Enhanced applicability of loop transformations

Enhanced Applicability of Loop Transformations

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus,
prof.dr.ir. C.J. van Duijn, voor een commissie
aangewezen door het College voor Promoties in
het openbaar te verdedigen op
maandag 24 september 2007 om 16.00 uur

door

Martin Palkovič

geboren te Bratislava, Slowakije

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. H. Corporaal
en
prof.dr.ir. F. Catthoor

This work was carried out at IMEC.

Cover design: Martin Palkovič

Printed by: PROCOPIA nv

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Palkovič, Martin

Enhanced applicability of loop transformations / by Martin Palkovič. – Eindhoven : Technische Universiteit Eindhoven, 2007.
Proefschrift. – ISBN 978-90-386-1584-4
NUR 959
Trefw.: ingebedde systemen / dataopslag / compilers ; optimalisering / real-time computers ; toepassingen.
Subject headings: embedded systems / storage management / optimising compilers / Pareto optimisation.

To my parents, my sisters and Jana

Acknowledgements

La reconnaissance est la mémoire du coeur.
Jean Baptiste Massieu
(1743-1818)

When reaching a milestone in your life, which a PhD certainly is, you should look back in time and acknowledge people who have been important to you and have contributed to your personal and professional growth. Maybe returning to your childhood is a little bit too far, but perhaps somewhere, in 1986, at elementary school, trying to write my first programs in BASIC using the PMD-85 computer (MHB 8080A 2.048MHz CPU, 48kB RAM and 4kB ROM), there was the beginning of what I am trying to finish now. Two years after my first programs, my parents bought me my first computer, an ATARI 800 XE, so I could continue my programming experiments at home. I realize now that at that time it cost a few times their monthly salaries, and maybe they do not yet realize how grateful for that I was and still am. Next to that, I also had excellent math and physics teachers at elementary school, who encouraged my further interest in those subjects and technology in general.

Later at comprehensive school, I specialized in mathematics, physics and informatics. I also met great teachers there who supported me in my interests and enriched my professional growth. I would like to express my gratitude to all of them. In the early 90's, when the Coordinating Committee for Multilateral Export Controls (CoCom) became obsolete, the first computers based on the Intel 80286 and 80386 appeared in Czechoslovakia (with some delay, because at that time the Intel 80486 was already available in the U.S.). I believe I still have still a leaflet from the shop with the price for a 286 machine at home. It was more than 120.000 Czechoslovak koruna's at that time, which is roughly 5000 USD when using exchange rates for the Czechoslovak koruna of 1990. Fortunately, those machines were available at the Technology club (T-club) of the Faculty of Electrical Engineering at the Slovak University of Technology, which I intended to became my "alma mater" anyhow. And (again thanks to my parents) I chose a cheaper but still relatively good Amiga 500 for home usage.

I believe I experienced the best times of my student life when I was enrolled at the Faculty of Electrical Engineering and Information Technology of the Slovak Univer-

sity of Technology. It was a pleasure for me to study all those interesting subjects of technology and I believe the teachers there did a very good job, for which I am really grateful. Here, I also have to thank my father, who "forced" me to study International Economics in parallel at the University of Economics. These studies gave an extra dimension to my vision of the world. After the third year at university, I felt that I would like to apply the theory I had learned in a practical working environment and gain some experience abroad. Thanks to the International Association for the Exchange for Students for Technical Experience (IAESTE) I spent two wonderful summer months in Aachen working in the motivating environment of the Zentrallabor für Elektronik in Forchungszentrum Jülich, Germany. I am thankful to all my colleagues there for all the interesting discussions. Encouraged by the positive experience, after the fourth year of my studies I also looked for a summer internship abroad. From Prof. Daniel Donoval I heard about the Inter-University Micro-Electronics Center (IMEC), where some of the PhD students from our Department of Microelectronics were already doing their research. Thus, I asked him if it would be possible to visit IMEC as an undergraduate student for a summer internship, with the potential to return there the following year for a master's thesis if IMEC was interested. The answer from IMEC was positive and so I came to Leuven, Belgium, in 1999.

After doing the summer internship at IMEC, I returned one year later for my master's thesis. I have to express my gratitude to Miguel Miranda and Arnout Vandecappelle who made my first days and years at IMEC much easier. Also I have to thank Miguel Miranda for the daily guidance during my master's thesis and Prof. Francky Catthoor for all the discussions and feedback during those days.

After finishing my master's thesis I got the opportunity to stay at IMEC as an employee working on the Data Transfer and Storage Exploration (DTSE) methodology, in particular the Global Loop Transformations (GLT) step. Even though the Geometrical Model (GM) used by this step was powerful and the transformations were easily expressible I believed it had severe limitations. This belief became even stronger after my return from my three week visit to Tanguy Risset's Compsys group at ENS Lyon, France, in 2003. Thus, my main task at IMEC was to discover how to apply the techniques based on GM to realistic applications. The result of this research is the dissertation you are now reading. Of course, it would not have happened without the help and support of my colleagues at IMEC and cooperating universities, the responsible persons at TU/e and IMEC who allowed me to became a PhD candidate and the support of my two promotors, Prof. Henk Corporaal and Prof. Francky Catthoor.

At IMEC I have many people to thank and even those who are not mentioned by name know they deserve my gratitude. Erik Brockmeyer, Rogier Baert, Arnout Vandecappelle, Vincent Nollet, Eddy De Greef, Sven Wuytack, and Robo Paško, are just a few who helped me in my daily work. Also Michel Eyckmans for being my activity lead. It has been a pleasure. I would also like to mention Diederik Verkest, Johan Vounckx, and Wilfried Verachtert for making this research possible.

At cooperating Universities I have to thank Sven Verdoolaege (at that time from the Katholieke Universiteit of Leuven, now at the Universiteit of Leiden, the Netherlands) for very interesting discussions about GLT and GM and all the help with his toolchain; Per Gunnar Kjeldsberg and his PhD student Qubo Hu from the Nor-

wegian University of Science and Technology in Trondheim, Norway and Benny Thörnberg from the Mid-Sweden University, Sundsvall, Sweden, for very intensive and fruitful cooperation in the field of high-level memory estimators; Valentin Stefan Gheorghita from the Technische Universiteit Eindhoven for enthusiasm and cooperation in scenario research and many others from the Computer Science Department at the Katholieke Universiteit of Leuven and the Electronics and Information System Department at Ghent University.

I would like to express my appreciation to Erik Brockmeyer for sharing an office with me for more than 6 years and all the discussions, professional or non-professional, we had during those years.

Special thanks go to my two promotors, Prof. Henk Corporaal and Prof. Francky Catthoor for all the discussions, support and feedback during my research and during the writing of this dissertation. Also special thanks to Prof. Koen De Bosschere, Twan Basten and Albert Cohen for being members of my reading committee and for the final feedback, which certainly improved the quality of this text.

Special thanks also to Sven Verdoolaege for reading the whole text, for his detailed feedback and for all the interesting discussions during my research; Eddy De Greef for reading the conclusions section, its translation to Dutch, and for his patience when explaining the ATOMIUM library internals to me; Andrew Fort for proof-reading some parts of this dissertation and Tom Ashby for proof-reading the summary and these acknowledgments.

Life is not only work, and thus I would like to thank people who were part of my social life here in Leuven; Théodore Marescaux, for being not only a good colleague but also a good friend hanging around since 1999; Radim Cmar, Miro Čupák, Petr Dobrovolný, Štefan Kubiček, Robo Paško, and Rišo Stahl for all the coffee breaks and professional help at work; The previous, Soňa Pallayová and their families for creating a small home for me here in Leuven, sharing time outside work with me and always being helpful when I needed it; Prof. Francky Catthoor for all the excellent prepared bike and cross-country skiing trips and for showing me that Belgium can indeed be hilly and that during winter you can find 60 cm of snow if you go to the right places.

Finally, I would like to thank again (and again) my parents for much more than is written in this text, for their love, understanding and all their support during my student years. The same gratitude goes also to my two great sisters, Helena and Soňa, and my beloved Jana. Without them, life would not be as beautiful and fun as it is.

Martin Palkovič
Leuven, July 2007.

Enhanced Applicability of Loop Transformations

Summary

Data transfers and storage of large arrays in background memories are dominating contributors to the chip area and power consumption of all modern multimedia embedded systems. Modern high-level memory optimizations contribute to the cost-efficient realization of these systems. In these optimizations an important step involves loop transformations across the global program scope. These transformations can be performed on a geometrical model extracted from the program. The geometrical model captures all the memory access dependencies in the program. Loop transformations in general modify the order in which the iterations and statements within a loop body are executed. This could be beneficial for different reasons such as enabling more parallelism or improving locality of the accessed data.

Due to the limitations of current geometrical models, the applicability of the transformations is limited. In this dissertation, we propose several applicability-enhancing techniques for loop transformations. First, hierarchical rewriting separates and encapsulates the details of the application into functions, reducing the complexity of the problem by hiding undesired constructs.

Second, we instantiate and extend the scenario technique for loop transformations. A scenario is defined as a selected set of paths in the program which we choose to exploit in the same way. A careful exploitation of scenario information, similar to inlining, path predication or hyperblock creation, can significantly enlarge the exploration space for optimizations. Unlike path predication or inlining, however it can work across several conditional branches, merge several condition bodies, and still control the exponential code explosion. Applying scenarios introduces several tradeoffs. The most obvious is that of code duplication vs. more optimizations (similar to tail duplication during hyperblock creation) when additional loop transformations are enabled by scenario usage.

The exploration space of the scenario creation technique grows exponentially with the number of paths in the control-flow graph of the application. In this dissertation

we propose several heuristics for the scenario creation technique. These heuristics have different time requirements and accuracy limitations. Thus the designer has the possibility to choose the heuristic based on his time and accuracy constraints and the size of the problem.

In addition, most current (global scope) loop optimizations target the best solution for locality. In this dissertation we show that targeting the best solution for locality is not necessarily optimal for a particular platform instance, and that trade-offs should be involved during loop transformations when the platform is unknown. This dissertation provides real-life examples of trade-offs during loop transformations and gives an overview of the joint research work in high-lever estimators for loop transformations which will make the loop transformations trade-off oriented.

Contents

CHAPTER 1

Introduction

Nil tam difficile est quin quaerendo investigari possiet.
Publius Afer Terentius
(185BC-159BC)

Nowadays, electronic devices are ubiquitous in the world around us. An electronic device can be found nearly in any gadget we take in our hands these days. Handheld devices such as mobile phones, smart phones, handheld television, personal digital assistants (PDA)s and global positioning system (GPS) devices are working in urban areas, on the countryside, when traveling by car at 120 km/h or walking slowly 4km/h. The capturing devices such as digital cameras support different resolutions and can capture and encode both, slowly and rapidly moving objects.

These devices have to work independently on the environment, e.g., urban areas or the countryside, 120km/h or 4km/h, in which they are present. This results in the existence of a lot of different standards which are suited for different environments and have to be integrated into one device. Embedded programmable processors, whose software can be changed, aim to be cost effective and flexible solution for current challenges compared to traditional hardware-only approach using Application Specific Integrated Circuit (ASIC)s. They should be also much more power and performance effective for a given application domain compared to general purpose processors.

The specification of an application for embedded programmable processor is usually written in a high-level programming language. In the past the application was directly hand-written in the machine language or translated directly from a high-level language using a standard compiler. Nowadays, the strict development time constraints due to decreased time to market does not allow to hand-write the machine code. Unfortunately, current compilers do not produce machine code with the same quality as hand-written machine language.

To bridge the gap between high-quality hand-written machine language and code produced by the compiler source-to-source transformations exist. They perform high-level optimizations improving the quality of the source code resulting also in more power and performance effective translated code.

An important part of these high-level optimizations are Loop Transformations (LT). However, these transformations are not always applicable in the context of modern embedded systems with a lot of control-flow and trade-offs involved. This dissertation aims at solving the applicability problem of LT. It proposes three independent techniques enhancing the applicability of LT in the context of modern embedded systems. The synergetic effect can be achieved by combining these techniques.

This chapter provides an introduction to the application domain targeted in this dissertation, the embedded system virtual platform used, and challenges in current source-to-source transformation techniques. Section 1.1 presents the application domain description. Section 1.2 discusses the embedded system platform description focusing on the memory hierarchy subsystem. Section 1.3 introduces the design flow used in this dissertation. Then it defines the problem statement and objective and the contributions of this dissertation, all supported by an illustrative example. At the end of this chapter a thesis overview is provided.

1.1 Application domain description

The main application target domain of this dissertation consists of embedded data-dominated applications, i.e., applications which deal with large amounts of data and data transfers. We can subdivide this target domain into three classes: multimedia applications, front-end telecommunication applications and network component applications [28]. In this dissertation we focus on multimedia applications which have the following main features [28]:

- Deep and large loop nests. Multimedia applications typically contain many deep and large loop nests for processing the multidimensional data. Due to the rectangular shape of images and video streams, the loop bounds are often constant or manifest, i.e., only dependent on the enclosing loop iterators. The loops are typically quite irregularly nested though.

- Mostly manifest affine loop bounds, conditions and array indices. Manifest means that they are a function of the enclosing loop iterators only. Affine means that this function is a linear combination of these loop iterators and a constant. However, more and more recent multimedia applications also contain data dependent conditions and indexing, and *while* loops. This is not reflected in the state-of-the-art design methodologies for multimedia applications and it is one of the major topics of this dissertation.

- Multidimensional arrays. Multimedia applications mainly work on multidimensional arrays of data. Also large one-dimensional arrays are often encountered. Often, the multidimensional arrays are already linearized to a one-dimensional array.

- Statically allocated data. Most data is statically allocated. Its exact storage location can be optimized at compile time. Also this is starting to change, especially in emerging newer multimedia algorithms. These new algorithms have, besides of the data that is statically allocated, i.e., the data which is analyzable at design time, also the data that is dynamically allocated at run time. This results in the mixed case where both, statically and dynamically allocated data are within a single application. However, the mixed case is a topic of future research and it is not covered in this dissertation. We only deal with the pieces of the code that have fully statically allocated data.

- Temporary data. Most of the data is temporary, only alive during one or at most a few iterations of the algorithm. When the data is not alive anymore, the memory space where it was located can be reused and it can be overwritten by other data.

