Scratchpad-oriented address generation for low-power embedded VLIW processors

Guillermo Talavera Velilla

Ph.D. Thesis Dissertation

Universitat Autònoma de Barcelona

Department of Microelectronics and Electronic Systems

October 2009

Jordi Carrabina, lecturer in the Microelectronics and Electronic Systems Department of the "Universitat Autònoma de Barcelona",

CERTIFIES:

That the present thesis has been realized under his direction by Guillermo Talavera Velilla in partial fullment of the requirements for the degree of "Doctor en Informàtica (opció microelectrònica) per la Universitat Autònoma de Barcelona".

Bellaterra, October 2009.

A mis padres y hermana,

por su amor incondicional.

A Ruth,

porque, en calidad de "doctoranda consorte", ha sufrido

este doctorado más de lo que le correspondía.

Gracias por estar a mi lado todos estos años.

A Héctor, Irene... ¡y los que vengan!

¡Encantado de haberos conocido!

Espero que si esto no os sirve de ejemplo,

por lo menos os sirva de excusa.

Acknowledgements

"I can't remember to forget you."

Leonard Shelby. Memento.

This thesis came about with the help of many different people without whom this book would not have been possible. I would like to thank them all for their efforts, discussions, patience and friendship.

I sincerely thank Prof. Jordi Carrabina for giving me the opportunity to pursue my PhD in the Department of "Microelectrònica i Sistemes Electrònics" at the Universitat Autònoma de Barcelona. I would also like to thank him for all the discussions we had during these years and for giving me the freedom and support to choose my path.

There are a lot of people from the "Cephis Group" [cep] (or really close) that I must be grateful to. First of all, I would like to thank Toni Portero for all these years working together, helping and encouraging each other during "hard times". I would also like to thank Màrius Montón and Borja Martínez, it was a big pleasure to have them as colleagues in some of the courses we taught together. Besides, I really learned a lot from them. I also learned a lot from Enric Pons (a.k.a. Obi), and, after some time, he convinced me that pr0n can also be an art. Xavier Fitó, I had him as student and at first I was affraid, but sometime latter I discover that he is not so dangerous and we became a good friends. Màrius, Borja, Obi, Xavi and Elena Garcia, thank you very much for all the time we spent and all the coffees we had "saving the world", we had great times. I met you all here "at work" and now I really consider you my friends, thank you also for your friendship, it really mean a lot to me.

There are other people from the group that I would like to thank, Quim Saiz, Marc Moreno, Aitor Rodriguez, Roger Puig, Eduard Cespedes, Juan Carlos Chak Ma, ... and all the rest of the group, thank you for your friend-ship, discussions, and the nice times we had at the university. Carlos Montero, thanks for letting me win from time to time when playing squash... it is a real boost for my moral.

I would also like to thank other people I meet at the university: Vicenç Soler for taking care of me the first years at the university and showing me the "tricks" I needed, Eleni Kanellou for her deep proof reading of this thesis. Elena Valderrama, it has been a pleasure to share some lectures with you. Jorge Ramirez thanks for all these years of help with the computers and labs, Marta Garsaball, Jordi Jovani and Antonio Guerra for their support and help with all the administrative tasks. I also would like to thank Jordi Tirado, because he is a good friend, and a better teacher, and he had a lot of patience guiding me for some months; and Xavier Naveira (and Maria), because he changed from student to friend really fast, and since then it is always nice to talk to him.

Of course, nothing would have been the same without my trips to Leuven. There are plenty of reasons to show my permanent gratitude to a lot of people there and I really wish I do not forget anybody. Francisco Barat (a.k.a. Pancho), Murali Jayapala, Tom Vander Aa, thanks very very very much for all the explanations, patience, help and friendship. I learned really a lot from you in a very nice environment, I will never forget that. Nevertheless, next time you want to do a PhD creating a coarse-grained reconfigurable instruction set processor (CRISP) architecture and its compiler, please sit down, relax, take a cup of tea and wait until all this creative need disappears: without CRISP, my life would have been much more boring, but easy too ;). Also thanks again to Pancho and Yulia for all these evenings watching movies and playing with singstar, there I discover that I am not such a bad singer... at least I can manage with Madonna.

Praveen Raghavan, thank you very much for your support, help and explanations, I would have never made it without your help. Andy Lambrechts, Anthony Leroy, Karel Van Oudheusden (a.k.a DayLight), Javed Absar, Vincent Nollet, thanks a lot, I learned a lot from all you, and I had great fun while learning. My stays in Leuven would have been completely different without my deep friends Will Moffat and Theodore Marescaux (a.k.a. Theo), I really hope one day we can meet again and have our talks, beers and fun we had these years.

In addition, I also meet some other Spaniards in Leuven: Javier (a.k.a. Javi, a.k.a. Doctor Xano) and Martin Resano (and Yoli and Espe), Nacho Gomez (and Adri), Elena Perez, I enjoyed meeting you in Belgium. Now I have even more excuses to go to Madrid or Zaragoza. To you too, thanks for all the time we spent together, all the conversations, all that I learned and your friendship. A particular recognition to Javier because he endured all my doubts and fears and his advice is always good.

I would also like to thank all my friends, because they are always there. It won't be easy to write all their names, but basically, thanks to all the ones from the physicist group (Roger, Eva, Bernat, Marta, Jorge Juan, Joan, Alexis, Rául, Enric, etc). The ones from chemistry group (Alicia, Elena, Raquel, Sara, Sonia, Marta, Miriam, Raquel, Mireia, Jordi "Tanki", Santi, Dani, Ferran, Jordi "Jorge", Miki, Sito, etc.). And the ones who have always been there (Oscar, Santi, Xavier, Rául, Pep, etc.).

A very special and deep gratitude and recognition to Francky Catthoor, I learned a lot from his knowledge and vision of technical aspects, but what is even more important, I would never be able to thank him enough for sharing his wisdom in all the other aspects of life and for giving me another vision of reality.

Abstract

Nowadays Embedded Systems are growing at an impressive rate and provide more and more sophisticated applications. An increasingly important set of embedded systems are real-time portable multimedia and digital signal processing communication systems: cellular phones, PDAs, digital cameras, handheld gaming consoles, multimedia terminals, netbooks, etc. These systems require high performance specific computations, usually with real-time and Quality of Service (QoS) constraints, which should run at a low energy level to extend battery life and avoid heating. A flexible system architecture is also required to successfully meet short time-to-market restrictions. Hence, embedded systems need a programmable, low power and high performance solution in order to deal with these requirements.

Very Long Instruction Word architectures seem a good solution for providing enough computational performance at low-power with the required programmability to speed the time-to-market. Those architectures rely on compiler effort to exploit the available instruction and data parallelism to keep the data path busy all the time. With the density of transistors doubling each 18 months, more and more complex architectures with a high number of computational resources running in parallel are emerging. With this increasing parallel computation, the access to data is becoming the main bottleneck that limits the available parallelism. To alleviate this problem, in current embedded architectures, a special unit works in parallel with the main computing elements to ensure efficient feed and storage of the data: the Address Generator Unit, which comes in many flavors.

The purpose of this dissertation is to prove that optimizing the process of address generation is an effective way of solving the problem of accessing data while decreasing execution time and energy consumption.

As a first step, this thesis evaluates the effectiveness of different state-of-the-art devices commonly used in the embedded domain, argues for the use of very long instruction word processors and presents the compiler and architecture framework used for our experiments.

This thesis also presents a systematic classification of address generators, a review of literature according to the classification of the different optimizations on the address generation process and a step-wise methodology that gradually reduces energy reusing techniques that already have been published. The systematic architecture exploration framework and methods used to obtain a reconfigurable address generation unit are also introduced.

Results of the reconfigurable address generator unit are shown on several benchmarks and applications, and the complete step-wise methodology is demonstrated on a real-life example.

"You met me at a very strange time in my life"

Tyler Durden - The Narrator.

Fight Club

Contents

List of Figures

List of Tables

List of Acronyms

ADOPT: ADdress OPTimisation

AE: Address Equation

AGU: Address Generation Unit

ALU: Arithmetic and Logic Unit

ASIC: Application Specific Integrated Circuit

ASIP: Application Specific Instruction Set Processor

CGRA: Coarse Grain Reconfigurable Architecture

COFFEE: COmpiler Framework For Energy-aware Exploration

CSE: Common Sub expression Elimination

DFG: Data-Flow Graph

DLP: Data Level Parallelism

DM/DMH: Data Memory/Data Memory Hierarchy

DMA: Direct Memory Access

DSIP: Domain Specific Instruction Set Processor

DSM: Deep SubMicron

DSP: Digital Signal Processor

DTSE: Data Transfer and Storage Exploration

FEENECS: Flexible Extremely ENergy Efficient Configurable System

FPGA: Field Programmable Gate Array

FSME: Full Search Motion Estimation

FU: Functional Unit

GOP: Group Of Pictures

GPP: General Purpose Processor

ICMO: Instruction/Configuration Memory Organization

ILP: Instruction Level Parallelism

IM/IMH: Instruction Memory/Instruction Memory Hierarchy

ISO: International Organization for Standardization

JPEG: Joint Photographic Experts Group

LB: Loop Buffer

LCs: Local Controllers

ME: Motion Estimation

MIMO: Multiple-Input and Multiple-Output

MPEG: Motion Picture Experts Group (also a standard for video encoding and decoding)

MPSoC: Multi-Processor System-on-Chip

NOP: No OPeration

PC: Program Counter

PE: Processing Element

QoS: Quality of Service

RF: Register File

RTL: Register Transfer Level

SIMD: Single Instruction Multiple Data

SoC: System-on-Chip

SPM: ScratchPad Memory

TLM: Transaction-Level Modeling

VHDL: VHSIC (Very High Speed Integrated Circuits) hardware description language

VLIW: Very Long Instruction Word (also used as a shorthand for Very Long Instruction Word processor)

VWR: Very Wide Register

CHAPTER 1

Introduction

"A child of five would understand this.

Send someone to fetch a child of five."

Groucho Marx

Over the last decade, the demand for embedded systems has been growing at an impressive rate and represents around 100% of the worldwide production of microprocessors [Tur99]. We can sense the presence of such systems in automobiles, house-hold appliances, consumer electronics and several others. An increasingly important set of the embedded devices are multimedia and communication systems like cellular phones, PDAs, multimedia terminals, handheld gaming consoles, etc.

Multimedia handheld devices require high performant specific computation, usually with real-time and Quality of Service (QoS) constraints, which should run at low energy to have a long battery life. Having a flexible architecture is also needed to meet short time-to-market restrictions and to easily update to new versions of the applications, or even adding a new application with similar requirements. Hence, the ideal multimedia device will present high quality multimedia content, and will be networked, portable, inexpensive and easy to use. Moreover, in order to cope with the dynamism of current and future multimedia applications, modern (and even more future) embedded systems demand a programmable and high performant solution running at low energy consumption to deal with all these requirements.

1.1. Embedded systems

Maybe the simplest definition of an embedded system is a system where the computing is not intended to be general purpose: embedded systems are specifically designed for a

single application or a limited set of applications. In many cases, the principal component of such systems is a programmable processor, and often, it is a Very Large Instruction Word (VLIW) processor (alone or integrated with other processor cores).

Reasons for not choosing a general-purpose processor depend on several metrics: performance, energy, power, size and cost. When the computation requirements are not very high, using a general-purpose processor might be over-dimensioned, however, a significant number of embedded processors (e.g., digital signal processors [DSPs]) offer more performance on specific applications than a general-purpose processor can provide. Handheld embedded systems provide more and more functionalities and users demand a quality comparable to their non-mobile counterparts, but still expect a long battery life; hence, low-energy consumption is a key issue and general-purpose-processors give versatility in detriment of energy efficiency and then batteries will last less. Power is increasingly important, especially in portable applications: the cooling and package cost required to make a general-purpose processor perform cellular phone functions would be prohibitive. Regarding the size, many embedded devices are smaller than general-purpose chips and modules. Finally, concerning cost, even the cheapest Pentium processor (ATOM [INT]) costs more than many consumer electronics items.

1.2. Application domain: Multimedia

A huge segment of the embedded systems market is driven by multimedia applications, like mobile and most hand-held devices, and this is the target domain of this dissertation. Multimedia applications consist of several digital signal processing algorithms (e.g. audio, graphics and video processing algorithms) and are characterized by the following features:

- Compute Intensity: Media applications require a high number of arithmetic operations per memory reference [OKM+02]. An MPEG4 encoder with medium complexity requires an estimated 8 GOPS of computational processing power [BGN97].

- Real-Time Response: Most applications require an immediate response to user interactions. For instance, when an user invokes a video decoding application in a hand-held device the response should be immediate. Also, transition from one application to another should be seamless without any significant delay.

- High Memory Bandwidth: Large amounts of data have to be processed in audio, image and video applications. For a simple QCIF (144 lines x 176 pixels) video encoder @25 fps the required memory bandwidth is about 2Mbps per second. For advanced applications this figure is much higher.

- Parallelism: Many computations on the data are independent. A high degree of parallelism can be achieved at instruction, data and task levels [OKM+02], nevertheless full parallelism can not be reached due to the data-dependent nature (conditions and while loops) that most modern multimedia applications exhibit.

- Locality: High locality in data and instruction references. Most of the compute intensive kernels in these applications are repetitive numeric computations (mostly via for-loops) [BGN97].

These characteristics can also be viewed as metrics that might be used for architectural style exploration. An architecture best suited for this application domain can be chosen based on these metrics. Other metrics should also be considered to satisfy the system constraints.

1.3. Address generation on embedded systems

Embedded applications, such as speech and image recognition, high bandwidth wireless communications or multimedia applications, are often characterized by having a complex array index manipulation scheme and a large number of data accesses [Kuh04]. These data sets are typically stored in main memory, which means that the processor needs to generate the address of the memory location in order to retrieve and store them.

Between the main memory and the computing elements, the memory hierarchy can be built using caches or ScratchPad Memories (SPM). On-chip SRAM caches consume 25% to 45% of the total CPU power [PND98a] and the allocation of data on those memories is done at run-time. Scratchpad memories are software controlled and its allocation is done at compile time. Compile time allocation improves energy reduction and predictability and allows an easier analysis and optimization of the application. Hence, this type of memories reduces considerably the energy consumption (average reduction of 40%) and the area-time product (46%) [BSL+02a] but relies on compiler/programmer effort. Because of these energy and area-time savings, scratchpad memories are widely used in the embedded systems domain.

Address calculations often involve linear and polynomial arithmetic expressions which have to be calculated during program execution under strict timing constraints. Memory address computation can significantly degrade the performance and increase power consumption: $50\% - 75\%$ of the power consumption on embedded multimedia systems is caused by memory accesses [WCNM96, MNCM97]. Hence, it is very important to carry out these accesses and related addressess computations in an effective way.

Some current embedded architectures have addressed this problem by including a dedicated unit that works in parallel with the main computing elements ensuring efficient feed and storage of the data from/to the data path. Even if different types of these units exist, they can be classified as Address Generation Unit (AGU). The development of AGUs is performance and power critical and has a big impact in VLIWs, ASIPs and Digital Signal Processors (DSP) architectural ability to access memories.

Nowadays, more and more sophisticated architectures with a high number of computational resources running in parallel are emerging [KP03b, KP03a, KMN+04, FWW99]. This trend will continue in near future, e.g. according to the International Technology Roadmap, at the end of the decade, semiconductor chips will grow to 4 billion transistor running at 10GHz and chips will incorporate hundreds of processing elements [sia05].

Embedded computing will also be affected by this increasing parallel computation and the access to data will become the main bottleneck that limits the available parallelism. Future AGUs will have to deal with enormous memory bandwidth in distributed memories and will have to achieve global trade-offs between the bandwidth required, the number of cycles needed to fetch data (reaction-time), the energy or the area. Also, coming AGUs will have to deal with a growing amount of concurrency and more data-dependent and complex control flows.

The main goal of this work is to obtain an optimal set of AGU structure and compiler parameters for any given data-flow application (or application set) for low-power embedded VLIW processors using scratchpad memories. Optimization will take as main parameter energy consumption and as a result we will obtain a complete design methodology.

The major contributions of this dissertation are listed in section 1.4 and section 1.5 gives an outline of the text.

1.4. Contributions

In this section the major contributions of this work are summarized, in chapter 7 a more detailed analysis is done.

VLIW platform election: when targeting the embedded domain, different common options appear and choosing the right computing element is a key factor to success in any design. We have analyzed the different platforms used in the embedded multimedia domain and made a comparison between them on a real, representative and complete application. To make a fair comparison, we have accomplished platform dependent and independent optimizations on each platform. Based on this analysis, we can derive when each platform brings the most benefits and we justify the election of VLIW processors for multimedia embedded applications.

AGU review and classification: we proposed a systematic classification of address generators and a literature review classification to illustrate the complementarity or overlap of different optimizations on the address generation process. We focused on AGU architectures and on compilation techniques to optimize the address generation process for scratchpad memories due to the power restrictions of the embedded domain. We considered the address generation process for DSPs and VLIW architectures which have to deal with computing intensive algorithms where data access is a main issue.

AGU integration: with the increasing level of complexity of silicon devices, evaluating all the options manually becomes increasingly difficult and tool support for evaluating the design space is necessary. In this chapter, we describe a methodology and framework used to create an address generation unit template that targets the embedded multime-

dia domain. To create such unit, we studied different applications representative of the domain and we identified the hardware elements needed to optimize the address generation process. This template has been integrated into a complete high-level architecture and compiler framework, where we can realistically simulate complete applications.

Energy aware design methodology: we also presented a step-wise procedure that gradually reduces energy consumption of multimedia applications targeting the embedded domain. We show how, after following the different steps of the methodology, the improvements in energy and execution time reach 90%. Even if this analysis was done on an MPEG4 encoder application, the methodology and optimizations proposed can be easily extended to any data-flow oriented application.

1.5. Structure of the dissertation

The following chapters of this thesis illustrate in detail the issues and problems presented in this introduction.

Chapter 2 "VLIW-DSIPs: key processor for embedded multimedia": compares the mapping of the same multimedia application into different platforms typically used in the embedded domain: an Application Specific Instruction Processor (ASIP), a soft-core processor with specific functional units implemented on an FPGA, an Application Specific Integrated Circuit (ASIC) and an embedded platform on a single chip formed by a high performance DSP and a processor. The analysis is done taking into account the whole processor platform including memories, and from this study, we can extract when each platform is most suited.

Chapter 3 "State of the art on address generation": analyzes the different types of address equations, the design metrics, constraints and costs used to evaluate an address generator and gives a systematic classification of address generators. Based on this study, we propose a literature review classification to illustrate the complementarity or overlap of different optimizations on the address generation process.

Chapter 4 "High Level Architecture and Compiler Requirements: COFFEE framework": presents the retargetable compiler and hardware simulator framework used to fulfill this work. First, this chapter presents the context of this work and the current trends in processor architecture space. Then it shows various architectural proposals for different processor components to reach the same energy efficiency as that of an ASIC. Finally, it puts these different architectural parts together to present the FEENECS architecture template.

Chapter 5 "AGU template": introduces a methodology and framework used to create the template of the AGU targeting the embedded multimedia domain. We studied different applications representative of the domain and we identified the hardware

elements needed to optimize the address generation process. At the end of the chapter we present a optimized AGU taking as basis the template used.

Chapter 6 "Complete Optimization Methodology": presents a methodology and a flow that combines steps in a sequential way with constraint propagation that gradually reduces energy consumption. This flow is shown from scratch for a complete and real application.

Chapter 7 "Summary and conclusions": finally summarizes the main results and exposes some global conclusions. Limitations and possibilities for future research are also discussed.

CHAPTER 2

VLIW-DSIPs: key processors for embedded multimedia

"Computers are useless. They can only give you answers."

Pablo Picasso

From a designer's perspective, system level specifications, goals and constraints have to be translated from higher abstraction levels to lower abstraction levels. At each level, many decisions that will affect the end design must be taken [CVB98, GVNG94] and the designer is confronted with a wide design space. Consumption, performance, retargetability and development time are some of the elements that need to be analyzed and well balanced to choose the right main processing elements.

In this chapter, we[1] present the mapping of a representative multimedia application (MPEG-4 Main Profile) into different target platforms typically used in the embedded domain. We will see when the choice of a VLIW processor is most suited and we will identify that, after the state-of-the-art optimizations, the address generation of the application remains the bottleneck.

2.1. Introduction

Nowadays, embedded multimedia applications are widely present in our lives. Those applications are usually specified in System-Level Design Languages (like Java, UML, C++, MATLAB, SystemC TLM), starting from a reference or golden model, that needs,

[1] This thesis focuses on VLIW-ASIPs and on the address generation process of those processors. Nevertheless during the some time I collaborated with Antonio Portero [Por09] on the comparison between different processors. This chapter summarizes the work done together.

in many cases, to be executed in real-time cost/energy-sensitive devices such as mp3-mp4 players, mobile phones, personal data assistants (PDAs), etc. Due to the requirements of the applications, those devices have to provide very high performance at low energy consumption because of battery life

Choosing the right main processing element is a key issue for the success of those devices where consumption, performance, retargetability and developing time are some of the elements that need to be analyzed and balanced. Actual designs can include different processors: application specific integrated circuits or instruction set processors (ASICs or ASIPs[2]), general purpose processors, various digital signal/media/image processors, embedded FPGA, etc. with their own local memories, architecture and with a complex interconnection scheme[LRJ+09]. Attending to the increasing computational complexity of multimedia applications, present and future Multi-Processor System-on-Chip (MPSoC) platforms have to satisfy many requirements: high computation, enormous quantity of memory accesses and high communication flow between different processing elements for non-deterministic applications. Moreover, embedded multimedia applications must satisfy hard (or soft) real-time constraints while maintaining an acceptable quality of service for the users at the lowest energy consumption.

To satisfy these computing requirements, most current SoCs will contain several heterogeneous types of processor cores and memory units ([Pea06, TIO]) connected through a hierarchical shared bus, point to point, bus matrix or a NoC [BM02]. These complex platforms can handle real-time video and audio compression and are usually based on heterogeneous solutions containing at least one DSP for multimedia-data-flow acceleration and one processor core for control flow and reactivity.

In this chapter, we propose a comparison of different implementations coming from an original unique reference (golden) model of the application. Current comparisons found in literature [BRK07] are based on small kernels or not fully platform-optimized applications. Without real aggressive optimizations, tuned for each individual style, the comparisons of the results typically give an unfair advantage to one or another platform. As far as we know, this work is the first to compare different design options for modern multimedia applications, coming from a unique, realistic and complete application: namely an MPEG-4 Video Main Profile (MP) specification. We mapped the same application into different platform styles, applying strong platform independent and dependent optimizations; comparing optimized implementations from a single unique real source gives realistic and significant results since we are making an impartial comparison.

The main purpose of this chapter is to quantitatively recognize the fundamentals that system designers require at a very early stage in the design path, which platform method/subclass is the most convenient choice to implement the target system. The outcomes are exposed for a video compressor application but they can be extended to similar data-flow dominated systems; most multimedia applications at the present time fall into that category. In addition, in this work we are comparing values among different

[2]ASIP stands for Application-Specific Instruction-set Processor; in this work we consider ASIP as synonym of DSIP that stands for Domain-Specific Instruction-set Processor. See section 2.5.2 for a extended explanation.

platforms and the results coming from a single original description, with sufficient effort spent on each of them to come to a sufficiently optimized solution. As a result, these choices are evaluated also in an objective way.

2.2. Representative multimedia application: MPEG4 encoder

MPEG-4 is a global multimedia standard that delivers professional-quality audio and video streams over a wide range of bandwidths, from cell phones to broadband, HDTV and beyond. MPEG-4 was defined by the Moving Picture Experts Group (MPEG), the working group within the International Organization for Standardization (ISO) that specified the widely adopted standard [ISO01].

The model implemented in the different platforms is based on a video compressor with main profile and permits I, P and B slice compression, with 4:2:0 chroma format, and 8 bit pixel sample depth [ISO01, Kuh04, PE02]. This model comes from an original "golden model" code developed following the MPEG standard completely, where the main parts were implemented from scratch and the other parts of the model were obtained from open sources already developed. The most computationally-intensive algorithms were produced using the Matlab framework [mat]. These modules are the motion estimation (ME), the motion compensation (MC), discrete cosine transform (DCT), the quantization (Q), and zig-zag. For the rest of this thesis, we will call these algorithms the kernel part of our application. These algorithms are data-flow oriented and hence very intense in terms of computation. They are an excellent potential target to be accelerated. Other algorithms involved in the MPEG compression are the entropy coder and the algorithms that mount the MPEG video stream; these algorithms are more control-flow oriented and thus harder to accelerate. We used Matlab to develop the application kernel since it provides an excellent framework for data and signal processing, mainly for synchronous data flow models (among others). Once this model was finished, we used it as a reference code for coherence and subsequent verification when we developed a C++ version of the model. At this abstraction level, some platform independent optimizations where developed and the refined C++ model was used as a transition step to two new versions: a SystemC [sys] version and a C version.

These two new versions will be used to map the application on hardware platforms using an ASIC and a FPGA (SystemC version) and on software ones by means of a commercial DSP and an ASIP (C version) [LAJ+04]. Figure 2.1 shows a diagram of the path followed.

As mentioned above, the model has two distinct parts: the control flow part and the kernel part. When the ASIC and the FPGA are targeted, the SystemC model related to the kernel is passed through a behavioral hardware synthesis tool which creates a low level RTL description in a hardware description language. This description can be used later for logic and physical synthesis in the FPGA or the ASIC, and will serve for performance estimations of the hardware parts, although it is not fully optimized. The

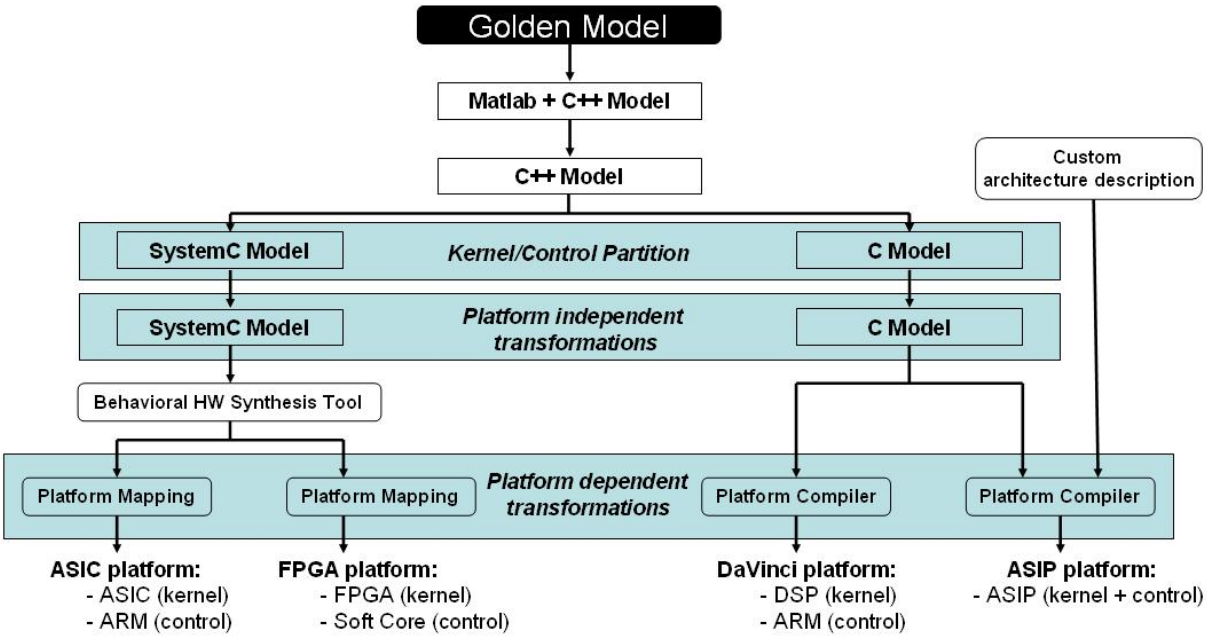


Figure 2.1.: MPEG design flow from concept to platform

control part of the application does not need the power of a hardware implementation and a RISC processor core (an ARM9 in the case of the ASIC and a NIOS II in the case of the FPGA) can handle this part of the algorithm perfectly well; as it is more control-flow oriented, a software implementation is more straight-forward. Before both models (SystemC and C) are mapped onto the targeted platforms, they are exposed to some platform-independent and platform-dependent optimizations to obtain the maximum benefit for each platform style.

2.3. Targeted platforms

The objective of early SoC-design analysis and hardware/software co-design is to obtain the right allocation of functions to hardware and software and to correctly size hardware resources to meet the design requirements. At this early stage of development, some major configuration decisions must be made which will affect the final implementation of the design, and hence the platform results. When an engineering team starts a project, there is a broad design space where to decide and the different possible implementations have distinct advantages and drawbacks that lead to huge differences in the quality of the results.

In this section we describe the main distinct platforms that engineers can use and the different metrics needed to evaluate their suitability for each solution. In this work, we

will not consider the economic aspect of the different implementations, which are well known in industry, and we will just focus on the technological differences of the targeted platforms that can handle our driven application (MPEG4) efficiently.

For this work, we have chosen four platforms that are widely used in the embedded domain: an Application Specific Instruction Processor (ASIP), a soft-core processor with specific functional units implemented on an FPGA, an Application Specific Integrated Circuit (ASIC) and a single chip embedded platform formed by a high performance DSP and a RISC processor. Any of these four platforms can potentially fulfill the requirements of our code with the required optimizations, but all have advantages and drawbacks. In this section, we will first analyze the parameters needed to evaluate the different platforms and then study each of the platforms.

2.3.1. Design metrics

In our work we focus on the technological metrics of the different platforms under study. Those metrics can be evaluated in terms of costs and constraints. We have two different types of costs: implementation costs and design costs.

In the first category, implementation costs, we have area, speed and energy consumption. For a given technology, the area of a chip determines the cost of a VLSI process and this was historically the most important cost of a design. Nowadays, with the increasing capacity of integration, the design area is not that crucial, but still has to be taken into account (it is directly related to the manufacturing costs). In multimedia applications, the main contributions to this cost are due to the processing elements and memories. In today's designs, especially in the embedded domain due to battery life, energy consumption is the limiting factor that drives design decisions and it depends on several parameters from physical to system level: technology, area, frequency, supply voltage, bit activity, algorithm, leakage, etc.

Design costs are related to the system complexity development: programmable solutions have the advantage of software development over the slow hardware development of other solutions. An easy compilation and a retargetable framework is an advantage that can help accelerate time-to-market of products or can facilitate upgrades or new versions of the applications.

Constraints in the embedded domain are of two kinds: flexibility and performance. The flexibility of a chip determines its capacity to deal with different applications and the performance is a real-time constraint related with the time needed to compute a set of operations (with a given energy budget). Beside this definition, in data-flow applications latency and throughput are also used as time-related measures. The latency, in multimedia systems, can be expressed as the delay between data fetched from a sensor or memory and its processed output for its further exploitation (transmission, storage, display etc). Throughput is the amount of data successfully processed per unit of time, and is controlled by the available bandwidth, as well as the available signal-to-noise ratio and hardware limitations. In our case, performance is related to the time needed to encode a macro block and is linked with the latency and number of available resources and the communication between memories (where images are located) and functional

units. Latency in our system is lower than in other Multimedia systems since data is processed at macro-block level and not at image level (with a similar throughput). That way, we do not have to wait to have all images in memory in order to process them.

2.3.2. Platform Independent optimizations

Memory accesses cause 50%-75% of the power consumption on embedded multimedia systems [WCNM96, MNCM97], hence, optimizing global memory accesses of an application is crucial for achieving low power systems. Catthoor et al. propose in [CBGN98, Cat99, Cat02] the so-called Data Transfer and Storage Exploration (DTSE) methodology. The goal of DTSE is double: on the one hand DTSE reduces the storage requirements of embedded applications and minimizes the absolute amount of memory needed to execute the application; on the other hand, DTSE optimizes the locality of data accesses at a very high level in order to reach a high utilization of small but efficient memories which are close to the processor. A more detailed explanation on DTSE techniques is shown in section 6.2.

In order to improve the implementation of our reference model, we carried out some DTSE optimizations. First, the code has been rewritten without any pointers and without using dynamic memory allocation because synthesis tools can not synthetise dynamic memory in hardware. We have also adapted data types so that, for example, if integers do not exceed the value 255, we can only use 8-bit registers. We have changed global loops (image size, for example) to compute the needed calculations at macro-block level; in this manner, the L1 cache size needed decreases for all platforms, and that helps the possible pipeline at block level. Transformations to reduce the size of the data structures inside L1 memories mean that memory inside different platforms has to be small enough to fit in the chip. In addition, image computation is carried out at macro block level (not at image level). These optimizations were implemented equally for the C and the SystemC (platform independent) versions.

2.4. Customized Platforms Architectures

In the following section, we detail the implementation done on customized platforms: the FPGA and the ASIC. Those solutions are based on a behavioral SystemC code synthesis. We perform an architectural exploration at functional units level during the synthesis process. Then, we obtain different compromises in terms of area versus power consumption and execution time of the different algorithms involved in the video compression. This synthesis provides hardware code at RTL level that can be synthesized for a FPGA or an ASIC. The tool used for hardware synthesis is Cynthesizer 5.4 from ForteDS [For], which generates production-quality RTL using a high-level SystemC-TLM (Transaction-level Modeling) design description as input. The technology libraries used in our case are the Faraday UMC 90nm [Far07] ones (worst case technology).

The first step in the synthesis is to carry out a technology-aware architectural exploration of each part of the different algorithms involved in the MPEG application. As

multimedia applications are based on very intensive loops, unrolling the inner loops of the algorithm makes the available parallelism visible and maximizes the usage of the resources. When a loop is unrolled, several things occur and they affect the kind of hardware produced. First, Cynthesizer replicates the operators inside the loop by the number of iterations in the loop's upper and lower limits. Cynthesizer also creates a controller that regulates any iteration-specific items in the replicated hardware. After unrolling the inner loops, we synthesized the resulting code in the different customized platforms optimizing the results for speed.

