the principal components. The coefficients are also known as loadings.

- **Scores:** The second output, scores, contains the coordinates of the original data in the new coordinate system defined by the principal components. This output is the same size as the input data matrix.

- **Variances:** he third output, variances, is a vector containing the variance explained by the corresponding principal component. Each column of scores has a sample variance equal to the corresponding element of variances.

- **Hotelling's T2:** The last output of the princomp function, t^2, is Hotelling's T2, a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

With the aid of the pareto function we can get a scree plot of the percent variability explained by each principal component as shown in Fig. C.2.

```
pareto(percent_explained)
xlabel('Principal Component')
ylabel('Variance Explained (\%)')
```

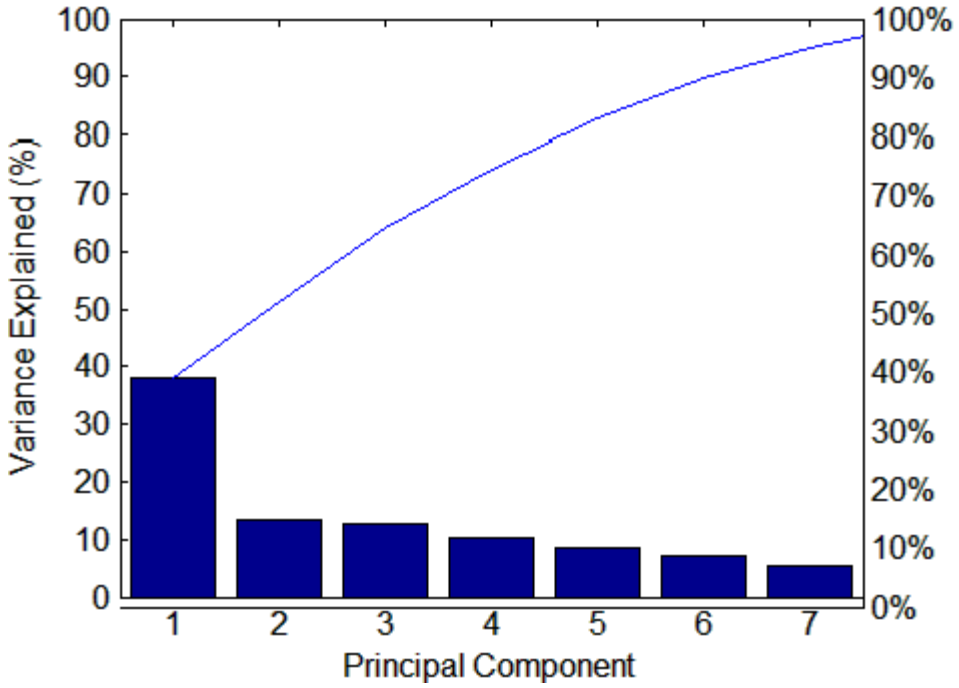

Fig. C.2: Scree plot of the percent variability explained by each principal component.

Figure C.2 shows that the only clear break in the amount of variance accounted for by each component is between the first and second components. However, that component by itself explains less than 40% of the variance, so more components are probably needed. Hence, it is clear that the first three principal components explain roughly two-thirds of the total variability in the standardized data ratings, so that might be a reasonable way to reduce the dimensions in order to visualize the data.

Finally, by utilizing the biplot function we can visualize both the principal component coefficients for each variable and the principal component scores for each observation in a single plot as shown in Fig. C.3.

```
biplot(coefs(:,1:2), 'scores',scores(:,1:2),...
'varlabels',categories);
axis([-.26 1 -.51 .51]);
```

Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, the first principal component, represented in this biplot by the