- Real-time behavior with a form of periodic behavior. Multimedia applications typically have to satisfy soft real-time constraints. They encode or decode the input image stored in multidimensional arrays in the time loop. A time loop is a loop within which a video or audio frame is encoded or decoded. The time loop starts when the first frame is encoded/decoded and ends when the last frame is encoded/decoded. Its loop trip count depends on the number of frames in the encoded/decoded sequence. This designates the application to have a form of periodic behavior.

To demonstrate the described features on real-life applications, in the sequel we briefly describe the structure and code characteristics w.r.t. the list above of two multimedia algorithms, an MPEG-1 Layer 3 (MP3) audio decoder and a Quadtree Structured Difference Pulse Code Modulation (QSDPCM) video encoder. These applications are used in most of our experiments and also give a good overview of the typical applications in the multimedia domain representing both, the audio and the video part. The functional description of these applications is shown in Appendix A.

The MP3 audio decoder application has real-time periodic behavior. It receives a decoded audio frame and decodes it until no frame is present at the input. The frame is decoded in several steps. Each step is composed of a nested loop (or several non-perfectly nested loops) with maximum loop nest depth of 5. The iteration count for the loops is in hundreds of iterations. The loop bounds and array indices are mostly manifest and affine. Typical for this application are a lot of non-manifest *outermost data-dependent conditions*. We will provide a technique to deal with them in this dissertation in Chapter 5. The application uses three-dimensional arrays and a lot of temporary data storing the frame between the two subsequent processing steps of the decoder.

The QSDPCM video encoder reads input video frames and encodes them using the information from the previous video frame. It also has real-time periodic behavior and encodes the frame in several steps. Each step is again composed of a nested loop (or several non-perfectly nested loops) with a maximum loop nest depth of 8. The iteration count depends on the size of the frame, but for a frame it is again in the hundreds to thousands of iterations. The loop bounds and array indices are mostly manifest and affine. Typical for this application are a lot of non-manifest

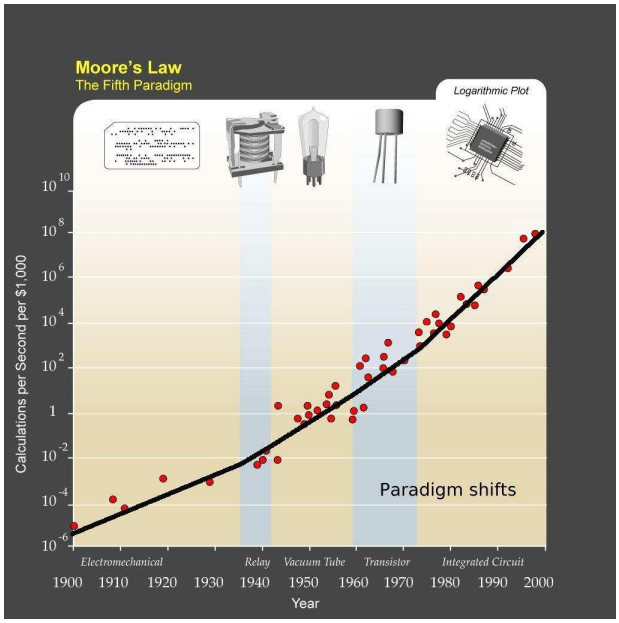


Figure 1.1: Kurzweil's expansion of Moore's Law shows that due to paradigm shifts the underlying trend holds true from integrated circuits to earlier transistors, vacuum tubes, relays and electromechanical computers.

innermost data-dependent conditions. We will provide a technique to deal with them in this dissertation in Chapter 4. The application uses two-dimensional arrays and a lot of temporary data storing the frame between the two subsequent processing steps of the decoder.

1.2 The platform description

The programmable microprocessor architectures proposed in [216] and starting in the 1971 with the Intel 4004, came a long way in the past 35 years. The number of transistors on an integrated circuit has grown from 2300 in the year 1971 to more than 5×10^8. The growth follows Moore's law, i.e., that the transistor density of integrated circuits, with respect to minimum component cost, doubles every 24 months [239]. The performance of the microprocessors follows this trend and improved 35% per year until 1986, and 55% per year since 1987 [99]. This increase is due to different factors. In the past a constant improvement was given by increasing the clock rate. In addition, the introduction of instruction pipelining when going from a Complex Instruction Set Computer (CISC) to Reduced Instruction Set Computer (RISC) and exploiting the Instruction Level Parallelism (ILP) for superscalars and Very Long Instruction Word (VLIW) caused further improvement in the performance measured

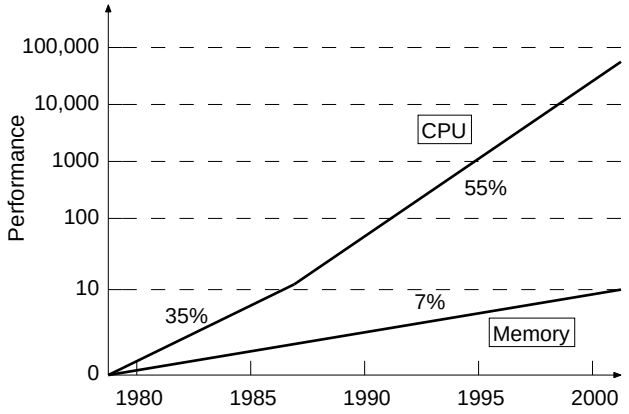


Figure 1.2: The memory gap [99].

as useful work (or instructions) executed per unit time. Nowadays, the clock rate hardly raises. Higher clock speeds may increase the temperature to unacceptable limits. In addition the clock speeds are affected by the scaling problems of wires. Thus, it becomes almost impossible to produce a Central Processing Unit (CPU) that runs reliably at speeds higher than 4.3 GHz or so [239]. Further performance improvements are achieved by using larger caches in the new processors, more functional units [146, 147], and multiple computing cores.

Integrated circuits with the corresponding Moore's Law were not the first computing paradigm. Computing devices have been consistently accelerating price-performance from the *mechanical calculating devices* used in the 1890 US Census, to Turing's *relay-based* "Robinson" *machine* that cracked the Nazi enigma code, to the CBS *vacuum tube computer* that predicted the election of Eisenhower, to the *transistor-based machines* used in the first space launches, to the *integrated-circuit-based personal computers* [131]. This is demonstrated in Figure 1.1 where each paradigm shift caused a further growth of the slope.

Not all aspects of computing technology develop in size and speed according to Moore's Law. Random Access Memory (RAM) speeds and hard drive seek times improve on average 7% each year [99]. This opens the gap between the CPU performance increased so far with 55%/year and the memory performance increase limited to 7%/year. This memory gap is depicted in Figure 1.2. The gap is now so big that it is being referred to as the "memory wall" [220]. Despite the use of advanced memory subsystems during the last 15 years, the "memory wall" is still present [144]. Thus, an intelligent use of the structure and capacity of the memory subsystem by advanced compilation and source-to-source transformation techniques becomes more and more important.

The productivity of software developers most assuredly does not increase exponentially with improvements in hardware, but by most measures has increased only slowly and fitfully over the decades. Software tends to get larger and more complicated over time requiring automated tools helping the designer to manage the increased complexity of the design process.

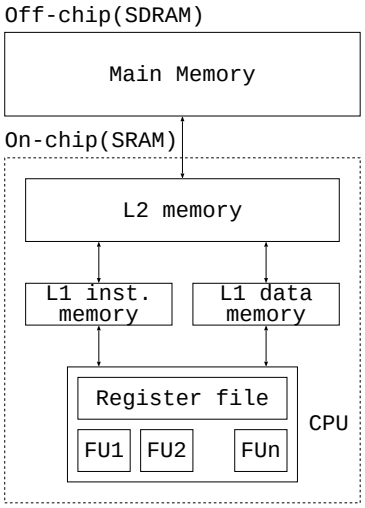


Figure 1.3: A general platform architecture used in this dissertation.

In this dissertation, we focus on the combination of the "memory gap" issue [215] and the software productivity issue. To deal with these issues at a high-level of abstraction, we do not need a concrete instance of a platform. The platform used should have a CPU, at least one level of on-chip memory and an external off-chip main memory. Such a platform architecture with two levels of on-chip memory is depicted in Figure 1.3 and it is a general abstraction of current state-of-the art multimedia platforms. The CPU data-path in the platform has multiple functional units which can be scheduled in hardware (superscalar) or in software (VLIW). The program is stored in the instruction memory and the data is stored in the data memory. The memory subsystem has several layers, at least one resides on-chip and one resides off-chip. The memory is software controlled. However, hardware controlled caches are also allowed in the platform and are used when software control is not possible due to too much unpredictability in the application. Such an abstraction of the platform was already used in previous work dealing with the "memory" gap problem, e.g., [130, 44].

1.3 Mapping problem

The mapping problem is depicted in Figure 1.4a. We have an application as described in Section 1.1 and we have a platform as described in Section 1.2. Our goal is to translate the application specification, described in a high-level language, to the machine language, achieving low energy and very good performance. To achieve this goal, modern (optimizing) compilers are used to translate the high-level language description to the machine language as depicted in Figure 1.4b. However, the resulting machine code is still far from hand-written machine code w.r.t. energy consumption and/or performance. To improve the result of the compiler, source-to-source transformation frameworks are used as depicted in Figure 1.4c. In Subsec-

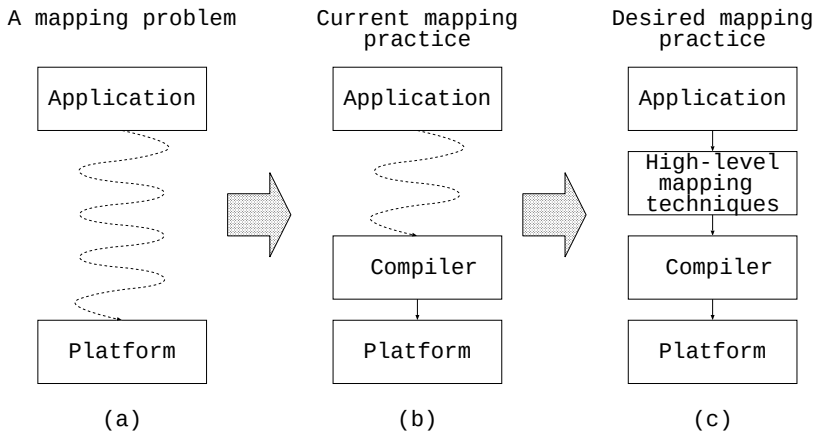


Figure 1.4: A mapping problem.

tion 1.3.1, we introduce the state-of-the-art in the compilation process. In Subsection 1.3.2, we introduce the missing compiler optimizations that are covered by the high-level source-to-source transformation frameworks where the description of the algorithm remains in the same language. However, after the compilation process better results are achieved w.r.t. the energy and performance compared to directly compiling the initial specification. (see Figure 1.4c). The introduction to the source to source optimizations is in Subsection 1.3.2. This dissertation targets research in the area of those source-to-source transformations.

1.3.1 Optimizing compiler

The goal of the compiler is to translate the description in the high-level language to the machine language. This is done in several phases of the compilation process. In the front-end of the compiler, the high-level language is translated into a platform independent internal representation. This representation usually has the form of an Abstract Syntax Tree (AST). In the middle-end of the compiler, this representation is transformed to a platform dependent representation. This representation is referred to as a Register Transfer Language (RTL). In the back-end of the compiler, the assembly code is generated from the RTL description.

The output of a compiler should be, in the ideal case, of equal quality to hand-written assembly/machine code. This is rarely achieved. However, the code that is produced by compiling can be made to run faster, to take less space and to consume less power by applying program transformations, commonly called optimizations. Code optimization can take place at many levels, ranging from the algorithmic level down to the assembly level. Higher level optimizations produce more dramatic results, but lower level ones treat and exploit the target machine idiosyncrasies.

Compiler optimizations fall into two general categories: platform independent and platform dependent. The platform independent optimizations are performed on the platform independent internal representation. At this level, the compiler can improve loops, address references, etc. The platform dependent optimizations, that

include register allocation and utilization of machine idioms, are performed on the platform dependent internal representation and can be largely embedded in the code generation process. At this level, the compiler uses the machine specific information to make good use of the machine resources and idioms.

For the embedded software domain we target, the optimizations performed have to be even more advanced than for the general-purpose domain. This is necessary, because only a small overhead of compiled code versus hand-written assembly code is generally acceptable in the embedded compiler community. With the advent of more sophisticated code optimization technology [135] the overhead of compiled code versus hand-written assembly code has been reduced compared to the past. These novel code optimization methodologies have high runtime requirements. However, in embedded code generation, higher compilation times are acceptable, which may lead to a paradigm shift in code optimization technology. Also, in the embedded domain, the detailed characteristics of the target machines are taken into account resulting in tight development of the compiler with the target architecture [233, 235].

The different optimizations are applied within different optimization passes during the compilation. Nowadays, compilers can have a large number of those passes. E.g., the *gcc* compiler framework contains 127 possible optimization passes and approximately the same amount of parameters to control the amount of optimization to be performed [232]. Those different passes interact together. This causes synergy in a positive, but unfortunately, also in a negative way. The positive effect is when one pass is enabling more freedom for the second pass, e.g., the loop unrolling enables better instruction scheduling. The negative effect is when one pass limits the freedom for another pass or one pass eliminates the effect of other passes, e.g., the loop unrolling limits the freedom for other high-level loop optimizations like interchange and fusion. We foresee this as a big problem also w.r.t. the optimizations on different levels. Recently, adaptive compilation has been proposed, where the optimization passes are selected and tailored to the compiled application [40]. A similar approach is iterative compilation [124, 125, 83, 97], where many variants of the source program are generated and the best one is selected by actually profiling these variants on the target hardware. Other authors [2, 175] use machine learning to gradually improve the optimization results.

Despite of the large number of optimization passes, current compilers often neglect whole groups of optimizations. Important examples are optimizations for better utilization of the memory hierarchy subsystem. The reason is that the hardware control of the current memory elements of this subsystem (caches) determines only at run-time which data to transfer and where to transfer, preventing a large group of design-time memory optimizations. The hardware controlled caches are one of the most power consuming elements in the platform architecture. Nowadays, power dissipation starts to be the limiting factor in embedded handheld devices. Thus, the embedded system domain is trying to avoid using caches and prefers software controlled Scratchpad Memory (SPM)s. These require different (design-time) optimization techniques which are not implemented in current (embedded system) compilers. This triggers the need for source-to-source transformations of these low energy memory elements with increased emphasis on the energy reduction optimizations. Nowadays, these methodologies are implemented separately as source-to-source optimization frameworks. The application is transformed (source to source) in such a

way that the new source code after compilation performs better and/or consumes less energy by effectively exploiting the memory hierarchy subsystem. Ideally, these transformations should be part of the optimizations in the compilation process.