2.4.1. FPGA platform

We decided to target the MPEG kernel on the Altera-FPGA PCI Express Development Kit, Stratix II GX Edition [Str] as a HW-SW platform for reconfigurable systems and ASIC prototyping. Altera's NIOSII [Nio] soft-core allows designers to integrate custom instructions, hence, system designers can fine-tune the hardware micro-architecture to meet performance goals and furthermore, they can easily handle instructions as macros in C/C++. We modeled the NIOSII with capacities similar to the ones from ARM926 [A04]. The synthesized NIOSII has a data/code cache of (8K/8K) similar to ARM and, a hardware multiplier/divider. It also has a 6-stage pipeline with branch prediction. We have not added a Floating Point Unit (FPU) because our computation does not need floating point calculations: all them were already transformed to fixed point equivalent operations. As peripherals, we selected a SDRAM controller, a SRAM bridge, on-chip RAM and a performance counter (only for debugging purposes). For our comparison, we decided to target the MPEG kernel on the QuartusII-FPGA PCI Express Development Kit, Stratix II GX Edition [Str] as a HW-SW platform for reconfigurable systems and ASIC prototyping. FPGAs are slower than their application-specific integrated circuit (ASIC) counterparts and draw more power because the number of transistors per logic gate is higher and because interconnections use switches and this leads to the chip consuming higher static and dynamic power. Some of their advantages are a shorter time to market, as they save the ASIC mask-based fabrication process, the ability to re-program in the field to fix bugs, and lower non-recurring engineering (NRE) costs. Sometimes, designs are developed on regular FPGAs and then migrated into a fixed interconnection version that resembles an ASIC, called a structured-ASIC, or directly into an ASIC.

Metrics and Energy Estimation Model The RTL verilog code description generated by Cynthesizer is passed through a logic synthesis tool (Synplicity) with the Altera StratixII libraries. Then, we obtain another Verilog description with technology information about the FPGA. This description is loaded in Quartus II for the back-end process to that platform. EDA tools provided by the FPGA vendor offer a tool suite for power analysis and optimization that allows estimating device power consumption and even heat dissipation from early design concept through design implementation [Pow]. Optimizing the register transfer level code generated from the behavioral synthesis tool can be a tedious and difficult process since the automatically generated code is complex

and the interpretation is tricky and dull. Hence, manual optimizations at this level are quite difficult since they have been already made at higher level (behavioral SystemC). The produced code is prepared automatically accordingly, and its reading is knotty. We have generated the NIOSII architecture and programmed the C code.

Communication between the kernel part and the control part is done with FIFOs following a previous work [MHB08]. In this work, high level FIFO channels are automatically replaced by real hardware FIFOs. This approach provides a mechanism of communication for those parts of the common description of the system that will be implemented in SW by a processor, with the pieces that will be synthesized to hardware, without any modification of the system description structure and preserving blocking synchronization. In the proposed system, there is a FIFO with a macro block size customized to the MPEG characteristics; that connects the kernel part (Q-zigzag algorithm) and the Huffman algorithm that is computed on the NIOSII.

From the processor perspective, the FIFO is shown in its memory map as unique address position, or equivalently, as a device with only one register. So, block transfers are done simply by consecutive writes or reads to this position. The blocking mechanism is handled by connecting the *wait_ request* signals of the NIOSII system bus with full-empty signals of the real FIFOs. From the kernel point of view, just after the synthesis stage, high level FIFO channels are replaced by a RTL wrapper designed for synthesized modules to access the FIFOs, maintaining the blocking mechanism. Huffman is computed in the processors and results are written to another memory space that is another FIFO with similar size, where the compressed image is produced and afterward, sent again to a L1 memory.

Platform Dependent Optimizations Manual optimizations at this level were carried out to manage L1 memory (from image to macro block size). As a result, the amount of memory required to implement the MPEG compressor has been significantly reduced thanks to the DTSE transformations. After different optimizations, the memory needed by the application is not too high and the design fits in a medium size FPGA. This transformation was carried out at SystemC level, taking into account the size and structure of the different MPEG memories and according to the Altera RAM block types (M-RAM, M512, M4K and LCs). Available memory bandwidth has to be as high as possible in order to compute the maximum number of pixels in parallel that loop unrolling permits. Unrolling DCTs makes different multiplications run in parallel. We fitted these multiplications in the FPGA MAC blocks (y=a*b+y'), which accelerated their computation.

2.4.2. ASIC platform

Cadence's BuildGates [Cad07] tool was used for ASIC synthesis. Using the technology logic cell library, the synthesis tool performs the process of transforming the ASIC register-transfer level (RTL) description into a technology-dependent netlist. The netlist is the standard cell representation of the ASIC design at the logical level. It consists of the standard-cell library element instances, and port-connectivity between them. Proper

synthesis techniques ensure mathematical equivalence between the synthesized netlist and the original RTL description.

Since both RTL and gate netlists views are only useful for abstract (algebraic) or approximate-time simulation, but not device fabrication, the physical representation (layout) of the chip must be designed as well. The layout view is the lowest level of design abstraction in common design practice. From a manufacturing perspective, the standard cell's VLSI layout (even if cell layouts are black boxes) is the most important, as it is closest to an actual "manufacturing blueprint" of the standard cell.

Metrics and Energy Estimation model The synthesis of the Verilog RTL code generated with Cynthesizer from the original model is used by BuildGates to generate a Verilog structural description in which the basic elements are cells (standard cells and macrocells) belonging to the library designed for the target technology. At this abstraction level, all modules are hardware descriptions and therefore a complete exploration of all nodes is possible. BuildGates provides a complete power report when the synthesis is completed (still without physical back-annotation). The tool provides information about the internal cell, nets, and leakage power. Memory power is obtained from the datasheet provided by the Faraday Libraries when they are created. Then, the energy per activation and leakage power for the MPEG is obtained with a gate level simulation with Verilog-XL.

Platform Dependent Optimizations We carried out several DTSE transformations at MPEG design time to fit the code structures into the L1 memories. Therefore, the percentage of on-chip memory integrated with respect to the global requirements of the MPEG, ranges from 25% for the implementation optimized for size, to 45% for the implementation optimized for speed. Architectural exploration analysis at this level using simulations (for verification of results) is undoable due to the extended time necessary to carry out simulations. Simulations that take seconds at TLM level (as for example the compression of an image) would take days at RTL level. Therefore, the simulation must be carried out with a very small image size, for example using a single macro block instead of the whole image, according to the code transformations done at the beginning of our work.

In our design, we built an ASIC for the MPEG kernel part in 90nm UMC Faraday technology (Faraday cell libraries used for the UMC technology). We fulfilled the placement and routing of the chip using Cadence tools. Further verification must be done around the back-end process, such as pre- and post-layout simulations, to ensure large enough fault coverage (around 95% of nodes verified with the test vectors). In this case, memories where restricted by the ones available in the cell library, and the control part of the algorithm was implemented in an ARM9 processor tailored for the same technology, following the same strategy as in the FPGA case for the NIOSII soft-core processor.

2.5. Instruction Set Processor platforms

A unique C++ model is directly translated to a C description to be mapped onto the DSP and ASIP platform. Even if this C description has special characteristics (such as: no pointer or no dynamic memory allocation), the code is still not fully optimized and some optimizations are needed to extract the maximum performance of the application. Both implementations, DSP and ASIP, have an Instruction Level Parallel compiler that does standard compiler optimizations. Compilers targeting the embedded domain face performance improvements on area, energy and performance. Thus, compiler optimization has become an essential component of high-performance and embedded computer systems. An extensive bibliography on traditional compiler optimizations can be found in [Muc97, AK02, ALSU06].

Beside the usage of compiler optimizations, since multimedia applications are based on intensive data computation, the reduction of transfers between different memory levels can considerably reduce energy consumption. Data Transfer and Storage Exploration (DTSE) optimizations [CBGN98, Cat02] aim at improving spatial and temporal locality to minimize the load of shared buses, which is the main source of power consumption [WCNM96]. These transformations are usually achieved at a high expense on addressing and local control. Some source code optimizations reduce control flow. Control flow optimizations are based on mathematical models and modify source code, trying to minimize the number of iterations that control-flow statements evaluate. Loop nest splitting minimizes the number of executed if-statements in loop-nests of embedded multimedia applications. Advanced code hoisting moves portions of the inner loops to outer ones. Ring buffer replacement eliminates small arrays serving as buffers for temporary data. These three techniques reduce control flow, arithmetic calculations and execution time and hence, they minimize energy consumption ([FM03, Fal05, FM04]).

2.5.1. Digital Signal Processor - Based Platform

There are significant differences between the different families of DSPs and a good election is highly decisive. Due to the nature of our code, as a representative DSP-based platform, we selected the Texas Instruments DaVinci platform [DMS]. The DaVinci solution is a DSP-based System-on-Chip (SoC) platform optimized to fulfill the performance and feature requirements for a broad spectrum of digital video applications. More specifically in our case we used the TMS320DM644 [TIO, Inc02a] which, in addition to the DSP core, has an ARM926 [A04] processor. The core of the DSP is a VLIW processor core specially designed to exploit instruction level parallelism and to maximize channel density in I/O chip communications. It contains two identical clusters with four FUs each and represents a typical clustered DSP.

Framework The Code Composer Studio [Inc02b] is the development framework provided by Texas Instruments for the TMS320DMxx family. The Setup CCStudio provides an interface with the development board that contains the DSP. Independently of the

board used, the framework permits the processor, the working frequency and external memory on which the application will run to be configured. The Digital Video Evaluation Module [TIO, Inc02a, oCS08, Bha08] enables developers to immediately evaluate the DaVinci processors and building digital video applications, it also allows developers to write production-ready application code for the DSP and ARM926EJ-S.

Metrics and Power Model The code memory requirements, after an accurate DTSE exploration, ensure that the required memory structures are small enough to fit them all in the DSP L1 internal RAM. The kernel part is loaded in the DSP while the control part is loaded in the ARM (similarly to the hardware implementations). The bandwidth of the coded memory structures is large enough to allow the DSP to use all its potential, that is, the eight Functional Units (FUs) with software pipelining.

The power consumption of TMS320DM6446 [Bha08] can vary widely depending on the use of on-chip resources. Thus, power consumption cannot be accurately estimated without an understanding of the components of the DSP in use and the usage patterns for these components. Texas Instruments offers a spreadsheet where, by providing the usage parameters that describe what is being used in the DSP and how, we can obtain accurate consumption numbers for the power-supply and thermal analysis. This model breaks down power consumption into two main components: static and dynamic. Using this model, different applications that use the DSP differently can obtain accurate predictions over the entire spectrum of possible power consumption in the platform. To use the spreadsheet, simply select the case temperature for which you want to estimate power and fill in the appropriate module use parameters. The spreadsheet takes the provided information and displays the power consumption details for that configuration.

Platform Dependent Optimizations Usually, programmers write code following the model or functionality and make the code as readable as possible. For a SW designer using a DSP, the easiest method to optimize the application is to let the compiler handle their different optimizations. In fact, these optimizations result in a huge reduction in the execution time, and hence consumption. Among other traditional optimizations ([Muc97, AK02, ALSU06]), the compiler is in charge of analyzing the different loops and determining if unrolling certain loops is worthwhile, in which part of the code there are good candidates for software pipelining, etc. For each analysis, the compiler must evaluate the factor that the code will be improve by. Nevertheless, the results achieved with the compiler are not always the best possible results. This can occur for two reasons: first, good "human-readable" code is usually not so good for the compiler because it can hide possible optimizations; and second, the compiler does not have enough information for carrying out more aggressive optimizations and uses more conservative solutions to assure that the algorithm is executed correctly.

SIMD instructions: Some Texas Instruments DSPs are capable of using Single Instruction Multiple Data (SIMD) operations. SIMD operations are instructions employed to achieve data level parallelism in a similar way as vector processors. SIMD operations in Texas Instruments compilers are based on intrinsics. Intrinsics are functions handled

especially by the compiler and usually substitute a sequence of automatically-generated instructions, in which the compiler has an intimate knowledge of the intrinsic function and can, therefore, optimize it better. The inclusion of SIMD instructions can be automated by the compiler; however, in most cases hand tuning of the code is needed since the compiler does not have enough information to detect when it is possible to use SIMD.

In house libraries: TI, as most DSP vendors, supplies libraries specialized in several aspects, such as the IMGLIB that provides some functions commonly used when working with image manipulation. Among these, there are functions that implement IDCT, FDCT, quantization, etc. Using these functions leads to a considerably faster execution. Basically, this library (and related functions) exploits maximum SIMD instructions and uses the maximum bandwidth of the data buses. To use these functions correctly, the parameters for passing data have to fulfill certain requirements: data passed to the function must have a specific format and should be aligned in the memory. If data types of the algorithm are not compatible with the required format, these functions cannot be used.

The most computing intensive algorithm is the ME with the SAD computation. We manually unroll the inner loops to "help" the compiler to extract parallelism. Before using instructions that take profit of SIMD architecture, the bottleneck was sharing data between on-chip memory and FUs. The use of SIMD instruction set permits to make more than one operation simultaneously on data vectors. An example is DOTPU4*src1* , *src2* that realizes multiplication with 4 vectors operands between operands *src1* and *src2* and afterward realizes the scalar addition of the results. With those instructions, the number of executed instructions is reduced while the number of read writes to memory is kept constant. With this, the bottleneck remains exclusively in the data memory and data cross paths, responsible of moving data from the register file to the functional units. This happens always with the ME computation, where execution requirements between blocks are few operations but very intensive, causing the ME to account for 60% of the total time of coding. More intensive use of SIMD instructions can be made in the code, for example, using the DCT function that is part of the TI library.

2.5.2. Application/Domain Specific Instruction Set Processor-ASIP/DSIP

An ASIP is a processor dedicated to one application or a small set of applications. Even if we consider that ASIPs target a less broad set of applications than Domain Specific Instruction Set Processors (DSIPs) we use both terms as synonyms since usually this distinction is not contemplate. ASIPs (or DSIPs) have a customized instruction set or specific hardware resources and require an ILP compiler. Since ASIPs are optimized towards certain applications, they combine high performance and efficiency of a dedicated solution with the flexibility of a programmable solution, and hence, they fill the gap between highly optimized platforms like ASICs and the more flexible solution offered by DSPs. Developing an ASIP requires a large hardware and software design effort, but it

can be reused easily, since new versions of the product only need to compile the new source code [FFY04].

Framework Some ASIPs have a configurable instruction set and these cores are usually divided in two parts: static logic, which defines a minimum ISA, and configurable logic, which can be used to design new instructions.

Low energy is one of the key design goals of the current embedded systems for multimedia applications and Very Long Instruction Words (VLIW) ASIPs, in particular, are known to be very effective in achieving high performance with reasonably low power, which is the main objective in the domain of interest.

For our experiments, we used a retargetable VLIW-ASIP compiler and simulator framework based on Trimaran [tri99] called COFFEE (explained in detail in chapter 4). This framework can map applications to a broad range of processors and memory configurations with different instruction set architectures and can also simulate and report detailed performance and energy estimates.

Possible compiler research optimizations for these architectures are high-level machine independent code transformations and "back-end" machine dependent optimizations such as instruction scheduling, register allocation, etc. The architecture used for this work is VLIW-based and, as for all VLIW processors, the processor executes instructions that are composed of parallel operations that are executed in the different Functional Units (FUs) (sometimes using coarse-grain reconfigurable logic). The processor's data path is divided into clusters (or slices), which contain one or more register files, one or more functional units, and bypass logic. This well-known division of the data path into clusters reduces the energy consumption and the data-path delay by reducing the complexity of the register files and the bypass logic. Since reducing the number of cycles is also an effective way of reducing the energy consumption, all data-path operations can be predicated to enable efficient execution of loops with conditionals. Furthermore, chains of data-dependent operations can be executed in a single clock cycle, using software controlled functional unit chaining to decrease the effective latency of a set of operations, and hence reduce the number of clock cycles. Thanks to this chaining, some register file accesses can be prevented, which reduces the energy consumption further. The COFFEE framework allows broad compiler research and instruction set exploration to obtain an optimal ASIP architecture for the required application. A detailed description on the framework is explained in chapter 4.

Design Metrics and Energy Estimation Model At this abstraction level, the complete hardware description is not yet needed and therefore exploration can be faster. To obtain early estimates with sufficient accuracy, we carried out the following methodology: different instances of the components of the processor (Register File, ALU, pipeline registers, etc.) were designed at RTL level with an optimized VHDL description. In our case, for each instance, logic synthesis was carried out with the UMC@90nm general purpose standard cell library from Faraday [Far07], also used for the ASIC power model. The result of the process is a library of parametrized energy models. Energy per acti-

vation and power leakage for the different components are estimated from the activity information from gate level simulation and the parasitic information. Memories used for this analysis are highly optimized custom hardware blocks and hence, the standard cell flow cannot be used. We created a library of memory blocks with the corresponding energy consumptions (dynamic and leakage) using a commercial memory compiler (from Artisan). Finally, our pre-computed library contains the energy for various components of the processor using standard cell flow, and for memories, using the commercial memory compiler. A detailed description of the complete energy model is also described in chapter 4.

Platform Dependent Optimizations In addition to the platform independent optimizations explained in Section 2.3.2, we carried out a generic architectural research exploration (see chapter 6 for more details).

Scratchpad memories (SPM) are high-speed memories where the compiler generates explicit instructions to move data from and to the following levels of the memory. SPMs have been proven to be more energy, area and performance efficient than caches [BSL+02a, AC06] and they are an optimal choice for embedded systems. Moreover, in any typical multimedia application, significant amount of execution time is spent in small program segments. Energy can be reduced by storing them in a small loop buffer instead of the big instruction cache *[LMA99, AJB+04, JBA+05, VJC+06].*

Several steps of DTSE insert complex control flow. Removing this control flow is crucial to get an optimal implementation of any program; the techniques explained in [FM04, Fal05] address this problem.

The data path can be composed of one or more clusters. It contains one or more register files and one or more FUs. By clustering, the routing length and interconnect complexity inside a cluster are reduced (i.e. good for both power and speed), at the price of increased compilation complexity due to the additional cluster-to-cluster data transfers. To get the optimal configuration of the data path a research exploration was done [FFY04]. The application under study has some recurrent operations that consume excessive cycles: multiply-and-accumulate operations and divide operations, very popular in multimedia applications, can be accelerated using specific hardware support. This will improve energy reduction significantly and also boost the speed up decreasing the number of cycles needed for the application. We added hardware support for multiply-and-accumulate and divide operations.

Once the memory and data computation have been optimized, the bottleneck resides in generating the application code addresses [TJCC08]. Address computation is extremely important in multimedia systems and often involves linear and polynomial arithmetic expressions that have to be calculated during program execution under strict timing constraints. Since multimedia applications are based on intensive deep-nested loops in which the majority of the code is executed, a specialized address generator unit was developed to speed up all address related computations in the inner-most loops. Address generation in these parts of the code is easy and does not require a lot of hardware support, hence minimum hardware and a small number of registers are sufficient to

generate the addresses of the inner-most loops. Addresses for the rest of the application are computed in the default (more complete) address generation unit, which is similar to a normal functional unit (see chapters 5 and 6).

2.6. Final platforms results and comparison

In our work, we have chosen four platforms widely used in the embedded domain. Any of those four platforms can potentially fulfill the requirements of our code with the required optimizations, but all of them have advantages and drawbacks. In this section, we will first analyze the parameters used to evaluate the different platforms and after that, we will study each one of those platforms.

2.6.1. Results

We did a fair comparison mapping the same realistic application on different platforms with the same technology (90nm). Different implementations came up from the same original source code where we performed platform independent and platform dependent optimizations to exploit the maximum benefit from each targeted device, therefore, comparing results between optimized application-platform, making an impartial comparison. Depending on the available design time, area, performance, energy consumption, flexibility and architectural requirements, designers can choose the most suitable platform style for their final implementation. The results shown in the following two tables 2.1 and 2.2 are for two different MPEG implementations: the first implementation refers to the FSME algorithm and the second one to faster ME, with a logarithmic search. The former is representative of quite regular algorithms with a high access locality. The latter is representative of algorithms with a more irregular control flow and less local access, leading also to a more difficult address scheme. In both cases, for the FPGA and the ASIC, we did an analysis for a basic version, optimized for size, and another version optimized for speed. These two hardware implementations have been targeted to an FPGA and to an ASIC. The ASIP also has two different implementations (as seen in section 2.5.2). The framework used is mostly based on optimized macro-blocks which can be parametrized (a.k.a ASIP-Macro). This is the most suitable style to use because an ASIP template (with parameters) can be heavily reused for different chip instances. So, the effort added to develop the macro library is well worth for the most critical blocks in the architecture like the data and instruction memories, register-files, and multipliers. The glue logic in between those macros will of course still be implemented with a conventional standard cell flow.

For comparison, we also give the results of the same ASIP based on a standard cell implementation (a.k.a ASIP-Std-Cells). Even if we recommend implementing the ASIP based on optimized macro-blocks, the ASIP based on standard cells gives a more direct comparison with the ASIC since we used the same technology running at the same frequency. In the following tables, we show the results in terms of energy consumption, execution time and area for the different implementations (ASIC, ASIP, DSP and

		ASIC Opt size	ASIC Opt.speed	ASIP (standard Cells)	ASIP Macro	DSP (Da Vinci)	FPGA (opt size)	FPGA (opt speed)
Energy (mJ)	Memories	15,9	22,6	23,4	20,1	not available	not available	not available
	Leakage	0,5	0,3	0,5	0,2	not available	not available	not available
	"core"	0,04	0,04	2,6[1]	0,95[1]	not available	not available	not available
	Total	**16,5**	**22,9**	**26,4**	**21,3**	**536**	**2626,5**	**1188,3**
Time related	**cycles (in millions)**[2]	88,4/60,9 (14,4)	38,6/25,3 (14,4)	84,6/55,5 (14,0)	84,66/55,5 (14,0)	282,36/185	88,4/60,9	38,6/25,3
	Execution time (s)[2]	0,20/0,14 (@450MHz)	0,14/0,06 (@450MHz)	0,19/0,12 (0,03) (@450MHz)	0,17/0,11 (0,03) (@500MHz)	0,48/0,31 (@590MHz)	2,0/1,4 (@43,34MHz)	0,92/0,61 (@41,82MHz)
	Frame Rate[4] **(qCIF)**	15,3/22,2	35,0/53,3	15,9/24	17,7/27	6,3/9,6	1,5/2,2	3,3/4,9
Area (mm^2)		2,89	6,25	6,05[3]	2,42	11,4[5]	↑↑	↑↑
Design time	**First Implementation**	**HIGH** Synthesis front-end + back-end		**HIGH** (first version) ≈ ASIC development + compiler development		**LOW** Code compilation	**MEDIUM** Synthesis front-end	
	Following implementations	**HIGH** Synthesis front-end + back-end		**LOW** (following versions) Code compilation		**LOW** Code compilation	**LOW** Synthesis front-end	

[1]:the ASIP core contains the energy of the FUs and the register file.
[2]:the first number represent the cycles or execution time related to the kernel of the application (the number between brackets refers to the control part).
[3]:area for the ASIP based on std cells is an estimation.
[4]:first value GOP IPB second value IPP (IPB/IPP)
[5]:information obtained from the references [BKM06] and [AKA+02] .

Table 2.1.: FSME MPEG Summarized results

		ASIC (opt size)	ASIC (opt speed)	ASIP Std.Cells	ASIP Macro	DSP (Da Vinci)	FPGA (opt size)	FPGA (opt speed)
Energy (mJ)	Memories	2,3	5,9	9,7	8,9	not available	not available	not available
	Leakage	0,06	0,07	0,19	0,09	not available	not available	not available
	"core"	0,01	0,01	1,78[1]	0,87[1]	not available	not available	not available
	Total	**2,39**	**5,97**	**11,69**	**9,97**	**93**	**535,4**	**434,5**
Time related	**cycles (in millions)**	18,1/14,9 (14,3)	14,5/11,3 (14,3)	39,6/29,7 (14,1)	39,6/29,7 (14,1)	49/36,8	18,1/14,9 (14,3)	14,5/11,3 (14,3)
	Execution time (ms)	43/36 (@450MHz)	35/27 (@450MHz)	88/66(31) (@450Mhz)	79/59(28) (@500MHz)	83/62 (@590MHz)	416/342 (@43,54MHz)	337/263 (@42,98MHz)
	Frame rate [4] (qCIF)	74,7/90,3	93/119	34,1/45	37,8/50	36,1/48,2	7,2/9,6	8,9/11,9
Area (mm^2)		2,95	6,67	5,1[3]	1,73	11,4[5]	↑↑	↑↑
Design time	**First Implementation**	**HIGH** Synthesis front-end + back-end		**HIGH** (first version) ≈ ASIC development + compiler development		**LOW** Code compilation	**MEDIUM** Synthesis front-end	
	Following implementations	**HIGH** Synthesis front-end + back-end		**LOW** (following versions) Code compilation		**LOW** Code compilation	**LOW** Synthesis front-end	

Table 2.2.: FAST MPEG Summarized results

FPGA). First of all, we should remember that, as explained throughout this chapter, the ASIC was implemented from a high level SystemC description of the application. According to our experience, the behavioral synthesis tool, even if it allows for a very fast development, leads to a final design larger and slower than what an average engineer (or design team) could achieve by direct manual coding in a hardware description language such as VHDL or Verilog (obviously after investing significant design effort). This implies that the design coming from SystemC (the ASIC and FPGA implementation) is larger, slower and has higher energy and leakage consumption than what a "good hand-coded design" can achieve. Despite the fact that these results can be improved, they still give a fast and representative idea of a final project result where design time is limited, i.e. for most practical chip projects today. It is already a remarkable result that the domain-specific programmable processor (the ASIP) and the standard-cell based ASIC platforms give, for each algorithm, results of the same order of magnitude, with differences depending on configurations (ASIP-macro, ASIP-std-cells, ASIC-basic or ASIC-optimized). As the application is data-flow dominated, the energy needed to access the memories is similar in the ASIPs and the basic implementation of the ASIC since memories are of the same size and technology and also the global architecture can be kept largely the same due to the ASIP style. That is one main difference with the more general focus of the DSP style. The optimized (for speed) ASIC consumes much more energy in the memories since more memories are used. This boosts the execution time at the penalty of increasing considerably the energy consumption and chip area. The FSME algorithm is more computing intensive and the access to the memories gives the major contribution to the final energy consumption for the ASIC and the ASIP versions. For this algorithm, the difference in cycles between the ASIC-basic implementations and the ASIP is of the same order of magnitude. For the Logarithmic ME algorithm the difference between the results of the ASIPs and the ASICs is higher. This is because this algorithm has much less data transfers and the contribution of the control has larger impact on the cycles needed to complete the encoding: ASIPs are better focused on data access and computing intensive applications and suffer from irregular control flow. Some techniques such as predication alleviate the performance problem but increase the energy consumption. Other, more recent solutions have been proposed [PAM+07] which alleviate both the speed and energy problems but that are not yet available in the main-stream commercial solutions. As we can see in [RLJ+], the usage of a distributed loop controller in addition with condition support improves considerably the performance and energy consumption (around a 30%).

2.6.2. Energy considerations

The energy is lower in FAST than in FSME (table 2.1 and table 2.2) for all platforms because less computations are needed to process the pixel data structures and the control operations are similar. The main energetic contribution is due to the memories. In the case of the ASIC we take into account that all memories are switching at the same time. Therefore, this scenario is considerably worse than in ASIPs implementations where just the simulated switching was taking into account. The leakage of energy for ASIP and

ASICs are similar: ASIC opt. for speed has more energy consumption because it trades-off power and execution time. The energy consumption is higher when the execution time is lower producing that the dynamic energy would be similar. ASIP macro architecture is more optimized and therefore the leakage power is lower. This is mainly due to two reasons: ASIP is running at a higher frequency (500MHz instead of 450MHz), and, finishing faster the computation, the time the design is leaking is considerably reduced leading to a half of the energy consumed (for leakage only) compared to the ASIC version. As mentioned in the previous paragraph, a hand-coded ASIC must give better results, in terms of energy consumption and leakage, so the difference with the ASIP version will be also reduced. To reduce the dynamic energy of the ASIC implementation we could decrease the voltage supply. This will lead to a reduction in the dynamic energy at the cost of decreasing the frequency and hence increasing the time needed to compute the algorithm. Also, we have to take into account that by raising time with the device working, the leakage increases. Therefore, it is not that worth to try to improve dynamic energy in the ASIC, at least not for the type of data-intensive applications considered here. The biggest difference between ASIP and ASIC implementations is in the energy of the core: the ASIP also contains the energy due to the register file contribution and this implies that this energy is several times higher than the ASIC which does not have a "power-hungry" register file.

The FPGA is, by far, the worst in terms of energy and clock speed. This difference is much higher than what has been reported in earlier comparisons based on simple kernels [RAdSJ+00]. It is clear that even much optimized fine-grain cell alternatives, as proposed also earlier in literature [WZG+01], would not be able to bridge this huge gap. There is the possibility in the FPGA to improve the energy consumption producing a structured ASIC with the Hard-Copy utility [ALT08]. The ASIC produced can decrease in energy consumption around a 70%. Although the produced structured-ASIC has decreased the energy consumption, it still has really higher consumption compared with the other implementations. The results on the DSP show that the core consumption consumes around a factor 10 more than the ASIP implementation. This is mainly due to two reasons. First of all, the platform dependent optimizations, basically the inclusion of specialized hardware (for the multiply and accumulate and division operations and the specialized address generator unit). Second, we have to emphasize that DSP memories are fixed and built for general purposes and for the ASIP case, we tuned all the memory hierarchy for this application (sizes, loop buffer, number of ports, etc.).

2.6.3. Time related considerations

The execution cycles in the FSME for an ASIC optimized for size are 88,4M, very similar to the ASIP version (84,6M). A better result is given by the ASIC optimized for speed which gives 38,6M. That is twice faster than the best ASIP approximation. The DSP results can be improved since we have used partially the low level assembler TI libraries to implement the function of the video coder. The execution time is given in a worst case where the GOP includes an I frame, another P and finally a B frame QCIF(176x144) size and 8 bits pixels. The values given in the table (in million cycles) are two: one for the

GOP IPB and another for the GOP without B frames just a IPP. This GOP has at least 30% lower computation and, therefore, it has a considerable million-cycles reduction. Then, the frame rate has two possible numbers, one for the GOP IPB and another for the IPP. In brackets (in million cycles too), there is the number of cycles needed to compute the control part; this computation can be done in parallel with the kernel part. We have to take into account that the kernel part is faster than the control one. We have to improve it, decreasing the needed cycles to compose the control computation, because we do not want them to be dominant in our application.

When the frame rate is above 24 images per second, we will establish (according to video and TV standards) that we are working in real time. For the FSME, we get real time just for the ASIC optimized for speed solution. But for the FAST implementation, we get real time implementation for all the solutions except the FPGA due that its achieved maximum working frequency is too low. Improving frame rate would be possible by producing GOPs with just I frames. The number of frames that we can achieve with ASIC opt. size and ASIC opt. speed for an III GOP (without prediction and bidirectional frames) is 123 and 203 images/sec (QCIF size). The corresponding number is 116 images/sec for the ASIPs macro solution, 113 images/sec for DSP and finally 27 images/sec for FPGA opt. for speed. Therefore, we can get real time for mobile CIF (352x288 pixels) in ASIC, ASIP and DSP. One must remember that this coder computes in worst case scenario since all the macro-blocks are computed. If we change the implementation, for example, when data or residual values are lower than a given threshold, then we can decide not to compute them. Then, we could increase a lot the speed of the solution but losing some quality performance.

2.6.4. Area considerations

In the ASIC, area is doubled due to FUs parallelization that makes the speed increase for more than 3 times for FSME and 25% for the FAST implementation. This is due to the fact that in MPEG, ME is the dominant algorithm and, Amdahl's law [Amd67] argues for accelerating those parts of the system. In contrast, in the FAST implementation, ME is not as dominant as the rest of algorithms and the increase on speed improving this part is not as impressive as with FSME. The FSME ASIP solution has an area similar to the size-optimized ASIC. The ASIP std. cells solution, the area is 5,1 to 6,05 mm^2 and the speed is half compared to the corresponding ASIC solution. The DSP solution has a higher area than ASIC and ASIP because the DSP is more "general purpose" than just a video encoder and has more resources (that are not used by our application). The area in the FPGA is the highest of all the solutions because of its regular and repetitive structure (larger cost in transistors per gate) and because of the configuration capabilities.

2.6.5. Design time considerations

The design time for the ASIC implementations is quite long since hardware development is slow and the verification process consumes 70% of all the design time. It is clear that

this solution gives the best results in terms of energy consumption and execution time (with the same clock frequency); but its main drawback is that new versions of the design, adding new functionalities, or changing the algorithms, need an almost completely redesign process, with the cost on human resources, development time, etc. that this entails. With more advanced deep submicron technology nodes, the processing costs will become even higher, so the ASIC option is becoming very difficult to motivate, except for a few extremely high volume markets. Moreover, end-users that acquire a new device are used to "benefiting" from the upgrades on the functionality. The development of an ASIP is even higher for the first design since, in addition to the design and development of the hardware, the ASIP requires a compilation framework. The main advantage is that this one-time effort is worthy since upgrades on the algorithms or new functionality just require to recompile the extended code. This is a cunning advantage for the embedded domain, where totally new applications are limited (inside an application domain), but they change quite fast with upgrades, new protocols, and more functionality. End-users can benefit from upgrades, or even new features, just by downloading a new version of the firmware. Mapping applications onto commercial platforms (DSP or ASIP) benefits of a really fast development since no hardware development is needed but the results are far from the customized platforms. Nevertheless, optimizing the source code improves this situation.