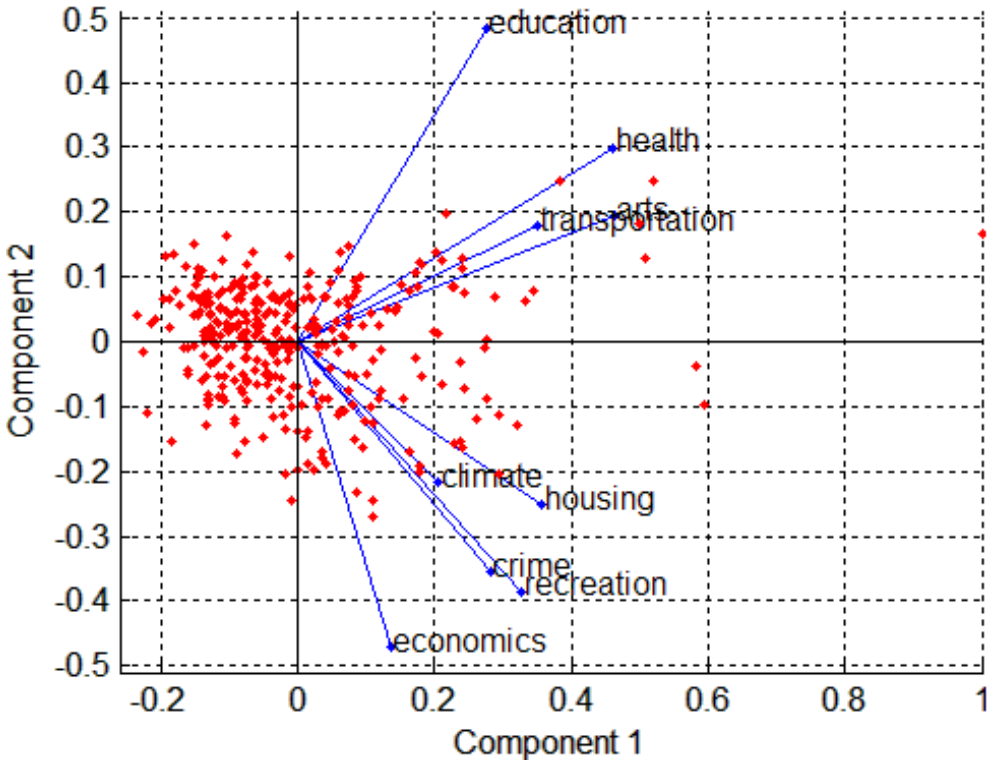

Fig. C.3: Visualizing the results of the PCA with the Biplot.

horizontal axis, has positive coefficients for all nine variables. That corresponds to the nine vectors directed into the right half of the plot. The second principal component, represented by the vertical axis, has positive coefficients for the variables education, health, arts, and transportation, and negative coefficients for the remaining five variables. That corresponds to vectors directed into the top and bottom halves of the plot, respectively. This indicates that this component distinguishes between cities that have high values for the first set of variables and low for the second, and cities that have the opposite.

Each of the 329 observations is represented in this plot by a point, and their locations indicate the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled to fit within the unit square, so only their relative locations may be determined from the plot.

C.2 Applications Pseudonyms

Table C.1 shows the pseudonyms of the employed applications for the multivariate analysis.

Tab. C.1: Pseudonyms for the applications used for PCA.

App. Abbrev.	Description
A1	Dot product of a vector of 128 16-bit elements
A2	Matrix multiplication for 2 100x100 square matrices
A3	Computes a real FIR filter, Input data and filter taps are 16-bit
A4	Apply Sobel filter of 3x3 window to an image of 8192 pixels
A5	Performs a thresholding operation on an input image of 8192 pixels
A6	Takes histogram of an image of 8192, 8-bit pixels
A7	Performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients
A8	Performs a mixed radix forwards FFT using a special sequence of coefficients
A9	Performs a point by point multiplication of the 3x3 mask with an input image
A10	Performs IDCT on 8x8 DCT coefficient blocks
A11	Performs IDCT on 8x8 DCT coefficient blocks with the aid of all possible SIMD
A12	Elastic Graph Matching used in the project of bad weeds recognition and elimination
A13	performs a 3x3 median filter operation on 8-bit unsigned values
A14	performs a 3x3 median filter operation on 8-bit unsigned values with the aid of all possible SIMD
A15	performs a FIR filter whose sum can be larger than 32 bits
A16	performs a FIR filter whose sum can be larger than 32 bits with the aid of all possible SIMD
A17	performs a FDCT on a list of 8x8 8-bit pixels
A18	performs a FDCT on a list of 8x8 8-bit pixels with the aid of all possible SIMD

D LIST OF ACRONYMS

α	Dispatching Rate
β	Processing Rate
ε	Internal Memory Read Referencing Rate
λ	Internal Memory Write Referencing Rate
γ	L1D cache Memory Miss Rate
δ	L1P cache Memory Miss Rate
μ***C***	Micro-Controller
μ***P***	Micro-Processor
ALU	Arithmetic Logic Unit
ARMA	Auto-Regressive Moving-Average
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-Set Processor
CCS	Code Composer Studio
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CTMDP	Continuous-Time Markovian Decision Processes
DCT	Discrete Cosine Transform
DMM	Digital Multi-Meter
DSK	DSP Starter Kit
DSP	Digital Signal Processor
DVS	Dynamic voltage scaling
EEPROM	Electrically Erasable Programmable ROM
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FLPA	Functional Level Power Analysis
FPGA	Field Programmable Gate Array

GPP	General Purpose Processor
GSM	Global System for Mobile Communication
IDCT	Inverse Discrete Cosine Transform
IDE	Integrated Development Environment
IIR	Infinite Impulse response
ILP	Instruction Level Parallelism
ILPA	Instruction Level Power Analysis
IMU	Instruction Management Unit
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
MIPS	Mega Instruction per Second
NOP	No Operation
PC	Personal Computer
PCA	Principal Component Analysis
PDA	Personal Digital Assistant
PFA	Power Factor Approximation
PSR	Pipeline Stall Rate
PU	Processing Unit
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SPLOOP	Software Pipelined Loop
SRAM	Static Random Access Memory
UMTS	Universal Mobile Telecommunication System
VLIW	Very Long Instruction Word
ZOL	Zero-Overhead loop