1.3.2 High-level mapping techniques

The high-level mapping techniques targeting the memory subsystem consisting of SPMs gained attention in recent years. We are not going to list here all the high-level mapping techniques present. Rather, we focus on a few state-of-the-art contributions.

The DTSE methodology developed at IMEC targets the optimal mapping of the application w.r.t. the memory footprint reduction and data transfers on a predefined memory subsystem using source to source transformations. The DTSE methodology will be discussed in more detail in Chapter 2. A lot of work in the efficient use of the memory hierarchy has been done at Uni. California, Irwine [170]. They have analyzed the accesses to variables and chose a set of variables to be placed within the scratch-pad memory. At Uni. Dortmund, a technique integrated into a compiler has been developed which analyzes the application, partitions an array variable whenever its beneficial, appropriately modifies the application and selects the best set of variables and program parts to be placed within the scratch-pad [212]. This work uses static analysis for both data and program parts. At Penn State University, a compiler-controlled dynamic on-chip scratch-pad memory (SPM) management framework using both loop and data transformations has been proposed [115]. The work focuses on the data parts only and uses dynamic copying of these data parts. However, the algorithm is only applicable under simplifying constraints, i.e., perfectly nested loops, exactly known loop bounds and array subscripts being affine functions of all loop indices along with additional constraints. Early work in this area based on static analysis has been performed at Uppsala Uni. [189]. Their approach, which focuses on the data parts only, shows that a static analysis is sufficiently precise and no dynamic analysis is needed. The instruction level power analysis and optimization of software is discussed in [198]. Other system level power optimizations for embedded systems are listed in [172, 22, 34].

The high-level mapping techniques consist of platform independent optimizations and platform dependent optimizations. Platform independent optimizations include data-flow transformations removing the redundant memory accesses, and loop and control-flow transformations enabling transformations for the subsequent steps. Platform dependent optimizations contain data and instruction memory organization and transfer, spatial data and instruction locality improvement and cache organization related issues. Most of this work however requires compile time analyzable code limiting the applicability of those high-level mapping techniques.

The importance of loop transformations for memory aspects in the embedded domain has been recognized quite early in the compiler theory [17]. The loop transformations reorganize the control-flow in the loops to enable better optimizations of the subsequent steps during compilation. Very early work on this has started already at the end of the 70's [137] but that was only a classification of the possible loop transformations. The loop transformations have enabled the parallelization or

have improved the temporal locality of data accesses. There is a lot of work in this domain that is surveyed in Chapter 7. At IMEC, the loop transformations have been systematically applied for more than 10 years [203, 44, 213]. However, the structure and complexity of novel applications limits the applicability of these transformations and opens new challenges for enhanced applicability of loop transformation techniques.

1.4 Problem statement and objective

Loop transformations are an important part of high-level mapping techniques such as the DTSE methodology. To perform DTSE optimizations, the complexity of the optimization problem is reduced by selecting the subset of the program which is the target of memory optimization. This subset involves mainly memory accesses, loops and affine and manifest control-flow. The remaining parts of the program are hidden for optimization purposes. In the current DTSE flow, the selection and separation of the program subset relevant for memory optimization is performed manually by the designer. This takes a lot of time and is error prone. Therefore, formalization of this code separation supported by tools is needed to segregate the original source code. Our approach presented in Chapter 4 makes this task for the designer much easier.

The DTSE methodology is targeting rather static (compile time analyzable) applications or application parts. Nowadays, applications are becoming dynamic and have a lot of different execution paths. The actual execution path depends on the mode selected by the application or user. A mode determines which parts of the application will be used, i.e., it determines the execution path of the application. In the past, multimedia applications usually worked in one concrete mode. Modern multimedia applications have multiple modes which are not used equally and do not require the same resources and optimizations.

These dynamic modern applications with a lot of modes cause problems for the traditional DTSE methodology. The DTSE methodology can still be applied, but only on each static part, where control can be determined fully at compile time, separately. However, the static parts are getting smaller. All of this prevents the traditional DTSE from performing global optimizations reducing the main strength of the DTSE methodology. Thus, for those applications, DTSE can perform only local optimizations. In Chapter 5, a scenario methodology instance for loop transformations in the global program scope is proposed and elaborated. A scenario is defined as a selected set of paths in the program which we choose to exploit in the same way [225, 85]. The scenario technique should be applied on top of the DTSE methodology and should "open the eyes" for global optimizations in DTSE when a lot of unbalanced modes are present in the application.

During the platform independent source-to-source transformations of the DTSE methodology, decisions are made without propagating important estimation from the platform dependent transformations. The cost functions used in the platform independent steps lead to one particular decision for that transformation step. As it will be shown in Chapter 6, this can lead to suboptimal solutions. We propose considering the effects of the remaining DTSE steps at the higher levels of the methodology using high-level estimators and not going to one "optimal" solution at the

platform independent steps. E.g., during the early GLT step of the DTSE methodology, one concrete decision is made which is not optimal for every platform instance. Instead, a set of decisions which cover the optimal decisions for any platform instance should be propagated to the adjacent steps of the methodology. With such an approach, we can obtain better results as we will see in Chapter 6.

1.5 Solution on an illustrative example

In this section we demonstrate the solution to the problems mentioned in Section 1.4 on two illustrative examples. We will not use any code in our first example. Instead we will use an analogy with a laundromat where clothes are washed. In our analogy, the clothes corresponds to the code and the washing process to the optimization (loop transformation) of this code. The code "washing machine" analogy was also used by Hugo De Man at the DATE'02 keynote speech [52].

We extend this analogy in Figure 1.5a and define two types of clothes, the white ones that we are going to "wash" and the dark ones that we would like to keep away because of coloring damage. The clothes can only be washed in one public laundromat where different people (Alice, Bob, Carl, Dirk, Erik, Frank and Geert) are coming during the weekend. Each person can bring one laundry basket to the laundromat and not every person is washing each weekend. Which people can meet in the public laundromat during a particular weekend is depicted by following the arrows between a person's laundry baskets in Figure 1.5a starting from the person on the top (Alice) and ending at the person on the bottom (Geert). Thus, Alice, Carl and Geert are washing every weekend. Before going to the laundromat, each person has to separate the white and the black clothes to avoid coloring damage. Thus, at home each person has to remove the dark clothes from the laundry basket. This is demonstrated in Figure 1.5b. How to do so is explained in Chapter 4.

From the Figure 1.5b we can also see that when Alice is in the laundromat, Bob could also be there depending on condition *a*. If Alice would know that Bob is going that weekend to the laundromat, i.e., *a is true*, she could phone him and ask him if he would be so kind and also wash her clothes to save the washing powder and gasoline when driving to the laundromat. This is depicted in Figure 1.5c. We created two scenarios, one assuming that Alice, Bob, Dirk, Carl, Frank and Geert are in the laundromat and the other one assuming that Alice, Bob, Erik, Carl, Frank and Geert are in the laundromat. The actual situation in the laundromat will be known only during the weekend. As can be seen from Figure 1.5c, in the first scenario, Bob and Dirk can combine their laundry for sure and in the second scenario, Alice, Bob, Erik and Carl can also combine their laundry for sure because they will surely wash together if that particular scenario occurs. Of course, in the first scenario Alice can also wash together with Carl and Geert, given that there is no dependency between Alice and Bob (e.g., Bob still has Alice's white socks). Note, that the combination of baskets and arrows in Figure 1.5a represents the Control Flow Graph (CFG) of the application. The details about scenarios will be described in Chapter 5. *Scenario 1* allows us to combine the two baskets of Bob and Dirk with one T-shirt and one pair of shorts. *Scenario 2* allows us to combine two baskets of Alice and Bob with the T-shirt with the basket of Erik with one T-shirt and one pair of shorts and with

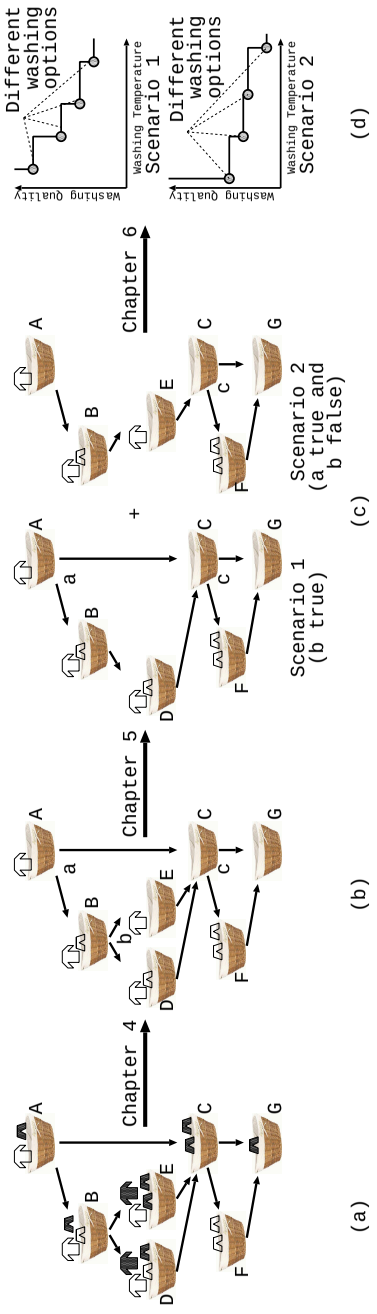


Figure 1.5: The solution on an illustrative example.

the empty basket of Carl. In the code, the combination of baskets with more white clothes means enlarging the exploration space for our transformations.

The clothes in the laundry baskets in each scenario (note that some baskets have been combined) are being washed. However different people like to wash their clothes differently and something else is optimal for them. Some people prefer to wash at 30° not to lose the elasticity of the cloth, for other people it is better to wash it at 60° to be sure that it is fully clean. Also, for optimizations, there is not a unifying best optimization for any platform. That is why we provide a set of best solutions that vary over different platforms. This is demonstrated in Figure 1.5d and it will be discussed in Chapter 6.

Now we will demonstrate the steps from Figure 1.5 on a small piece of real code which is depicted in Figure 1.6a. In the first step in Figure 1.6b, the code that is not the target of the DTSE optimizations is hidden into function **f()**. Then, in the second step (see Figure 1.6c), the code is specialized into two parts, *Scenario 1* and *Scenario 2*. These specialized codes contain only certain paths from the original CFG (see also Figure 1.5). They also enable more DTSE optimizations compared to the previous code in Figure 1.6b. Note that *Scenario 1* and *Scenario 2* together contain all the paths in the original CFG (see also Figure 1.5). On these codes, different loop transformations are applied resulting in different solutions trading-off Data-size cost for Control-flow complexity cost in the 2-dimensional exploration space as depicted in Figure 1.6d. This approach enables more optimizations compared to the approach without scenarios and multidimensional (Pareto) optimal solutions which enable better utilization of the platform after the platform is known.

1.6 Thesis contributions

This dissertation contributes mainly to the platform independent stage of the DTSE methodology. The primary goal of this dissertation is filling the gaps in the platform independent stage and extending some steps in this stage towards future dynamic applications. The dissertation contributes in four areas:

- DTSE requires prepared code where loop level constructs and arrays to be optimized need to be clearly separated from the remaining part of the application code. In Chapter 4 we formalize and implement this separation of the code.
- DTSE primarily targets static applications where the execution order can be analyzed (and modified by DTSE) at compile time. In Chapter 5 we propose a systematic methodology of scenario usage for extending the applicability of the GLT step, which is one of the most important steps in the methodology, towards more dynamic applications.
- The GLT step in the DTSE methodology currently targets only one particular goal which is the minimal lifetime of the array elements in the application so that the memory locations can be reused. In Chapter 6 we present case studies demonstrating that this goal does not always lead to the optimal solution. The presented case studies show that many trade-offs exist during the GLT step.

Figure 1.6: The solution on an illustrative example - a small piece of real code.

- The trade-off oriented GLT requires a different steering mechanism compared to traditional GLT. In Chapter 6 we propose the coupling of high-level estimators to the GLT framework as a steering mechanism for trade-off oriented GLT and contribute to the definition of the requirements for those high-level estimators together with other researchers.

1.7 Thesis overview

The dissertation is composed of eight chapters. In this section we provide short overview of the chapters:

Chapter 2 presents the DTSE methodology and DTSE related techniques. Note, that this chapter provides only an overview of the methodology, a more detailed description can be found in the DTSE books [32, 28]. It also positions our contributions within the whole DTSE methodology flow.

Chapter 3 explains the Geometrical Model (GM) and GLT framework in more detail. Especially, the different GLT frameworks are analyzed and the gaps in these frameworks are defined resulting in the contributions of this dissertation.

Chapter 4 formalizes the algorithms for separation of the code to be optimized from the rest of the code. This separation of the code is called hierarchical rewriting in the DTSE methodology. We have implemented the algorithms using the ATOMIUM framework.

Chapter 5 proposes a systematic methodology of scenario usage for enhancing the applicability of the GLT step in the DTSE methodology. We implemented the proposed methodology using ATOMIUM framework and Boost Graph Library (BGL). In this chapter, a solution for *while* loops is also proposed as a combination of the preprocessing and scenario approach. In that sense, this extension has a cross-level nature with the previous chapter.

Chapter 6 provides case studies which show the need for trade-off oriented GLT. Compared to the "best locality" traditional GLT step, which provides only one GLT solution, the trade-off oriented GLT step provides a set of optimal GLT solutions in the multi-dimensional exploration space. This is achieved by using high-level estimators as the steering mechanism for GLT. The explanation of those high-level estimators and their coupling to the GLT framework is also discussed in this chapter.

Chapter 7 surveys related work in hierarchical rewriting and condition hiding, scenarios and Global Loop Transformations (GLT).

Chapter 8 provides a summary and conclusions, and proposes possible topics for future research.