The more generic DSP processors, like DaVinci, provide a quick implementation if the requirements of the application (in terms of real time, energy consumption) are not very tough. If the application is more complex, the FPGA solution must be considered again since this platform permits a HW/SW full parallel and concurrent implementation. But it must be stressed that, in the FPGA solution, energy consumption is the highest, and therefore, it is not a good embedded solution when battery life or power consumption are limiting factors. As expected, if you have time and resources, ASIC or ASIP solution can provide a more optimized solution.When the development time cannot be too high, an off-the-shelf solution, such as the DaVinci is the best one. FPGA solution is just for prototyping when some parts have to be accelerated with hardware and because power consumption is unacceptable for battery-powered embedded systems. Some results can be obvious but numbers are extracted from the same source code and provide us with a realistic view among implementations. This work is realized with a video encoder but this methodology can be extended to other multimedia space solutions such as 3D video games, or other fields that are not multimedia such as biomedicine, robotics or artificial intelligence where the technology requirements could be rather dissimilar. Other improvement would be to make this work not just for one task but for several tasks since in this case the concurrency management would have given different results.

2.7. Conclusions

As seen in this chapter, different solutions are present when targeting the embedded multimedia domain. All solutions have drawbacks and advantages and the constraints

argue as much for choosing one solution as they do for another. One major objective of our work is to quantitatively identify all the elements needed to provide the system designer with a very early estimation, and what platform style/subclass is the best choice to implement the target system. In addition, we provide actual comparative values among the different platforms and the results come from a single original description, with enough effort spent on each of them to come to a sufficiently optimized solution. Therefore, these options are compared also in a fair way. The results are exposed for a video compressor solution but they can be extended to any data-flow dominated system. Most multimedia applications nowadays fall into that class.

A widespread misunderstanding nowadays is to think in an embedded FPGA as a final solution for small volume of production. It is necessary to keep in mind that FPGA consumes more than other solutions even in the case of a structured ASIC solution. Even if power consumption is not a negligible issue, the most limiting factor of the FPGA solution compared with the ASIC one is the factor ten in terms of clock frequency. For that reason, if real-time requirements are very tight, this FPGA solution can still not compete with more aggressively optimized solutions such as ASIC or ASIP.

Figure 2.2 shows the remaining different design styles (DSP, ASIP and ASIC) and the key metrics on which they are efficient. ASICs are known for their high performance and energy efficiency. DSPs on the other hand are flexible as well as able to deliver the performance, but they are not energy efficient. Application Specific Instruction-set Processors (ASIPs) on the other hand try to combine all three metrics. ASIPs try to reach the same energy efficiency and performance of an ASIC while still being flexible. To design such an efficient ASIP or embedded processor, it is necessary to observe the high level requirements and trends in the domain of processor design. However the design effort for mapping code on an ASIP is higher.

An off-the-shelf DSP gives good results and, if meets the requirements, this is the most suitable solution since it has a fine compromise between energy consumption, area and design time and can handle the real-time requirements imposed. To achieve the same performance, an FPGA will be excessively power hungry and an ASIP or ASIC will demand too much developing time, even if the performance and consumption will be improved. The main problem of choosing a DSP is that this solution cannot be easily scaled if the requirements of performance increase. In this case, when a major change of the performance will be needed (for example, for following versions of a product) the architecture style exploration in the project needs to be done again from scratch.

The ASIC solution makes sense when the number of chips to produce is very high and it provides the best compromise in terms of energy, speed and also area. ASIP and ASIC must be the solutions if we desire to come up with the state-of-the-art hardware architectures with better performance and hence to be used for top requirements as for example: real time video compression of 1080i (1920x1080) pixels images size, 60fps, I, P and B frames type or even in the future 2160p (3840×2160).

An ASIC brings (or can bring) more performance but incremental engineering is expensive, slow and filled with delays. Moreover, an ASIC cannot keep up with changing market conditions and can not adjust to new standards that might be needed for the following versions of the product. The ASIP requires more developing time in the first

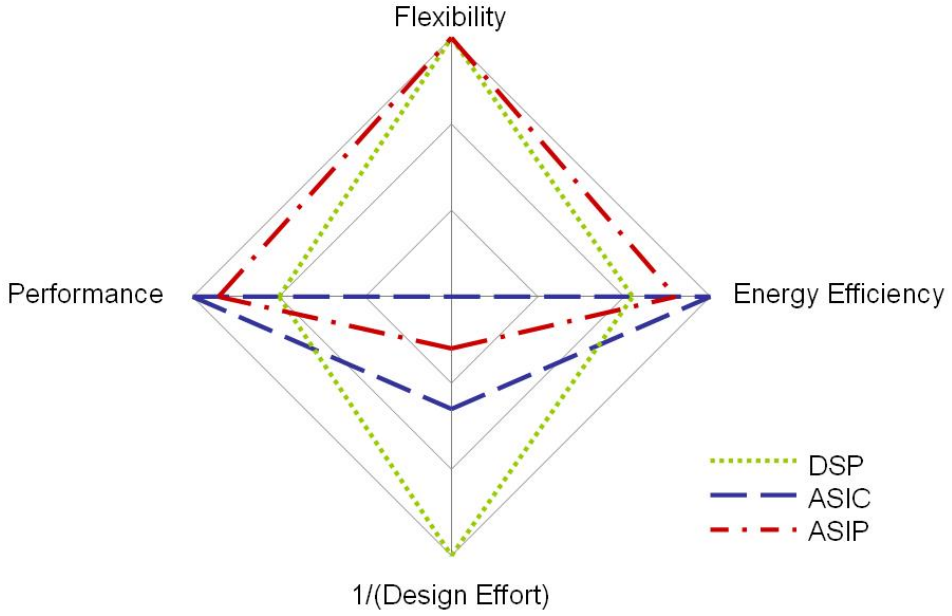


Figure 2.2.: Different design styles target different design metrics

version of the product, since hardware, software and compiler must be developed. After that, any upgrade or change in the product just needs a compilation of the new application.

The ASIP solution requires more development time in the first version of the product, since hardware and a compiler framework must be developed (see chapter 4). After that, any upgrade or change in the product just needs a compilation of the new application. The choice of an ASIP is easily justified when none of the commercial solutions provides enough performance at the required energy-efficiency (given the battery limitations) to satisfy all our requirements. An ASIP solution can also make sense if commercial devices can not satisfied the requirements or if the image size, and hence the performance and energy efficiency must be boosted. It is also the desirable solution if we know in advance that several additional follow-up products will be designed by the same team on the same platform. Then, even if off-the-shelf customized components are available for assembling the platform for each of these different product versions, a larger first investment in a flexible domain-specific platform will pay-off (see chapter 6) for the effort and time saved in the following product versions. Obviously, even when the DSP meets the performance requirements, that quantitatively evaluated energy reduction and the corresponding chip cost can also already be an incentive to go the ASIP in systems that are battery-constrained which is the normal case in embedded multimedia domain.

CHAPTER 3

State of the art on address generation

"Experience is simply the name we give our mistakes."

Oscar Wilde

In this chapter we will focus on AGU architectures and on compilation techniques to optimize the address generation process for scratchpad memories due to the power restrictions of the embedded domain. We will especially consider address generation for DSPs and VLIW-like architectures which have to deal with computing intensive algorithms where access to data is a main issue. The main contribution of this chapter is to provide a systematic classification of address generators and a review of literature according to the classification to illustrate the complementarity or overlap of the optimizations on the address generation process.

3.1. AGU model and classification

Programmable architectures oriented to exploit parallelism, Digital Signal Processors (DSPs) and multimedia processors (a mixture of RISC and DSP processors), usually follow the VLIW paradigm: a number of Functional Units (FUs) running in parallel, following a schedule generated by a compiler [Leu00a]. These architectures focus on real-time performance and often deal with infinite, continuous streams of data.

To access the data, an address generator unit works in parallel with the main data calculation units to ensure efficient feed and storage of the data from/to the data path. Besides the access time and the parallel access constraints, the main problem is the efficient generation of the address sequences for a given application.

The generation of an address sequence is done from an address equation, which is a function extracted from the software description of the algorithm where the parameters are indexes (I_n) of nested loops or range address (r_m): the bounding box where to generate addresses.

$$AE = f(I_1, I_2, ...I_n, r_1, r_2, ...r_m)$$

In a broad sense, an address generation unit is the unit that uses the Address Equation (AE) to generate an Address Sequence (AS). The resulting Address Sequence contains "what" address to access and "when" to access it. Figure 3.1 shows a general address generation unit.

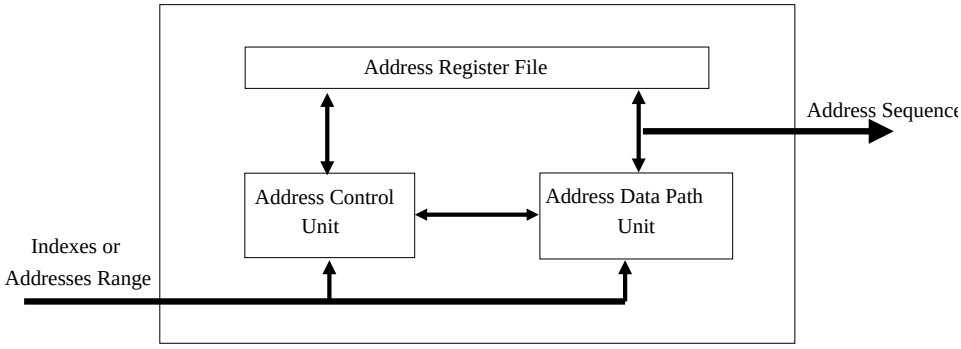


Figure 3.1.: A general Address Generation Unit (AGU)

In a system with several memories several AEs can coexist. Each one of those AEs describes the mapping of their parameters to the respective data location. In this case, one or multiple AGU must provide the address sequences needed.

3.1.1. Types of Address Equations

The AE is a function extracted from the software description of the algorithm. The different AEs can be categorized in terms of their regularity or their flexibility.

Regularity

The regularity of the AE is correlated with the complexity of the index expressions.

Affine AE: An AE is affine when the address equation is a linear expression of the indexes I_n and constants C_n as shown in the following equation:

$$AE = C_0 + C_1.I_1 + C_2.I_2 + ... + C_n.I_n$$

This is the typical case for addresses generated by a number of manifest nested loops. In figure 3.2 we can recognize three AE. For arrays y and w the AEs are

```
for (i=0; i<=N1-N2; i++){
  y[i] = 0.0;
  for (j=0; j<N2; j++){
     y[i] += w[j]*x[i+j];
  }
}
```

Figure 3.2.: Example of affine address equation: a FIR filter Code.

a direct function of the loop indexes i and j respectively and for array x, it is a function of i and j with the coefficients $C_0 = 0$ and $C_1 = C_2 = 1$.

A more complex example is shown in figure 3.3. In this case, the address equation calculates the address indexes for the arrays c, *block* and *res*. Assuming c is a 4 bytes integer, the address equation of c can be written in the following form $AE_c([k][j]) = c[k * 4 + j * 4 * 8]$, and under this form we can clearly see that the AE of c is affine with parameters $C_0 = 0, C_1 = 4$ and $C_2 = 32(8x4)$.

```
for (i=0; i<8; i++)
   for (j=0; j<8; j++){
      tmp 0.0;
      for (k=0; k<8; k++)
         tmp+=c[k][j]*block[8*i+k];
      res[8*i+j]=tmp;
   }
```

Figure 3.3.: Example of affine address equation: piece of code from the MPEG2 decoder kernel

Piece-wise affine AE: An AE is called piece-wise affine AE when parts of the AE can be written as linear expression of the indexes and constants. This is the case of AE in a nested loop with conditional statements based on the iterators (manifest conditions).

In figure 3.4 a piece of code from the MPEG-4 video decoder core is shown. In this case, the conditional expressions limit the possible optimizations in the search space.

Non-linear AE: An AE is called non-linear when there is no linear relation between the AE and the address indexes. This is the most general case of AE. Figure 3.5 shows an example of real code where we can see square expressions ($j*j$ and $i*i$). The non-linearity of those expressions highly constraint the search space for the optimization of address generation and related hardware, but this type of AE does not occur very often in real life application codes.

```
int *DCstore;
DCstore = (int*)malloc(LB*6*15*sizeof(int));
...
initialize (Xtab, Ytab, Ztab, Xpos, Ypos);
loop{
   if(comp==1||comp==3){
      blockA=DCstore[0+15*Xtab[comp]+90*(((mbnum/MB)*MB+(mbnum%MB+Xpos[comp]))%
          LB)];
   }else{
      blockA=mg*8;
   }
   if(comp==3){
      blockB=DCstore[0+15*Ztab[comp]+90*(((mbnum/MB+Ypos[comp])*MB+(mbnum%MB+
          Xpos[comp]))%LB)];
   } else{
      blockB = mg*8;
   }
   if(comp==2||comp==3){
      blockC=DCstore[0+15*Ytab[comp]+90*(((mbnum/MB+Ypos[comp])*MB+mbnum%MB)%LB
          )];
   }else{
      blockC = mg*8;
   }
   mbnum++;
}
```

Figure 3.4.: Example of piece-wise affine AE extracted from the MPEG-4 video decoder core. Conditional expressions limit the search space for address arithmetic optimization

```
for (j=0; j<l_code; j++)
for (i=0; i<l_code; i++){
   if(i<=c1)
      if(i==c1)
         rdm=h2[l_code-1-i];
      else rdm=rr[i*(i-1)/2];
   else rdm=rr[j*(j-1)/2];
   ...mul(rdm, ...);
   if(i<=c2)
      if(i==c2)
         rdm=h2[l_code-1-i];
      else rdm=rr[j*(j-1)/2];
   else rdm=rr[i*(i-1)/2];
   ...mul(rdm, ...);
```

Figure 3.5.: Example of non-linear AE: fragment for the GSM codebook code.

Flexibility

The flexibility of the address equation gives the idea of the range of the flexibility needed by the AGU to create the address sequence.

manifest or predefined with constants: This is a special case that occurs for architectures that target a specific algorithm, or a small number of algorithms. In those cases, the AE is manifest and then address sequences patterns are predefined

before the hardware design and the knowledge of the program can be efficiently exploited by constructing optimized AGUs.

parameterizable or predefined with parameters: This is the case for AS controlled by some input parameters. E.g. this is the case for domain specific designs where the AGU can be parametrized to the application since it has to support a limited number of algorithms.

dynamic address equation: This is the most general case of address sequence where the AE depends on AGU's external events and results (e.g. data-dependent addresses). Here, the AE and thus the AS can be modified during run-time execution. This is the general case for address sequences in processors or DSPs. In the embedded systems domain, based on application scenarios that are most currently occurring, most dynamic address equations can be modeled as manifest or parameterizable address equations [PBV+05, PCC05, GSBC05]. For the rest of the cases, a fully run-time backup scenario without compiler support must be used.

AE design space exploration

Figure 3.6 summarizes section 3.1.1 and shows the various possible combinations of regularity and flexibility of the AE. The complexity of building an efficient AGU increases as we move away from the origin of the graph. For each point in the design space, an optimal AGU can be found to meet the design metrics targeted by the design.

3.1.2. Design Metrics

The quality of an AGU can be evaluated in terms of five different metrics that can be categorized in terms of costs and constraints.

Implementation Costs

Area: the area of a chip determines the cost of a VLSI process. The contributions to the area of an AGU are due to the architecture that implements the arithmetic operations and/or the size of AGU instructions in memory, in case of programmable solutions.

Energy consumption: energy consumption is a metric that depends on several other parameters from physical to system level: technology, area, frequency, supply voltage, bit activations, algorithm, leakage, etc. Energy per task is the most important metric in embedded systems. In this document we will not discuss all the possible parameters, but just the ones accessible from an architectural perspective.

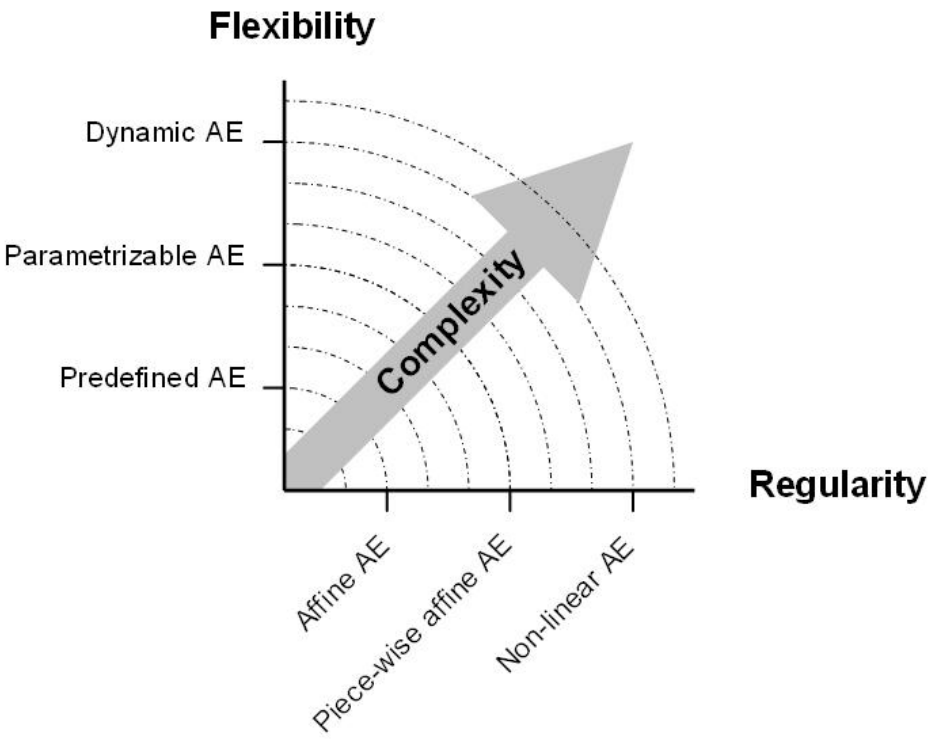


Figure 3.6.: Design Space of AGUs

Historically, the area of a design was the most important cost but nowadays, with the increasing capacity of integration, the area of the AGUs is not crucial and will not be a critical metric in future designs, nevertheless area is a design metric to be taken into account. Currently, energy consumption is the limiting factor that drives design decisions, specifically in the embedded domain due to battery life.

Design Costs

Ease of compilation: programmable solutions have the advantage of software development over the slow hardware development of other solutions. An easy compilation and a retargetable framework is an advantage that can help to boost time-to-market of products or can facilitate upgrades or new versions of the applications.

Constraints

Flexibility: the flexibility of an AGU determines its capacity to deal with different algorithms as seen in section 3.1.1.

Performance: Performance is a real time constraint related to the reaction time (la-

tency) to compute one memory address and the bandwidth or throughput: the number of address equations generated per period. The bandwidth of an AGU depends on the number of available resources. An optimal AGU should provide sufficient address expressions to feed the data path per time unit.

Maximal power/temperature: this constraint is relatively new and of increasing importance with actual and future silicon technologies. The maximal power and temperature that the device can reach is directly connected with the cost of the packaging which has to support such high temperatures.

3.1.3. AGU classification

In the literature we can find different names for Address Generator Units (AGUs), for example, Address Calculation Unit (ACU), Address Arithmetic Unit (AAU), Data Address Generator (DAG), Memory Management Unit (MMU), Direct Memory Access (DMA), etc. In this document we will generally refer to all these address generators as AGUs and in this section we will explain the different possible hardware implementations. We can distinguish two big types of AGU architectures, the ones based on tables and the ones based on a data path.

Table based AGUs

If the AE is short, a Lookup-Table (LUT) can be used to implement this simple map of the AE. This approach needs a controller to complete any deterministic address sequence. In the simplest case, the controller can be a simple increment/decrement counter, but often a Finite State Machine (FSM) is needed. When the AE is more complicated and a LUT solution becomes too big, the logic (that can be custom, programmable or configurable) to complete the AE can be implemented using a Programmable Logic Array (PLA) since those devices have a smaller size compared to the same implementation in a LUT. The two following types of address generator based on table based AGUs can be implemented for custom or flexible (programmable or configurable) logic.

Memory based AGU (mAGU): Short address sequences can be directly mapped in a lookup-table (LUT), this approach is also called "counter/table based AGU" . In the simplest case, the controller of the mAGU can be a simple increment counter, but often more resources are needed.

Incremental AGU (iAGU): If the AE is very regular and can be expanded in sequences of the address values (A_i) then the AE can be implemented by modulo/binary counters with the outputs modified by custom logic that can be implemented in a Programmable Logic Array (PLA) (figure 3.7). This approach has simple

control implemented in the counter and due to the implementation on a PLA the size is smaller compared to the mAGU implemented with a LUT.

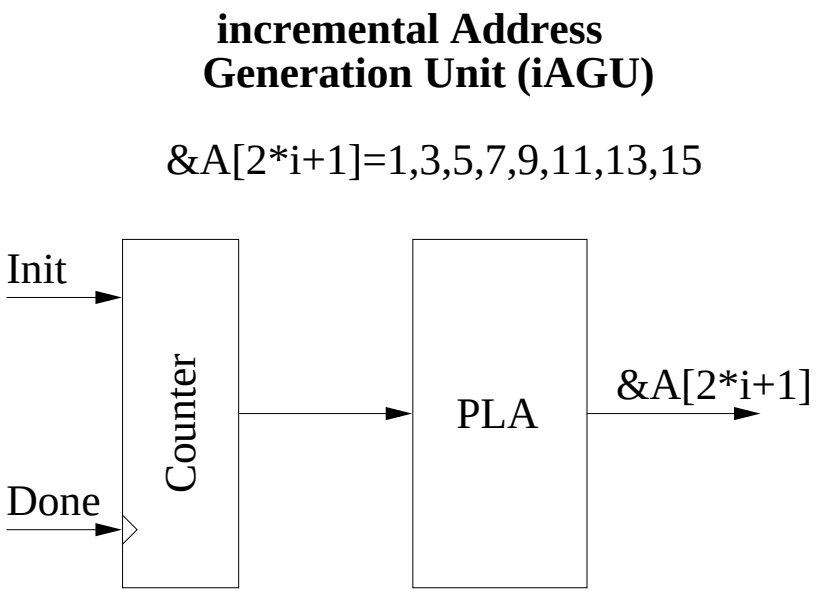


Figure 3.7.: Example of an Incremental AGU

Data-path based AGUs

When the address equation is complex and regular or when it has to support some control or programmability, data-path based AGUs are the most suitable implementation style since they can cover wider address spaces.

Custom AGU (cAGU): In applications where the AE is a direct function (AE affine) of the loop iterators, pointers, flags coming from other units (e.g the data path) or when the AE follows a complex or long sequence, then it can be mapped onto arithmetical operators, and thus, implemented in custom hardware.

This is an optimal architectural style for irregular memory accesses that consume too much mapping logic if implemented in an iAGU/mAGU or for long array-based address sequences. These cases are usually based on a set of nested loops, where the address parameters are combined to give addresses at each iteration of the loops (figure 3.8). A cAGU exploits fast post-modification techniques of address pointer reference to reduce the overhead of indexed array reference generation, but this becomes an expensive alternative when the size and number of memories increases.

Programmable AGU (pAGU): Programmable AGUs are targeted for an optimal calculation of loop array indexes. These AGUs have dedicated registers available

that can be programed. Hence they are the most suitable architectural style for parameterizable or dynamic address equations (3.1.1). Indirect addressing is by far the most used addressing mode in programs running on these systems, since it enables the design of small and faster instructions [AOC02]. Some pAGUs also support loop counters implemented in hardware. The flexibility of pAGUs pays an area overhead depending on the following parameters:

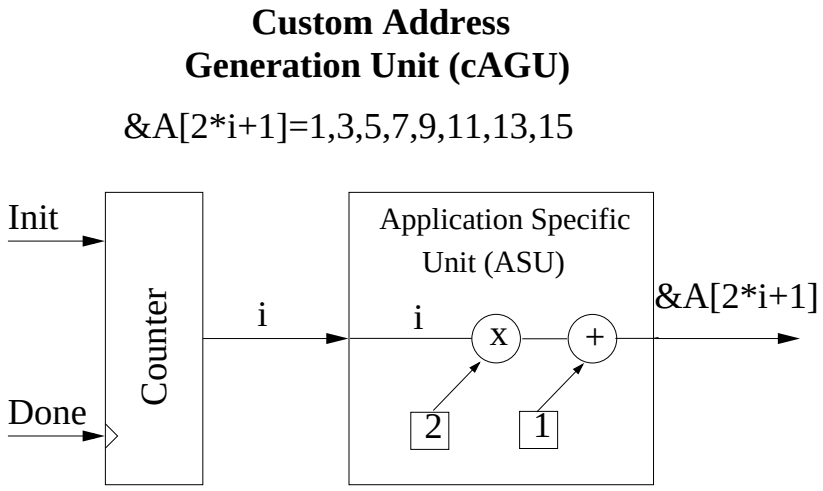


Figure 3.8.: Example of a Custom AGU

- number of memories to address.
- size of the arithmetic blocks (adders, multipliers,etc.).
- address register files.
- program memory and instruction decoders.

As an illustration example of this overhead we point that more than 25% of the area of the Coolflux DSP is used by the AGUs [Phi04].

AGUs on DSP: Address generator units in DSPs differ from that of standard processors: usually DSP AGUs are a specific case of programmable AGUs based on increment/decrement counters, whereby the address for the next memory access is updated during the current memory access (figure 3.9). They normally include register-indirect modes with post-increment for accessing data arrays in memory, and circular ("modulo") addressing capability for managing circular buffers. These addressing modes provide efficient management of data arrays to which repetitive algorithms are applied. Many architectures such as TI TMS320C64X [Inc06] provide indirect addressing modes with auto-increment/decrement arithmetic. These features allow efficient sequential access of memory and increase code density, since they subsume address arithmetic instructions.

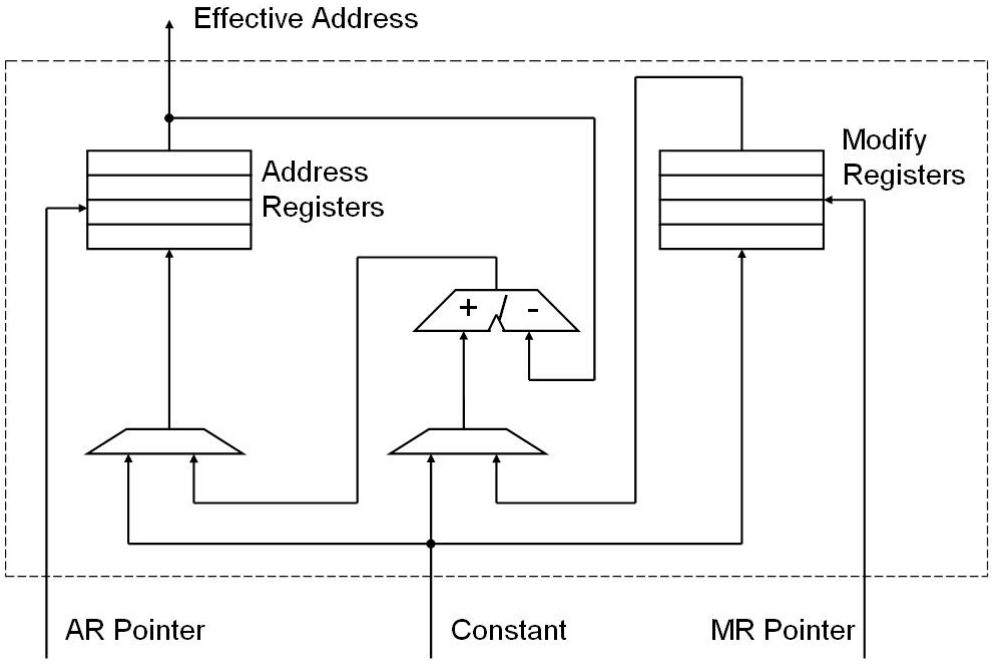


Figure 3.9.: A typical AGU for DSPs

Off-chip memory related address generators

Two other types of "general" address generators (with different flavors) are briefly described in this section for completeness of the topic since they are out of the scope of this thesis. An extensive description of them is given in [HPon].

Direct Memory Access (DMA) Unit:
Direct memory access (DMA) is a feature that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit. It allows devices to transfer data without subjecting the CPU to a heavy overhead: a DMA transfer essentially copies a block of memory from one device to another. While the CPU initiates the transfer, it does not execute/generate the address sequences needed for the transfer. Otherwise, the CPU would have to copy each piece of data from the source to the destination. This is typically slower than copying normal blocks of memory since access to I/O devices over a peripheral bus is generally slower than normal system RAM.

A typical usage of DMA is copying a block of memory from system RAM to (or from) a buffer on the device. Such an operation does not stall the processor, which as a result can be scheduled to perform other tasks in parallel. DMA transfers are essential to high

performance embedded systems.

Nowadays, DMA functionality is restricted to off-chip transfers, but a DMA-like functionality is also becoming very important for on-chip scratchpad usage, and in some research papers they are indeed heavily exploited already [DBD+06]

Memory Management Unit:
Memory Management Unit is a term usually used to designate a hardware device or circuit that supports virtual memory and paging by translating virtual addresses into physical addresses. Among the functions of such devices are the translation of virtual addresses to physical addresses (i.e., virtual memory management), memory protection, cache control, bus arbitration, and, in simpler computer architectures (especially 8-bit systems), bank switching. Usually MMUs are in charge of the SDRAM to on-chip data communication. In this work we do not consider MMU since this paper focuses on on-chip data addressing.

3.2. Optimizations on Address Generators

Efficient memory accesses are crucial for any architecture which deals with real-time-data-dominated algorithms. Support for multiple address equations often leads to the mapping of each address equation onto different dedicated or programmable hardware, resulting in large area and power overhead and/or timing problems. Hence, the address equation should be optimized first to exploit area savings possible by sharing hardware (while meeting timing constraints imposed), optimizing power or improving performance. Some optimizations at different levels can considerably improve address generation. In this section, we will review the state-of-the-art optimizations for address generation considering that other general optimizations, as for example data transfer and storage exploration optimizations [Cat02], have already been applied.

3.2.1. Architecture and micro-architecture optimizations

AGUs can boost the transfer of data to the calculation units since they compute in parallel the address of the next memory accesses. To provide enough data bandwidth, usually several programmable AGUs run in parallel. If the number of AGUs increases, the program area grows, and then ROMs and instruction decoders and the area overhead introduced can become a dominant factor. Indirectly, this also leads to energy overhead and usually real trade-offs are present. Many of the state of the art optimizations aim at reducing the hardware overhead.

Address optimizations

Until the beginning of the 90's, little research was done on address generation optimizations and first optimizations were oriented to reduce area on table-based AGUs. With

the possibility to integrate SRAM memories on chip, area savings became less important and energy became the main problem. Even if area is not anymore a limiting factor some of the techniques can be reused for energy reduction.

Table-based AGUs (3.1.3) implement manifest address sequences and although they cannot deal with dynamic applications, the knowledge of the specific address equation enables hardware design time optimizations. In [VBR+93, GDF89] the authors assumed that every AE is mapped to a separate address unit. Miranda et al. in [MCM94] presented an area optimization technique for application specific address generation units for large memories in real time signal processing systems. Their techniques rely on a system level exploration of the trade-offs involved on algorithmic specifications: when the signal processing application is being defined, savings are exploited by sharing index expressions among several individual AEs.

Miranda et al. in [MCM94, MKCdM97, MCJM98, MCJdM96] present the ADress OPTimization (ADOPT) environment. The framework gives a formalized methodology and an automated technique to support address arithmetic optimizations in flow-graph expressions for distributed memory architectures. The ADOPT environment targets architecture and system level optimizations at the same time and the methodology relies on two stages: one independent of the target architectural style and a second one stage specific to the selected AGU. ADOPT targets two different architectural styles for the generation of the AE: incremental AGUs and customized ACU (3.1.3 and 3.1.3).

In [ST98] Schmit et al. present optimization techniques for the address generation of memories containing multiple arrays. The techniques are based on the rearrangement of bits instead of the traditional technique of addition of a base address and a variable to calculate the addresses of the arrays elements.

The trade offs between area and performance for known address sequences have been addressed in [HCC02]. Their work studies the impact on area and performance of memory access related circuitry in eliminating row and column address decoders from the memory and incorporating the necessary hardware decoding in the address generation circuit. They show how circuit delay can be nearly halved at the expense of increased area.

BSG: Bit Sequence Generators

Predefined AEs can also be implemented using bit level sequence generators. In the most straightforward case, this will lead to the direct mapping of each bit level AE onto a dedicated hardware block. This strategy could lead to a large area overhead and bit level optimizations are mainly oriented to the reuse of hardware between bit level address generators. The work of Grant [GD91] and Lippens [LMdWV91] shows, for very regular AE, optimizations at bit level for counter/table based architectures. In [GML94] the same authors present the address generation techniques of the ZIPPO toolbox, which relies on hardware multiplexing optimizing word level and bit level address among several AEs. The authors show that address generation optimizations can result in area savings between 10%-60%.

Loop Accelerators: LA

Data-dominated applications use a large amount of memory and typically those applications consist of deeply nested for-loops where the main algorithm is usually located in the innermost loop. Complex media algorithms for e.g., usually have a large number of two dimensional array and vector accesses. Mathew et al. in [MD04] expose in their work how Amdahl's law argues in favor of accelerating those parts of the algorithm, and present a low power loop accelerator based on clustered distributed address generation. The loop accelerator is responsible for the calculation of addresses once the program enters the part of the program with time consuming 2D loops and needs a high operand flow in the functional units. In the rest of the program, the addresses are calculated in the generic address generation unit.

3.2.2. Compiler optimizations

Any processor targeting the embedded domain has strong restrictions on area, energy and performance. Exploiting design time optimizations alleviates run time issues with various benefits. First of all, it reduces chip area, and thus energy, compared to the run-time hardware support that is needed by superscalar processors. Another benefit is that complexity is usually easier and faster to deal with in a software design than in a hardware design. Hence, the chip may be cheaper, quicker to design and easier to debug. Software also benefits from possible upgrades while improvements on superscalar dispatch hardware requires to change the processor. For all these reasons, optimizing compilers has become an essential component of embedded and high-performance computer systems and extensive bibliography is available [Muc97, KA02, ALSU06]

Address calculation optimizations

Programmable AGUs (section 3.1.3) benefit from flexibility but have an important area penalty due to the arithmetic units, registers and counters needed to give enough versatility and must mainly rely on software and compiler optimizations.

Traditional compiler approaches aim at address pre-calculation and post-calculation [ASU86]. Address pre-calculation consists of an addition of a base address with a variable. In this method, the address must be calculated before the reference. For innermost loop kernels this approach presents a important penalty on the performance of the algorithm. Address post-calculation reduces an array reference to an address pointer reference and incrementing/decrementing the resulting address for the next reference of the array. This method allows the calculation of the next address in parallel with a main data operation. First optimizations on address post-calculation where studied by Liem et al. in [LPJ96, LPJ97]. In their work, the authors show code transformations that produce code with optimized index array references for fast post-modification combining address variables with the same addresses offset among loop iterations. These optimizations improve the efficiency of programmable AGUs.

For data dependent address generation (3.1.1) using a programmable AGU, address

optimizations can be achieved via data ordering and address register allocation. Data ordering determines the order of data stored in the memory. Address register allocation assigns an address register to each data access for address generation. The goal of these optimization is to maximize the usage of auto-increment/decrement and hence reduce the number of address loading instructions. Many research papers and industrial compiler groups work in auto-increment/decrement optimizations but they are mainly oriented towards timing optimization, not power. In [CL98] Cheng and Lyn present an address optimization technique for loop execution for DSPs with auto-increment/decrement architecture. They propose a new graph model that takes care of constraints on memory allocation and data ordering. In [Leu00b] other optimization techniques targeting DSPs are deeply explained.

In [RKHK02] the authors present arithmetic and address computation optimization for a set of typical very regular kernels that appear frequently on several kinds of applications. Their optimization resides in the observation that, for those kernels, the elements accessed are usually stored close to one other in memory. Their work studies scalar conversion and common sub-expression optimization between successive iterations of loop bodies. Actual compilers do not realize this kind of analysis and for successive iterations the value of common sub-expressions is computed at each iteration.

Register file optimizations

The efficient usage of registers in programmable AGUs (3.1.3), focusing on AGUs for DSPs, has been studied by several authors. Leupers et at, in [LM96], show algorithm optimizations by computing appropriate memory layouts for program variables. For that, they proposed an improved cost function for edge selection on the address graph. This work has been extended by Sudarsanam in [SLD97] who presented a methodology to study the effect on code size and performance associated with the number of available address registers and the range of the auto-increment for simple offset assignment problems but for increment/decrement ranges superior to one. Offset optimization techniques rely on constructing a layout of local variables in memory, such that the addresses of variables can be accessed using auto-increment operations on address registers. Wess, et al. in [Wes99], present a template address generation unit for DSPs and apply neighborhood search techniques with simulated annealing to overcome the local minimal in the search process, to the address assignment problem. That technique optimizes the address layout to minimize the offsets of consecutive memory accesses.

Later on, Leupers in [LD98] presents a genetic algorithm that optimizes arbitrary register file sizes and auto-increment ranges and introduces the allocation of modify registers into offset assignment. For fixed architectures, with a fixed set of registers, Basu et al., in [BLM98], present an heuristic for the optimization of the number of instructions needed for the calculation of array addressess in the program loop.

More recently, in [AOC02] the authors proposed the usage of auto-increment addressing modes to reduce the problem of allocation of arrays in the register file. It extends previous work in the area by merging life ranges of address registers in a pairwise way beyond basic blocks. In this work also, no power optimization is targeted.

3.2.3. Source code transformations

Address optimizations

The problem of address generation is related to indexed signals and the main difficulty is to find the most efficient address computation. Address optimization can reduce area, but also access time and the number of calculations needed to obtain the next memory address.

The ADOPT framework introduced previously, targets incremental AGU (section 3.1.3) and custom AGUs (section 3.1.3) but the techniques can be reused on programmable AGU. For those architectures, it is possible to perform several optimizations at a behavioral level of the code (at a high level language). All those transformations are performed at source level, meaning that the result code is also C. The generated code is near optimal in terms of minimal code execution and has reduced arithmetic expressions which minimizes the area overhead, introduced by the usage of a large amount of address generation units, and the time required to generate memory addresses which is a main issue for data-intensive applications.

System level optimizations can be found in different works: [MCJM98] presents address equation/cluster splitting, merging, sharing and induction variable analysis. In [GMCG00] the authors explain loop-invariant code motion, global algebraic transformations, Common Sub-expression Elimination (CSE) and loop invariant code hoisting. Pointer substitution, advanced code hoisting, algebraic cost minimization and non linear operator strength reduction are introduced in [GMV+00]. All these techniques aim at code transformations that reduce or simplify the address calculations in the address path.

Control flow optimizations

Data Transfer and Storage Explorations (DTSE) transformations are crucial to efficiently map data-intensive applications onto programmable platforms [Cat02, CDKO01, GH96]. DTSE transformations modify the initial code to minimize the load of shared memory buses which is the main source of power consumption [WCNM96]. This transformations are usually achieved at a high expense on addressing and local control. After removing DTSE and address related bottlenecks, the control flow issues represent an important overhead on code execution. Control flow transformations are suitable for any implementation of AGUs and focus on two issues: improving the degree of mutual exclusiveness in nested condition trees, and in optimizing the decoding of the condition testing in deeply nested conditional constructs.

DTSE transformations add costly integer modulo operations and divisions to the initial addressing code. A pointer substitution technique [GMV+00] for piece-wise linear addressing solves this problem. This optimization is platform independent and transforms modulo operations onto conditional code and linear induction variables. The control overhead introduced by DTSE or by this pointer substitution technique has been addressed in [PMCV01] . The same author in [PMD+02] and [PMC02] addresses control flow optimizations for performance on a MPEG-4 video decoder algorithm and

the trade-offs on power and performance on address generation by selective function inlining.

Other control flow optimizations, fully complementary with the previous ones, have been studied by Falk in [FM03, FV04, Fal05, FM04]. The optimizations presented are based on mathematical models combined with genetic algorithms: loop nest splitting minimizes the number of executed *if-statements* in *loop-nests* of embedded multimedia applications; advanced code hoisting moves portions of the inner loops to outer ones and ring buffer replacement eliminates small arrays serving as buffers for temporary data. The three techniques reduce control flow, arithmetic calculations and execution time and hence minimize energy consumption.

3.3. Conclusions

In this chapter we analyzed the different types of address generators and we proposed a classification. According to this classification we then summarized the state of the art of the optimization on the address generation process at different levels: source code, compiler and architecture and micro-architecture.

In the embedded domain, devices have to deal with complex applications. So, evaluating what architecture is most suited is not an easy task and tool support is needed. In chapter 5 we will introduce a framework to analyze and create an optimized AGU-template targeting the embedded domain. In chapter 6 we will plug this template into the compiler and architecture framework described in chapter 4.

CHAPTER 4

High Level Architecture and Compiler Requirements: COFFEE framework

"Imagination was given to man to compensate him for what he is not;

and sense of humor to console him for what he is."

Francis Bacon

While this thesis focuses on some parts of the processor, it impacts all parts of the platform. To ensure that this work is consistent with optimizations in other parts, it is necessary to look at all parts of the platform. Therefore this chapter bundles the boundary conditions and conclusions of a team of PhD students working on the exploration and optimization of different parts of the embedded platform. A common analysis and trade-off discussion of proposed architecture extensions ensures the consistency between the different parts. The different PhD students focus each on different parts, but take the effect of local modifications on the rest of the platform into account.

The result of this common analysis is presented in this chapter together with all proposed modifications form the FEENECS (Flexible Extremely ENergy Efficient Configurable System) architecture template (Section 4.3). By explicitly looking at the requirements and restrictions of current compilers, and their link with the architecture, high-level relations have been derived in order to improve the compilability. Based on these relations and on the specific features of the FEENECS architecture template, a matching methodology proposal is drafted and detailed in chapter 6 .

As the proposed architecture modifications and compiler steps are the work of a team of PhD students that are based on observed trends in technology and state of the art, they are *not fully supported* by experimental data, especially for parts that are not the

focus of this thesis. However, they are motivated using high level relations and, when possible, an example or reference is used to give more detail to the discussed concepts.

The rest of this chapter is organized as follows: Section 4.1 presents the context of this work and the current trends in processor architecture space. Section 4.2 presents various architectural proposals for different processor components to reach the same energy efficiency as that of an ASIC. Section 4.3 puts these different architectural parts to present the FEENECS architecture template and section 4.4 briefly describes the energy estimation model used. Finally, section 4.5 exposes the conclusions of this chapter.

4.1. Architecture Exploration and Trends

Architectures form the bridge between the application and the technology. Therefore, in order to optimize an ASIP processor architecture, the designer must take into account the application requirements. An effective ASIP architecture exploration has to cover a wide range of architectures to find the one which is Pareto optimal for the application and system cost trade-offs (e.g. reduce the energy consumption while providing the required real-time constraints and quality). From the implementation side, it is important to take the physical design method (e.g. custom design vs. standard cell design) and high level technology inputs (e.g. poor interconnect scaling in Deep SubMicron (DSM) technologies, leakage) into account early in the design flow to ensure a more optimal outcome. If the implementation allows the designer to give guidelines on the floorplan, it is important to take this into account. Architecture exploration therefore forms the corner stone of any processor design.

Note that variability and reliability are also crucial issues in DSM technologies, but the mitigation of these effects can be handled in a complementary way [HMFW07, HMWF09] that is compatible with our approach. Therefore this part will not be tackled in this thesis.

4.1.1. Interconnect scaling in future technologies

A number of papers have appeared that compare the scaling of interconnect to the scaling of logic (transistors) for scaled technology. According to many papers [DeM05, JNH06, SK99] and the latest ITRS report [ITR07], a clear difference exists between logic and interconnect scaling and interconnect scales much worse. This leads to potentially reduced gains when scaling to deep submicron technologies ($65, 45, 32, 22\ nm$). The poor scaling of interconnect and vias is due to varying physics limiting factors, ranging from the k-value between the wires to DSM issues like surface scattering and grain boundary scattering. This affects both local and global wiring and therefore, this trend needs to be taken into account for future architectures.

In this chapter, it is assumed that interconnect does indeed scale worse than logic and that the impact of this interconnect scaling in terms of performance and energy cost is increasing. Therefore, some rather disruptive modifications to the architecture are proposed (e.g. replacement of traditional register files with new foreground mem-

ory structure, as discussed in Section 4.2.3). The architecture modifications that are proposed in this chapter are propagating the higher cost of interconnect through to various levels of the design (from heavily communicating components, based on application knowledge, down to layout).

However, in case the cost of interconnect (especially the local interconnect) does not increase with respect to logic, the gains of the proposed solutions will still exist with respect to traditional designs, but may relatively be reduced. Therefore some of the traditional state-of-the-art solutions will still be part of the valid trade-off solution to choose from. It may also be the case that because a disruptive change often takes more effort, the current state-of-the-art architecture may still be taken.

4.1.2. Representative architecture exploration examples: What are the bottlenecks?

During and after architecture exploration, the designer can obtain an energy breakdown of the different components of the processor architecture. Figure 4.1 shows the energy breakdown for a high-performance Coarse Grain Reconfigurable Architecture (CGRA) processor implemented in 130nm running a MIMO application. Figure 4.2 shows the energy breakdown for an embedded VLIW processor running the MPEG2 decoder.

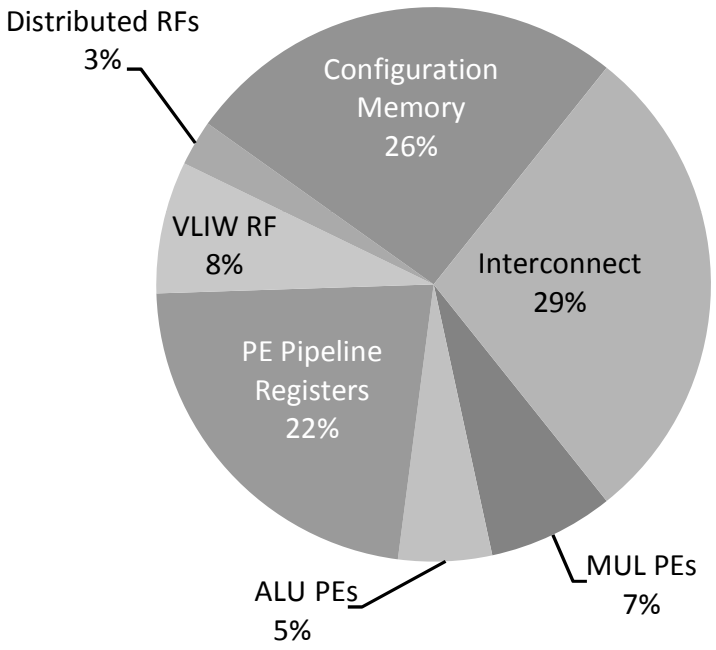


Figure 4.1.: Energy Breakdown for a high performance CGRA (8x8 PEs) running a MIMO benchmark, with a clock of 200MHz

While both breakdowns are from different application and processors, similar conclusions can be drawn. From both figures, it can be seen that the energy consumption is

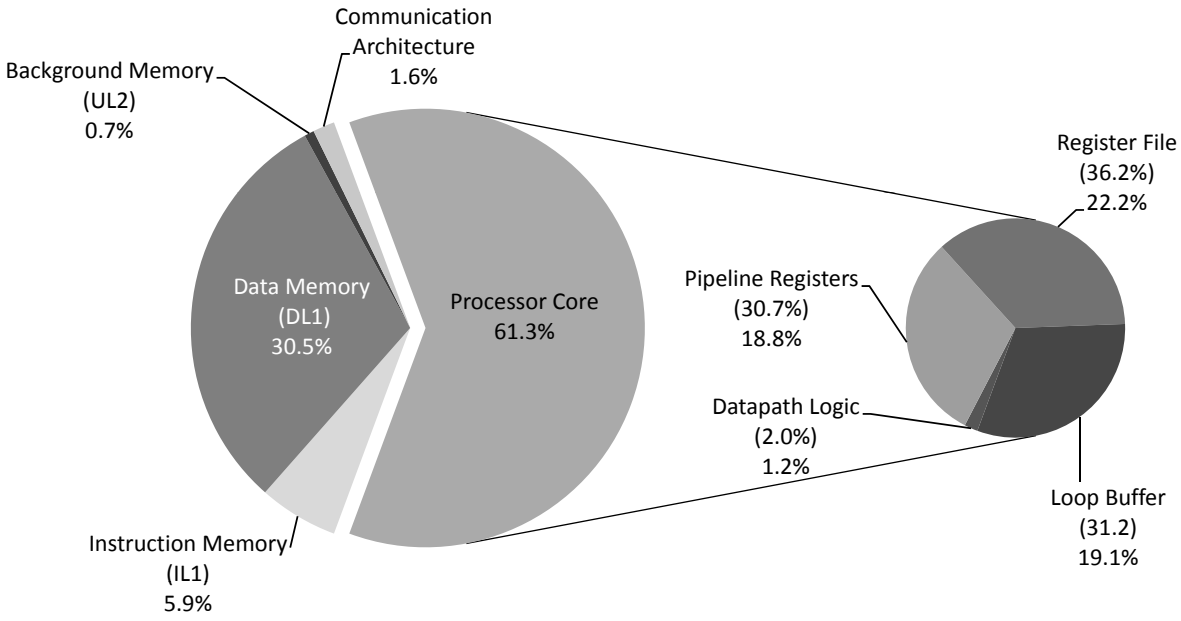


Figure 4.2.: Energy Breakdown for heterogeneous VLIW processor with 8 slots running an MPEG2 decoder, with a clock of 600 MHz. The numbers between brackets indicate the percentage for the processor core only.

not really dominated by a single architectural component.

In the CGRA, a large part of the energy consumption is spent on the interconnect (the architecture instance shown is a high-performance instance with a rich interconnect topology). This is because a significant part of the communication between different slots in the VLIW has been pushed from the VLIW register file to the interconnect and the PE Pipeline Registers. This leads to a relatively smaller part of the CGRA energy that is directly consumed by shared register files. In fact, in a CGRA the foreground memory organization consists of the register files, pipeline registers and the interconnect and therefore still remains a critical issue.

It is assumed here that in order to reach the required high performance, the 8 slot VLIW has to be clocked at a significantly larger clock speed (here 600 MHz compared to 200 MHz for the CGRA).

The CGRA is more parallel than conventional VLIWs and a larger part of the energy consumption is directly spent on the data-path logic (in this case 12% for both ALU and MUL PEs, compared to only 2% for the VLIW), which indicates a more efficient energy usage. Note none of the pie-charts gives direct information about the difference in absolute energy consumption between the VLIW and the CGRA.

In conclusion, both pie-charts show that the configuration memory or loop buffer, register files, data-path logic, data-path pipeline registers and interconnect consume almost equally important parts of the global energy. Therefore, all parts need to be

considered together while optimizing.

4.2. Architecture optimization using cross-abstraction and cross-component relations

A clear need exists to think globally and across different abstraction layers while designing an architecture. The properties required at each of the architecture components and the reasons for these properties can be motivated across the different abstractions layers. This cross-abstraction knowledge can lead to different optimizations than would be selected if every part of the system were optimized in isolation, which we call *entanglement*. The following subsections present a possible solution for individual architectural components and its reasoning across different abstraction layers.

4.2.1. Algorithm design

Even though this section targets architecture optimizations, the cross-abstraction information can be propagated down from the algorithm design. Following the increase in processor parallelism and the inability to exploit irregularity in algorithms (with many conditions), algorithmic designers have been pushed to design more regular algorithms. In case of power-efficient ASIC design, once again designers are forced to use regular algorithms, as adding flexibility is expensive in ASIC design. However, when these trends leads to a drastic increase in the number of operations that need to be executed, the energy penalty is significant (e.g. compute motion estimation in MPEG as a full search or hierarchical search). Even though the parallel architecture can be better filled and performance can improve, the resulting efficiency in terms of energy per task is still (significantly) lower. Therefore, some of the modifications that are proposed in this section explicitly address the need to better support irregularity in an efficient way, directly on the parallel architecture. These modifications include a split in address and data computations onto separate slots (Section 4.2.6), distributed control loop buffers (Section 4.2.4), and efficient parallelization of irregular data [Rag09]. When the architecture supports more irregularity, different versions of an algorithm can be designed, taking into account the context. One promising example with large gains (up to a factor 10 in energy efficiency and performance) can be found in [MDB+09], where the conditions of the wireless channel are taken into account in order to select different implementations of an FFT algorithm to match the requirements instead of executing a full FFT in all cases.

The rest of this section will focus on the different architecture components and discuss the propagation of constraints from the application down to the layout.

4.2.2. Data Memory Hierarchy

Application: As the amount of data required increases substantially from one generation of the application to the next, e.g. a higher data rate in wireless applications

or a higher resolution in image/video applications, it is important to efficiently handle the transfers of this large amount of data. Typical embedded applications exhibit both spatial and temporal data locality, which can be exploited (using source code transformations as discussed in [ECWF00]) to reduce the cost of the data memory hierarchy by optimizing the reuse.

Architecture: When the application can be analyzed at compile time, the data transfers can be managed by the programmer or compiler and a scratchpad can be used instead of a cache [SWLM02, BSL+02b, PND98b, Mar03]. In the case of a scratchpad, the data transfers from the higher level memories to the scratchpad memory are programmed explicitly and handled by a DMA engine. This programming overhead is acceptable as it can be done at design-time. In rare cases where embedded applications exhibit truly random accesses and their access pattern can not be analyzed at compile time, they can still use (hardware controlled) caches [Abs07].

Implementation: From the perspective of circuit (and layout) design it is not always possible to design one large memory. Because of the increasing cost of long interconnect [JNH06, SK99, ITR07], very large monolithic memories are increasingly difficult be to built in scaled technologies. Therefore the memory can be partitioned into banks and internally into sub-banks. Furthermore the use of multiple levels of hierarchy and the DMAs should also be taken into account while floor planning.

4.2.3. Foreground Memory Organization

In case of VLIWs, register files form the core part of the data communication across the different slots and clusters as a storage element. Alternatively, as in CGRAs, part of that communication can be pushed to the interconnect between different PEs. Therefore, in both, centralized and distributed register files, the PE pipeline registers and the inter-slot or inter-PE interconnect are all considered to be part of one architectural component and they need to be optimized together. From here on register files and the connections between the slots are together called the *foreground memory organization* in the rest of the thesis.

As *foreground memory* forms one of the core parts of the processor architecture and given that it is one of the biggest bottleneck, new ways to optimize it need to be considered.

Application: *Different types of data:* Current embedded applications contain various types of data, from array data that have high spatial locality to scalar data that store e.g. a single coefficient or a temporary value. Unlike the data layout of the data memory hierarchy, that explicitly allocates (parts of) arrays to scratchpads and optimizes the transfers with local copies, at the register file all data is treated the same. Array data is stored together with temporary variables, without any concept of data layout. By introducing the data layout concept at the level of the traditional register file, the specific

properties of different data elements in terms of spatial locality, temporal locality, size, life time etc. can be exploited. In conventional architectures and compilers this is not yet done.

Architecture:

- *Heterogeneous register file:* To be able to split the different types of data and treat them accordingly, a heterogeneous foreground memory architecture is needed. A separate register file for scalar variables will enable to perform a more optimized data layout for the streaming data (e.g. a wider register file for SIMD data and a narrower scalar register file). Splitting off the address computations into separate slots (see below) enables the use of an optimized register file for the address path.
- *Energy cost per access:* From the architectural perspective, about 50% of the power consumption of a typical L1 data memory (around 64K) is spent in the decoding logic. The other 50% is spent in the actual data storage, in the memory cells [AH00, Eva95]. The spatial locality that is available in array data of the streaming type can be exploited by loading them together into the register file by a wide load/store from the memory. The overhead of the address decoding in the memory will thereby be distributed over a number of words.

Additionally, the energy cost per access scales with the number of ports [RDK+00], as an increase in the number of ports leads to an increased load for every register file cell. Therefore, multi-ported register files can be replaced by a clustered register file architecture, with multiple register files that have less ports each. From the cost per access point of view, a form using only single-ported register file cells is therefore advisable. In today's architectures this is not achievable due to restrictions in both architecture and compiler, that would lead to poor performance and utilization. Therefore a more disruptive foreground memory change is needed to reach this goal, where architecture and compiler modifications are coupled/entangled.

Implementation: Finally, from the layout perspective, it is important that the wire lengths of the interconnect inside the register file and of the complete foreground memory organization (including the connections to the other slots) should be optimized. Accesses to the foreground memory are very frequent and heavily communicating components are close together. This activity-aware floor planning is compatible with the approach presented by [J.G08], but here the concept is applied at a finer granularity.

Proposed solution: Some intermediate values in parallel architectures often have a single cycle life-time, so, it does not make sense to store the variable into the register file organization. In this case it is more efficient to use the forwarding network that writes the result of one Functional Unit after the execute pipeline stage directly into the pipeline register before the execute stage of the next FU. This type of forwarding is commonly found in DSPs and forms an integral part of the proposed foreground memory organization.

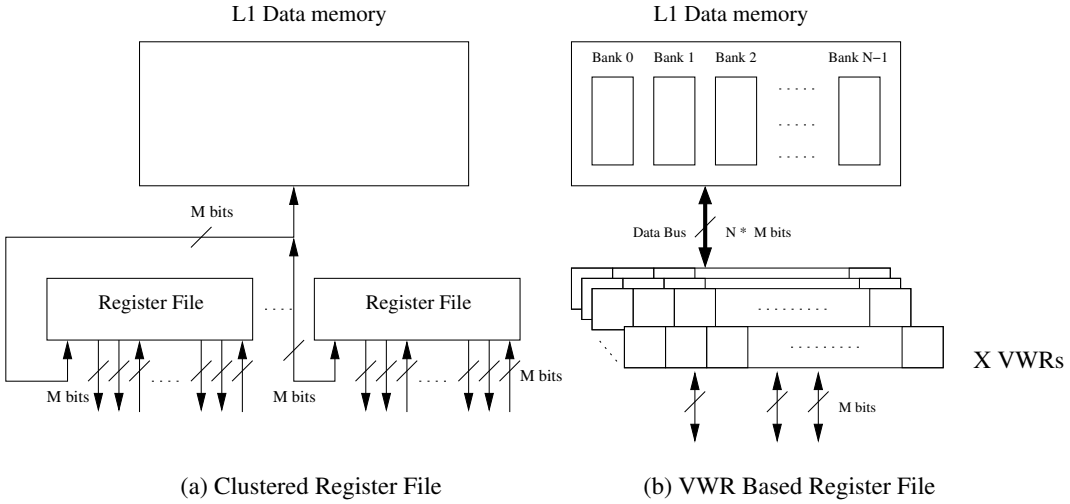


Figure 4.3.: Very Wide Register: A register file solution for streaming data with spatial locality

For variables with a longer lifetime, a novel architecture solution is proposed. Figure 4.3(a) shows a typical clustered register file where the interfaces between the memory and the register file on one hand and between the data path and the register file on the other hand are of equal width. Data can be copied word by word into the register file, with no restrictions on the data layout in the L1 data memory. Figure 4.3(b) shows the proposal of a Very Wide Register (VWR), a foreground memory organization optimized for streaming applications that exhibit a large amount of spatial locality. The VWR has a wide interface (width of a complete SPM row) towards the memory and a narrow interface (width of a data-path word) towards the data path. From the data-path side, through an interconnection network, SIMD FUs can read out data words that internally contain many sub-words. The wide interface towards the memory allows a large amount of data to be transferred to/from the VWR with a single memory decode. Wiring capabilities are not a limiting factor due to the high number of metal layers available in current DSM technologies. This implies a reduced decode overhead for the L1 memory, but requires a careful data layout and increased compiler complexity to maximize the number of useful data words to be present in this transfer. One multi-ported register file can be replaced by a set of VWRs. While there are effectively two ports, one towards the memory and one towards the data path, the VWR storage cells remain single-ported as both the ports are not simultaneously accessed. This single-ported cell nature of the VWR internally reduces the energy cost per access compared to a multi-ported register file, at the cost of an increased complexity for the compiler. The physical layout of the VWR and SPM can be matched (pitch alignment) to reduce the interconnect length between both. Therefore the layout of the physical implementation should be such that the VWR and SPM are placed next to each other.

The gains that can be expected when moving from a traditional register file architecture to a VWR-based foreground memory organization depend on the assumption that

the cost of interconnect increases when scaling to DSM technologies and the improvements will be less if the cost of interconnect can be reduced by technology modifications.

4.2.4. Instruction/Configuration Memory Organization (ICMO)

A traditional instruction memory organization consists of an L1 instruction memory which is controlled by a program counter (PC). As increasingly parallel architectures require more instructions to be fetched every cycle, the instruction memory needs to be wider and therefore consumes a lot of energy.

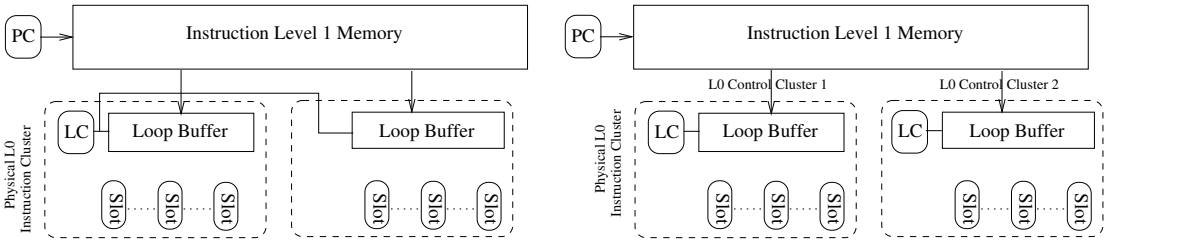
Application: Applications typically consist of different control flows merged into a single combined control flow, e.g. the address-generation part of the program is mixed with different data producer-consumer chains into one sequence of operations. Mixing the different flows seemingly simplifies the programming, but the efficiency of the instruction memory can be improved if the different flows can be handled separately. To enable this in current architectures is however not possible.

Additionally, most applications contain both control intensive parts and regular kernels that perform the most computationally intensive parts. The kernels are structured as nested loops and form the core of most embedded applications. By matching the instruction memory organization to the kernel structure, the overall efficiency can be improved.

Architecture: Traditional instruction memory organizations consists of a monolithic L1 instruction memory, controlled by a single program counter (PC). Recently, academic and some industrial architectures have introduced a small instruction memory closer to the processor, called *loop buffer* or *L0 memory* [SHmWH01, IBM05, UWW+99]. This small L0 memory contains the instructions for a kernel, together with a small zero-overhead loop (ZOL) control or a loop controller (LC). During loop execution, instructions are fetched from the loop buffer and the L1 instruction memory can be put to sleep (by either Vdd throttling or clock gating). Due to the extra copy that is needed to move the kernel instructions from the L1 to the loop buffers, there is a trade-off involved. For some instructions in control-intensive parts of the code it will be difficult to gain back the cost of this extra copy. Therefore, these control-intensive parts are still directly fetched from the L1. This approach is compatible with the use of non-volatile memories for the L1 instruction memory in order to reduce leakage energy consumption in the L1 memory.

The concept of a loop buffer can not only be used for different slots of the data path, but is also useful for other platform components that need to be programmed, e.g. the DMA, interconnect networks, etc. However, the activity of e.g. the DMA and the data path can be very different. Therefore sharing a single loop buffer and keeping the control centralized will normally lead to inefficiency.

To obtain a more optimal solution, the control of loop buffers for different parts of the architecture can be split in order to match the application constraints, resulting in



(a) VLIW processor physically distributed loop buffers with a centralized single loop controller

(b) VLIW processor withphysically distributed loop buffers and distributed loop controllers

Figure 4.4.: Distributed Loop Buffer: An instruction memory solution for optimal instruction issuing

distributed control loop buffers. By splitting the control, components that are not active for a certain number of cycles can be kept under low-leakage/sleep mode.

Implementation: *Physically distributed loop buffers* are placed (during layout) close to the parts of the processor they control. This can be achieved using activity aware floor-planning techniques like [J.G08] and distributing the loop buffers as explained in [Jay05a].

Figure 4.4(a) shows a state-of-the-art physically distributed loop buffer as explained in [Jay05a]. Figure 4.4(b) shows the proposed physically distributed loop buffer with distributed control which would enable different parts of the program to be controlled in an efficient way, based on its corresponding activity.

The Very Wide Register and the distributed loop buffer are the main contribution of Raghavan PhD Thesis [Rag09].

4.2.5. Data-Path Parallelism

Although the data path operations are not the most important contribution to the energy consumption in Figures 4.1 and 4.2, the data-path style has a large effect on how the register files and instruction/configuration memory organization will be used. The organization of the data path determines the complexity and, consequently, the cost of the interconnect. Current embedded processors exploit parallelism in order to provide a sufficient amount of performance, while still keeping the energy consumption under control. They can however differ in the way they do this. As not all approaches are as energy efficient, this section discusses the trade-offs involved.

Application: The computationally intensive kernels of most embedded applications contain parallelism at different levels: e.g. across different pixels, blocks of pixels or frames of a video sequence. Different types of parallelism follow from the way it is extracted from the application. When different iterations of a loop are being executed in parallel, this is called Loop Level Parallelism (LLP). The parallel execution of different

instructions, either from outside or inside a loop, is called Instruction Level Parallelism (ILP). Finally, the execution of multiple instructions of the same type on different data is called SIMD or Data Level Parallelism (DLP). The amount of parallelism of different types that can be extracted depends on the application dependencies. Ideally, processors should contain a mixture of slots of different widths: very wide SIMD units for regular kernels that contain a large amount of DLP, together with medium wide units for kernels with more control and scalar units for non-DLP code.

Architecture: Embedded processors can be designed to exploit one or a combination of these types of parallelism. Both embedded VLIW processors and CGRAs provide parallel slots or PEs and use software pipelining to convert the LLP into ILP (See Figure 4.5). *Separate instructions* (although combined into a single very large instruction in VLIW terminology) control the parallel slots or PEs, which gives a lot of freedom to the compiler with respect to how to overlap the different iterations (still respecting the dependencies) and how to place the different instructions on the slots. As a downside, exploiting ILP requires managing a large number of separate units and the communication between them for every instruction of the overlapped loops, which leads to a significant amount of energy that is spent in the ICMO (Instruction/Configuration Memory Organization).

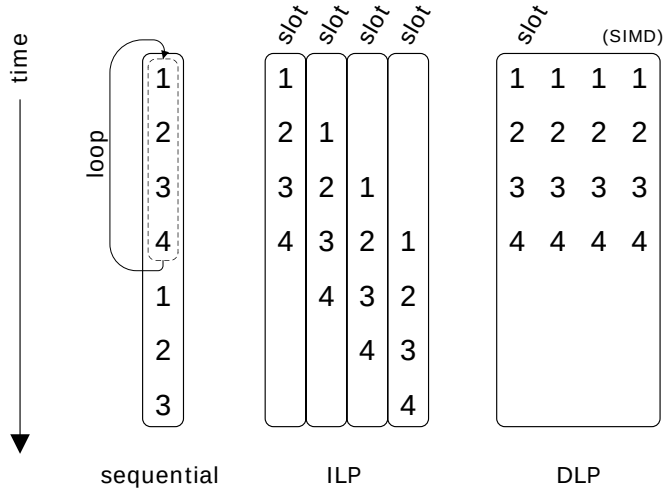


Figure 4.5.: Converting Loop Level Parallelism (LLP) into Instruction Level Parallelism (ILP) or Data Level Parallelism (DLP)

When subsequent iterations can be overlapped completely, the parallelized operations are of the *same type* and subword parallelism or SIMD can be used. This means that they can be executed on the same FU (wider and slightly modified to separate the data). The advantage of exploiting DLP is that the resulting SIMD instruction executes multiple (e.g. 4 as shown in Figures 4.5) operations in parallel, but is still steered by a single instruction, hence Single Instruction Multiple Data or SIMD. This leads to a more energy efficient parallelization than exploiting ILP.