CHAPTER 2

The DTSE methodology

Goed geheugen. Om zoveel te kunnen onthouden moet hij weinig hebben beleefd.
Jan A. Emmens
(1924-1971)

The memory subsystem and bus usage consumes over 50% of the energy consumption in embedded systems [53, 148]. This is especially true for modern multi-media systems such as image processing or video encoding/decoding which manipulate large multi-dimensional data sets resulting in a large amount of data storage and transfers. Therefore, optimizing the global memory accesses of an application, using e.g., the DTSE methodology developed at IMEC, is a crucial task for achieving effective low-power realizations. The goal of the DTSE methodology for system-level power optimization is to determine an optimal execution order for the data transfers together with an optimal memory architecture mapping for storing the data of the given application [32, 28]. This leads to a reduction in the number of main (off-chip) memory accesses and more efficient on-chip local memory (cache or SPM) utilization. The cost functions currently incorporated for the storage and communication resources are both power and area oriented [30]. Due to the real-time nature of the targeted applications, the throughput is normally a constraint. Improving the global memory accesses generally also has a positive influence on the performance because it reduces the (external) bus traffic and it improves the cache hit rates [129].

The DTSE methodology is split into several substeps combined in two groups: platform independent and platform dependent steps. The platform independent steps transform the program independently of the parameters of the memory (data storage) target platform, which is, in effect, chosen or constructed based on the results of these steps. The platform is subsequently used to further optimize the program in the platform dependent steps. The platform independent steps optimize the data flow, the regularity and locality of data accesses in general, and make the data reuse possibilities explicit. The subsequent platform dependent steps take physical proper-

ties of the target background memory architecture into account to map and schedule the data transfers in a cost-efficient way.

The starting point is an executable system specification with multi-dimensional array accesses. The output is a transformed source code specification, potentially combined with a (partial) netlist of memories which is the input for the final platform architecture design/linking stage when partly customizable or configurable memory realizations are envisioned. The transformed source code is input for the software compilation stage in the case of instruction-set processors. The flow is based on the idea of constraint orthogonalization [27], where in each step a problem is solved at a certain level of abstraction. The consequences of the decisions are propagated to the next steps and as such decreases the search space of each of the following steps. The order of the steps should ensure that the most important decisions and the decisions that do not put too many restrictions on the other steps are taken earlier. The former criterion is quite obvious. The latter is also intuitively clear because if we would perform the decisions which impose many constraints at the beginning, these constraints would limit the exploration freedom of the remaining steps. This general approach is different from the iterative optimization approaches, e.g., [2, 175], which we mentioned already in Chapter 1. There, the applied transformations are dependent on each other, i.e., we cannot unambiguously determine the order of the transformations. Breaking them up in consecutive steps leads to potentially severe suboptimality. This is generally known as the phase coupling dilemma where quality and scalability have to be traded-off [199, 128]. In our approach we circumvent that phase coupling by carefully selected splits. Still, those phase coupled approaches can be well utilized inside the particular steps of DTSE where the constraint orthogonalization does not apply any further.

The DTSE methodology is partly supported with tools in the ATOMIUM system exploration environment. For the platform dependent part the tools are quite robust, well tested and used for real-life designs. For the platform independent part, most of the tools are prototypes only. The transformations are applied to original source code after program partitioning/pruning. The partitioning/pruning reduces the complexity of the exploration space and ensures that only the relevant parts of the code with (large deeply nested) loops and multidimensional memory accesses are the target of the DTSE optimizations.

The complete DTSE methodology is described in detail in [32] for customized architectures and in [28] for programmable architectures. The global DTSE framework is shown in Figure 2.1. To situate our research interests in this global framework, we have highlighted the steps we have contributed to in this dissertation with bold text. To put those steps in the global context we give in this chapter a summary of the DTSE methodology. Section 2.1 provides an overview of the platform independent steps of DTSE and Section 2.2 provides an overview of the platform dependent steps. Section 2.3 discuses other DTSE related methodologies and Section 2.4 summarizes the open issues in the DTSE flow.

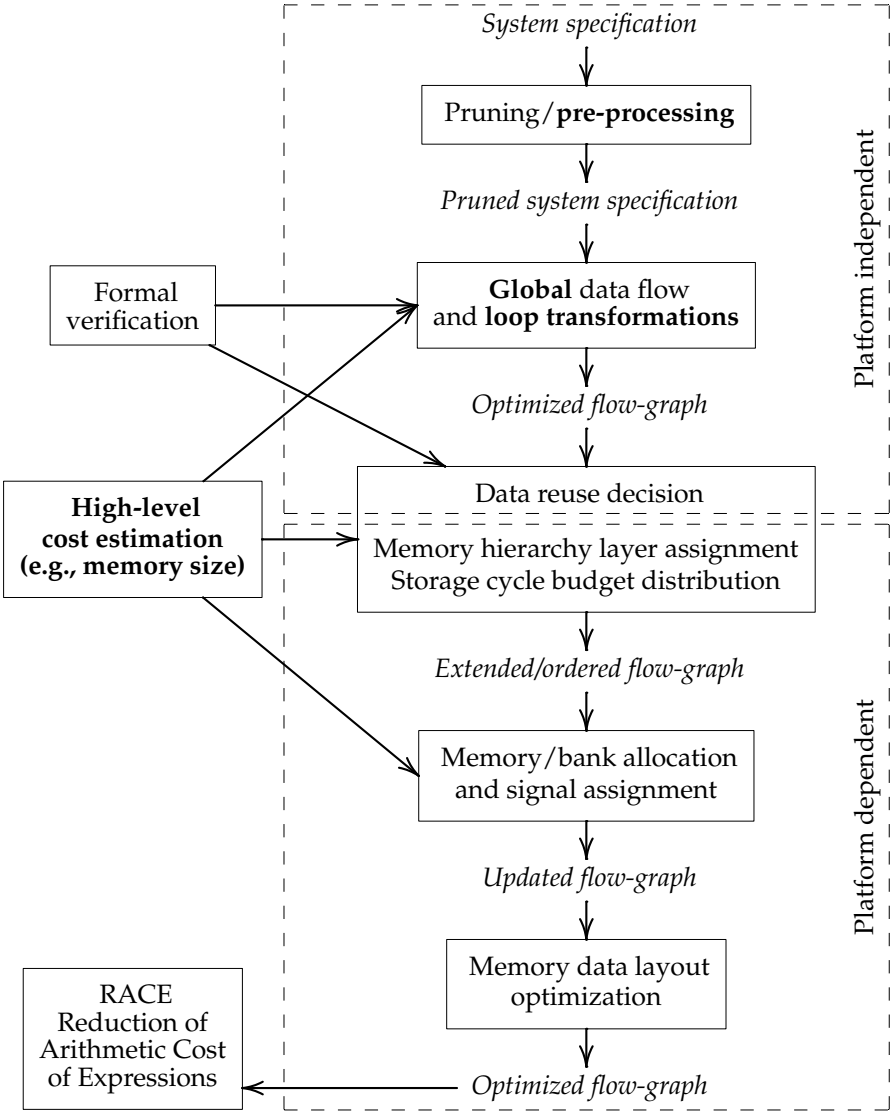


Figure 2.1: DTSE methodology for data transfer and storage exploration: global overview [201, 213].

2.1 Platform independent steps

Platform independent steps of the DTSE reduce the number of array accesses and enable later platform dependent optimization steps. They are beneficial for any platform used later in the design-flow. The platform independent steps are pruning and preprocessing, global data-flow transformations, global loop and control-flow transformations and data reuse exploration.

1. **Pruning and preprocessing**

   This step precedes the actual DTSE optimizations; it is necessary to identify and isolate the parts and data structures in the program which are data-dominant and thus relevant for the DTSE. The preprocessing/pruning step also presents this code in a way which is optimally suited for transformations [32]. Thus mostly loops with large bounds and exhibiting good reuse of data and data structures such as array variables are exposed. All the freedom is exposed explicitly, and the complexity of the exploration is reduced by hiding constructs that are not relevant. Apart from areas of power oriented gain, the parts of the program, which are a bottleneck for obtaining better performance need to be identified. These are mostly the data structures that have very little locality but are accessed heavily.

   During the pruning the ATOMIUM analysis tool can be used. This tool identifies the used part of the code dynamically during the profiling phase using given testbench. Thus the testbench for the application has to be carefully selected. ATOMIUM dynamic pruning prunes all the code we are not interested in for further optimization. The ATOMIUM analysis identifies and profiles all array accesses in the program. This gives us the first hint on which parts of the code we should target our optimization effort. After the pruning, the preprocessing, i.e., to expose explicitly all the optimization freedom, is performed.

   During the preprocessing, pointer accesses are converted to array accesses [63, 75], other constructs that cannot be modeled by the geometrical model are hidden away [162], functions are selectively inlined [1] and the code may be rewritten in Dynamic Single Assignment conversion (DSA) form [69, 209]. Although DSA is not a strict requirement for the following steps, it does increase the freedom potentially allowing better transformations to be performed. For all those preprocessing techniques, mostly prototype tools exist.

   We contribute to the preprocessing step by proposing a methodology for dealing with data-dependent conditions. We separate the treatment of outer and inner data-dependent conditions. Inner conditions are hidden by using our technique for hierarchical rewriting and hiding of data-dependent conditions. Outer control-flow is treated with the instantiation and extension of general scenario methodology [225, 85]. This enables better utilization of the GLT step of the DTSE methodology as will be discussed in sequel.

2. **Global data-flow transformations**

   The set of system-level data-flow transformations that have the most crucial effect on the system exploration decisions has been classified, and a systematic

methodology has been developed for applying them [31, 33]. Two main categories exist. The first one directly optimizes the important DTSE cost factors by removing redundant accesses and reducing intermediate buffer sizes. The second category serves as an enabling transformation for the subsequent steps because it removes the data-flow bottlenecks wherever required, especially for the global loop transformations step.

The main goal of the global data-flow optimization step is to reduce the number of bottlenecks in the algorithm that prevent optimizing code restructuring transformations from being applied and remove access redundancy in the data-flow. The transformations consist mainly of advanced signal substitution avoiding unnecessary copies of data, modifying computation order in associative chains enabling certain loop transformations, shifting of "delay lines" through the algorithm to reduce the storage requirements, and recomputation issues to reduce the number of transfers and storage size. For the data-flow transformations only a limited set of prototype tools exists.

3. **Global loop and control-flow transformations**

   The goal of the global loop and control-flow optimization step is to reduce the global lifetimes of the signals and to increase the locality and regularity of the data accesses. Signal is any time-varying quantity in the program that could be either scalar valued or vector valued. In the DTSE methodology we focus on vector-valued signals, i.e., array variables. Locality of data accesses means that the accesses to the same memory location have to be close in time and the regularity means that the order of consumption should be the same as the order of production. The transformations remove system-level buffers introduced due to mismatches in production and consumption ordering (regularity problems). They allow also the data to be stored later in the design flow in smaller memories closer to the data paths.

   The Global Loop Transformations (GLT) step, the related preprocessing step and the link to high-level estimation leading to trade-off oriented GLTs are the main subjects of this dissertation. The transformations in the GLT aim at improving the data access regularity and locality for multi-dimensional array signals and at removing the system-level buffers introduced due to mismatches in production and consumption ordering (regularity problems). The state of the art in GLT focuses on one cost function resulting in one optimal solution for GLT [204, 76, 44, 213]. In this dissertation we show that an extension of this approach towards different cost functions resulting in trade-offs is needed. We propose to use high-level estimation to steer those trade-off oriented GLTs. For the GLTs a prototype tool and a more robust tool built on top of ATOMIUM framework exists. The results of this dissertation have also contributed to the development of these tools.

   The GLTs are applied globally across the full code and not only individual loop nests, also across function scopes because of the selective inlining applied in the preprocessing step. Still, they are applied only within the Static Control Part (SCoP)s which will be defined in Chapter 3. In the preprocessing we propose a novel techniques that enables additional GLT going beyond the scope of SCoPs, e.g., across data dependent conditions. For these novel techniques

also prototype tools based on the ATOMIUM framework, Standard Template Library (STL) and BGL have been developed.

It is crucial that the GLT step is applied before the data reuse exploration and in-place steps. Loop transformations change the execution order such that the production and the consumptions of data elements are moved closer together in time. The result is that the data reuse copies in the memory hierarchy can be made smaller since data is kept in the copy for a shorter time period, and higher data reuse factors can be achieved. Also, the intra/inter in-place optimizations which exploit the reuse of the memory location of the array element/array which is not used any more in the program by newcoming array element/array, can benefit from limited lifetime enabled by GLT.

4. **Data reuse exploration**

The goal of the data reuse decisions step is to better exploit a hierarchical memory organization to benefit from the available temporal locality in the data accesses. An important consideration here is the distribution of the data over the hierarchy levels such that frequently accessed data can be read from smaller and less power consuming memories. This obviously has a positive effect on the total power consumption of the application because the most frequently accessed data is then read from less power consuming memories. Also the smaller memories can then be closer to the data paths thereby reducing the dissipation in the interconnect, especially if off-chip memory accesses are replaced by on-chip memory accesses.

In this step the data locality introduced by the previous global loop transformation step is exploited. Data reuse possibilities are made explicit by analyzing virtual multi-copy hierarchies (including bypasses) for the trade-off of power and memory size cost. Heavily reused data will be copied to smaller power-efficient on-chip memories, while costly accesses to external memory are reduced.

The basic methodology of [58] and [222] is systematic, though it has restrictions on the actual data reuse behavior that can be handled. [200] and [28] have extended this methodology by introducing some vital cost parameters to describe a more complete search space. They further explored the relationship between these parameters and the cost function for power and memory size, and proposed heuristics to steer the search for a good solution.

[200] formalized the extended search space by introducing an analytical model for the cost parameters as a function of the index expressions and loop bounds. This avoids long simulation times and more importantly, it allows for the identification of exactly which array elements have to be copied to a sub-level for optimal data reuse. This has lead to a fully automated design technique for all loop-dominated applications to find optimal memory hierarchies and generate the corresponding optimized code [201].

2.2 Platform dependent steps

Platform dependent steps of the DTSE uses the information about the predefined memory organization to perform further optimizations. Some substeps only apply for an (embedded) customizable memory organization which is becoming available on several platforms by partly powering down overdimensioned memory blocks that are not fully needed. The platform dependent steps are Memory Hierarchy Layer Assignment (MHLA), Storage Cycle Budget Distribution (SCBD), Memory Allocation and Assignment (MAA) and memory data layout optimizations.