A common approach to extract DLP from a loop is to overlap multiple iterations of a loop. The amount of DLP that can be extracted this way corresponds to the number of iterations that can be fully parallelized without breaking dependencies between the iterations. This freedom is usually limited.

Architectures can exploit both ILP and DLP. State-of-the-art embedded VLIW processors support SIMD [Tex00, vdWVD+05], while subword parallelism in CGRAs is far less common.

In conclusion, the total number of separate slots can be reduced for the same performance by using DLP. Therefore the relative contribution of the cost of the instruction/configuration memory to the total energy consumption can be reduced. Secondly, the reduction in the number of slots leads to a reduction in the complexity of the interconnect. Because of this motivation, the processor architecture should first exploit all available DLP and only then, if required, a limited amount of ILP can be used to reach the required real-time performance. The ICMO should also be organized in a distributed fashion and customized to the different slots in order to be efficient.

Implementation: As with the register file architecture and physically distributed loop buffers, it is important to place the data-path slots as close as possible to where they are needed. E.g. the units that will compute the addresses (address generation units can be placed closer to the interface to the memory and do not need to be grouped with the data computation units). Additionally, the most heavily communicating slots can be placed next to each other in the floor plan. In order to optimize the implementation of the data-path units, without going to a full custom design, so-called Data-path Generators (DPG) can be used. Given that it is more favorable to exploit SIMD, each of the sub-word data paths can be optimized using semi-custom logic instead of random standard-cell based place and route. Such a semi-custom/custom based data path will consume significantly less area and energy, as demonstrated in [WGN01].

4.2.6. Data path - Address path

Application: Operations that contribute to the execution of the target application are different from the ones that compute the memory addresses for loads and stores. Both types of operations have different characteristics, like e.g. dynamic range and different dependence chains. By separating them, their execution can be made more efficient.

As various data optimizations like DTSE [PCD+01a, FSG+98] substantially increase the addressing complexity, the address calculations can consume a significant amount of resources and should be looked at in detail. Hence, platform-independent source code transformations have been proposed to reduce this overhead (as shown in chapter 3). On top of that, a separate platform-dependent compiler phase will be added in chapters 5 and 6 to further mitigate this problem.

Architecture: Most processors perform data and address computation operations on the same slots. As the operations that compute the addresses and the data computations

of the application algorithm follow separate dependence chains, they can be separated onto different sets of slots. Only the load/store operation forms a synchronization point between the two paths.

Address calculations have different characteristics than operations on data. The dynamic range of calculations on addresses (fixed range of e.g. 16 bits depending on memory size) and iterator values is not necessarily the same as the dynamic range of the data (e.g. 8 bit data for pixels). Separating the data path and address path enables the FUs to be of different widths. Additionally the instruction set of address and data path can be customized, in order to optimally support the required operations. This will lead to larger performance and energy gains.

Implementation: Using the same high level layout directives that have been mentioned above, the data path and address path FUs can be grouped with their respective register files (or VWRs), memory interfaces and loop buffers.

A more detailed description of the split between address path and data path can be found in [Tanon].

4.3. Putting it together: FEENECS Architecture Template

Figure 4.6 combines the optimizations for different components that have been presented in the previous Sections. The presented processor design is still a template and architecture exploration within this template is required to find the optimal architecture for one application or a set of applications. Based on the performance requirements and on initial feedback from synthesis for a certain technology node, the pipeline depth of this in-order processor has to be fixed.

The most notable components of Figure 4.6 are the data memory hierarchy, that consists of an L2 background data memory that is accessed over a generic global communication architecture (NoC, Bus or other). Data is transferred from this L2 memory into the L1 Data memory, a scratchpad memory, by the DMA. From there on, complete lines of the SPM are moved to the VWRs for data parallel computations and single values can be copied to a scalar RF. The AGU units are also placed close to the background or foreground memory units between which organize the transfers. Preferably they are even split over read and write ports so that they can be fully local to the port they are providing addresses to. The data-path FUs also come in two types: (1) a set of Complex FUs that balance DLP and ILP (a set of SIMD FUs) and use extensive forwarding to reduce VWR accesses, and (2) a set of control FUs that are of scalar type and access the scalar RF. On the instruction side, the L1 Instruction memory can be accessed directly by control code, but for the kernels, a set of physically distributed loop buffers are used (placed next to the components they control). To improve the efficiency of the instruction memory, these loop buffers also have distributed control and can synchronize at certain points using local controllers (LCs). As they do not follow

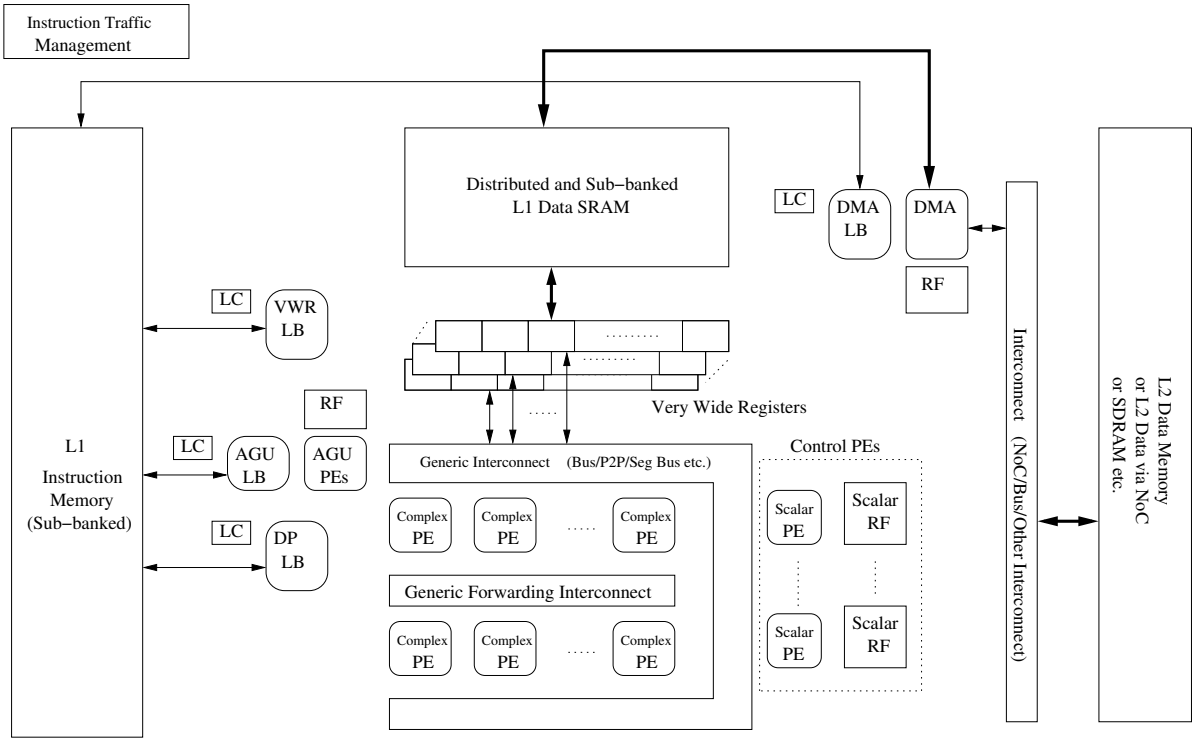


Figure 4.6.: Complete high level efficient architecture: FEENECS architecture template

a single program counter (PC), the number of NOP (no operation) entries is heavily reduced in applications that exhibit very different activation frequencies for the different components.

The communications between different processors of this type (inter processor communication) is compatible with other related work like [ADD+08], while the intra processor connections can be optimized using techniques like segmented buses etc. [Pap06].

In this architecture template, the implementation and layout guidelines that have been discussed in the previous sections have been used to group the different architectural components based on their communication requirements in order to reduce the interconnect cost. Hierarchically structured hardware design using *Data-path Generation* [RWT05] and *Energy-aware floor-planning*[J.G08] can be used to propagate these constraints down to the physical implementation.

The set of architecture modifications that is presented above has led to the filing of a patent, as described in [RC06].

In order to make use of any efficient architecture, a scalable and retargetable compiler that can compile to it is required. The next section explains a high level compiler flow to perform a phase decoupled compilation for such a processor template.

4.4. Energy estimation model

The COFFEE framework enables early estimates with enough accuracy since at this abstraction level, the full hardware description is not yet needed and therefore exploration can be faster. After architectural exploration and optimizations, when the hardware is fixed, a complete and more accurate gate level simulation and estimation is then possible.

In the COFFEE framework, different instances of the components of the processor (Register File, ALU, pipeline registers, etc.) were designed at RTL level with an optimized VHDL description. For each instance, logic synthesis is done with the UMC90nm general purpose standard cell library from Faraday [Far07] also used for the ASIC power model (in chapter 2). The result of the whole process is a library of parametrized energy models. The energy per activation and leakage power for the different components are estimated from the activity information from gate level simulation and the parasitic information. Since memories are highly optimized custom hardware blocks, the standard cell hardware design flow cannot be used. Instead, different memories were modeled using a commercial memory compiler from Artisan [ARM].

Finally, the precomputed library contains the energy annotations (dynamic and leakage) for various components of the processor using standard cell flow, and for memories, using the commercial memory compiler. A detailed description of the complete energy model is fully described in [RLA+08].

4.5. Summary

In this chapter we have seen the main features and possibilities of the COFFEE compiler and architecture framework. This framework will be used in chapter 5 and 6 to obtain the simulation results of the different benchmarks and applications used.

CHAPTER 5

AGU template

"That's thirty minutes away. I'll be there in ten."

The Wolf. Pulp Fiction

Different optimizations can boost performance and reduce energy consumption. Data Transfer and Storage Exploration (DTSE) optimizations [Cat02, CBGN98, CD00, Cat99, GH96] are crucial to efficiently map data intensive applications onto programmable platforms. DTSE transformations modify the initial code to minimize the load of shared memory buses, which is the main source of power consumption [WCNM96]. The data memory access related impact on energy and cycle count is usually dominant in such data intensive applications. Hence, it is motivated that applying these DTSE transformations to the source code is initially performed without worrying yet about the impact on other components in the platform. Hence, after the DTSE stage a direct implementation of the resulting code would lead to a high expense on addressing and local control. That is not desirable, so, in the overall methodology of [Cat02, CBGN98], it is proposed to complement the DTSE methodology with a postprocessing ADress OPTimisation (ADOPT) stage where this overhead is largely removed again. In [FM04, Fal05], the author gives a detailed explanation on how to reduce control flow at source code level. With those techniques, the control flow overhead introduced by DTSE is significantly reduced (even with respect to the original code). Also related to the address generation itself, several techniques have been proposed to alleviate the cost, as shown in our review of chapter 3. Combining all these postprocessing steps should give us already a significant reduction of the overhead. But at the level of the address generation itself, we believe that we can go further in the improvement to provide even more optimal results.

In this chapter, we introduce the methodology and framework used to create a more energy-optimized address generation unit template that targets the embedded multimedia domain. That template can then be tuned to the specific application by hand to

produce very energy-efficient AGU instances. To create such a unit, we have studied different representative applications of the domain and we have identified the hardware elements needed to optimize the address generation process. At the end of the chapter we present an optimized AGU instance taking as basis the template used.

5.1. AGU Mapping Framework

For the AGU template, we have used a mapping framework based on the task partitioning framework described in [TSU+] by Taniguchi et al. In this section, first of all, we will summarize the task partitioning framework used for dynamic reconfigurable architectures, its main features and limitations. After that, we will analyze the proposed reconfigurable AGU model used and finally, we will show the AGU mapping framework which is an extension of the global task partitioning framework.

5.1.1. Task Partitioning Framework for Dynamic Reconfigurable Architectures

To evaluate various dynamic reconfigurable architectures, Taniguchi, et al. [TSU+] proposed a Parameterized Reconfigurable Processor model (PRP-model) and a task partitioning optimization algorithm based on Simulated Annealing (SA) for architecture exploration, corresponding to their proposed PRP-model.

The PRP-model consists of an homogeneous array of Processing Elements (PEs), internal memories with different capacities, and configuration memory to store configuration data (instructions). Every specification of the PRP-model is defined by parameters and an overview of the PRP-model is shown in Figure 5.1.

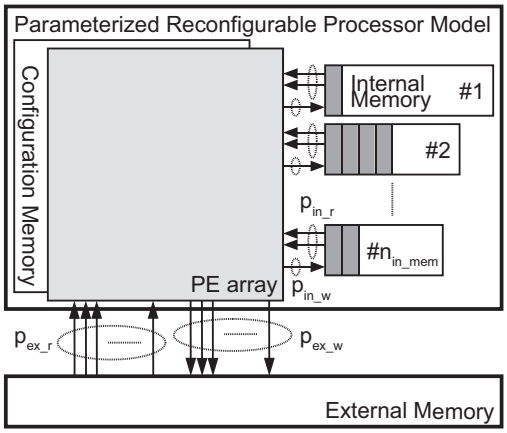


Figure 5.1.: Overview of PRP-model

The task partitioning optimization algorithm, based on Simulated Annealing, divides tasks into subtasks to minimize execution cycles considering reconfiguration overhead

for the PRP-model. Figure 5.2 shows an overview of published task partitioning framework. This generic framework can be used to instantiate and map onto a reconfigurable architecture. This allows evaluation of various architectures for specific applications by changing PRP-model parameters. For given DFG and PRP-model parameters, the task partitioning framework divides DFG into sub-DFGs (SUB0, SUB1, and SUB2 shown in Figure 5.2) to minimize execution cycles. By setting several PRP-model parameters and applying the task partitioning algorithm, designers can easily evaluate various reconfigurable architectures.

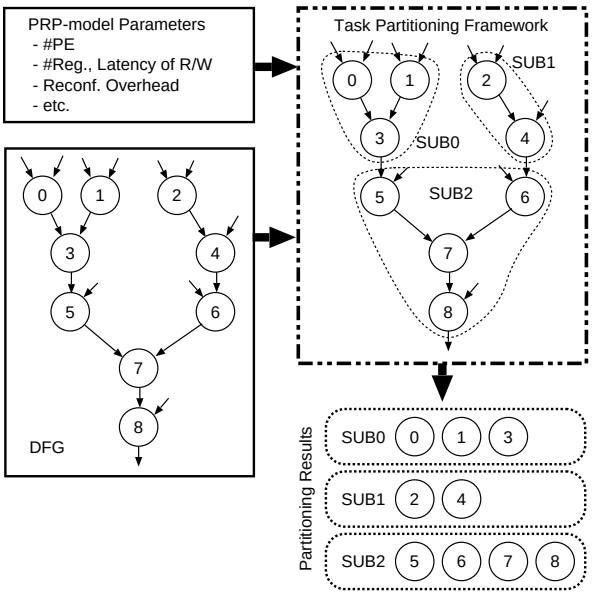


Figure 5.2.: Overview of Task Partitioning Framework

However, this task partitioning framework has some limitations in solving the AGU mapping and exploration problem. One main limitation is its highly abstracted model which assumes that all the PEs in the array are homogeneous. To calculate complex addressing with a strict energy constraint, architecture optimization considering heterogeneous PE array is necessary because modulo and multiplier are more energy consuming than addition, subtraction, and shift, and it is difficult to implement them for each PE. Therefore, in order to reuse the task partitioning framework for AGU mapping and exploration, we have significantly extended the initially proposed PRP-model, which assumes only a homogeneous PE array. In order to arrive at truly energy-optimized AGUs, we need to include a more heterogeneous model, and we have modified the task partitioning algorithm to map applications effectively on this hybrid array. To further retarget the task partitioning framework, we have also added different types of functional units which are mostly found in DMAs and other state-of-the-art AGUs. All these innovative extensions will now be described in more detail.

5.1.2. Reconfigurable AGU Model

The reconfigurable AGU model, which corresponds to a heterogeneous PRP-model, has n_{PE} PEs and their corresponding pipeline registers. Each PE has a heterogeneous function with its specific latency and is fully connected to any other PE output (except its own output). Because of this full connection, placement and routing don't have to be considered .

Figure 5.3 shows one example of the proposed reconfigurable AGU model which has four PEs. PE0 and PE1 have a latency of 1 cycle and implement *add* and *sub* instructions indicated by '+' and '-'; and PE2 has 4-cycle-latency *multiply* instruction indicated by '*'; and finally, PE3 has 30-cycles-latency *modulo* instruction indicated by '%'. Because of the full connection, the order in which the PEs are organized is not important. For example, the array would have the same functionality if PE0 was swapped with PE3. Thus, we can consider the AGU model in which any PE can be swapped with any other, as shown for example in Figure 5.3.

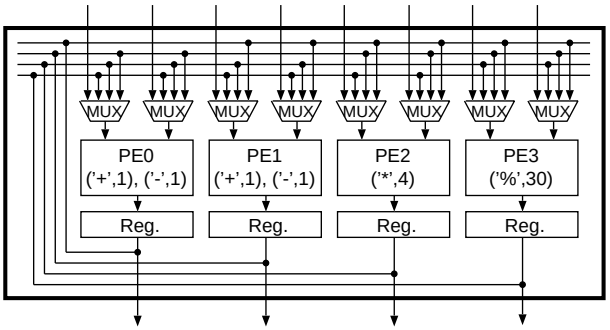


Figure 5.3.: Reconfigurable AGU Model

5.1.3. AGU Mapping Framework

Figure 5.4 shows an overview of the AGU mapping framework. For a given DFG of address calculation and reconfigurable AGU specification, the AGU mapping framework does the scheduling and the mapping of the DFG onto a certain architecture using the SA algorithm to minimize cycle count. By AGU mapping framework, the DFG of address calculation is partitioned into sub DFGs (AC0, AC1, AC2, and AC3 shown in Figure 5.4) and each node is assigned to one PE in the configuration. For more information on the algorithm used for scheduling and allocation the reader can refer [TSU+07].

When the algorithm is performing scheduling and allocation, for each iteration in the simulated annealing, the algorithm moves an assigned instruction to another PE by four types of MOVE operations. Assume that the *add* instruction indicated with '+' and assigned at PE1 in ACn (shown in Figure 5.5), is selected in SA iteration. We have four MOVE possibilities for this operation (shown in Figure 5.6):

- move to free PE in ACn-1 in Figure 5.6 (a),

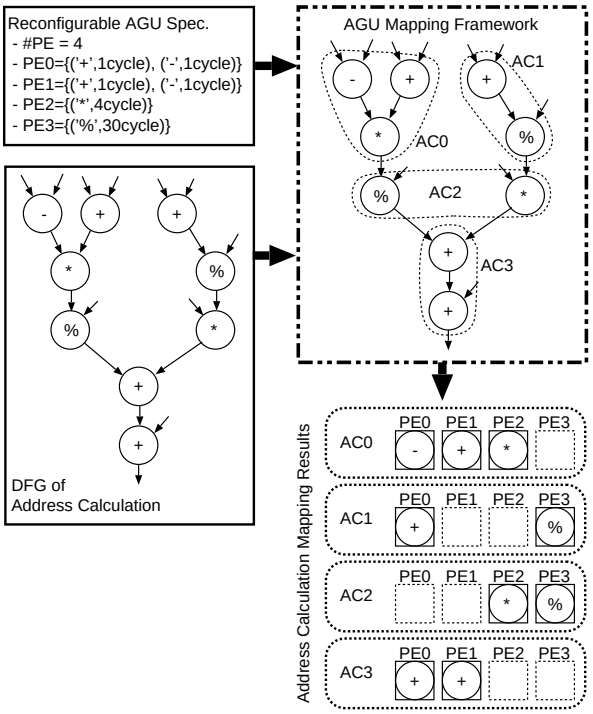


Figure 5.4.: Overview of AGU Mapping Framework

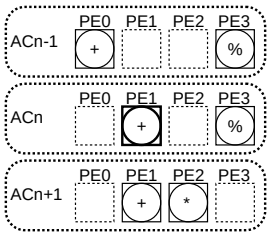


Figure 5.5.: Initial State for Example of MOVE

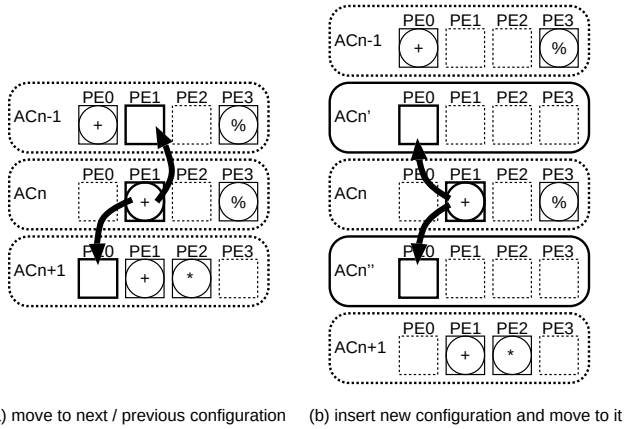


Figure 5.6.: Example of MOVE

- move to free PE in ACn+1 in Figure 5.6 (a),
- insert new configuration ACn' between previous configuration and move to there in Figure 5.6 (b),
- insert new configuration ACn" between next configuration and move to there in Figure 5.6 (b).

When the last instruction in the ACn (shown in Figure 5.7 (a)) is moved to a PE in another configuration, an empty configuration where no instructions are assigned may occur. Then, the AGU mapping framework removes the empty configuration (like Figure 5.7 (b)).

SA normally iterates by modifying the solution and accepting better solutions until the at-end condition is satisfied. To explore the given solution space effectively, invertible MOVEs should be prepared. The proposed AGU mapping framework includes removing empty configurations (shown in Figure 5.7). Therefore, we added insertion of a new configuration, which is the inverse of removing empty configurations (shown in Figure 5.6 (b)). Insertion of new configurations does not bring any effect for cycle counts immediately, but this will usually bring new opportunities to get better solutions for further iterations.

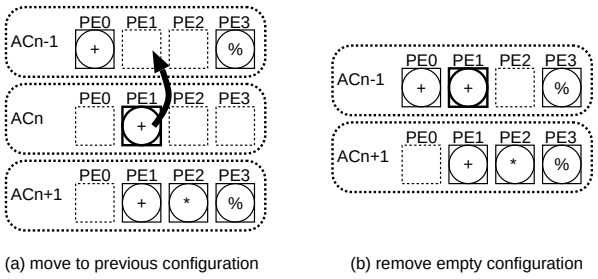


Figure 5.7.: Removing Empty Configuration

5.2. AGU Exploration Framework

In this section we explain the proposed AGU exploration technique. The AGU exploration framework iterates over the AGU mapping (as explained in the previous section) using the AGU mapping framework over different architecture candidates in the solution space. To prune the search space of AGU architecture candidates, we assume that each PE is capable of implementing a set of small DFGs. The PE implementation pattern set includes all possible combinations of PE implementation patterns. Note that the exploration of the pattern itself is outside the scope of this paper. Various works, like [BCA+04, YM04] and others, tackle this problem. The AGU exploration framework iterates over all possible combinations of the PE implementation patterns corresponding to one architecture configuration. The AGU mapping framework is in turn called to perform scheduling and allocation on this architecture in order to evaluate its costs.

Table 5.2 shows an example of a PE implementation pattern set. PE implementation patterns $(+)$ and $(-)$ indicate a PE which executes *add* instructions and *sub* instructions, respectively. PE implementation pattern $(+, -)$ designates a PE which executes *add* or *sub* instructions.

Let n_{ptn} and m be the number of PE implementation patterns and assumed PEs, respectively. The number of architecture candidates in the solution space equals $(n_{ptn})^m$ because each PE has n_{ptn} implementation patterns. However, because of full connection of reconfigurable AGU model, the AGU can implement the same functionality as the AGU whose PEs are swapped. Therefore, the number of architecture candidates with m PEs can shrink to ${}_{n_{ptn}}H_m$, which means repeated m-combinations from n_{ptn} elements. Finally, the number of architecture candidates N_{cand} is described as follows:

$$\begin{aligned} N_{cand} &= \sum_{m=1}^{max_{PE}} {}_{n_{ptn}}H_m \\ &= \sum_{m=1}^{max_{PE}} \frac{(n_{ptn}+m-1)!}{m!(n_{ptn}-1)!}, \end{aligned} \tag{5.1}$$

where max_{PE} is the maximum number of PEs. Usually, the number of functional units in embedded processors is limited, and the maximum number of PEs max_{PE} may realistically not increase so much. The number of PE implementation patterns (n_{ptn}) is expected to increase when we consider special instructions for more effective address calculation. Then, N_{cand} may explode because N_{cand} increases in factorial order of n_{ptn}.

Table 5.2 shows all architecture candidates for a given PE implementation pattern (shown in Table 5.2) for the case of $max_{PE} = 3$. Notice that each architecture candidate has a different functionality, that is, it contains different number of *add* and *sub*. When we assume an input DFG which consists in only *add*, architecture candidates No. 1, 6, and 15 cannot execute given DFG because they do not contain any *add*. In the same way, by focusing on the number of instructions like *add*, *sub*, etc. in each candidate, architectures can be explored effectively.

Pattern	Specification
$(+)$	*add*
$(-)$	*sub*
$(+, -)$	*add, sub*

Table 5.1.: Example of PE Implementation Pattern

No.	#PE	Architecture Candidate	$\#add$	$\#sub$
0	1	$(+)$	1	0
1	1	$(-)$	0	1
2	1	$(+,-)$	1	1
3	2	$(+),(+)$	2	0
4	2	$(+),(-)$	1	1
5	2	$(+),(+,-)$	2	1
6	2	$(-),(-)$	0	2
7	2	$(-),(+,-)$	1	2
8	2	$(+,-),(+,-)$	2	2
9	3	$(+),(+),(+)$	3	0
10	3	$(+),(+),(-)$	2	1
11	3	$(+),(+),(+,-)$	3	1
12	3	$(+),(-),(-)$	1	2
13	3	$(+),(-),(+,-)$	2	2
14	3	$(+),(+,-),(+,-)$	3	2
15	3	$(-),(-),(-)$	0	3
16	3	$(-),(-),(+,-)$	1	3
17	3	$(-),(+,-),(+,-)$	2	3
18	3	$(+,-),(+,-),(+,-)$	3	3

Table 5.2.: Architecture Candidates: All Combination of PE Implementation Patterns shown in Table 5.2 in case of $max_{PE} = 3$

Let min_i and max_i be the minimum and maximum number of instructions of type i that can be instantiated, respectively. Let n_i^{arch} be the number of instructions of type i in an architecture candidate $arch$. AGU exploration framework tries to perform a mapping only for architecture candidates which satisfy following equation.

$$\forall i(n_i^{arch} \geq min_i) \wedge (n_i^{arch} \leq max_i) = 1 \tag{5.2}$$

When n_i^{DFG} means the number of instructions of type i contained in given DFG, we decide min_i and max_i as follows.

$$min_i = \begin{cases} 0 & \text{if } n_i^{DFG} = 0 \\ 1 & \text{otherwise} \end{cases} \tag{5.3}$$

$$max_i = \begin{cases} 1 & \text{if } n_i^{DFG} = 0 \\ n_i^{DFG} & \text{otherwise} \end{cases} \tag{5.4}$$

Based on the Equation 5.2, the AGU exploration framework can effectively eliminate architecture candidates that do not fall into the relevant design space, as specified by the above constraints. This will reduce the worst-case exploration effort significantly.

To enumerate architecture candidates and to apply the task partitioning algorithm

for each one, an effective enumeration algorithm for all of them is needed. The proposed architecture candidates enumeration algorithm tries to enumerate all repeated combination of given PE implementation patterns. Our basic idea for obtaining all repeated combinations effectively is the pruning of the repeated permutation tree. Figure 5.8 shows the tree structure of the 3-repeated permutation of 3-patterns: (A), (B), and (C). In all repeated permutations, for example in Figure 5.8, following repeated permutations can be regarded as the same repeated combination: {(A), (A), (B)}, {(A), (B), (A)}, and {(B), (A), (A)}. To avoid duplicate representations, we represent all repeated combinations in alphabetical order by elements. In previous example, supposed permutations are regarded as one repeated combination: {(A), (A), (B)}. Then, we can obtain all repeated combinations without any duplication by tracing all permutations which only satisfy alphabetical order by element. Figure 5.9 shows a tree of all repeated combinations, all repeated permutations which satisfy alphabetical order. Then, the problem is how to trace the pruned tree effectively.

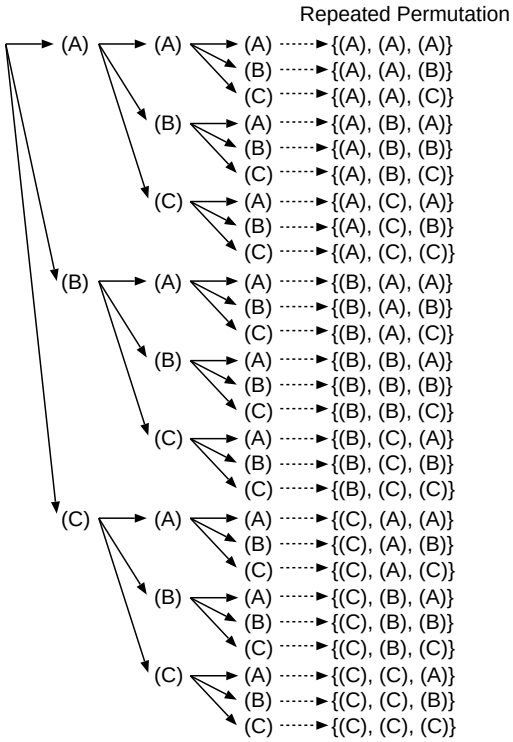


Figure 5.8.: Tree for 3-Repeated Permutations of 3-Patterns: (A), (B), and (C)

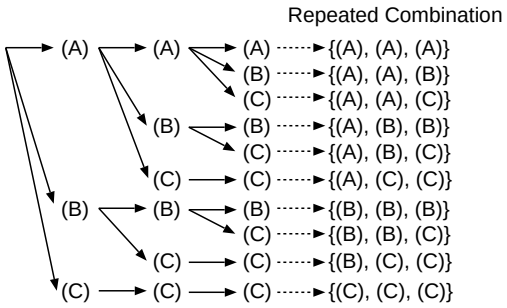


Figure 5.9.: Tree for 3-Repeated Combinations of 3-Patterns: (A), (B), and (C)

To trace the tree shown in Figure 5.9, we have constructed the architecture enumeration algorithm in such a way that it includes the architecture enumeration and mapping for each architecture shown in Figure 5.10. Let max_PE, PE_Impl, and $Cand$ be the number of limited PEs, a set of PE implementation pattern, and an architecture candidate which is a set of PE implementation pattern, respectively. The proposed algorithm is constructed by two functions: *enumerate*() and *enumerate_sub*(). The *enumerate_sub*() deals with pruned search recursively, and *enumerate*() applies tree search such that the number of combinations ranges from 1 to max_PE. *enumerate_sub*() includes two special variables to trace the tree effectively: *level* and *start*. The *level* controls a depth of recursive call corresponding to the number of elements in a combination. The *start* is to keep alphabetical order by propagating the previous element.

```
enumerate(max_PE,PE_Impl){
  for(i=1;i<=max_PE;i++){
    Cand={}
    enumerate_sub(i,0,Cand,PE_Impl);
  }
}

 enumerate_sub(level,start,Cand,PE_Impl){
  if(level>0){
    for(i=start;i<#PE_Impl;i++){
      Push i-th PE_Impl to Cand;
      enumerate_sub(level-1,i,Cand,PE_impl);
      Pop from Cand;
    }
  }
  else{
    if(Cand satisfies Eq. (5.2))
      Do AGU mapping to Cand;
  }
}
```

Figure 5.10.: Architecture Candidate Enumeration Algorithm

Figure 5.11 shows a part of relation of *enumerate_sub*() function calls for the following assumption: the number of patterns in a combination is 3, and 3 PE implementation patterns, (A), (B), and (C) are prepared. Each rectangle including "(A)," "(B),"

and "(C)," connected arrow with dash line, or "Map" represents each function call of *enumerate_sub*(). Arrows with solid line between each rectangle means functions call and return.

For each *enumerate_sub*() call, a pattern which is the same as *start* or successor of *start* is pushed to *Cand*. Then, the *level* is decremented and *enumerate_sub*() is recursively called again. Once called *enumerate_sub*() has returned, the pushed pattern is popped from *Cand*, and the next pattern is pushed to *Cand*. Then, *enumerate_sub*() is called recursively once more. Finally, when the level reaches 0, mapping is applied to *Cand*. Applying this procedure recursively, we can trace a pruned tree like Figure 5.9 and effectively explore the entire solution space. In this way, for all repeated combinations, all repeated permutations which keep alphabetical order, are effectively enumerated.

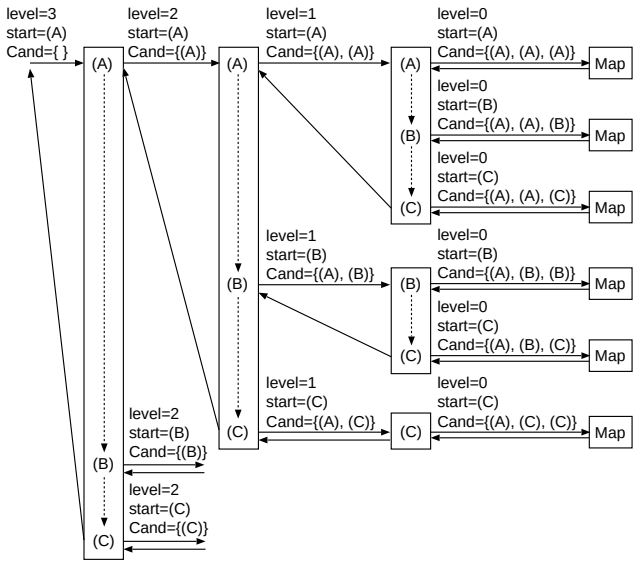


Figure 5.11.: Function Call of enumerate_sub( ) for Tree Search

5.3. Benchmarks and Applications

A benchmark is a standard program or set of programs which can run on a computer in order to assess the relative performance. It must be representative of the application domain to stress the key processor and compiler features necessary to run efficiently. In this work, we used several benchmarks and some complete applications to test the effectiveness of the proposed AGU.

Benchmarks:

The MediaBench [LPMS97] benchmark suite includes representative applications from the multimedia domain that are expected to be part of representative future multimedia applications. This includes video, image and audio coding and decoding, encryption algorithms, and 3D rendering.

JPEG: JPEG is a standardized compression method for full-color and gray-scale images. JPEG is lossy, meaning that the output image is not exactly identical to the input image. Two applications are derived from the JPEG source code; cjpeg does image compression and djpeg does decompression.

ME: The ME (Motion Estimation) is a key part of video compression used by MPEG 1, 2 and 4 as well as many other video codecs. The ME is the process of determining motion vectors that describe the transformation from one 2D image to another; usually from adjacent frames in a video sequence. The motion vectors may relate to the whole image (global motion estimation) or specific parts, such as rectangular blocks, arbitrary shaped patches or even per pixel. The motion vectors may be represented by a translational model or many other models that can approximate the motion of a real video camera, such as rotation and translation in all three dimensions and zoom.

MPEG2: MPEG-2 is the current dominant standard for high-quality digital video transmission. The important computing kernel is a discrete cosine transform for coding and the inverse transform for decoding. The two applications used are mpeg2enc and mpeg2dec for encoding and decoding respectively.

GSM: European GSM 06.10 provisional standard for full-rate speech trans-coding, prI-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits.

ADCPM: Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding. adpcm decode is the decoder and adpcm encode is the encoder.

Real-life applications:

In addition to the benchmarks we also want to evaluate the proposed techniques with a consistent set of real-life applications.

Cavity_detector: The cavity detection benchmark [BTC89] is part of a medical imaging application to detect cavities on tomography scans. The part of the application that is used as a benchmark is a chain of custom imaging filters.

QSDPCM: Quad-tree Structured Difference Pulse Code Modulation algorithm is an inter-frame compression technique for video images. It involves a hierarchical motion estimation step, and a quad-tree based encoding of the motion compensated frame-to-frame difference signal.

MPEG4: The MPEG4 is the complete application that drove chapter 2 and the optimizations and results explained will be analyzed in detail in chapter 6.

5.4. Experimental results and final template

Data-flow dominated applications are based on intense computations in the inner most loops of the codes, and Amdahl's law [Amd67] argues in favor of speeding up these parts of the algorithms. Moreover, accessing memories to bring data to the data path is costly in terms of energy. As we saw in chapter 3, many optimizations are possible, both architecture-dependent and independent. Those optimizations are crucial to improve speed and the energy efficiency but introduce a high expense on addressing and local control, for example by adding complex modulo operations, especially in the addressing part of the codes.

To obtain a good general address generation unit capable of dealing with many multimedia applications in an efficient way, we have used the applications and benchmarks mentioned in the previous section in the AGU mapping framework with different architectures. The three real-life applications (cavity detector, QSDPCM and MPEG4) and the ME benchmark have been optimized with some DTSE and some control flow techniques explained in [FM03, FV04, Fal05, FM04] before the AGU exploration.

In this work, we have considered the three inner most loops being k, j and i the iterators of the loops from the outer to the inner iterator as we can see in figure 5.12.

```
for (k=init_val_k; k<lim_val_k; k++){
  for (j=init_val_j; j<lim_val_j; j++){
    for (i=init_val_i; i<lim_val_i; i++){
      code_to_be_executed;
    }
  }
}
```

Figure 5.12.: Aspect of the inner most loops of the different benchmarks

Table 5.3 shows the needed operations used in the inner loops of the different benchmarks or applications and these operations are the ones needed to be mapped to the PEs of the AGU.

We can then construct the template AGU which will be capable of dealing with these operations speeding up the process of address generation. Table 5.4 shows the proposed PEs for the AGU template targeting the applications and benchmarks used, and figure 5.13 shows the template in a similar fashion to figure 5.3.

Benchmark / application	Source code optimizations	Operations needed
JPEG	NO	i+1 i-1
MPEG2	NO	i+1 i-1 i*j i>>j
GSM	NO	i+1 i-1 i*j i>>j
ADCPM	NO	i+1 i-1 i>>j
ME	Yes	i+j i*j (i+j)%N_0 (i+j)/N_0*N_1+N_2
Cavity Detector	Yes	i-j-k i*j i+j+k i+j i-j i+j*N_0+N_1
QSDPCM	Yes	i+j i*j i+j+k-N_0 i*N_0+j (i+j)*N_0+k
MPEG4	Yes*	i+N_0 i*j i>>j

* The MPEG4 application was manually optimized by applying the techniques explained in chapter 6.

Table 5.3.: Operations needed for the different benchmarks and applications

PE	Implements	latency
PE0	±	1
PE1	±	1
PE2	<< >>	1
PE3	*	4
PE4	(a+b)%c	30
PE5	(a+b)/c	30

Table 5.4.: Operations on the PE

Once the topology of the PEs of the AGU has been determined, we can then integrate this AGU in the machine description used in the COFFEE framework (chapter 4) to simulate the behavior of the processor. For those experiments, we used two different processors. The first processor (a.k.a base processor) was build in a similar fashion to the TI TMS c67 DSP family [TI-]. This DSP has two clusters with four functional units

each, as we can see in figure 5.14. Each cluster has one functional unit with a multiplier, and the remaining functional units have a general ALU and a shifter; besides hardware support for cluster copy and branch operations. The second version of the processor (processor with AGUs) substitutes, in each cluster, two simple functional units for the reconfigurable AGU (5.15).

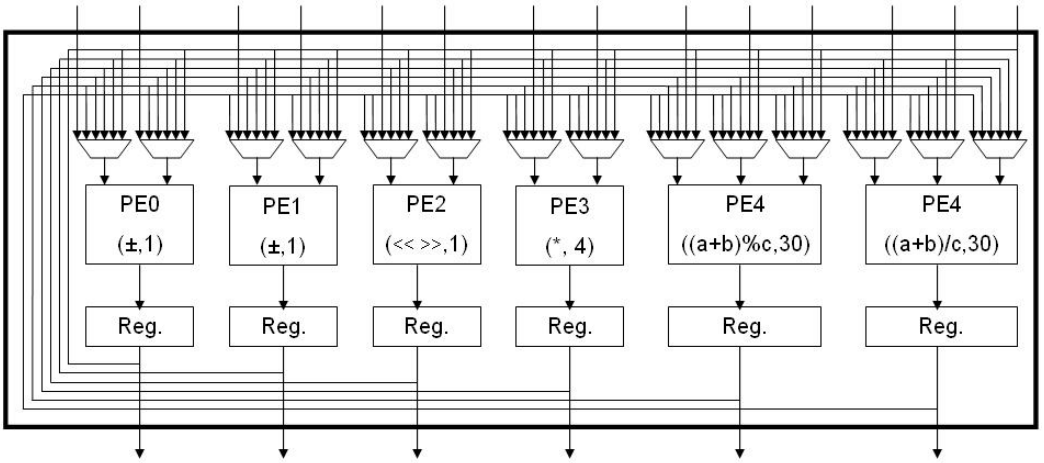


Figure 5.13.: Reconfigurable AGU template

Not all the hardware resources are used in all cases. For example, QSDPCM and ME need the complete set of PEs on table 5.4 but MPEG2, MPEG4 and cavity detector do not need the PE4 and PE5 [*(a+b)%c* and *(a+b)/c*] since they are are not used in these algorithms. For our experiments we used the complete template (figure 5.13) for the QSDPCM and ME applications and a reduced version of the template without the PE4 and PE5 for the rest of the applications. If we know in advance that just a small set of applications will be executed, tuning the template for the specific applications will give us better results, in terms of area, because of the reduction of unnecessary hardware and in terms of energy, because with less hardware resources accessing the foreground memories, less ports are needed and the energy per access of the memory is smaller[1].

[1]In register files, bit cells are grouped to form individual registers, and the registers grouped to form the overall register file. Register files are usually organized as two-dimensional grids of wires, one dimension for the control paths and the other for the data paths. In register files with multiple ports, there are correspondingly many additional control and data lines, since the control lines must be asserted separately and the data lines must be implemented separately for independent access to the bit cells (and by extension, the registers) to take place. Each intersection between a control line and its corresponding data line contains gates that connect the date line to the value of the bit cell when the control line is asserted, reading from or writing to the bit cell as appropriate.

The linear size of the bit cell scales directly with the number of ports, because that many lines (either control or data) must be able to connect the value stored in the bit cell. Thus, the area of a register file grows with the square of the number of ports (each port requires new routing for control and data). Register files are limited by routing and not by transistor density. In addition, the read access time of a register file grows approximately linearly with the number of ports. As the number of ports increases, the internal bit cell loading becomes larger, the larger the area of a register file

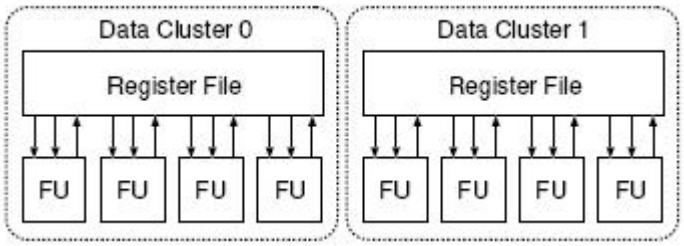


Figure 5.14.: Base processor

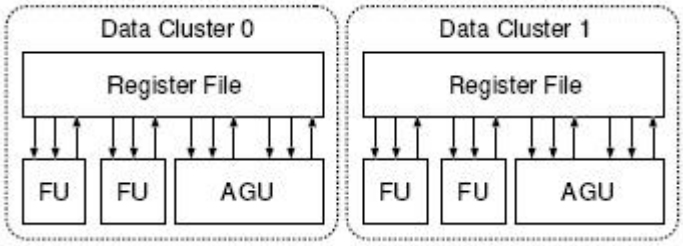


Figure 5.15.: Processor with the reconfigurable AGU

Figures 5.16 and 5.17 show the different implementations of the benchmarks and applications (except for the MPEG4 application that will be explained in detail in chapter 6). The "original" and "optimized" results were run with the base processor (figure 5.14) and the AGU with the "processor with the reconfigurable AGU" (figure 5.15).

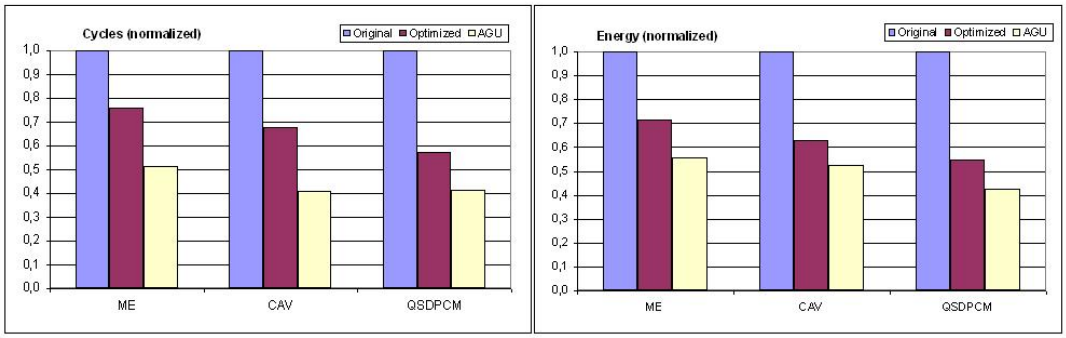


Figure 5.16.: a) Cycles and b) Energy comparison of the different benchmarks/applications after optimizations and AGU inclusion.

causes longer wire delays, and longer wires and larger cells yield to more power-hungry circuitry.

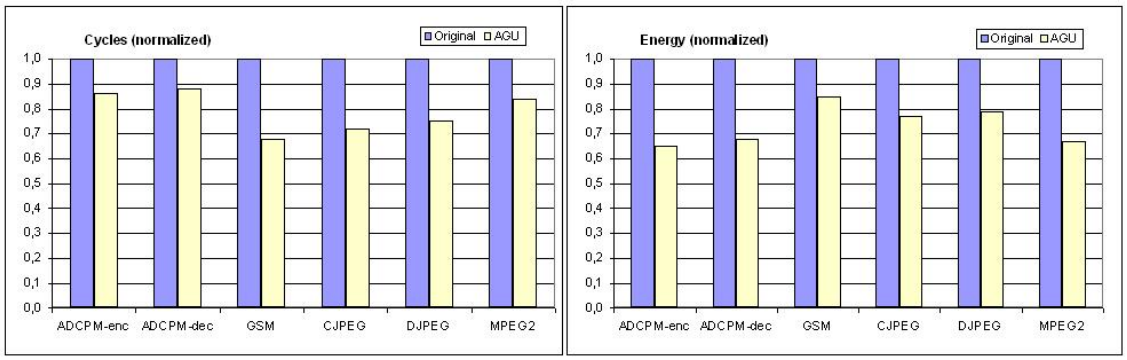


Figure 5.17.: a) Cycles and b) Energy comparison of the different benchmarks/applications after AGU inclusion.

As we can see, the usage of a specific AGU improves both cycles and energy consumption. In the case of the optimized codes (figure 5.16), we can see gains in the execution cycles of 30% compared to the optimized versions and between 50 and 60% compared to the original versions. Regarding the energy, the improvements over the optimized code are around 20% and compared to the original implementations around 50%. These results show the effectiveness of adding specialized AGUs dedicated specifically for the costly operations needed, results of the DTSE optimizations.

In figure 5.17, we notice that even in the case of a benchmark without that complex addressing operations (no DTSE optimizations accomplished), which could be easily managed without an AGU, we can notice a considerable improvement: between 10% and 30% in execution cycles and between 15% and 33% in terms of energy. Adding the AGUs offers more explicit parallelism to the compiler which can then execute more instructions at same time, and hence, finish earlier the needed calculations.

The hardware overhead introduced by the AGUs is shown in figure 5.18 The first configuration corresponds to the base processor (figure 5.14) which is used as reference. For the comparisons we just considered the functional units and processing elements (not foreground memory). For the experiments we used the processor presented in figure 5.15 but with different configurations of the AGUs. For the QSDPCM and ME we used the complete template and the overhead introduced is considerable: around 200%. This is basically because with this configuration we removed four basic functional units (with a general ALU and a shifter) and we replace them with two AGUs with a modulo and a divider processing element each; and the hardware needed to implement these operations is large. This important overhead could have been partially reduced using the next configuration ("processor with 2 AGUs with shared modulo and divider operations"), since modulo operations are based on divider operations and, hence, related hardware is similar. In this case, the hardware overhead would have been smaller, but the execution time and the energy needed would have been higher. As seen in chapter 3, area for AGUs is not anymore a crucial metric and will not be a critical metric in future designs, nevertheless it has to be taken into account. Even with the configuration of the 2 complete AGUs, the processor core accounts for less than 10% of the total processor

taking into account data and instruction memories, register files, pipeline registers and functional units.

The rest of the applications and benchmarks used the "processor with 2 AGUs without modulo or divider operations". In this case, the hardware overhead just accounts for 15% of the processor core.

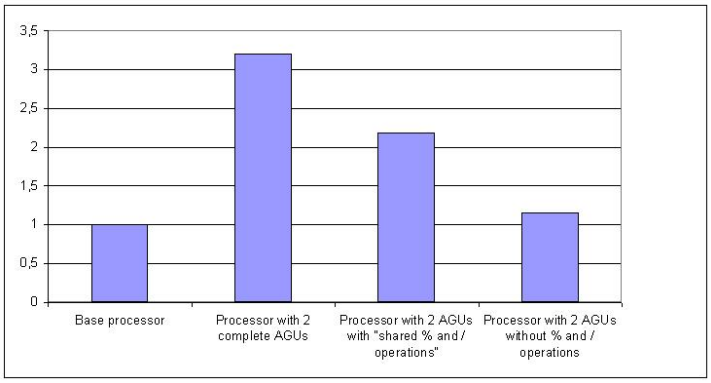


Figure 5.18.: Hardware overhead introduced by the different configurations of the processor

5.5. Optimized "stand-alone" AGU

The reconfigurable AGU template used in the previous section was used as a basis to construct another further optimized AGU. The previous template uses hardware support to compute the addresses needed by the applications. Nevertheless, more efficient loop control can still improve the results. Taking as a basis the template and the corresponding PE operations, and with a careful study of the inner most loops of the applications, we designed the AGU shown in figure 5.19.

This AGU is capable of dealing with the same operations as the reconfigurable AGU of the previous sections. It works with two modes of operation. If in the addressing codes, dividers, non-linear modulo operations or control conditions are present, then the AGU works as a normal functional unit and, when running deep-nested loops, instructions are issued from the loop buffers. In this case, the behavior -and results- will not differ much from the reconfigurable original version.

When there are no control operations, no divisions and no modulo operations (or the ones present can be simplified using the techniques explained in section 6.2.2), in this case, the AGU needs special instructions to load the context (information about the iterators, MUX configurations, etc) and then it behaves as a stand-alone AGU, capable of generating the addresses required without fetching the instructions from the loop buffer in a similar way to the work presented in [MD04].

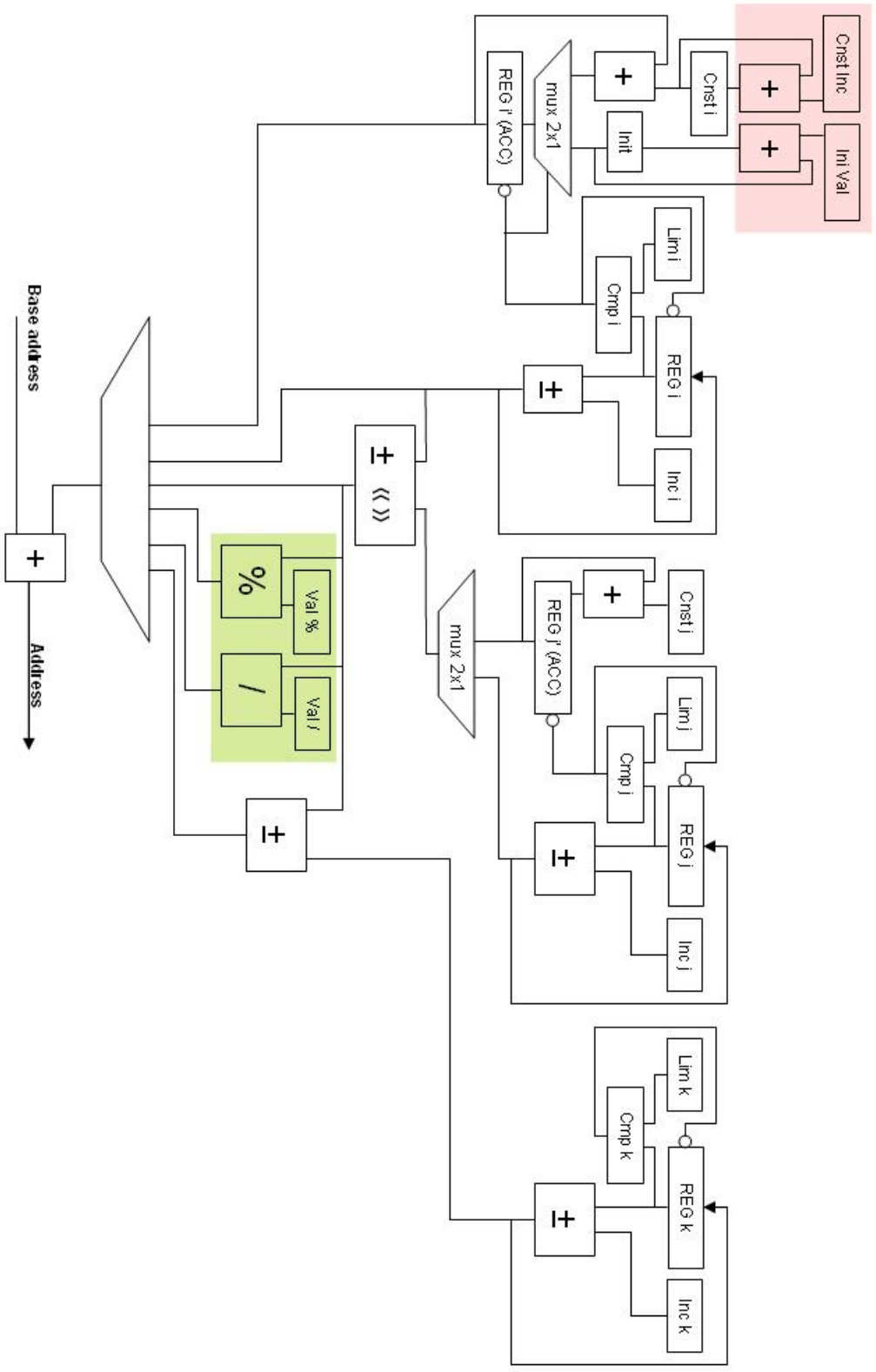


Figure 5.19.: Proposed AGU

At the moment, it is not easy to add this behavior of the AGU to the framework, and each benchmark or application that is going to be simulated requires a considerable manual effort. Experiments with the cavity detector benchmarks show an improvement of around 50% (both in terms of energy and cycles) compared to the reconfigurable version of the AGU (with the same processors, just changing the AGU behavior). Compared to the the processor with the template AGU, this new template version is 2% bigger for each configuration (with modulo and divider operations, with shared modulo and divider or without modulo and divider operations respectively). Again, if the modulo and divider operations are not needed, the overall hardware overhead (compared to the base processor) is minimal.

This is basically due to the following: the registers and configuration memory of the AGU are loaded with few instructions, and, after that, one AGU is capable of feeding the data path with one address per cycle until the end of the execution and does not need to fetch the instructions from the loop buffer.

5.6. Conclusions

In this chapter, we have detailed the AGU mapping and exploration technique based on a reconfigurable AGU model. We have shown that the proposed AGU mapping framework maps a given application DFG on a specified reconfigurable AGU model. We have also presented our AGU exploration framework which is capable to explore many architecture candidates which satisfy constraints within the AGU mapping framework.

With this framework we have identified the main computing elements required to perform an effective address generation and we have used the architecture and compiler simulator framework, explained in chapter 4, to validate our AGU template. Finally, taking as a basis the template found, we have constructed an AGU template that improves considerably the results obtained when dealing with sequential address expressions.

CHAPTER 6

Complete Optimization Methodology

"Darling, the legs aren't so beautiful,

I just know what to do with them."

Marlene Dietrich

As mentioned in chapters 1 (section 1.3) and 2, different optimizations can boost the performance and reduce the energy consumption of our applications. High performance at low energy usage is crucial for the multimedia handheld domain where applications should run at an acceptable quality and users expect to have a long battery life.

In this chapter, after a small introduction about the application under study, we analyze the required optimizations to obtain near-optimal results of a real multimedia application running on a VLIW processor. We especially emphasize on reducing the data memory access and related addressing overhead, including also the instruction memory overhead, at different abstraction levels. Even though optimizations can be achieved in three different contexts (source code [FM04], compiler [ASU86] and architecture [FFY04]), in this chapter we present a step-wise script (or methodology) that gradually reduces energy reusing techniques that already have been published. This flow combines steps in a sequential and unidirectional way (no feedback loops or backward iterations are required, once a step is applied) with constraint propagation between the steps. From a first "straightforward implementation" the overall improvement of the flow boosts execution time and decreases energy consumption up to 90% for a standard application.

6.1. MPEG-4

Media processing refers to the computation required for the creation, encoding, decoding, processing, display and communication of digital multimedia information such as images, audio, video and graphics. As a representative application of the multimedia domain, we selected an MPEG-4 encoder which will be the driving example of all our work.

MPEG-4 is a standard used primarily to compress audio and visual (AV) digital data. Introduced in late 1998, is the designation for a group of audio and video coding standards and related technology agreed upon by the ISO/IEC Moving Picture Experts Group (MPEG) under the formal standard ISO/IEC 14496. MPEG-4 is being used in web (streaming media) and CD distribution, conversation (videophone), and broadcast television, all of which benefit from compressing the AV stream.

MPEG-4 absorbs many of the features of MPEG-1 and MPEG-2 and other related standards, adding new features such as (extended) VRML support for 3D rendering, object-oriented composite files (including audio, video and VRML objects), support for externally-specified Digital Rights Management and various types of interactivity. AAC (Advanced Audio Codec) was standardized as an adjunct to MPEG-2 (as Part 7) before MPEG-4 was issued. MPEG-4 is still a developing standard and is divided into a number of parts. Unfortunately, the companies promoting MPEG-4 compatibility do not always clearly state which part of the standard they offer compatibility for.

The key parts to be aware of are: MPEG-4 part 2 (MPEG-4 SP/ASP, used by codecs such as DivX, XviD and 3ivx and by Quicktime 6) and MPEG-4 part 10 (MPEG-4 AVC/H.264, used by the x264 codec, by Quicktime 7, and by new DVD formats like the already deprecated HD DVD and Blu-ray Disc). Most of the features included in MPEG-4 are left to individual developers to decide whether to implement them. This means that probably no complete implementations are available of the entire MPEG-4 set of standards. To deal with this, the standard includes the concept of "profiles" and "levels", allowing a specific set of capabilities to be defined in a manner appropriate for a subset of applications. More information can be obtained in [Koe99, PE02]. The model used to be implemented in the different platforms is based on a video compressor with main profile and permits I, P and B slice compression, with 4:2:0 chroma format, and 8 bit pixel sample depth [ISO01, Kuh04].

In our work, we compressed 4 images in QCIF size (176x144) compression simulations using an IPBB Group of Pictures (GOP). The algorithms related to the temporal compression (changes between one frame and the next frame(s)) are the motion estimation and motion compensation. The motion estimation used is the Full Search Motion Estimation (FSME) where all the motion vectors candidates are computed to get the minimum Sum of Absolute Values. A detailed explanation of the MPEG4 and related algorithms can be found in [Kuh04]. Even if the choice of FSME can be discussed, those results can be extrapolated to the fast implementation of the motion estimation in chapter 2 and gains in the same order of magnitude can be expected in the case of a hierarchical motion estimation.

6.2. Background data memory optimization

The VLIW-DSIP framework used in our work (chapter 4) allow us to explore and optimize the code of our application (or set of applications), compilation options and architecture for a specific purpose. In our case, since low energy is one of the key design goals of the current embedded systems, the optimizations will aim to decrease energy consumption of the MPEG4 encoder.

Memory address calculations often involve linear and polynomial arithmetic expressions which have to be calculated during program execution under strict timing constraints and can significantly degrade the performance and increase power consumption: 50%-75% of the power consumption on embedded multimedia systems is caused by memory accesses [MNCM97, WCNM96]. Hence, it is very important to carry out these accesses and related addresses computations in an effective way.

A memory cluster is defined as a number of memory blocks with common input and output ports. The memory blocks themselves could have a complex internal memory organization (e.g. interleaved memory banks) and local memory control (like loop control, cache replacement policy or FIFO control). Each cluster has a set of ports connecting to other memory clusters. It also has an optional decoder which could be used to reduce the instruction traffic on the memory bus (which is also power consuming) and to reduce the size of the required program memories (which are expensive in terms of energy and area). The COFFEE framework allows us to change and modify different parameters and configurations of the memory hierarchy, from different configurations (cache, a scratchpad or a multi-level memory hierarchy) sizes of the different caches, register file, number of read/write ports, etc.

6.2.1. Scratchpad Memories

In a typical processor platform, memories are organized in levels in order to reduce energy consumption [BB95, OK06, HPon]. The memory hierarchy distinguishes each level in the "hierarchy" by response time since response time, complexity, and capacity are related. Lower levels are designed with faster smaller memories like caches, scratchpads or SRAMs, and higher levels are designed with slower, denser and larger memories like SDRAMs. For instance, a memory access to level-1 memory is in the order of 1-5 cycles, while an access to a level-2 SDRAM takes around an order of magnitude more. So, small memories have faster access times and consumes less energy compared to larger memories. Figure 6.1shows an example of a three level memory hierarchy.

Between the main memory and the computing elements, the memory hierarchy can be built using caches or scratchpad memories. On-chip SRAM caches consume 25% to 45% of the total CPU power [PND98a] and the allocation of data on those memories is done at run-time. Scratchpad memories are high-speed memories where the compiler generates explicit instructions to move data from and to the following levels of the memory hierarchy, often using DMA-based data transfer [BSL+02a, AC06].

Since scratchpad memories are software controlled and allocation is done at compile

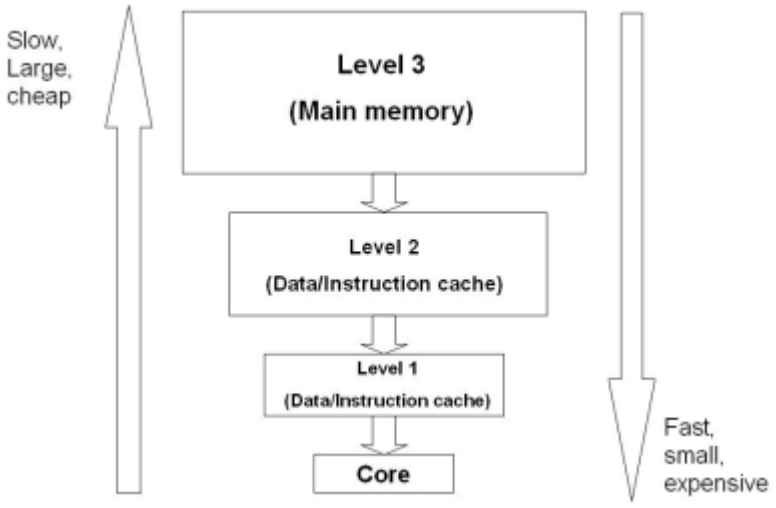


Figure 6.1.: Example of a three level memory hierarchy

time, this alleviates run time issues and provides various benefits. First of all, chip area is reduced, and thus energy, compared to the run-time hardware support that is needed by cache based schemes (e.g. superscalar processors). Another benefit is that complexity is usually easier and faster to deal with in a software design than in a hardware design. This type of memories reduces considerably the energy consumption (average reduction of 40%) and the area-time product (46%) [BSL+02a, AC06] but relies on compiler/programmer effort. Since SPMs have been proven to be more energy, area and performance efficient than caches [KKC+04] and compile time allocation improves energy reduction and predictability and allows an easier analysis and optimization of the application codes, scratchpad memories are widely used in the embedded domain.

6.2.2. Data Transfer and Storage Exploration

Besides the use of scratchpad memories, optimizations targeting the reduction the transfers between the different levels of the memory hierarchy and exploiting the locality of the data can considerably enhance energy savings. Data Transfer and Storage Exploration (DTSE) optimizations [Cat02, CBGN98, CD00, Cat99, GH96] are crucial to efficiently map data intensive applications onto programmable platforms. The goal of DTSE is double: on one hand DTSE reduces the storage requirements of embedded applications and minimizes the absolute amount of memory needed to execute the application; on the other hand, DTSE optimizes the locality of data accesses at a very high level in order to reach a high utilization of small but efficient memories which are close to the processor.

The DTSE methodology consists of several steps depicted in figure 6.2. The first step in the methodology is a memory oriented data flow analysis followed by global data flow and loop transformations in order to reduce the required amount of background memories. After that, follows data reuse transformations exploit a distributed memory hierarchy. Storage cycle budget distribution is performed, in order to determine

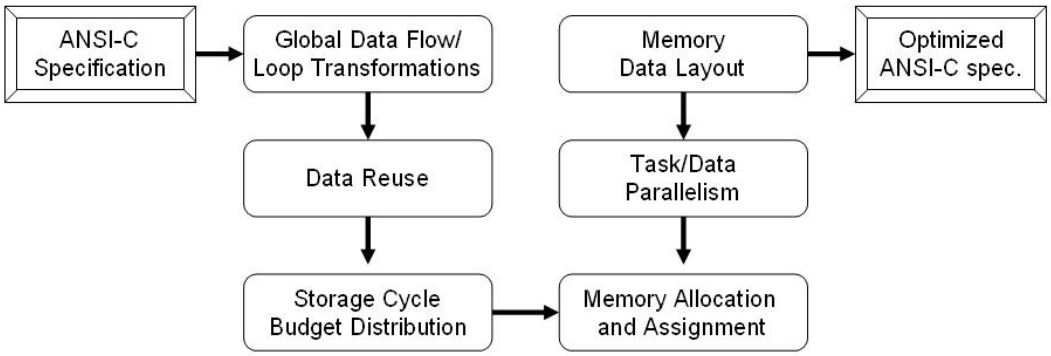


Figure 6.2.: Overview of the DTSE methodology

the bandwidth requirements and the balancing of the available cycle budget over the different memory accesses in the algorithmic specification. The goal of the memory hierarchy layer assignment phase is to produce a netlist of memory units from a memory library as well as an assignment of signals to the available memory units [BMCC03]. For multi-processors systems, a task/data-parallelism exploitation step [CDWD01] takes place minimizing the communication and storage overhead induced by the parallel execution of subsystems of an application. Finally, data layout transformations (e.g. in-place mapping) help to increase the cache hit rate (when caches are used) and to reduce the memory size by storing signals with non-overlapping lifetimes in the same physical memory location.

In this work, we have implemented several DTSE transformations:

- code rewriting without pointers and dynamic memory allocation
  - This transformation does not belong actually to the DTSE steps but to the Data Type Refinement (DTR) stage which precedes all the DTSE stages [Cat00].
- change of data types to maximize the usage of 8-bit registers
- image computation transformations (loop transformations) to compute calculations at macro-block level instead of image level
- Arithmetic Cost Minimization (ACM): [PMD+02] aims to exploit algebraic and modulo transformations for minimal number of calculation instances without changing control flow
- Non-linear Operator Strength Reduction (NOSR): [GMV+00] focuses on substituting the non-linear expressions, e.g. modulo operation with a combination of conditionals and induction variables

All these DTSE optimizations belong to the Processor-level DTSE (P-DTSE) stage which deals with a single-threaded application task running on one or more functional units or processors without exploiting data-parallelism. When the target application and platform does not meet these assumptions, then also other flavors of the DTSE approach should be applied. So, to ensure a more complete covering of the topic, also 2 other stages will be briefly introduced here: Thread-frame-level DTSE (T-DTSE) and Data-level DTSE (D-DTSE):

- T-DTSE: this extension of DTSE focuses on optimizing the transfer and storage of data when multiple concurrent (non-deterministically triggered) threads are executed on one or more processors [ZPD+07, MWP+01].