1. **Memory Hierarchy Layer Assignment (MHLA)**

   The MHLA step maps the most beneficial candidates from data reuse copy trees to a virtual memory hierarchy subsystem. During MHLA [26], the data reuse copy trees resulting from the data reuse exploration and the corresponding transfers are partitioned over several hierarchical memory layers, based on the bandwidth and high-level memory size estimation. The high-level memory class of each of the memory layers is determined (e.g., on-chip, off-chip, ROM, SRAM or DRAM and other RAM "flavors"). The Data Reuse Analysis (DRA) and the MHLA steps have been integrated in the robust Memory Hierarchy (MH) tool based on the ATOMIUM framework.

2. **Storage Cycle Budget Distribution (SCBD)**

   The goal of the SCBD step is to ensure that the (usually stringent) real-time constraints are met with a minimal cost penalty. The major substep involves Storage Budget Optimization (SBO) to determine which data should be made simultaneously accessible in the memory architecture such that the real-time constraints can be met with minimal memory bandwidth related costs.

   This step mainly determines the bandwidth/latency requirements and the balancing of the available cycle budget over the different memory accesses.

   Additional loop transformations are performed to meet the real-time constraints, such as merging of loops without dependences, software pipelining and partial loop unrolling [185]. These loop transformations normally do not influence the access order of data elements, so also the data reuse behavior remains the same.

   The data reuse transformations introduce dependences in the code which constrain the freedom for SCBD transformations. However, a certain transformation freedom is made available by defining the data reuse copies in single assignment form. This allows the SCBD transformations to move copy update code out of a loop kernel for performance reasons. This extends the lifetime of the data, since the data is copied earlier to the copy-candidate than actually needed. As a result, the performance gain has to be traded off with a slightly larger final copy size cost [47].

   The initial data types (arrays or sets) are grouped/partitioned in basic groups, a sub-step called Basic Group (BG) structuring [60]. SBO performs a partial ordering of the flow graph at the BG level. It tries to minimize the required memory bandwidth for a given cycle budget. This step produces a conflict

graph that expresses which BGs are accessed simultaneously and therefore have to be assigned to different memories or different ports of a multi port memory [221, 223, 157].

3. **Memory/bank allocation and signal assignment (MAA)**

   The goal of the memory allocation and assignment step is to determine an optimal memory architecture for the background data. The step allocates memory units and ports (including their types) from a memory library and assigns the data to the best suited memory units, given the cycle budget and other timing constraints [14, 190]. The combination of the SCBD and MAA tools allows to derive real Pareto trade-off curves of the background memory related cost (e.g., power) versus the cycle budget [25]. This combination has been implemented in the Memory Architect (MA) tool based on the ATOMIUM framework.

4. **Memory data layout optimization**

   The goal of SPM and cache optimization is effective utilization of a processor's on-chip local memories by in-place [50] and data-layout [129, 130] transformations. In-place optimization consist of inter-signal (among data structures like arrays) and intra-signal (among elements of data structure like elements of array) in-place mapping. The transformation exploits the limited life-time of the data during program execution and thus reduces the capacity misses.

   This involves several sub-steps and focuses both on the SPM and the cache(s) and the main memory. One of the main issues involves in-place mapping of arrays and sub-arrays. In the worst case, all arrays require separate storage locations. When the lifetimes of arrays or elements in the array are not overlapping, the space reserved in the memory for these groups can be shared [51]. The single assignment arrays and array copies introduced in the data reuse step are in-placed during this step, leading to final optimal copy sizes. After the in-place data mapping step we now decide which signals will be locked in the data cache in case of a software controlled cache. The in-place technique (both intra and inter) for SPM or software-controlled caches has been implemented in the robust Memory Compaction (MC) tool based on the ATOMIUM framework.

   In the memory allocation and signal-to-memory assignment step, signals were assigned to physical memories or to banks within predefined memories. However, the signals are still represented by multi-dimensional arrays, while the memory itself knows only addresses. In other words, the physical address for every signal element still has to be determined. This transformation is the data layout decision. Main memory data-layout optimization exploits the memory organization data freedom and thus reduces the conflict misses.

   For hardware-controlled caches advanced main memory layout organization techniques have been developed, which allow to remove most of the present conflict misses due to the limited cache associativity [130]. Extensions on this methodology are based on the estimated copy size during the data reuse step [202].

2.3 Other related methodologies and stages

Except of the DTSE methodology several related methodologies and stages that complement the DTSE exist. The most important and the most coupled are high-level memory size estimation, formal verification techniques for system-level transformations and Reduction of Arithmetic Cost of Expressions (RACE).

- **High-level memory size estimation**

  The memory data layout optimization (see above) is the last step in the DTSE methodology and determines the overall required memory size of the application. However, during the earlier DTSE steps (GLT) the final execution order of the memory accesses is not yet fixed. Lower and upper bounds for the needed memory size for a partially defined loop organization and order, have been proposed by [122]. These can be used to steer the many possible loop transformations for a given application, and are also useful during the GLT step to help steering the exploration.

  While [122] mainly focuses on bounds on the memory needed for individual arrays, [180] considers the effect of simultaneously alive data-dependences to estimate the combined storage requirements for multiple arrays when the ordering of the accesses is fixed. [104] investigate bounds on memory requirement when only part of the global loop transformation has been fixed. In particular, they consider bounds over all possible loop fusion and loop shifting transformations.

  [105, 106] adds to the in-place estimation also DRA and MHLA decision estimation resulting in the unique framework that estimates the most important steps after the GLT. The estimation is linked to the GLT framework and incrementally estimates the effect of the incremental GLT. That means, only parts of the code affected by the incremental GLT are estimated again and combined with the previous estimation result. This framework is interesting not only for its completeness of estimation but also for its moderate time requirements.

- **Formal verification techniques for system-level transformations**

  In addition to the results on exploration and optimization methodologies, work has been done on system-level validation by formal verification of global dataflow, loop and data reuse transformations [181, 42, 186]. Such a formal verification stage avoids very costly CPU-time and design time re-simulation.

- **Reduction of Arithmetic Cost of Expressions (RACE)**

  The DTSE introduces a lot of addressing and control overhead. To deal with this overhead, the Address Optimization (ADOPT) methodology has been developed at IMEC [152]. That methodology has been extended to programmable processor contexts [94], including modulo addressing reduction [80]. The ADOPT methodology has been implemented in the RACE tool based on the ATOMIUM framework. However, the usage of the tool is limited due to strict input code requirements.

  The DTSE steps will typically significantly increase the addressing complexity of the applications, e.g., by introducing more complex loop bounds, index expressions and conditions. When this computational overhead is neglected, the

optimized system will, although being more power efficient, suffer a severe performance degradation.

However, most of the additional complexity introduced by DTSE can be removed again by source code optimizations such as constant propagation, code hoisting, strength reduction and others. Code hoisting moves loop-invariant computations out of the scope of the loop body, eliminating unnecessary recomputations of the same values. Strength reduction replaces expensive operations in terms of performance by alternatives using cheaper operations. For example, expensive modulo operations in addressing arithmetic can be replaced by an alternative implementation using cheaper increment and decrement operations [80]. The control flow complexity due to the introduction of conditional statements can also be removed [66, 67]. The final result is that for most applications not only the power consumption is reduced, but also the system performance is increased. However, also during the ADOPT optimization steps, trade-offs occur [160, 161].

2.4 Open issues

As we stated in Section 2.2 and in Section 2.3 the platform dependent steps and RACE are covered by tools. These tools are not prototype tools but rather mature tools which are used in real-life designs. Thus the research open issues remain in the platform independent part of the DTSE methodology (see Section 2.1), in the high-level cost estimation and in the formal verification (see Section 2.3). For the platform independent part of the DTSE methodology especially the preprocessing step and the global data flow and loop transformations steps are interesting research challenge. The GLT are closely related with high-level cost estimation. The high-level cost estimation is necessary to assess the impact of the GLT on the later design steps in the DTSE trajectory.

To solve the remaining open issues in the DTSE methodology we launched several research tracks. The formal verification research track was started in cooperation with the Katholieke Universiteit Leuven and is running already for some time [181, 42, 186]. The high-level cost estimation research track was started in cooperation with the Norway University of Science and Technology [122, 180, 104, 105, 106]. The current focus is on the Hierarchical Memory Storage Estimation (HMSE) which covers to the large extent the DTSE related issues. However, the high-level control-flow complexity estimation is still lacking.

This dissertation tries to fill the gap contributing in preprocessing for data dependent conditions (Chapter 4 and Chapter 5) and in trade-off oriented GLT and coupling the high-level estimators to the GLT framework (Chapter 6).

CHAPTER 3

Global loop transformations

Αγεωμέτρητος μηδείς εισίτω.
Plato
(427BC-347BC)

In compiler theory, loop transformations play an important role in improving cache performance and effective use of parallel processing capabilities. Most of the execution time of a scientific program is spent within loops. Thus a lot of compiler analysis and optimization techniques have been developed to make the execution of loops faster [238]. This description from Wikipedia emphasizes the importance of loop transformations for improving the performance of the system. However, recent advanced multimedia systems typically use also a large amount of data storage and transfers. This memory and bus usage consumes a major part of the energy in the system. The loop transformations may improve the regularity and locality of memory accesses. This enables better mapping of the arrays to software steered memory subsystem consisting of on-chip SPM and off-chip Static Random Access Memory (SRAM) main memory. The low energy consumption is especially desired for modern embedded systems.

The loop transformations are performed on a Geometrical Model (GM). In this model, all iterations of a particular statement are represented by a polyhedral shaped domain. The dependencies within the algorithm are represented as relations between those polyhedral shaped domains. Despite the conciseness of the model and effectiveness in dealing with generic loop transformations [218, 116, 46, 145], the model imposes strict limitations on the input code [19].

Section 3.1 presents the GM and discusses its limitations. Sections 3.2-3.8 explain a general Global Loop Transformations (GLT) framework and discuss in detail the different modules of the framework. Section 3.9 highlights the open issues when using current loop transformations techniques and frameworks. This dissertation tries to solve these open issues.

3.1 GM and its limitations

For the loop transformations, the geometrical model seems to be the right data model choice. When using this model, loop transformations can be performed highly effectively and efficiently by simple matrix manipulations [176, 217]. The model has been used in many research projects related to loop transformations [134, 71, 46, 72, 136, 56, 118, 141, 46, 38, 77, 191] and there exist several very good libraries [217, 176, 70, 68, 11] that support this model. We are not going to discuss these here, the state-of-the-art in this area can be found in Chapter 7.

In the GM model each statement is represented by its *iteration domain*, *statement description* and *index functions of arrays in the statement*.

Definition 3.1 *A* polyhedron *(also known as polyhedral set) in $\mathbb{R}^n$ is the intersection of a finite family of closed half-spaces in $\mathbb{R}^n$.*

Definition 3.2 *A* polytope *in $\mathbb{R}^n$ is the convex hull of a finite set of vectors (tuples or points in the vector space) or, equivalently, a bounded polyhedron.*

Definition 3.3 *An* iteration vector *is a tuple of the loop iterator values or, equivalently, a point in the iteration vector space.*

Definition 3.4 *An* iteration domain *of a statement is a set that represents all iteration vectors where the statement is executed. It can be described as the integer points in a polyhedron, if the loops are affine expressions of the outer loop iterators and the parameters and all loops increase by one in each iteration of the loop.*

Definition 3.5 *An* index function *is a function that maps the values of the iterators to an index of an array.*

The GM can handle only source code with manifest and affine for loop bounds and if conditions which determine the iteration domain polytope. Also the index functions of arrays in the statement have to be manifest and affine. The source code that can be modeled in GM has to fulfill the requirements of the SCoP [19]. The SCoP is a maximal set of consecutive statements without *while* loops, where loop bounds and conditionals may only depend linearly on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters. Note, that the term Polyhedral Dependency Graph (PDG) used in the literature is the GM with additional information about the dependencies among the iteration domain polytopes computed by the dependency analysis from the index functions. The dependency analysis and the PDG will be discussed in Section 3.5.

Figure 3.1a shows a simple C source code consisting of two statements (S1 and S2) in two loop nests. The iteration domain of S1 is depicted in Figure 3.1b and iteration domain of S2 is depicted in Figure 3.1c. The iteration domain of S1 as well as the iteration domain of S2 can be expressed with a set $\{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i,j \leq 3\}$. In the future we will implicitly consider that all the iterators are integer points and we will omit the $\in \mathbb{Z}$ part.

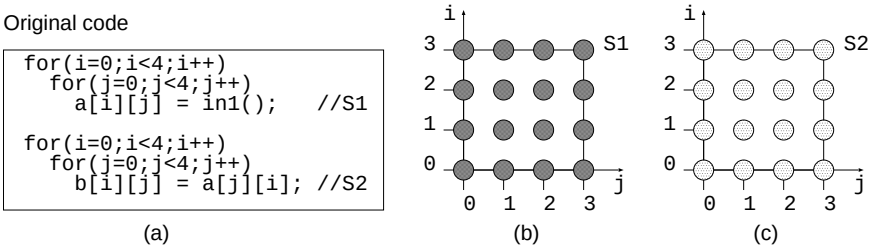


Figure 3.1: GM example for two loop nests: (a) Original code; (b) GM for S1; (c) GM for S2.

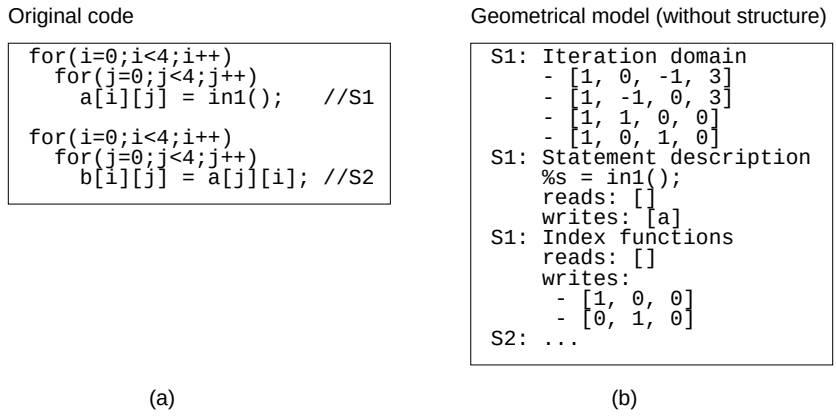


Figure 3.2: GM example for two loop nests: (a) Original code; (b) Parts of YAML description [242] of the GM for S1 in the Polylib notation [217].