- D-DTSE: this branch of DTSE exploits data parallelism in the mapping to processors with SIMD, vector or subword parallel support [SMC00, SMC02a, SMC02b, SMC03, OdBGB+03].

In the global system design flow, the P-DTSE stage is preceded first by the T-DTSE and then by the D-DTSE stages.

6.2.3. Control flow optimization

DTSE transformations modify the initial code to minimize the load of shared memory buses which is the main source of power consumption [WCNM96]. These transformations are usually achieved at a high expense on addressing and local control. In [FM04, Fal05], the author gives a detailed explanation on how to reduce control flow at source code level using the following techniques:

- **Loop nest splitting:** the goal of this transformation is to generate regular control flow in the innermost loops of data flow dominated applications by minimizing the executions of *if*-statements.

- **Advanced Code Hoisting:** this technique aims at moving portions of code from inner loops to outer ones. In contrast to existing code motion techniques, this is performed under consideration of control flow aspects. Depending on the conditions of *if*-statements, moving an expression can lead to an increased number of executions of this expression.

- **Ring buffer replacement:** DTSE generates small arrays serving as temporary buffer for data. Ring buffer replacement focus on the elimination of those arrays.

With those techniques, the control flow overhead introduced by DTSE is reduced (even in regard to the original code) and optimizing address generation will give us optimal results.

6.3. Address generation optimization

Multimedia applications are based on intensive deep-nested loops where the main algorithm is usually located in the inner-most loop. Chapter 5 has shown a specialized AGU developed to speed up all address related computations in the inner-most loops.

Once memory and data computation have been optimized, in the multimedia domain, the bottleneck resides in the address generation of the application [TJCC08]. Address computation often involves linear and polynomial arithmetic expressions which have to be calculated during program execution under strict timing constraints and complex media algorithms typically have a large number of two-dimensional array and vector accesses. To provide enough data bandwidth, usually several programmable address generator units run in parallel. If the number of AGUs increase, the program area grows, and then the instruction memory and instruction decoders, and the area overhead introduced can become a dominant factor. As an example of this overhead, we point out that more than 25% of the area of the Coolflux DSP is used by the AGUs [Phi04]. Indirectly, this increase in the area of the AGUs also leads to an energy overhead and usually real area versus energy trade-offs are present. Many optimizations aim at reducing this hardware overhead [MKCdM97, MCM94, MCJM98, MCJdM96] .

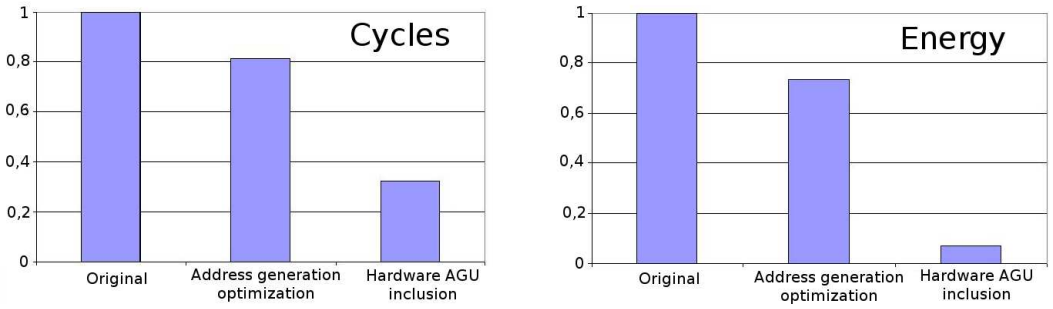
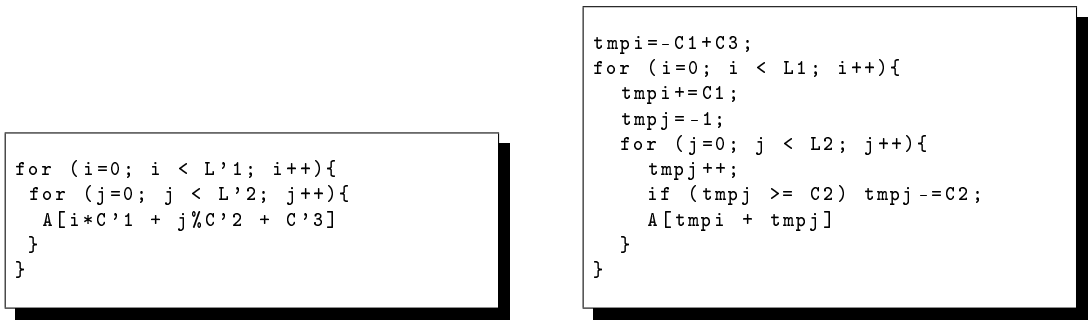


Figure 6.3.: Cycles a) and Energy b) improvement after address generation optimization and custom AGU inclusion for a MPEG4 Encoder.

After some manual and compiler Data Transfer and Storage Exploration optimizations have been applied, figure 6.3 shows the improvement on the overall code of addresses generation and related hardware optimization. The first optimization (address generation optimization) is basically due to loop transformations at source code level. Those transformations aim at exploiting the maximum parallelization of the computation of loops and to reduce, or simplify, the number of iterations. These transformations include global subexpression elimination, advanced code hoisting or loop nest splitting. These transformations reduce and simplify the computation needed by removing or reducing control flow, from inner loops to higher levels where the conditional branches are executed less often. A detailed explanation on these techniques can be found in [FM04] and similar work can also be found in [LZSS04, XSLS05].

Figure 6.4a) shows a very simple example of what code would look like after DTSE and control flow optimizations. The indexes of the arrays are usually calculated with



```
for (i=0; i < L'1; i++){
 for (j=0; j < L'2; j++){
  A[i*C'1 + j%C'2 + C'3]
 }
}
```

```
tmpi=-C1+C3;
for (i=0; i < L1; i++){
  tmpi+=C1;
  tmpj=-1;
  for (j=0; j < L2; j++){
    tmpj++;
    if (tmpj >= C2) tmpj-=C2;
    A[tmpi + tmpj]
  }
}
```

Figure 6.4.: Examples of inner-most loop source code a) after DTSE and control flow transformations b) after transformations preparing for hardware support.

modulo and multiplication operations. The example can be easily extended to multiple 2D arrays. With this kind of source code, the last transformation seen in 6.4b) alleviates the hardware needed for the calculation of the indexes. This gives a double benefit, first of all, it gives to the compiler freedom to exploit parallelism, and second, preparing the code in such a way gives impressive benefits adding some "simple" hardware support.

The second optimization in figure 6.3 (hardware AGU inclusion) shows the benefit of the inclusion of custom hardware after all the previous optimizations and code transformations have been applied on the MPEG4 encoder application. In a similar fashion to [MD04], we can accelerate those parts of the algorithm with a low-power loop accelerator based on clustered distributed address generation unit. The computation of the address generation in those parts of the code is easy and does not require a lot of hardware support; hence, minimum hardware and a small number of registers is enough for the generation of the addresses of the inner-most loops to boost the transfer of data to the functional units. The addresses of the rest of the application are computed in the default more complete address generation unit, similar (or equal) to a normal functional unit that requires more energy per operation but occur rarely during the execution of the application.

6.4. Data-path optimization

The data path configuration has a high impact on the performance of the processor. It might be composed of one or more data path clusters and might contain one or more register files and one or more functional units. By clustering, the routing length and interconnect complexity inside a cluster are reduced (i.e. good for both power and speed), at the price of increased compilation complexity due to the additional cluster-to-cluster data transfers. Between clusters, communication is still possible by means of inter-cluster interconnect. To get an optimal configuration of the data path, an architecture exploration must be performed for the application domain [Bar05].

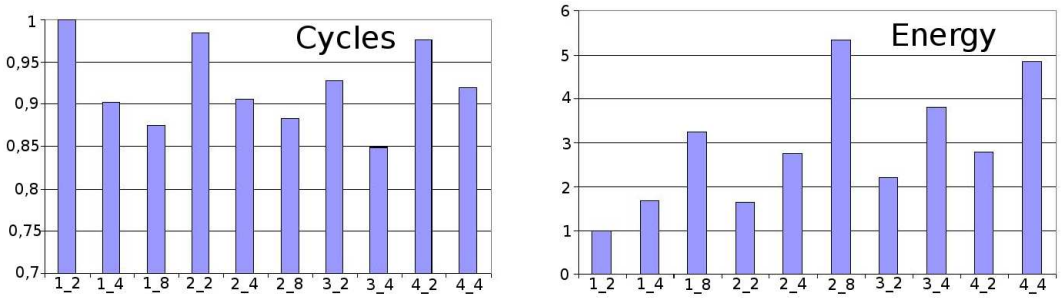


Figure 6.5.: Cycles a) and Energy b) for different cluster configurations (#Clusters_#FUs per cluster)

To illustrate the importance of the exploration, in figure 6.5 we can see the influence of the number of clusters and functional units on the energy and cycles. It is difficult to know a priori what that relation is and what will be the best configuration: having more functional units reduces execution time but increases energy, basically due to the rise of multi-ported register files; splitting up the register files (called clustering) reduces the energy consumption but can add a penalty in cycles due to intercluster communication [GBK07].

In addition to the general architecture exploration, some application-specific optimization have been studied. First of all, since multimedia applications have a lot of multiply-and-accumulate and divide operations, specific hardware support has been developed. The hardware and compiler support needed for that primarily improves performance decreasing the number of cycles needed for the application, and hence, indirectly, also energy.

6.5. Instruction loop buffering optimization

Loop buffering is an effective scheme to reduce energy consumption in the instruction memory hierarchy. In any typical multimedia application, significant amount of execution time is spent in small program segments. Energy can be reduced by storing them in a small loop buffer instead of in the big instruction cache [BHK+97, AJB+04, JBA+05]. But, as more instruction level parallelism is extracted from the application, wider data paths and wider loop buffers are needed. In the following figure, we can see the effect on the energy for different configurations of the loop buffer with the same processor.

Figure 6.6 shows the impact of different loop buffers sizes (*size x bitwidth*) on the overall energy consumption. As we can see, a good choice on the loop buffer size can significantly reduce the energy of the whole application, up to a factor 3 for the same processor architecture. A good loop-buffer architecture needs to be big enough to handle all the instructions in a loop or in some nested loops. An over-dimensioned size will result in costs of energy wasted, since the access to a bigger memory implies a bigger cost for each access. A smaller-than-needed size will have to be filled several times for the execution of the loop (or nested loops) resulting in unnecessary accesses to the next

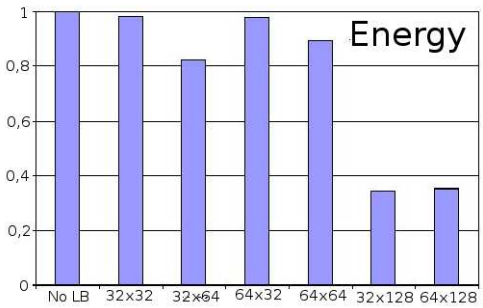


Figure 6.6.: Energy impact for different sizes of loop buffers for a fixed processor (1 cluster, 8 FU)

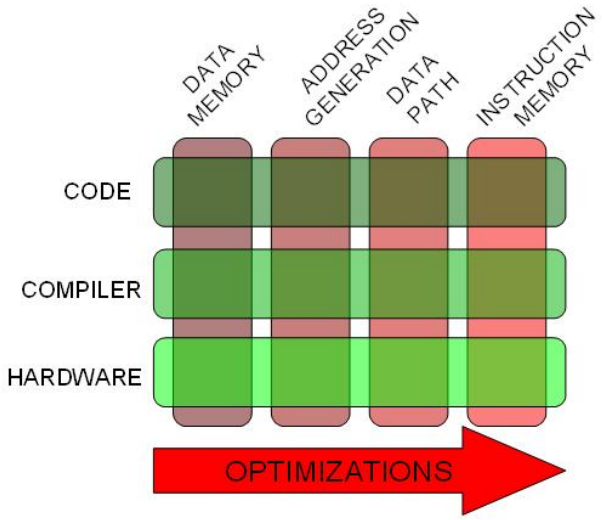


Figure 6.7.: Overview of optimizations

level of the cache.

A detailed explanation of the loop-buffer organization, techniques and heuristics can be found in [Jay05b, VA05].

6.6. Overall improvement and final energy distribution

In figure 6.7, we can see a picture of the optimizations accomplished. Those optimizations basically target (in this order) data memory, address generation, data path and instruction memory at three different levels: code, compiler and architecture. In this subsection, the different improvements of all the optimizations are shown.

Figures 6.8 and 6.9 show the energy and cycles improvement after the different optimizations applied progressively where each optimization is implemented having as an

starting point in the previous fully-optimized case.

Figure 6.8 shows the global energy improvement and the different contributions of the Instruction Memory (IM), Data Memory (DM), Register File (RF), Pipeline Registers (PR) and Functional Units (FU) for a baseline processor with 8 FU which gives a good trade-off between speed and energy consumption. For clarity of the graphic, the RF, PR and FU were grouped together since their contribution is rather small compared to DM and IM. Later, in figure 6.10, we will see and analyze the detailed energy distribution of the final implementation.

As we can see in figures 6.8 and 6.9, both the energy distribution and the number of cycles are considerably improved after the different optimization steps. We must say that both DM and IM in the baseline processor were quite overdimensioned to ensure that they do not unnecessarily constraint the results and give enough freedom to the optimizations regarding data layout and instruction scheduling, among others. As we can see in the original code, instruction memory is the dominating energy-consuming factor, beside the overdimensioning of the IM; at this point we did not use loop buffers and this has a extreme high penalty on the energy on algorithms with intense loop calculation.

Source code DTSE and control flow optimization improves the global energy consumption around 45% and the cycles needed around 65%. The DTSE optimizations basically are responsible for the DM improvement (from 0.37 to 0.14) and control flow has more effect on the IM since control flow optimizations rearranges loops maximizing the execution without conditions and the contribution of the IM goes form 0.58 to 0.39. Both, DTSE and control flow have a high impact on the speed of the algorithm since optimizing data transfer reduces the cycles used to fetch the data and control flow optimization reduces the number of branches that the code executes.

Data memory optimizations that were accomplished include other "non source-code" DTSE optimizations and improve the usage and effectiveness of the memories (size, numbers of ports, etc) and other compiler optimizations as the ones explained in the bibliography (on section 6.2). Basically, the usage of an optimized memory hierarchy to exploit temporal locality in the memory accesses on array signals has a very large impact on the power consumption of data dominated applications. Moreover, this is especially important in multimedia algorithms which have to deal with huge amounts of data and where the energy related with the transfers and storage becomes critical. DTSE optimizations minimize the transfer and storage of data, and, as expected, this has a big impact on the energy but a small effect on the speed since data computation remains the same.

Data path optimization introduces some custom instructions (together with their corresponding hardware and compiler resources, like MAC and divider operations) and reduces the number of cycles needed to compute the same code, but does not improve the energy since adding more hardware resources improves the performance but has a penalty on the energy consumption. After the data path has been optimized, memory accesses dominate the power consumption on embedded multimedia applications [WCNM96, MNCM97] and hence, it is very important to carry out these accesses and related address computations in an effective way. Since our application is data driven

and the kernel of the application is based on intensive deep-nested loops, adding an special dedicated address generation unit for these parts of the algorithm brings additional improvements in terms of energy and speed. The rest of the memory addresses computation occur rarely and are computed in the general, more configurable, but also more energy hungry, AGU. As we can see in the figures 6.8 and 6.9, after data path and AGU optimization, the energy used is halved and speed is improved around four times compared to the previous version.

Finally, as explained in section 6.5 and the related bibliography, we can see how the inclusion and optimization of the Loop Buffers (LB) also improves considerably the speed and energy consumption of the algorithm. The inclusion, or optimization, of loop buffers basically affects the IM since the instructions for the execution of the loop are stored in the small LB instead of a bigger instruction cache or scratchpad memory.

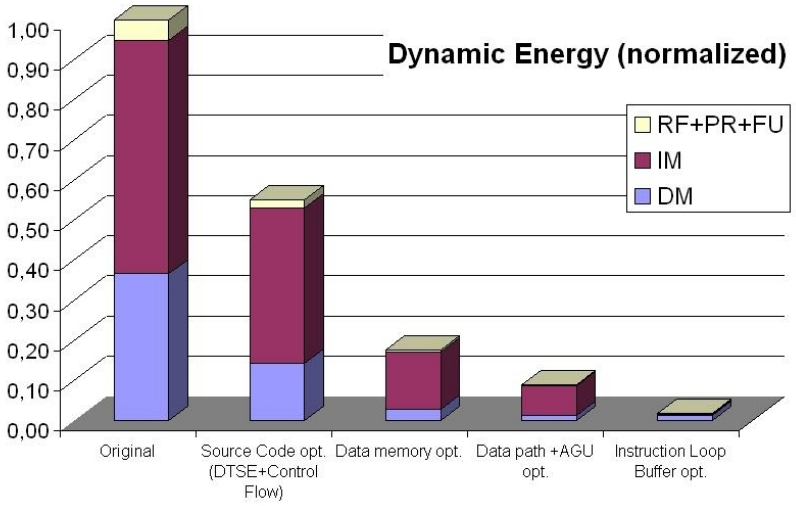


Figure 6.8.: Progressive energy reduction through optimizations for a fixed processor

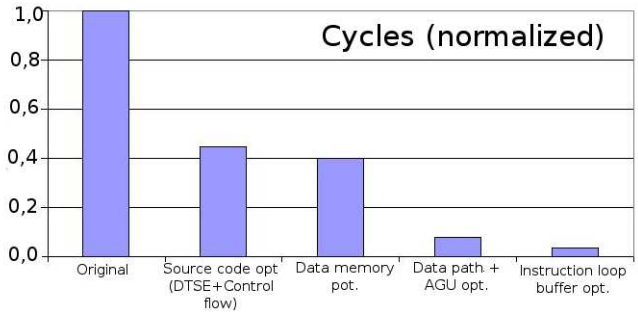
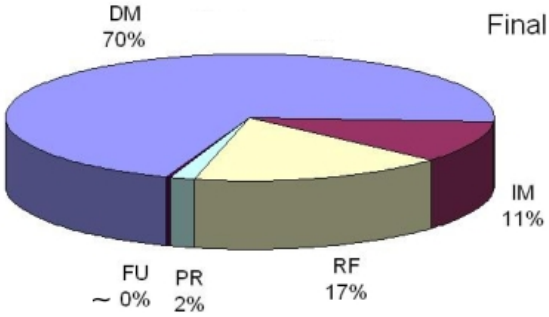


Figure 6.9.: Progressive execution-time reduction through optimizations for a fixed processor

After all the optimizations, figure 6.10 shows the final energy distribution of the MPEG 4 application for a GOP IPBB which corresponds to the last bar in figure 6.8. As seen

in the figure, around 70% of the energy of the application is due to the data memory. This is basically due to the fact that multimedia applications are based in continuous streams of data. The register file also accounts for a significant part of the energy of the application for the aforementioned reason. The instruction memory accounts for a small part of the energy distribution since the inclusion of loop buffers drastically reduces the consumption of energy of small but computationally-intensive loops. This final energy pie-chart shows that DM is the dominant factor.



Final energy distribution

		Dynamic Energy (mJ)	%
DM	= Data Memory	15,01	70,49
IM	= Instruction Memory	2,29	10,73
RF	= Register File	3,59	16,87
PR	= Pipeline Registers	0,33	1,57
FU	= Functional Units	0,07	0,35
Total		21,30	100,00

Figure 6.10.: Final energy distribution of the MPEG 4 application (GOP IPBB) on the optimized VLIW

6.7. Conclusions

In this chapter we presented a step-wise script that gradually reduces energy consumption of multimedia applications targeting the embedded domain. We show how, after the different steps, the improvement in energy and execution time reaches 90% and data memory dominates the new energy distribution. Even if this analysis was done on an MPEG4 encoder application, the methodology and optimizations done can be extended to any data-flow oriented application.

CHAPTER 7

Summary and conclusions

"May I never be complete. May I never be content.

May I never be perfect. Deliver me, Tyler,

from being perfect and complete."

Fight Club

This chapter summarizes the main results found in this dissertation and provides ideas for further improvements.

7.1. Summary of contributions

The main goal of this thesis was to obtain a complete methodology to optimize the energy consumption on data-flow applications for low-power embedded VLIW processors. The first part of the work intended to justify the usage of scratchpad memory based VLIW processors in the low-power embedded domain. For this comparison we mapped the same application into four widely used platforms: a VLIW-DSIP, an ASIC, an off-the-shelf DSP and a FPGA. The application used was a complete MPEG4 application which is well known in scientific literature and commonly used as reference example. Moreover, this application presents the typical characteristics of data-flow applications and, hence, the extracted results can be extended to other data-flow applications. Even if the application was mapped to four different platforms, each solution was optimized enough to allow for quantitative comparison in a fair way over four main metrics: execution time, energy consumption, area and design time.

The results show that the VLIW alternative has an energy consumption that can rival with the ASIC solution but pays an extra penalty in the design time. The VLIW solution makes sense if commercial devices can not satisfy the requirements or if the image size, and hence the performance and energy efficiency must be boosted. In any of these cases, the VLIW solution is the recommended solution if we know in advance that several additional follow-up products will be designed by the same team on the same platform. In this situation, even if off-the-shelf customized components are available, a larger first investment in a flexible domain-specific platform will pay off for the effort and time saved in the following product versions. This is the normal situation for embedded devices running multimedia applications, which need some flexibility to execute different applications -or upgrades of the applications- and usually evolve over time with new versions of the devices.

Another contribution of the thesis is to provide a systematic classification of address generators and a review of literature according to the classification to illustrate the complementarity or overlap of the optimizations on the address generation process. We focused on AGU architectures and on compilation techniques to optimize address generation process for scratchpad memories due to the power restrictions of the embedded domain. We considered the address generation process for DSPs and VLIW-like architectures, which have to deal with computing intensive algorithms where access to data is a main issue.

One of the main contributions of this thesis is to introduce into the COFFEE framework the results of the exhaustive architecture exploration method for reconfigurable architectures. With those results we proposed to find a template for an address generation unit for VLIW-like processors. Moreover, using this template we can construct a new AGU structure which is more efficient but needs more manual effort (that cannot yet be automated by the framework).

We also presented a step-wise script that gradually reduces energy consumption of multimedia applications targeting the embedded domain. We show how, after the different steps, the improvement in energy and execution time reaches 90% and data memory dominates this consumption. Even if this analysis was done on a MPEG4 encoder application, the methodology and optimizations done can be extended to any data-flow oriented application.

We have seen how ongoing research focuses on the optimization of the data memory, instruction memory, and novel register file organization. Firsts results on a data-dominated imaging application show that, once this three components have been optimized, they will consume less than 20% of the overall energy and data path and address generation units will then be (again) the future power hungry elements of the processors.

7.2. Future research

Research has the interesting property of generating more research opportunities, and is, therefore, by its own nature a never ending process. This is also the case for this dissertation. The following research ideas where spawned during the research process

but could not be elaborated.

- **Complete DTSE optimization:** In our work, we did not do the complete DTSE optimizations since some of the steps need extra analysis with automated tool support, and manual optimization may not be complete; e.g. global data flow and loop transformations were done in a trial and error procedure. Nevertheless, with more emphasis on these optimizations, DM and RF will be further reduced. Moreover, energy of the DM highly depends on the context, for example, if our encoded frames are send over a wireless network from our embedded device, the knowledge of the multimedia application and the transmission module together will bring us the opportunity to optimize even more since DTSE can be applied between different modules [Cat02]. The energy used by the register file can also be further improved using the techniques explained in [PAM+07] where a novel single-ported register file architecture is presented.

- **Extension to other benchmarks and applications:** The benchmarks used were considered representative of the multimedia domain. Nevertheless, other applications will also be evaluated. From the multimedia domain we could use other encoders or decoders of image and audio, encryption algorithms and 3D gaming applications. Moreover, since the research is valid for applications with the same requirements, we could extend the experiments to communication algorithms (that also have to deal with continuous streams of data) as for example, wireless algorithms.

- **Heterogeneous distributed AGUs:** In this work, we used distributed -but identical- AGUs. In near future, we plan to work on heterogeneous distributed AGUs.

- **Distributed loop buffers with different speeds:** Using a unified loop buffer structure improves the energy since, once arriving in the deepest loops, the instructions can be loaded from the LB instead of the next level of the foreground memory. Making a distributed loop buffer structure will even improve this results considerably. Moreover, the usage of different clock speeds for the different loop buffers will also be evaluated.

- **Deeper loops and loop control:** Nowadays, the proposed AGUs rely on the loop buffer instructions to deal with "if" conditions. This can be improved by extending the work to a more complex loop control scheme with several nested loops and loop control.

- **Multiprocessor AGUs optimization:** Future platforms will have several processors computing data. This will bring us the opportunities to exploit address generation optimizations in multi-processor systems. Before this, we plan adding the task level parallelism DTSE steps to optimize this parallel computation.

- **Address generation code compression instructions:** The idea of using compressed instructions, as in the ARM Thumb processor, will be evaluated.

- **Calculation of bits:** The techniques used during this thesis always work with complete words. The calculation of fewer bits instead of the complete word will be exploited in next future.

- **Merge of index expressions:** When merging different index expressions among several individual address equations, further optimizations can be performed and this possibility will be studied in the future.

- **Real implementation in hardware:** The know-how acquired in the work explained in chapter 2 will permit us to make a real implementation in hardware of a processor with optimized AGUs to obtain real comparisons on a working device. Certainly, the implemented AGUs will slightly differ from the ones presented in this work because they will be adapted to the target processor.

- **Physical location:** At a physical level, the framework does not consider the wiring contribution to the energy, and hence, the AGUs used in this work were always simulated as being located "near" the FUs of the data path with no energy to reach the memory decoders. Due to the size of the memories used in multimedia embedded domain, the address signal has to travel around the memory to reach the memory decoders. In future work, after a real implementation in hardware, we plan to study the impact of this travel and to put the AGUs near the instruction decoders.

7.3. General conclusions

Energy consumption is nowadays a main concern for embedded devices were users demand more and more sophisticated applications while maintaining battery life. To perform the required computation, sophisticated architectures with a high number of computational resources running in parallel are emerging [KP03b, KP03a, KMN+04] and this trend will continue in next future. Future applications will have to deal with enormous memory bandwidth in distributed memories and will have to achieve global trade-offs between the bandwidth required, the number of cycles needed to fetch data (reaction-time), the energy or the area.

Coming embedded systems will have to face different challenges. First of all, embedded systems will have to cope with the increasing requirements of next generation applications: high performance to meet requirements, programmability, since it is faster and easier to implement functionality in software, and power efficiency to extend battery life. Second, future systems will have to deal with deep submicron issues, due to technology scaling, that are becoming visible at higher abstraction levels [FHR99, DeM05].

VLIW architectures seem a good candidate [JdV00, FWW99] to deal with the increasing computation requirements at reasonable energy consumption and in the future, we expect enhancement on those architectures at all abstractions levels, even as a part of

heterogeneous MPSOC platforms.

In this dissertation we have seen how optimizing a VLIW processor-based platform at different levels can bring considerable savings. Nevertheless, further platform-dependent optimizations are possible on the register file and data memory parts. In [PAM+07], Raghavan et al. present a novel register file architecture: the Very Wide Register (VWR), which has single-ported cells and asymmetric interfaces to the memory and to the data path. The novel architecture presented is shown to obtain energy gains of up to a factor 10 with respect to conventional multi-ported register file over the different benchmarks. In [Kri09], the authors show how, after further optimizations on the data memory, very wide register and instruction memory have been completed, these three components will consume less than 20% of the overall energy for a data-dominated imaging application. The remaining components of the processor, the data path and the address generation units, will account for the rest of the energy and, therefore, it is very important to further optimize the energy consumption of the data path and the address generation units.

In this context, the presented work is of increasing importance, since reducing the overall energy consumption and accelerating execution time will become essential to meet the future requirements of multimedia (and in general, data-flow) applications.

In near future, we will see emerging distributed address generation units capable of dealing with these complex applications with extended support for complex condition control. At the architectural level, we expect more efficient and distributed address generation units for VLIW architectures. Good examples are the work on loop accelerators for VLIW architectures [MD04] or the MACGIC processor [MAC06, AMM+06], which is a low power, customizable, reconfigurable, and synthetizable DSP-IP core. We also expect more compiler optimization on address generation techniques, trying to minimize activity and improve spatial and temporal locality. Traditionally, code optimization was performed at a compiler level, and even if at this domain more optimizations are expected, source code optimizations are becoming of increasing importance. These new code optimizations are targeting non-conventional design goals like low power or control flow optimization [PMC02, FV04, PCD+01b].

Even if physical design is out of the scope of this survey, because power consumption concerns not just battery life but also packaging [HGS+06, LHS+07, HSS+08] and cooling costs and reliability, we also expect more improvements at this level. As an example, the work in [MAKB03] shows a new AGU which targets not just speed and energy reduction, but also the relaxation on thermal density. Variability, reliability and interconnect are also cause of major concern and we expect run-time managed techniques to deal with variability issues and the usage of segmented bus interconnect technologies to decrease the power consumption of intra-chip communication.

With the growing importance of compiler and source code optimization research for

embedded processors, research results will turn into commercial products. In the future we also expect a higher importance of VLIW, DSP and ASIP processors, with non-traditional compiler design goals trying to optimize low power and improve retargetability and with optimized address generation units trying to boost address generation at low power.

Appendices

APPENDIX A

Biography

Guillermo Talavera was born on 25th of January, 1976 in Barcelona, Spain. He pursued his undergraduate studies in the French School in Barcelona. He received a Bachelor degree in physics in 1999 and a Master of Engineering (ME) degree in Electronic Engineering in 2001 from the Universitat de Barcelona, Spain. In 2001 he joined the Universitat Autònoma de Barcelona (Barcelona, Spain) as an assistant teacher and in 2003 he received a ME degree in microelectronics (inside the Computer Science Program). From 2003 to 2009 he was continuing as assistant teacher in the university and a PhD candidate of Computer Science, in the Department of Microelectrònica i Sistemes Electrònics in the same university. As PhD candidate he has been in collaboration with the Inter-university Micro-Electronics Center (IMEC vzw), Heverlee, Belgium. He is a member of the IEEE, IEEE Computer Society and ACM. His research interests are in the field of low power embedded systems focusing on microprocessor architectures, compilers and system design automation. He can be contacted at gtalavera@ieee.org.

APPENDIX B

Publications

During this years of PhD, I have been involved in several research projects that lead to publications, moreover, due to my teaching duties I have been implicated in several publications regarding this activity.

Research publications

Journal papers

- G. Talavera, M. Jayapala, J. Carrabina, and F. Catthoor, "Address generation optimization for embedded high-performance processors: A survey", *Journal of Signal Processing Systems for Signal Image and Video Technology (formerly the Journal of VLSI Signal Processing Systems for Signal Image and Video Technology), May 2008 (online) Decembre 2008 (printed version) 2008.*

- G. Talavera, A. Portero, P. Raghavan, M. Jayapala, J. Carrabina, and F. Catthoor, "Power exploration and address generation optimization of multimedia applications on VLIW processors", *Planned for re-submission to the IEEE Transactions on Image Processing.*

- A. Portero, G. Talavera, J. Carrabina, and F. Catthoor, "Methodology for multimedia applications in multiplatform implementation for energy-flexibility space exploration", *Planned for re-submission to the IEEE Transactions on Computers .*

- A. Portero, G. Talavera, J. Carrabina, and F. Catthoor, "Data-dominant application implementation in multi-platform for energy-flexibility space exploration",

Planned for re-submission to the IEEE Transactions on Image Processing.

Conference papers

- A. Lambrecths, T. V. Aa, M. Jayapala, A. Leroy, G. Talavera, A. Shickova, F. Barat, F. Catthoor, D. Verkest, G. Deconinck, H. Corporaal, F. Robert, and J. C. Bordoll, "Design style case study for compute nodes of a heterogeneous NoC platform", *in 25th IEEE Real-Time Systems Symposium (RTSS)*, December 2004.

- G. Talavera, V. Nollet, J.-Y. Mignolet, D. Verkest, S. Vernalde, R. Lauwereins, and J. Carrabina, "Hardware-Software debugging techniques for reconfigurable Systems-on-Chip, *International Conference on Industrial Technology, 2004. IEEE ICIT '04.* vol. 3, Dec. 2004, pp. 1402- 1407 Vol. 3.

- G. Talavera, V. Nollet, J.-Y. Mignolet, D. Verkest, S. Vernalde, R. Lauwereins, and J. Carrabina, "Métodos de depuración HW-SW para sistemas on chip recongurables, *in Jornadas Sobre Computación Recongurable y Aplicaciones (JCRA)*, Barcelona, Spain, Septembre 2004, pp. 251-258.

- A. Lambrechts, P. Raghavan, A. Leroy, G. Talavera, T. Vander Aa, M. Jayapala, F. Catthoor, D. Verkest, G. Deconinck, H. Corporaal, F. Robert, and J. Carrabina, "Power breakdown analysis for a heterogeneous NoC platform running a video application", *in IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP)), 2005.* 16th , July 2005, pp. 179-184.

- A. Portero, G. Talavera, M. Monton, B. Martinez, and J. Carrabina, "NoC system for MPEG-4 SP using heterogeneous tiles" , in *Design of Circuits and Integrated Systems (DCIS)*, San Diego, California, USA. December 2006.