The iteration domains can be also represented in the Polylib notation [217]. In Figure 3.2 is the representation of the S1 iteration domain (Polylib notation) together with the statement description and the index functions of array in the statement S1 in the YAML Ain't Markup Language (YAML) format [242]. YAML is a data serialization format designed for human readability and interaction with scripting languages such as Perl and Python.

The iteration domain of S1 in Figure 3.2b is in the constraint format, where each line represents one constraint. The four lines/constraints express the four loop bounds of the two dimensional loop nest. Each constraint (equality or inequality) consists of a vector of n+2 elements, where n is the depth of the loop nest, and it has the format $(S, X_1, X_2, ..., X_n, K)$. This format represents the constraint:

$$
\begin{aligned}
if S = 0: &\quad X_1 i + X_2 j + \cdots + X_n + K = 0 \\
if S = 1: &\quad X_1 i + X_2 j + \cdots + X_n + K \geq 0
\end{aligned}
$$

The statement description identifies the statement the iteration domain belongs to and lists in sequential order each array access (read/write) which is substituted by a %s in the original statement description. The index functions are listed for each

Original code

```
for(i=0;i<4;i++)
  for(j=0;j<4;j++)
    a[i][j] = in1();   //S1

for(i=0;i<4;i++)
  for(j=0;j<4;j++)
    b[i][j] = a[j][i]; //S2
```

Geometrical model (with structure)

```
S1: Iteration domain
    - [0, 1, 0, 0, 0, 0, 0]
      (1*t0+0*i+0*t1+0*j+0*t2+0*1==0)
    - ...
S2: Iteration domain
    - [0, 1, 0, 0, 0, 0, -1]
      (1*t0+0*i+0*t1+0*j+0*t2-1*1==0)
    - ...
```

(a)

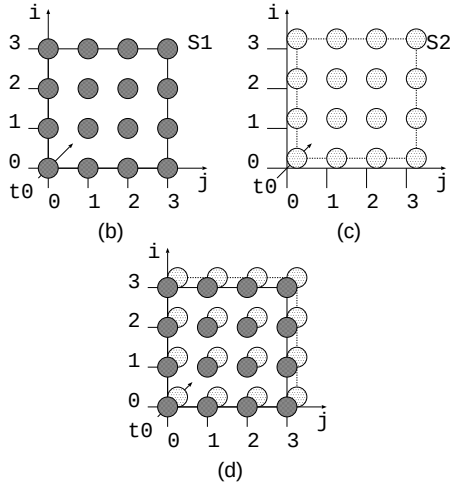


Figure 3.3: GM example for two loop nests: (a) Time dimension constraint for S1 and S2; (b) GM for S1 with time dimension; (c) GM for S2 with time dimension; (d) S1 and S2 placed in CIS.

array access (read/write). Each line corresponds to one dimension of an array. The index function has the format $(X_1, X_2, ..., X_n, K)$ representing the function:

$$X_1 i + X_2 j + \cdots + X_n + K$$

In the Geometrical Model (GM) in Figure 3.1 the information about some structure of the code is missing. From the iteration domain descriptions of the individual loop nests we cannot determine which loop nest is the first one and which is the second one. I.e., we cannot decide on the order of the two loop nests based on the iteration domain descriptions only. To represent the complete program in the GM we need to introduce the time (called as well pseudo or statement) dimensions between each two loop levels. The first time dimension is added at the first position. It orders statements at the global (outside loop) level. This is also depicted in Figure 3.3b where the added constraints distinguish between the first loop nest and the second loop nest. In the code, we can imagine it as introducing an extra virtual time loop t_0 iterating from 0..1 as the first statement at the global level and introducing a virtual if condition before each loop nest. The if condition would be *if*$(t_0 = 0)$ for the first loop nest and *if*$(t_0 = 1)$ for the second loop nest. Thus, only after the whole execution of the first loop nest, the t_0 iterator will increment and the second loop nest will be executed. This corresponds to the original execution ordering in the code in Figure 3.3a. The second time dimension is added between the first and the second loop dimension and determines the order of the statements within the outermost loop of the loop nest. The total number of time dimensions is d+1 where d is the depth of the deepest loop nest in the code. When having very deep loop nests the total number of dimensions, i.e., d real dimensions plus d+1 time dimensions, can be large. However, time dimensions can provide redundant information and then they are not needed. E.g., because the code in Figure 3.1 has two statements at the global

(top) level (the two for loops) the t_0 is needed to distinguish between those two for loops. However, at the first and second level, there is only one statement assigned to each of the two top for loops. Thus the time dimensions t_1 and t_2 are not needed to cover the complete structure of the code. Time dimensions have been already used in [71, 120, 19, 213].

After including all information about the structure of the code, the single statement iteration domains can be combined into a Common Iteration Space (CIS).

Definition 3.6 *The* common iteration space *is an n dimensional space, $n \geq r$, where r is the dimension of polytope with greatest dimension, where all polytopes are combined. The extra dimensions of the polytopes with $d < n$, where d is the dimension of the polytope, are set to a fixed value, e.g., 0.*

This combination is depicted in Figure 3.3d. In this CIS all the statements, i.e., S1 and S2 are placed. The order of the iterators is t_0,i,t_1,j,t_2 where t_0 is the outermost iterator and t_2 is the innermost iterator. We depicted only the iterators t_0, i and j because the t_1 and t_2 do not decide on any execution ordering and can be eliminated as discussed above.

In the following sections we discuss in detail the whole loop transformation framework using the GM we just presented as well as array data flow analysis and preprocessing issues. This leads us to the shortcomings in the Global Loop Transformations (GLT) flow which are highlighted in Section 3.9.

3.2 Loop transformation tool

Loop transformations are one of the basic optimization techniques used at the high-level to improve the source code implementation resulting in energy and/or performance gains. However, to perform these transformations manually is a tedious and error-prone task. Thus, the automation of the loop transformation framework is needed.

The loop transformations are usually performed on the GM especially due to the wide scope of loop transformations that can be handled in a uniform way. We have also selected this model as most suited for the GLT framework at IMEC. In general, every loop transformation framework consists of several modules (see Figure 3.4). The first module, called the **parser** translates the source C code to the Internal Representation (IR); e.g., AST. On the IR, the preprocessing is performed supported by different analysis methods or tools, e.g., Pointer Analysis and Conversion (PAaC), hierarchical rewriting, Factored User-Def chains (FUD) etc. The preprocessing is an optional block which enables that more code can be extracted to the GM. It is discussed in detail in the next section. The second module **GM checker/splitter** identifies the parts of the code that meet the SCoP requirements and separates them from the parts that do not meet the SCoP requirements. Note, that the part meeting the SCoP requirement can become larger when more preprocessing techniques are applied. In Chapter 4 and Chapter 5 we will present two novel preprocessing approaches that are one of the main contributions of this dissertation. The

extractor module translates the parts of the IR that fulfill the SCoP requirements, i.e., $IR_{GM1..n}$, from the internal representation to the geometrical model, i.e., to the $GM_{i,i\in 1..n}$. Each GM is then transformed to a GM^T in the **LT** module. The transformations are steered by the cost functions and supported by analysis. An important part is array dependency analysis which is performed on a GM. It identifies the dependencies among array accesses and their sizes. After performing the transformations on the $GM_{i,i\in 1..n}$ the $GM^T_{i,i\in 1..n}$ are transformed by the polyhedral **scanner** to the $IR^T_{GM1..n}$. Such a transformed IR is combined with the IR that did not meet the SCoPs requirements, i.e., IR_{NGM} (AST_{NGM}). Note that this IR also contains annotations about the positions within the code of the blocks that were extracted to GM and transformed. This is performed in the **combiner** block. The new IR^T (AST^T) is postprocessed and transformed to the new C^T source code.

In the next sections we will discuss in more detail the most interesting parts of this framework, namely the preprocessing, the GM extraction, the array dependency analysis, the GLT itself, the GM scanning and the postprocessing.

3.3 Preprocessing for GLT

From Section 3.1 it is obvious that the geometrical model is quite strict on the input source code (or internal representation) that can be extracted to the model. Only the blocks of the code that contain constructs supported by the model, i.e., *for*s and *if*s with manifest and affine bounds/control expressions and statements with affine and manifest memory accesses (arrays) are selected by the **GM checker/splitter** for the extraction to the model. The rest of the input source code is kept in the internal representation and not optimized. This part of the code is finally merged with the loop transformed parts with the **combiner**.

Nowadays, multimedia applications consist of large pieces of code which contain large deeply nested loops that process indexed signals. Current multimedia applications are not explicitly in the shape suitable for GM extraction and large parts of the code cannot be recognized as GM extractable by the **GM checker/splitter**. However, implicitly they are extractable. E.g., the indexed signals are hidden behind pointer arithmetics; large and deeply nested loops are distributed across several function calls; the indexed signals are intermingled with scalar signals that should be hidden for the high-level memory optimizations, and the affine if conditions and addressing are intermingled with the data dependent ones. To separate clearly and make explicit the GM extractable code is the main task of the preprocessing before the **GM checker/splitter**. Without the preprocessing the GM checker/splitter selects only those parts that are explicitly GM extractable in the original code. This is only a small portion of the code that could be extractable in reality, i.e., after the preprocessing. Thus, the preprocessing step is needed, which makes the implicitly GM extractable constructs explicit and hides the constructs that are not supported by the GM. This is done by the internal representation restructuring effort. Thus, we try to rewrite the internal representation of the application so that the undesired constructs for loop transformations are hidden and the desired constructs for loop transformations are made explicit before the GM part and the non-GM part are analyzed and split by the **GM checker/splitter**.

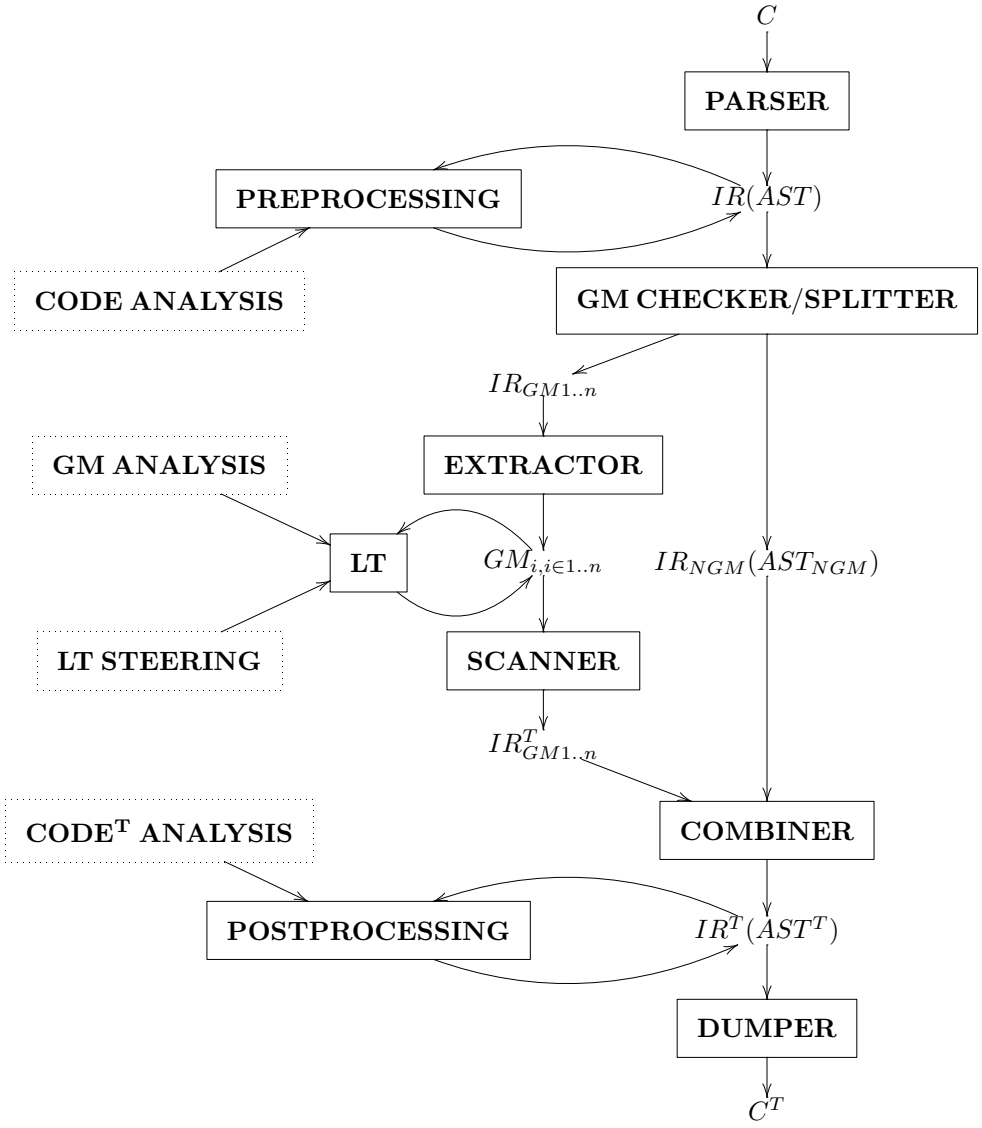


Figure 3.4: General loop transformation framework.

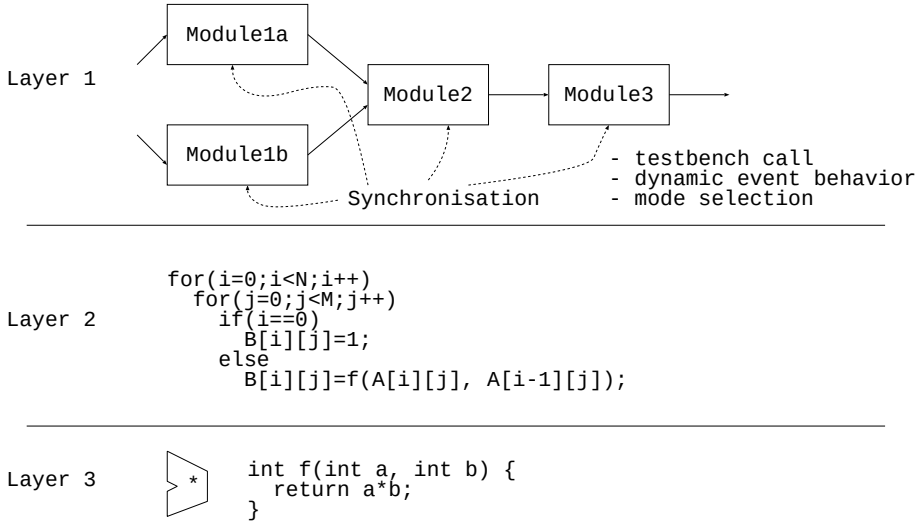


Figure 3.5: Dividing an application in the 3 layers.

The main tasks of the preprocessing are to reduce the complexity for the designer and/or tools, increase exploration freedom by making available search space explicit and by removing bottlenecks and hiding undesired constructs that are difficult to handle in GM. The preprocessing consists of several substeps such as hierarchical rewriting, hiding of undesired constructs, code expansion, array/pointer data-flow analysis, data flow chain removal, weight-based removal and partitioning.

In the hierarchical rewriting and hiding of undesired constructs substeps the application is rewritten in 3 layers. The first layer contains process control flow, the second layer contains loop hierarchy and indexed signals and the third layer contains arithmetic, logic and data-dependent operations. This is depicted in Figure 3.5. The first layer is usually well written already by the designer, however the second and third layer constructs are intermingled. To separate them systematically is a challenging task. We contributed to this task in Chapter 4 where we proposed and implemented a systematic technique to separate the two layers. An important part of the hierarchical rewriting is to make the addressing explicit. This requires Pointer Analysis and Conversion (PAaC) techniques [184, 63, 75] which transfer the implicit array accesses (pointers) to explicit ones (arrays). The hiding of undesired constructs also includes hiding the data dependent *if*s to the Layer 3 and using the worst case bounds for data dependent addressing.

The function inlining causes code expansion and creates the global search space for the GLT step. As discussed above, the loops can be distributed across several functions. The parts of the code that could contribute to the global search space should be inlined. This can be achieved with Selective Function Inlining (SFI) approach [1]. Note, that only the Layer 2 code should be inlined. Thus the inlining has to be selective, i.e., it should select only the code related to Layer 2 that should be inlined only at the call instances which are target of our optimizations.

The Dynamic Single Assignment conversion (DSA) [69, 209] does not enlarge the

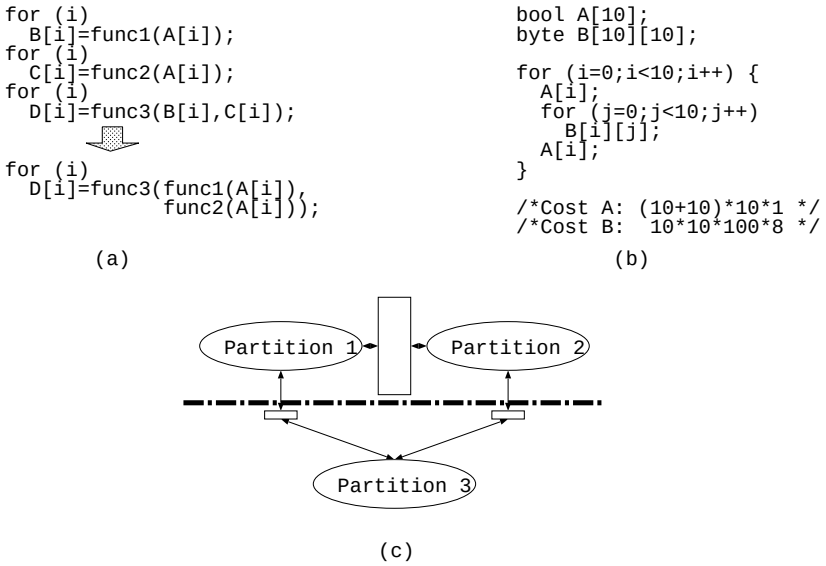


Figure 3.6: (a) Data flow chain removal; (b) Weight based pruning; (c) Partitioning.

code that can be extracted to GM. However, it simplifies the array data-flow (dependency) analysis (see Section 3.5) and thus enlarges the search space for the GLT. The remaining steps in the preprocessing are data-flow chain removal, weight-based removal and partitioning [32]. The data-flow chain removal means eliminating avoidable buffers in the program. This is depicted on Figure 3.6a where arrays B and C represent avoidable buffers that can be eliminated without changing ordering of the input and output of the chain being eliminated. No trade-off is involved when removing data-flow chains. The weight-based removal and partitioning simplifies the challenging task of DTSE optimizations by not considering small unimportant arrays in the exploration and partitioning of the problem using the divide and conquer principle. The weight based removal is based on the cost function $f_{access} * size * bits$ where f_{access} is number of accesses to a particular array, *size* is its size (number of elements) and *bits* is number of bits used for one element. The example of weight based removal is in Figure 3.6b. In this example, the cost of array A is (10+10)*10+1=200 and the cost of array B is 10*10*100*8=80000. Thus array B is more important than array A and array A because of its low cost should not be considered for DTSE optimizations. The partitioning cuts the application at edges without costly data transfers so the global view is not affected so much. An example is depicted on Figure 3.6c where a large buffer exists between Partition 1 and Partition 2. Thus these partitions should be optimized together. Partition 3 is only connected via the small buffers, thus it can be handled separately in the optimization process.

3.4 GM extractors

To get from the IR (AST) to the GM model we need a GM extractor. Some extractors combine the extractor itself, i.e., translation from IR to GM, with the parser, i.e., translation from C code to IR. Examples of such extractors are Polyhedral Extraction Routines (PER) and LooPo [61]. PER is an in-house product developed at IMEC which is part of the program analysis and transformation ATOMIUM framework [229]. The PER extractor itself works in several phases.

The first phase is called normalization. It tries to transform the AST constructs that cannot be easily modeled in the GM or pose difficulties for keeping track of relations between the AST and GM before the extraction to GM. This normalization has the same goal as the preprocessing, i.e., enlarge the scope for the extraction. However, it is applied only locally and thus its scope is rather limited.

The second phase is extraction of the node tree for the desired functions. After that, the extractor builds the AST-GM link (AG) model which we can consider as a very simple AST consisting only of constructs that can be directly translated to the GM. If the AG model creation was successful, the data structure extraction, the iteration domain extraction, the variable domain extraction, the access and the access domain extraction, the boundary extraction, the lexicographical information extraction and the flow dependency extraction will start in the listed order.

The PER extractor is a standalone tool. It is sufficient for the extraction, but it is not sufficient for the (re)creation of the source file. It lacks data type information or information about functions, types and variables outside the specified function. However, the tool is now being integrated in the ATOMIUM GLT framework where these shortcomings are going to be eliminated.

LooPo [61] is a prototype implementation of loop parallelization methods based on the polyhedral model. The tool includes both the parser (and extractor) and a dependence analysis module. It implements space-time mapping methods for nested for loops [134] and has also the capability of dealing with while loops, for loops with unknown bounds at compile time and if conditions in the loop nest [89]. LooPo can generate efficient target code by using High Performance Fortran (HPF) as a back end to LooPo.

The other extractors start directly from a IR. Examples of such extractors are Polyhedral Extraction Routines from SUIF (pers) and WHIRL to Polyhedra (w2p). The pers extractor has been developed at KU Leuven [213] and transforms Standford University Intermediate Format (SUIF) representation to the GM. It basically consists of a pass in the SUIF compiler [7] that collects the same information as PER does. However, it also keeps pointers to the internal SUIF data structure that can be used during code generation. After integrating PER into the ATOMIUM GLT framework PER will also keep the links to the ATOMIUM AST. The w2p extractor is a part of the WRaP-IT library [19] and transforms the WHIRL representation to the GM. WHIRL is the IR of the Open Research Compiler (ORC). The extractors differ not only in the input IR the GM is extracted from but also in the robustness and the features they support. Nowadays, we see that the GM extractors win its place also in mature and heavily used compilers such as gcc. The proposal of GENERIC and GIMPLE IR [149] and Tree SSA [155] opened the gcc compiler framework towards high level

optimization and modeling such as the GLT and GM [23, 174].

3.5 Array dependency analysis tool

The array dependency analysis can be part of the extraction, e.g., in the ATOMIUM PER extractor or it can be a separate tool after the extractor. E.g., the simple dependency analysis in perl (sda.pl) and the advanced dependency analysis in perl (ada.pl) (in the current framework called just dependency analysis in perl (da.pl)) are separate tools after the pers extractor. The Petit tool which is part of the Omega project [120] is also a separate array dependency analysis tool. The array dependency analysis can be found in any framework based on GM [56, 71, 118, 46, 38, 77]. It computes three types of dependencies, i.e., flow dependencies, anti-dependencies and output dependencies, among the iteration domains of statements. The definitions of these different types of dependencies are below. We will assume that all the iteration domains have the same dimension. This is automatically obtained when placing the iteration domains in the Common Iteration Space (CIS) together with using the time (pseudo) dimensions to preserve the original scheduling for the extracted GM.

Definition 3.7 *A* flow dependency *is a data dependency between a definition (a production, a write) and a use (a consumption, a read) of the same array variable element. The array variable element has to be defined before it can be used.*

Definition 3.8 *An* anti-dependency *is a data dependency between a use (a consumption, a read) and a (next) definition (a production, a write) of the same array variable element. The array variable element has to be used before it is redefined.*

Definition 3.9 *An* output dependency *is a data dependency between a definition (a production, a write) and a (next) definition (a production, a write) of the same array variable element. The order of the definitions has to be kept so that the array variable element contains the right data after the two definitions (productions, writes) have finished.*

Definition 3.10 *The* flow dependence relation $\delta_{S1,S2}$ *between two statements S1 and S2 is the set of all pairs of iteration vectors* $(\vec{i}, \vec{j})$ *that exhibit a flow dependence, i.e., when a definition (a production, a write) in S1 and a use (a consumption, a read) in S2 access the same array variable element.*

Definition 3.11 *Given two d-dimensional vectors* $\vec{i}$ *and* $\vec{j}$, $\vec{i}$ *is* lexicographically smaller *than* $\vec{j}$, *denoted* $\vec{i} \prec \vec{j}$ *if and only if there exists some k,* $1 \leq k \leq d$, *where d is the number of dimensions, such that* $i_l = j_l$ *for* $1 \leq l < k$ *and* $i_k < j_k$.

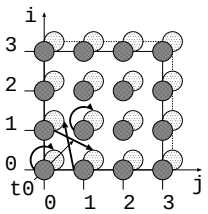
In multiple assignment code, we have to assure that the definition is executed before the use and that there is no intermediate write access to that array variable element, i.e.,

Original code

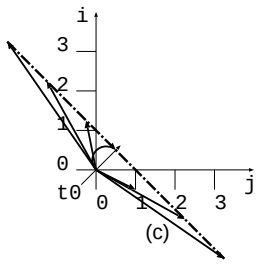
```
for(i=0;i<4;i++)
  for(j=0;j<4;j++)
    a[i][j] = in1();   //S1

for(i=0;i<4;i++)
  for(j=0;j<4;j++)
    b[i][j] = a[j][i]; //S2
```

(a)



(b)



(c)

Figure 3.7: GM example for two loop nests: (a) Original code; (b) The iteration domains and the dependence distance vectors in the CIS; (c) Dependence polytope.

$$\begin{aligned}\delta_{S1,S2} \quad = \quad & \{(\vec{\imath},\vec{\jmath}) \mid \vec{\imath} \in ID_{S1} \wedge \vec{\jmath} \in ID_{S2} \wedge W_{S1}(\vec{\imath}) = R_{S2}(\vec{\jmath}) \wedge \\ & \vec{\imath} \prec \vec{\jmath} \wedge \neg(\exists \vec{\ell} \in ID_{SX} : W_{SX}(\vec{\ell}) = W_{S1}(\vec{\imath}) \wedge \vec{\imath} \prec \vec{\ell} \prec \vec{\jmath})\}\end{aligned}$$

where ID_{S1} and ID_{S2} are iteration domains of statements S1 and S2, W_{S1} is the definition of an array in the statement S1, R_{S2} is the use of an array in the statement S2 and W_{SX} is an intermediate write access to that array variable in the iteration domain ID_{SX} of statement SX. Statement SX is executed after statement S1 and before statement S2.

Similarly we can define the anti-dependency relation and the output dependency relation. If we consider code in Dynamic Single Assignment conversion (DSA) form [69, 209] each array element is written only once. Thus, in such a code there are neither anti-dependencies nor output dependencies. Because each array element is written only once also the flow dependence relation is simplified.

$$\delta_{S1,S2} = \{(\vec{\imath},\vec{\jmath}) \mid \vec{\imath} \in ID_{S1} \wedge \vec{\jmath} \in ID_{S2} \wedge W_{S1}(\vec{\imath}) = R_{S2}(\vec{\jmath})\}$$

Definition 3.12 *The* dependence distance vector $\vec{d}$ *is the difference between* $\vec{\jmath}$ *and* $\vec{\imath}$*, where* $\vec{\imath} \in ID_{S1}$*,* $\vec{\jmath} \in ID_{S2}$ *and* $\vec{\imath}\delta_{S1,S2}\vec{\jmath}$*.*

Definition 3.13 *The* dependence polytope $DP_{S1,S2}$ *of a dependence* $\delta_{S1,S2}$ *is a convex hull of all dependence distance vectors between S1 and S2, i.e.,*

$$DP_{S1,S2} = conv\{\vec{d} \in \mathbb{Z}^d \mid \exists(\vec{\imath},\vec{\jmath}) \in \delta_{S1,S2} : \vec{d} = \vec{\jmath} - \vec{\imath}\}$$

The concept of flow dependence, flow dependence relation, dependence distance vector and dependence polytope is depicted in Figure 3.7. It shows a simple code we already used as an example in Section 3.1. In the first loop nest, in statement S1, the array element a[i][j] is defined and later in the second loop nest in statement S2 the array element a[j][i] is used. If (i,j)=(0,0) in the first loop nest and (i,j)=(0,0) in the second loop nest or (i,j)=(0,1) in the first loop nest and (i,j)=(1,0) in the second loop nest the same array element is accessed in the memory so there exists a flow dependency between the write in the first loop nest and read in the second loop

nest. The code is in DSA form so we can use the simplified definition to compute the flow dependence relation:

$$\begin{aligned}\delta_{S1,S2} = \quad & \{(t_{0S1}, i_{S1}, t_{1S1}, j_{S1}, t_{2S1}, t_{0S2}, i_{S2}, t_{1S2}, j_{S2}, t_{2S2}) \mid \\ & 0 \leq i_{S1}, j_{S1}, i_{S2}, j_{S2} \leq 3 \wedge t_{0S1} = 0 \wedge t_{0S2} = 1 \wedge \\ & t_{1S1}, t_{2S1}, t_{1S2}, t_{2S2} = 0 \wedge i_{S1} = j_{S2} \wedge j_{S1} = i_{S2}\}\end{aligned}$$

After simplification of this set, e.g., using Omega calculator [121], we get the flow dependence relation:

$$\delta_{S1,S2} = \{(0, i_{S1}, 0, j_{S1}, 0, 1, j_{S1}, 0, i_{S1}, 0) \mid 0 \leq i_{S1}, j_{S1} \leq 3\}$$

The (flow) dependence distance vector is the difference between two iterations that access the same array element in the memory, i.e., when (flow) dependence relation is valid. E.g., when (i,j)=(0,0) for the first loop nest and (i,j)=(0,0) for the second loop nest the dependence relation is valid and the difference (considering also time dimensions) is (1,0,0,0,0). When (i,j)=(0,1) for the first loop nest and (i,j)=(1,0) for the second loop nest the difference is (1,1,0,-1,0). Some of the dependence distance vectors are depicted in Figure 3.7b. Note, that the t_1 and t_2 dimensions were omitted here. They are always 0 for S1 and S2 and thus do not have any influence on the computed sets.