- A. Portero, G. Talavera, M. Monton, B. Martinez, M. Moreno, F. Cathoor, and J. Carrabina, "Energy-aware mpeg-4 single profile in HW-SW multiplatform implementation", *in IEEE International SOC Conference*, Austin, Texas, USA. Sept. 2006, pp. 13-16.

- A. Portero, G. Talavera, M. Monton, B. Martinez, F. Cathoor, and J. Carabina, "Dynamic voltage scaling for power efficient MPEG4-SP implementation", *in Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP).* Washington, DC, USA: IEEE Com-

puter Society, 2006, pp. 257-260.

- A. Portero, G. Talavera, F. Catthoor, and J. Carrabina, "A study of a MPEG-4 codec in a multiprocessor platform", *in IEEE International Symposium on Industrial Electronics (ISIE)*, 2006, vol. 1, July 2006, pp. 661-666.

Teaching publications

- G. Talavera, J. Saiz, and J. Carrabina., "Dispositivos y plataformas para docencia de informática y electrónica", *in Jornadas Sobre Computación Recongurable y Aplicaciones (JCRA)*, Barcelona, Spain, Septembre 2004, pp. 711-717.

- G. Talavera, B. Lorente, M. Monton, B. Martinez, J. Oliver, C. Ferrer, L. Ribas, J. Aguilo, and E. Valderrama, "Nuevas metodologías docentes y autoaprendizaje en la enseñanza técnica universitaria", *in Congreso Internacional de Docencia Universitaria e Innovación (CIDUI)*, Barcelona, Spain, 2006

- B. Lorente, G. Talavera, L. Ribas, and E. Valderrama, "Implantació d'una nova metodologia docent a les pràctiques de fonaments de computadors d'enginyeria informàtica", *in Congreso Internacional de Docencia Universitaria e Innovación (CIDUI)*, Barcelona, Spain, 2006.

- G. Talavera, X. Fitó, B. Lorente, A. Portero, M. Montón, B. Martínez, J. Oliver, C. Ferrer, L. Ribas, J. Aguiló, and E. Valderrama, "Adaptación metodológica a las nuevas directrices del EEES en la enseñanza técnica universitaria", in *Tecnologías Aplicadas a la Enseñanza de la Electrónica (TAEE)*, Madrid, Spain. 2006.

- A. Portero, J. Saiz, G. Talavera, R. Aragonés, M. Rullán, J. Aguiló, and E. Valderrama, "Aplicación del plan piloto en sistemas digitales en ingenier ía informática siguiendo las directivas del EEES", *in Tecnologías Aplicadas a la Enseñanza de la Electrónica. (TAEE)*, Madrid, Spain. 2006.

- G. Talavera, F. X. Fitó, B. Lorente, M. Montón, B. Martínez, C. Ferrer, and E. Valderrama, "Cas pràctic d'adaptació metodològica a les directrius EEES d'una assignatura d'enginyeria informàtica", in *III Jornada de Campus d'Innovació Docent. UAB*, Barcelona. Spain. 20 Setembre de 2006. .

- E. Valderrama, G. Talavera, M. Montón, B. Martínez, J. M. Fernández, and J. Muñoz, "Comparación de dos metodologías docentes utilizadas en los seminarios de fundamentos de computadores", *in XIV Jornadas de Enseñanza Universitaria de la Informática (JENUI)*, 2008.

Bibliography

[A04] A. Arm926ej-s technical reference manual, release d, 26 january 2004. B, D, E 2004. F.

[Abs07] Javed Absar. *Locality Optimization in a Compiler for Embedded Systems.* PhD thesis, IMEC vzw, ESAT, KULeuven, July 2007.

[AC06] Javed Absar and Francky Catthoor. Analysis of scratch pad and cache performance using statistical analysis. Jan 2006.

[ADD+08] A.Leroy, D.Milojevic, D.Verkest, F.Robert, and F.Catthoor. Concepts and implementation of spatial division multiplexing forguaranteed throughput in networks-on-chip. *IEE Transactions on Computers*, 57(9):1182–1195, September 2008.

[AH00] B. Amrutur and M. Horowitz. Speed and power scaling of SRAM's. In *IEEE Journal of Solid-State Circuits*, volume 35, February 2000.

[AJB+04] Tom Vander Aa, Murali Jayapala, Francisco Barat, Geert Deconinck, Rudy Lauwereins, Henk Corporaal, and Francky Catthoor. Instruction buffering exploration for low energy embedded processors. *Journal of Embedded Computing*, 1(3), 2004.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[AKA+02] S. Agarwala, P. Koeppen, T. Anderson, A. Hill, M. Ales, R. Damodaran, L. Nardini, P. Wiley, S. Mullinnix, J. Leach, A. Lell, M. Gill, J. Golston, D. Hoyle, A. Rajagopal, A. Chachad, Agarwala, R. Castille, N. Common, J. Apostol, H. Mahmood, M. Krishnan, Duc Bui, Quang-Dieu An, P. Groves, L. Nguyen, N.S. Nagaraj, and R. Simar. A 600 mhz vliw dsp. In *Solid-State Circuits Conference*, 2002.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, Boston, MA, USA, 2006.

[ALT08] ALTERA. Hardcopy series handbook, 2008.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[AMM+06] C. Arm, J.-M. Masgonty, M. Morgan, C. Piguet, P.-D. Pfister, F. Rampogna, and P. Volet. Low-power quad-mac 170 uw/mhz 1.0 v macgic dsp core. In *ESSCIRC'06:Proceedings of the 32st European Solid-State Circuits Conference*, 2006.

[AOC02] Guido Araujo, Guilherme Ottoni, and Marcelo Cintra. Global array reference allocation. *ACM Trans. Des. Autom. Electron. Syst.*, 7(2):336–357, 2002.

[ARM] ARM. *Artisan Memory Generator: http://www.arm.com/products/physicalip/memory.html.*

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bar05] Francisco Barat. *Crisp: A Scalable VLIW Processor for Low Power Multimedia Systems.* PhD thesis, 2005.

[BB95] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.

[BCA+04] Partha Biswas, Vinay Choudhary, Kubilay Atasu, Laura Pozzi, Paolo Ienne, and Nikil Dutt. Introduction of local memory elements in instruction set extensions. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 729–734, New York, NY, USA, 2004. ACM Press.

[BGN97] J. Bormans, T. Gijbels, and L. Nachtergaele. Initial assesment of video vm 5.0 memory requirements. In *ISO/IEC JTC1/SC29/WG11/MPEG97/M1914, Bristol*, April 1997.

[Bha08] Mukul Bhatnagar. Tms320dm6441 power consumption summary - spraau3. Technical report, Application Report, Texas Instruments,, April 2008.

[BHK+97] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki. Instruction buffering to reduce power in processors

for signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):417–424, December 1997.

[BKM06] Mladen Berekovic, Andreas Kanstein, and Bingfeng Mei. Mapping mpeg video decoders on the adres reconfigurable array processor for next generation multi-mode mobile terminals. In *GSPX*, 2006.

[BLM98] A. Basu, R. Leupers, and P. Marwedel. Register-constrained address computation in dsp programs. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 929–930, Washington, DC, USA, 1998. IEEE Computer Society.

[BM02] L. Benini and G. De Micheli. Networks on chip: a new soc paradigm. *IEEE Computer,*, vol. 35, no. 1, Jan. 2002.

[BMCC03] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor. Layer assignment echniques for low energy in multi-layered memory organisations. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 11070, Washington, DC, USA, 2003. IEEE Computer Society.

[BRK07] D. Baumgartner, P. Rossler, and W. Kubinger. Performance benchmark of dsp and fpga implementations of low-level vision algorithms. *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on,*, June 2007.

[BSL+02a] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[BSL+02b] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[BTC89] M. Bister, Y. Taeymans, and J. Cornelis. Automated segmentation of cardiac mr images. In *Computers in Cardiology 1989, Proceedings.*, pages 215–218, Sep 1989.

[Cad07] Command reference for buildgates synthesis and cadence pks timing analysis product version 5.15 january 2005. Technical report, Cadence, January 2007.

[Cat99] Francky Catthoor. Energy-delay efficient data storage and transfer architectures and methodologies: Current solutions and remaining problems. *Journal of VLSI Signal Processing*, 21:219–231, 1999.

[Cat00] Francky Catthoor. *Unified low-power design flow for data-dominated multi-media and telecom applications.* Springer-Verlag New York. ISBN 0-7923-7947-0, July 2000.

[Cat02] Francky Catthoor. *Data Access and Storage Management for Embedded Programable Processors.* Kluwer Academic Publishers, 2002.

[CBGN98] F. Catthoor, F. Balasa, E. D. Greef, and L. Nachtergaele. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publisher, 1998.

[CD00] Francky Catthoor and Nikil Dutt. Hot topic session: How to solve the current memory access and data transfer bottlenecks: at the processor architecture or at the compiler level? In *Proc of Design Automation and Test in Europe (DATE)*, March 2000.

[CDKO01] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omnes. *Programmable Digital Signal Processors: Architecture, Programming, and Applications.* Marcel Dekker, Inc., New York, USA, 2001.

[CDWD01] Francky Catthoor, Koen Danckaert, Sven Wuytack, and Nikil D. Dutt. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. *IEEE Des. Test*, 18(3):70–82, 2001.

[cep] Cephis web: http://cephis.uab.cat.

[CL98] Wei-Kai Cheng and Youn-Long Lin. Addressing optimization for loop execution targeting dsp with auto-increment/decrement architecture. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 15–20, Washington, DC, USA, 1998. IEEE Computer Society.

[CVB98] F. Catthoor, D. Verkest, and E. Brockmeyer. Proposal for unified system design meta flow in task-level and instruction-level design technology research for multi-media applications. In *Proc of 11th International Symposium on System Synthesis (ISSS)*, pages 89–95, 1998.

[DBD+06] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(3):279–291, 2006.

[DeM05] Hugo DeMan. Ambient intelligence: Giga-scale dreams and nano-scale realities. In *Proc of ISSCC, Keynote Speech*, February 2005.

[DMS] White paper digital media system-on-chip (dmsoc) tms320dm6441 sprs359d 2005-revised march 2008.

[ECWF00] E.Brockmeyer, C.Ghez, W.Baetens, and F.Catthoor. *Unified low-power design flow for data-dominated multi-media and telecom applications.* Kluwer Acad Publ. Boston, 2000.

[Eva95] P.D. Evans, R.J.; Franzon. Energy consumption modeling and optimization for SRAM's. In *IEEE Journal of Solid-State Circuits*, volume 30, pages 571 – 579, May 1995.

[Fal05] Heiko Falk. Control flow driven code hoisting at the source code level. In *ODES'05: Proceedings of The 3rd Workshop on Optimizations for DSP and Embedded Systems*, March 2005.

[Far07] Faraday Technology Corporation. *Faraday UMC 90nm RVT Standard Cell Library*, 2007.

[FFY04] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, 2004.

[FHR99] Michael J. Flynn, Patrick Hung, and Kevin W. Rudd. Deep-submicron microprocessor design issues. *IEEE MICRO*, 19(4), July-August 1999.

[FM03] Heiko Falk and Peter Marwedel. Control flow driven splitting of loop nests at the source code level. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 410–415, Washington, DC, USA, 2003. IEEE Computer Society.

[FM04] Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software.* Springer, 2004.

[For] Document, cynthesizer users guide for cynthesizer version 5.4 from forteds www.forteds.com.

[FSG+98] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, and A.Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Acad Publ. Boston, 1998.

[FV04] Heiko Falk and Manish Verma. Combined data partitioning and loop nest splitting for energy consumption minimization. In *SCOPES'04: Proceedings of The 8th Workshop on Software and Compilers for Embedded Systems*, Septembre 2004.

[FWW99] Jason Fritts, Zhao Wu, and Wayne Wolf. Parallel media processors for the billion transistor era. In *In Proceedings of the International Conference on Parallel Processing*, 1999.

[GBK07] Anup Gangwar, M. Balakrishnan, and Anshul Kumar. Impact of inter-cluster communication mechanisms on ilp in clustered vliw architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(1):1, 2007.

[GD91] D .M. Grant and P. B. Denyer. Address generation for array access based on modulus m couters. In *EDAC '91: In Proceedings of the 2nd ACM/IEEE European Conference on Design Automation (EDAC)*, pages 118–123, 1991.

[GDF89] D. Grant, P.B. Denyer, and I. Finlay. Synthesis of address generators. In *ICCAD-98: IEEE International Conference on Computer-Aided Design*, pages 116–119, 1989.

[GH96] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, pages 1277–1284, 1996.

[GMCG00] Sumit Gupta, Miguel Miranda, Francky Catthoor, and Rajesh Gupta. Analysis of high-level address code transformations for programmable processors. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 9–13, New York, NY, USA, 2000. ACM Press.

[GML94] D. M. Grant, J. V. Meerbergen, and P. Lippens. Optimization of address generator hardware. In *DATE '94: In Proceedings of the 5th ACM/IEEE European Design and Test Conference*, pages 325–329, 1994.

[GMV+00] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor, and D. Verkest. Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 623–632. IEEE Press, 2000.

[GSBC05] Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, and Henk Corporaal. Automatic scenario detection for improved wcet estimation. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 101–104, New York, NY, USA, 2005. ACM Press.

[GVNG94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice Hall, January 1994.

[HCC02] S. Hettiaratchi, P. Cheung, and T. Clarke. Performance-area trade-off of address generators for address decoder-decoupled memory. In *DATE '02:*

Proceedings of the conference on Design, automation and test in Europe, page 902, Washington, DC, USA, 2002. IEEE Computer Society.

[HGS+06] W. Huang, S. Ghosh, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: Thermal modeling for cmos vlsi systems. *IEEE Transactions on Very Large-Scale Integrated Circuits (TVLSI)*, May 2006.

[HMFW07] H.Wang, M.Miranda, F.Catthoor, and W.Dehaene. Synthesis of run-time switchable pareto buffers offering full range fine grained energy/delay trade-offs. In *J. of Signal Processing Systems*, Nov 2007.

[HMWF09] H.Wang, M.Miranda, W.Dehaene, and F.Catthoor. Design and synthesis of pareto buffers offering large range run-time energy/delay trade-off via combined buffer size and supply voltage tuning. In *IEEE Trans. on VLSI Systems*, volume 17, pages 117 – 127, Jan 2009.

[HPon] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffman, 2006 (Fourth Edition).

[HSS+08] W. Huang, K. Sankaranarayanan, K. Skadron, R. J. Ribando, and M. R. Stan. Accurate, pre-rtl temperature-aware processor design using a parameterized, geometric thermal model. *IEEE Transactions on Computers(TCOMP)*, September 2008.

[IBM05] IBM, http://www.research.ibm.com/cell/. *The Cell Microprocessor*, 2005.

[Inc02a] TI Inc. *TMS320C6000 Programmer's Guide*, 2002. http://www.ti.com/.

[Inc02b] TI Inc. *TMS320C64x DSP Two-Level INternal Memory Reference Guide*, 2002. http://www.ti.com/.

[Inc06] TI Inc. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (Rev. C)*, 2006. http://www.ti.com/.

[INT] INTEL. Atom processor.

[ISO01] ISO. International standard iso/iec 14496-2 2nd edition 2001-12-01 it (coding of audio-visual objects) part 2. Technical report, ISO, 2001.

[ITR07] ITRS. International techology roadmap for semiconductors 2007 edition: Interconnect. Technical report, ITRS, http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Interconnect.pdf, 2007.

[Jay05a] Murali Jayapala. *Low Energy Instruction Memory Organization.* PhD thesis, KULeuven, ESAT/ELECTA, 2005. To Appear.

[Jay05b] Murali Jayapala. *Low Energy Instruction Memory Organization for Embedded Processors.* PhD thesis, KULeuven, ESAT/ELECTA, 2005.

[JBA+05] Murali Jayapala, Francisco Barat, Tom Vander Aa, Francky Catthoor, Henk Corporaal, and Geert Deconinck. Clustered loop buffer organization for low energy VLIW embedded processors. *IEEE Transactions on Computers*, 54(6):672–683, June 2005.

[JdV00] Margarida F. Jacome and Gustavo de Veciana. Design challenges for new application-specific processors. *IEEE Des. Test*, 17(2):40–50, 2000.

[J.G08] J.Guo. *Analysis and Optimization of intra-tile Communication Network.* PhD thesis, ESAT/EE Dept., K.U.Leuven, August 2008.

[JNH06] M. Joshi, NS. Nagaraj, and A. Hill. Impact of interconnect scaling and process variations on performance. In *Proceedings of CMOS Emerging Technologies*, 2006.

[KA02] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[KKC+04] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. In *IEEE Trans on VLSI*, pages 281–287, March 2004.

[KMN+04] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A multi-level computing architecture for embedded multimedia applications. In *Proceedings of the IEEE Micro*, pages 55–66, 2004.

[Koe99] Rob Koenen. Mpeg-4 - multimedia for our time. *IEEE Spectrum*, Vol. 36, No. 2:26–33, February 1999.

[KP03a] Christoforos E. Kozyrakis and David A. Patterson. Scalable vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, 2003.

[KP03b] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 399–409, New York, NY, USA, 2003. ACM Press.

[Kri09] A. Kritikakou. Low-cost low-energy embedded processors for on-line biotechnology monitoring applications. Master's thesis, University of Patras (Greece) - IMEC (Belgium), 2009.

[Kuh04] Peter Kuhn. *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Estimation.* Kluwer Academic Publishers, 2004.

[LAJ+04] Andy Lambrechts, Tom Vander Aa, Murali Jayapala, Antony Leroy, Guillermo Talavera, Adriana Shickova, Francisco Barat, Francky Catthoor, Diederik Verkest, Geert Deconinck, Henk Coporaal, F. Robert,

and J. C. Bordoll. Design style case study for compute nodes of a heterogenenous noc platform. In *25th IEEE Real-Time Systems Symposium (RTSS)*, December 2004.

[LD98] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 3–8, Washington, DC, USA, 1998. IEEE Computer Society.

[Leu00a] Rainer Leupers. Code generation for embedded processors. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 173–178, Washington, DC, USA, 2000. IEEE Computer Society.

[Leu00b] Rainer Leupers. *Code Optimization Techniques for Embedded Processors Methods, Algorithms, and Tools.* Kluwer, 2000.

[LHS+07] Z. Lu, W. Huang, K. Skadron, J. Lach, and M. R. Stan. Interconnect lifetime prediction with temporal and spatial temperature gradients for reliability-aware design and runtime management: Modeling and applications. *IEEE Transactions on Very Large-Scale Integrated Circuits (TVLSI)*, February 2007.

[LM96] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *ICCAD*, pages 109–112, 1996.

[LMA99] Lea Hwang Lee, William Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proc of International Symposium on Low Power Electronic Design (ISLPED)*, August 1999.

[LMdWV91] P. Lippens, J. V. Meerbergan, A. V. der Werf, and W. Verhaegh. Phideo: a silicon compiler for high speed algorithms. In *In Proceedings of the European Conference on Design Automation*, pages 436–441, 1991.

[LPJ96] Clifford Liem, Pierre Paulin, and Ahmed Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 597–600, New York, NY, USA, 1996. ACM Press.

[LPJ97] C. Liem, P. Paulin, and A. Jerraya. Compilation methods for the address calculation units of embedded processor systems. In *In Proceedings of the Design Automation for Embedded Systems*, pages 61–77, Netherlands, 1997. Springer.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual*

ACM/IEEE international symposium on Microarchitecture, pages 330–335. IEEE Computer Society, 1997.

[LRJ+09] A. Lambrechts, P. Raghavan, M. Jayapala, Bingfeng Mei, F. Catthoor, and D. Verkest. Interconnect exploration for energy versus performance tradeoffs for coarse grained reconfigurable architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(1):151–155, Jan. 2009.

[LZSS04] Meilin Liu, Qingfeng Zhuge, Zili Shao, and Edwin H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 190–201, New York, NY, USA, 2004. ACM Press.

[MAC06] Low-power digital signal processing (macgic dsp) http://www.macgic.com, 2006.

[MAKB03] Sanu Mathew, Mark Anders, Ram K. Krishnamurthy, and Shekhar Borkar. A 4-ghz 130-nm address generation unit with 32-bit sparse-tree adder core. *IEEE Journal of Solid-State Circuits*, 38(5), may 2003.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers (Springer), Norwell, MA, USA, 2003.

[mat] Matlabworks website: http://www.matlabworks.com.

[MCJdM96] Miguel Miranda, Francky Catthoor, Martin Janssen, and Hugo de Man. Adopt: Efficient hardware address generation in distributed memory architectures. *isss*, 00:20, 1996.

[MCJM98] Miguel A. Miranda, Francky Catthoor, Martin Janssen, and Hugo J. De Man. High-level address optimization and synthesis techniques for data-transfer-intensive applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 6(4):677–686, 1998.

[MCM94] M. Miranda, F. Catthoor, and H. De Man. Address equation multiplexing for realtime signal processing applications. In *VLSI Signal Processing VII*, pages 188–197, La Jolla California, New York, 1994.

[MD04] Binu Mathew and Al Davis. A loop accelerator for low power embedded vliw processors. In *Proc of CODES and ISSS*, Stockholm, Sweden, September 2004.

[MDB+09] M.Li, D.Novo, B.Bougard, T.Carlson, L.Van der Perre, and F.Catthoor. Generic multi-phase software pipelined partial fft on instruction level parallel architectures. In *(accepted for) IEEE Trans. on Signal Processing*, 2009.

[MHB08] Marius Monton Macian, Borja Martinez Huerta, and Jordi Carrabina Bordoll. Síntesis de canales tlm para procesador nios-ii. In *VIII Jornadas de Computación Reconfigurable y Aplicaciones (JCRA)*, Madrid, Spain, Septiembre 2008.

[MKCdM97] M. Miranda, M. Kaspar, F. Catthoor, and H. de Man. Architectural exploration and optimization for counter based hardware address generation. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 293, Washington, DC, USA, 1997. IEEE Computer Society.

[MNCM97] D. Moolenaar, L. Nachtergaele, F. Catthoor, and H. De Man. System-level power exploration for MPEG-2 decoder on embedded cores : a systematic approach. *Journal of VLSI Signal Processing Systems*, pages 395–404, 1997.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[MWP+01] Pol Marchal, Chun Wong, Aggeliki Prayati, Nathalie Cossement, Francky Catthoor, Rudy Lauwereins, Diederik Verkest, and Hugo De Man. Dynamic memory oriented transformations in the mpeg4 im1-player on a low power platform. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 40–50, London, UK, 2001. Springer-Verlag.

[Nio] Niosii processor reference handbook, altera corp. v.7.2 october 2007.

[oCS08] TMS320DM6446 Digital Media System on Chip SPRS283F. Technical report, Texas Instruments www.ti.com, December 2005 revised March 2008.

[OdBGB+03] P. Op de Beeck, C. Ghez, E. Brockmeyer, M. Miranda, F. Catthoor, and G. Deconinck. Background data organisation for the low-power implementation in real-time of a digital audio broadcast receiver on a simd processor. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 11144, Washington, DC, USA, 2003. IEEE Computer Society.

[OK06] Vojin G. Oklobdzija and Ram K. Krishnamurthy. *High-Performance Energy-Efficient Microprocessor Design (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[OKM+02] John D. Owens, Ujval J. Kapasi, Peter Mattson, Brian Towles, Ben Serebrin, Scott Rixner, and William J. Dally. Media processing applications on the Imagine stream processor. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, 2002.

[PAM+07] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, and H. Corporaal. Very wide register: An asymmetric register file organization for low power embedded processors. In *"DATE '07: Proceedings of the conference on Design*, 2007.

[Pap06] Antonis Papanikolaou. *Application-driven software configuration of communication networks and memory organizations.* PhD thesis, CS Dept., U.Gent, Belgium, December 2006.

[PBV+05] Martin Palkovic, Erik Brockmeyer, Peter Vanbroekhoven, Henk Corporaal, and Francky Catthoor. Systematic preprocessing of data dependent constructs for embedded systems. In *Proceedings of PATMOS*, pages 89–98, 2005.

[PCC05] Martin Palkovic, Henk Corporaal, and Francky Catthoor. Global memory optimisation for embedded systems allowed by code duplication. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 72–79, New York, NY, USA, 2005. ACM Press.

[PCD+01a] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, April 2001.

[PCD+01b] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, 2001.

[PE02] Fernando C. Pereira and Touradj Ebrahimi. *The MPEG-4 Book.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[Pea06] Dac C. Pham and et al. Overview of the architecture, circuit design, and physical implementation of first-generation cell processor. *IEEE Journal of SSC*, Jan2006.

[Phi04] Philips PDSL, http://www.coolfluxdsp.com. *CF6 CoolFlux DSP*, 2004.

[PMC02] M. Palkovic, M. Miranda, and F. Catthoor. Systematic power-performance trade-off in mpeg-4 by means of selective function inlining steered by address optimization opportunities. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 1072, Washington, DC, USA, 2002. IEEE Computer Society.

[PMCV01] Martin Palkovic, Miguel Miranda, Francky Catthoor, and Diederik Verkest. *System Design Automation -Fundamentals, Principles, Methods, Examples*, chapter high level condition expression transformations for desing exploration, pages 56–64. Kluwer Academic Publishers, Boston, USA, March 2001.

[PMD+02] M. Palkovic, M. Miranda, K. Denolf, P. Vos, and F. Catthoor. Systematic address and control code transformations for performance optimisation of a mpeg-4 video decoder. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 547, Washington, DC, USA, 2002. IEEE Computer Society.

[PND98a] Preeti Ranjan Panda, Alexandru Nicolau, and Nikil Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[PND98b] Preeti Ranjan Panda, Alexandru Nicolau, and Nikil Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Por09] Antonio Portero. *Design Space Exploration of heterogeneous SoC Platforms for a Data-Dominant Application.* PhD thesis, E.T.S.E. - Universitat Autònoma de Barcelona, 2009.

[Pow] Powerplay early power estimator user guide for stratix ii, stratix ii gx, & hardcopy ii document version: 1.2.

[RAdSJ+00] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., Danny Patel, and Shad Roundy. Hoc ultra-low power wireless networking. *Computer*, 33,Issue 7,:42 – 48, July 2000.

[Rag09] Praveen Raghavan. *Low Energy Architecture Extensions for Embedded Processors.* PhD thesis, IMEC vzw, ESAT, KULeuven, June 2009.

[RC06] Praveen Raghavan and Francky Catthoor. Ultra low power asip (application-domain specific instruction-set processor) micro-computer. *EU Patent Filed EP 1 701 250 A1*, September 2006.

[RDK+00] Scott Rixner, William J. Dally, Brucek Khailany, Peter R. Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *HPCA*, pages 375–386, January 2000.

[RKHK02] J. Ramanujam, Satish Krishnamurthy, Jinpyo Hong, and Mahmut Kandemir. Address code and arithmetic optimizations for embedded systems. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 619, Washington, DC, USA, 2002. IEEE Computer Society.

[RLA+08] P Raghavan, A Lambrechts, J Absar, , M Jayapala, and F Catthoor. COFFEE: COmpiler Framework For Energy-aware Expoloration. In *Proc of HiPEAC*, Jan 2008.

[RLJ+] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest. Distributed loop controller for multi-threading in uni-threaded ilp architectures. *IEEE Transactions on Computers*.

[RWT05] RWTH Aachen – University of Technology, http://www.eecs.rwth-aachen.de/dpg/info.html. *DPG User Manual Version 2.8*, October 2005.

[SHmWH01] John W. Sias, Hillery C. Hunter, and Wen mei W. Hwu. Enhancing loop buffering of media and telecommunications applications using low-overhead predication. In *Proc of 34th Annual International Symposium on Microarchitecture (MICRO)*, December 2001.

[sia05] Semiconductor industry association, international technology roadmap for semiconductors: Design http://www.itrs.net/links/2005itrs/home2005.htm, 2005.

[SK99] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron ii: a global wiring paradigm. In *ISPD '99: Proceedings of the 1999 international symposium on Physical design*, pages 193–200, New York, NY, USA, 1999. ACM.

[SLD97] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp architectures. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 287–292, New York, NY, USA, 1997. ACM Press.

[SMC00] Rainer Schaffer, Renate Merker, and Francky Catthoor. Combining background memory management and regular array co-partitioning, illustrated on a full motion estimation kernel. In *VLSID '00: Proceedings of the 13th International Conference on VLSI Design*, page 104, Washington, DC, USA, 2000. IEEE Computer Society.

[SMC02a] Rainer Schaffer, Renate Merker, and Francky Catthoor. Exploitation of subword parallelism on the example of the staf algorithm. In *5th Workshop on System Design Automation*, pages 111–118, Pirna, Germany, April 2002.

[SMC02b] Rainer Schaffer, Renate Merker, and Francky Catthoor. Systematic design of programs with subword parallelism'. In *Parallel Computing in Electrical Engineering, International Conference on*, pages 393–398, Warsaw, Poland, September 2002.

[SMC03] Rainer Schaffer, Renate Merker, and Francky Catthoor. Causality constraints for processor architectures with subword parallelism. In *Digital Systems Design, Euromicro Symposium on*, pages 82–89, Antalya, Turkey, September 2003.

[ST98] Herman Schmit and Donald E. Thomas. Address generation for memories containing multiple arrays. In *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 17, 1998.

[Str] Stratix ii dsp development board reference manual altera corporation, august 2006.

[SWLM02] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. *Design Automation and Test in Europe (DATE)*, pages 409–414, March 2002.

[sys] Open systemc iniciative website : http://www.systemc.org.

[Tanon] Ittetsu Taniguchi. *Systematic Architecture Exploration Method for Low Energy and High Performance Reconfigurable Processors.* PhD thesis, Osaka University, Osaka, Japan, January 2009 (in preparation).

[Tex00] Texas Instruments, Inc, http://www.ti.com. *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000.

[TI-] Ti tms320c67 dsps.

[TIO] Texas instruments, "tms320c6000 cpu and instruction set reference guide", spru189f, october 2000.

[TJCC08] Guillermo Talavera, Murali Jayapala, Jordi Carrabina, and Francky Catthoor. Address generation optimization for embedded high-performance processors: A survey. *Journal of Signal Processing Systems for Signal Image and Video Technology (formerly the Journal of VLSI Signal Processing Systems for Signal Image and Video Technology)*, May 2008 (online) Decembre 2008 (printed version) 2008.

[tri99] *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism.* http://www.trimaran.org, 1999.

[TSU+] Ittetsu Taniguchi, Keishi Sakanushi, Kyoko Ueda, Yoshinori Takeuchi, and Masaharu Imai. Dynamic reconfigurable architecture exploration based on parameterized reconfigurable processor model. *In Giovanni De Micheli, Salvador Mir, and Ricardo Reis, editors, VLSI-SoC: Research Trends in VLSI and Systems on Chip. Springer Boston, 2007.*, 249:357–376.

[TSU+07] Ittetsu Taniguchi, Keishi Sakanushi, Kyoko Ueda, Yoshinori Takeuchi, and Masaharu Imai. Dynamic reconfigurable architecture exploration based on parameterized reconfigurable processor model. In Giovanni De Micheli, Salvador Mir, and Ricardo Reis, editors, *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, volume 249, pages 357–376. Springer Boston, 2007.

[Tur99] Jim Turley. Embedded processors by the numbers. *Embedded Systems Programming*, 12(5), 1999.

[UWW+99] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Burns, and Vincent Cao. Effective exploitation of a zero overhead loop buffer. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 10–19, New York, NY, USA, 1999. ACM Press.

[VA05] Tom Vander Aa. *Low Energy Instruction Memory Exploration.* PhD thesis, KULeuven, ESAT/ELECTA, 2005. To Appear.

[VBR+93] Jan Vanhoof, Ivo Bolsens, Karl Van Rompaey, Gert Goossens, and Hugo De Man. *High-Level Synthesis for Real-Time Digital Signal Processing.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[vdWVD+05] Jan-Willem van de Waerdt, Stamatis Vassiliadis, Sanjeev Das, Sebastian Mirolo, Chris Yen, Bill Zhong, Carlos Basto, Jean-Paul van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, Pedro Rodriguez, and Hans van Antwerpen. The tm3270 media-processor. In *MICRO '05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 331–342, Washington, DC, USA, 2005. IEEE Computer Society.

[VJC+06] T. VanderAa, M. Jayapala, H. Corporaal, F. Catthoor, and G.Deconinck. Instruction transfer and storage exploration for low energy vliws. pages 292–297, Oct.2006.

[WCNM96] Sven Wuytack, Francky Catthoor, Lode Nachtergaele, and Hugo De Man. Power exploration for data dominated video applications. In *ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design*, pages 359–364, Piscataway, NJ, USA, 1996. IEEE Press.

[Wes99] B. Wess. Minimization of data access computation overhead in dsp programs. In *In Proceedings of Design Automation for Embedded Systems*, pages 167–185, 1999.

[WGN01] O. Wiess, M. Gansen, and T.G. Noll. A flexible datapath generator for physical oriented design. In *Proc of ESSCIRC*, pages 408–411, Sep 2001.

[WZG+01] Marlene Wan, Hui Zhang, Varghese George, Martin Benes, Arthur Abnous, Vandana Prabhu, and Jan M. Rabaey. Design methodology for a low-energy reconfigurable single-chip dsp system. *Journal of VLSI Signal Processing*, 28(1), Jan. 2001.

[XSLS05] Chun Xue, Zili Shao, Meilin Liu, and Edwin H.-M. Sha. Iterational retiming: maximize iteration-level parallelism for nested loops. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 309–314, New York, NY, USA, 2005. ACM Press.

[YM04] P. Yu and T. Mitra. Scalable instructions identification for instruction-set extensible processors. In *Proc of CASES*, September 2004.

[ZPD+07] Z.Ma, P.Marchal, D.Scarpazza, P.Yang, C.Wong, I.Gomez, S.Himpe, C.Ykman, and F.Catthoor. *Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogeneous platforms*. Spring 2007.

"Si non e vero e ben trovato"

"Carmiña... oye, dejo esto, ¿eh?

Es muy estresante.

Interesante no, mujer, ¡estresante!"

Pazos. Airbag.