We can also compute the dependence polytope as defined above. All dependence distance vectors and the dependence polytope are in Figure 3.7c.

$$\begin{aligned}DP_{S1,S2} \quad = \quad & conv\{(d_0, d_1, d_2, d_3, d_4) \mid \exists(t_{0S1}, i_{S1}, t_{1S1}, j_{S1}, t_{2S1}, t_{0S2}, i_{S2}, t_{1S2}, \\ & j_{S2}, t_{2S2} \mid 0 \leq i_{S1}, j_{S1}, i_{S2}, j_{S2} \leq 3 \wedge t_{0S1} = 0 \wedge t_{0S2} = 1 \wedge t_{1S1}, \\ & t_{2S1}, t_{1S2}, t_{2S2} = 0 \wedge d_0 = t_{0S2} - t_{0S1} \wedge d_1 = i_{S2} - i_{S1} \wedge d_2 = \\ & t_{1S2} - t_{1S1} \wedge d_3 = j_{S2} - j_{S1} \wedge d_3 = t_{2S2} - t_{2S1})\}\end{aligned}$$

After simplification of this set we get the dependence polytope (see also the dash-dot line in Figure 3.7c).

$$DP_{S1,S2} = \{(1, d_1, 0, -d_1, 0) \mid -3 \leq d_1 \leq 3\}$$

After computing the iteration domains of the statements, introducing the time dimensions, putting the iteration domains to the CIS and computing the dependences we have a compact representation of the program. The program can then be represented as a Polyhedral Dependency Graph (PDG):

Definition 3.14 *A PDG is a tuple $G = \langle V, E, \mathcal{P}, \Delta \rangle$ consisting of the following elements:*

- *V is the set of nodes where each node represents a statement in the original program.*
- *Each node $n \in V$ is adorned by a polytope representing the iteration domain of that statement. The set of all such polytopes is $\mathcal{P}$.*

- *E is the set of edges. There is an edge between node $n_1 \in V$ and $n_2 \in V$, denoted as (n_1, n_2) if and only if there is a (flow) dependence relation $\delta_{S1,S2}$ between the two statements S1 and S2 the two nodes n_1 and n_2 represent.*
- *Each edge e = (n_1, n_2) is adorned by the dependence polytope $DP_{S1,S2}$. The set of all such polytopes is Δ.*

For the PDG, the time dimensions determining the scheduling of the statements are not necessary. However, then the original structure of the program is not known. During the GLT, the placement phase, which places the individual iteration domains in the new CIS, is required. This phase creates a new program structure from scratch based on the dependencies so that they are not violated. Both approaches, creating of the CIS based on the full information about the program structure and then adapting the known structure of the program as well as the construction of the program from scratch will be discussed in the next section.

3.6 Transformations on the GM

GLT changes the execution ordering of the statements without violating the data (flow) dependencies in the program. The execution ordering of the statements can easily be changed using the affine transformations. The affine transformation $\mathcal{T}_S$ maps each iteration vector $\vec{\imath}$ in the iteration space of statement S to a new iteration vector $\mathcal{T}_S(\vec{\imath})$

$$\mathcal{T}_S : \vec{\imath} \mapsto \mathcal{T}_S(\vec{\imath}) = T\vec{\imath} + \vec{t}$$

where the T is a linear transformation matrix and $\vec{t}$ is the translation vector. The T matrix does not to have be unimodular, i.e., square integer matrix with determinant +1 or -1. However, if the T matrix is not square then the transformed space can be larger (or smaller) dimension. Also, if determinant det T $\neq \pm$ 1, then the number of points in the domain will be scaled by the determinant. Thus, the transformation for non-unimodular matrix will not be a bijection, i.e. one-to-one mapping. In this work we will consider only unimodular linear transformation matrices. The affine transformation $\mathcal{T}_{S2}$ applied on the iteration domain S2 in Figure 3.8a

$$\mathcal{T}_{S2}(\vec{\imath}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \vec{\imath} + \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

is demonstrated in Figure 3.8b,c. Note, that the ordering of the dimensions in the T matrix and in the $\vec{t}$ is (t_0, i, t_1, j, t_2). Figure 3.8b depicts the linear transformation T and Figure 3.8c depicts the translation $\vec{t}$. Note the small shift of the whole domain to the origin of the coordinate system representing the shift $(-1, 0, 0, 0, 0)$. We filled the node (0,3) with darker fill to be able to observe the transformation of one particular

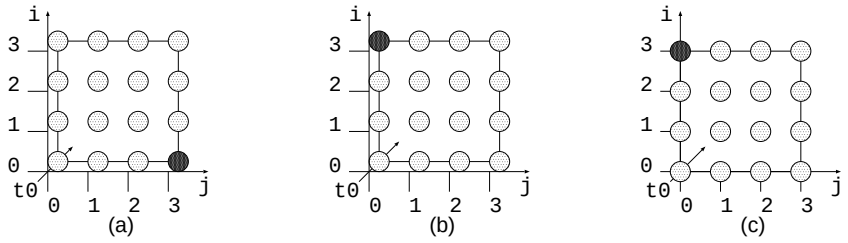


Figure 3.8: (a) Original iteration domain of S2; (b) The iteration domain of S2 after linear transformation; (c) The iteration domain of S2 after linear transformation and translation.

point in the iteration domain (and mainly to see the effect of the linear transformation).

In Figure 3.9 we can observe the effect of this transformation on the program (Figure 3.9a,d), on the iteration domain of S2 in the CIS and the dependencies (Figure 3.9b,e), and on the dependency polytope (Figure 3.9c,e). The linear transformation improved the regularity of the dependence in the program. This can be seen by the reduction of the dependency polytope dimension from 1D (line) to 0D (point). Note, that some approaches for regularity optimization are based on testing the dependency polytope dimension for most common linear transformations (interchange, inverse) [44]. The linear transformation we applied in our example was also the interchange of the two loops in the second loop nest. Despite improved regularity, the production and consumption of the array elements are still far apart. That is why we applied the translation vector. The -1 in the first time dimension t_0 of the translation vector $\vec{t} = (-1, 0, 0, 0, 0)$ fuses the two loop nests into one loop nest. Note that this transformation would not be possible without the regularity optimization. Still, the S1 has to precede S2 to keep the geometrical representation of the program valid. In the sequel we define when the geometrical representation of the program is valid and how do we recognize valid affine transformation:

Definition 3.15 *The geometrical representation of the program is valid if and only if all dependence distance vectors in the program are strictly positive, i.e.,* $\forall \vec{d} = \vec{j} - \vec{i} \succ \vec{0}$*. That means there should not be a consumption/use/read of an array element before the production/definition/write of that array element.*

Definition 3.16 *An affine transformation is valid if and only if the geometrical representation of the program after the affine transformation is valid.*

The validity of the transformed program in our example in Figure 3.9 is achieved by fission, i.e., loop splitting, of the last time dimension, so in reality the $\vec{t} = (-1, 0, 0, 0, 1)$. The +1 in the last dimension is necessary to distinguish between the statements in the innermost loop. The locality criterion tries in general to place production and consumption as close as possible and as still allowed by the validity constraint. However, in Chapter 6 we will show that it is not optimal from other aspects and can lead to a suboptimal solution when the application is mapped on a platform.

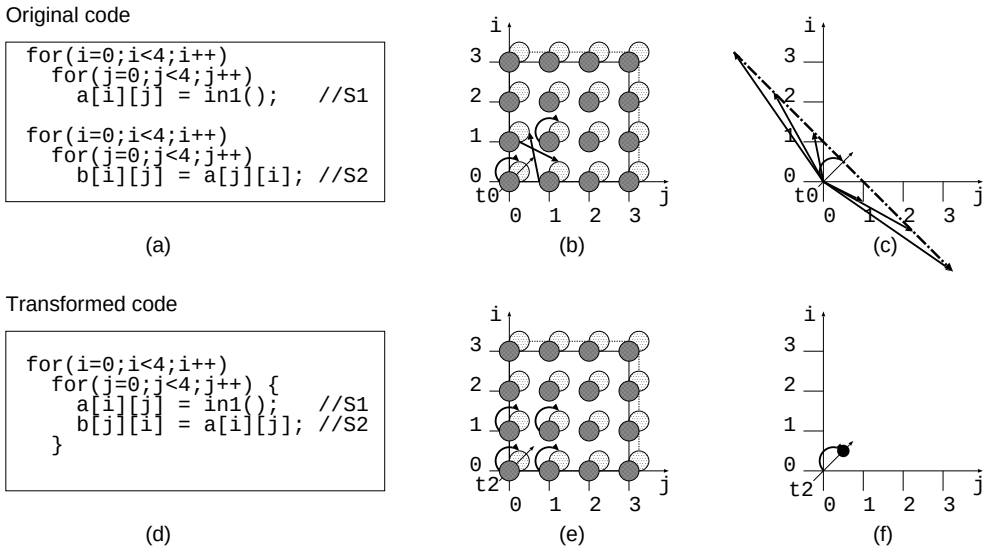


Figure 3.9: (a) Original code; (b) The CIS with the dependency vectors (original code); (c) The dependency polytope (original code); (d) Transformed code; (e) The CIS with the dependency vectors (transformed code); (f) The dependency polytope (transformed code).

An alternative approach is to start from the PDG without time dimensions and without the CIS. In this representation the individual iteration domains are transformed to improve the regularity of memory accesses, e.g., by selecting linear transformations that lead to the reduction of the dimensions in the dependency polytope. Only after the linear transformation phase the linearly transformed polytopes are combined together based on the locality criterion into a CIS and the ordering is selected. This approach has been presented by Danckaert [44]. Verdoolaege [213] has shown that for the locality improvement on a non-parallel architecture, this approach is equivalent with the incremental approach [213] where the original structure of the program is preserved in the CIS and the polytopes are incrementally linearly transformed and translated[1].

3.7 GM scanner

To get from the GM to the IR we need a GM scanner (see Figure 3.4). The full generation of the transformed IR (IR^T in Figure 3.4) is performed in two steps. In the first step, a simple IR is constructed. In the second step, the real IR is constructed. In some frameworks the second step contains also the **combiner** which combines the transformed parts of IR with the parts that have not been transformed.

[1]The previous approach of Danckaert is equivalent for architectures with Instruction Level Parallelism (ILP) scheduling only, where only time scheduling is present. However, for architectures with Data Level Parallelism (DLP) scheduling where also space dimension is present (like systolic arrays) Danckaert's approach is considered more general.

The GM scanners are usually based on the algorithm of Quilleré et al. [178]. Examples of the scanners based on this algorithm are LoopGen, Chunky Loop Generator (CLooG), WHIRL Loop Generator (WLooG), and SUIF Loop Generator (sloog). LoopGen is the original C++ loop generation tool and library developed by Quilleré et al. [178]. It takes the GM and produces the corresponding loop nest structure. However, the statement information is not present in this approach. CLooG is the reimplementation of LoopGen, which is easier to link into an application and includes some additional code generation options [18]. But, the statement information is also not present. The WLooG contained in WRaP-IT library uses CLooG to generate a WHIRL representation of the code after transformations. The sloog which is also based on CLooG, generates the SUIF representation of the code after transformations. It contains also pointers to the statements in the SUIF representation so the whole code can be (re)generated after transformations [213]. Some scanners, e.g., CLooG, contain also the dumper from the IR to the transformed C code. However, they do not contain the combiner, thus the non-geometrical part cannot be merged into the transformed code. The other scanners like sloog contain the combiner and leave the dumping to the external tool, e.g., SUIF to C (s2c) tool.

There are also code generation algorithms different from Quilleré et al. [178]. The code generation was first solved by Ancourt and Irigoin [9]. They used the Fourier-Motzkin elimination technique to compute loop bounds. The code generators derived from this technique are the LooPo code generator and the code generator in the Omega library [118]. However, for complex situations, the best solution is the Quilleré et al. algorithm [178].

In the following subsection we first explain the basics of Quilleré et al. algorithm. We will use the iteration domain descriptions from Figure 3.9e where the iteration domain of statement S2 has been shifted by 2 iterations in i and j dimensions and the time dimensions have been ignored. Such a situation is depicted in Figure 3.10 left. Later in the section we show how adding time dimensions affects the result of the scanning. Finally we will discuss the different options that can be used in the scanning process resulting in more or less compact output code.

3.7.1 Quilleré et al. algorithm

The Quilleré et al. algorithm generates loop levels by projecting the polyhedra onto the corresponding dimension. Next, it splits the projection into disjoint polyhedra and it sorts the resulting polyhedra to respect the lexicographic order. Lastly, it recursively generates loop nests that scan each polyhedron. This is demonstrated on a simple example of two iteration domains S1 : $\{[i,j] : 0 \leq i,j \leq 3\}$ and S2 : $\{[i,j] : 2 \leq i,j \leq 5\}$ in Figure 3.10. In this figure we first project the outermost dimension into disjoint polyhedra creating three disjoint 'i' iteration domains, $0 \leq i \leq 1$ where only S1 is executed, $2 \leq i \leq 3$ where both, S1 and S2 are executed and $4 \leq i \leq 5$ where only S2 is executed.

After this partitioning of the i dimension we look at the 'i' regions we created and split the projection of the second 'i' region into disjoint polyhedra for each 'j' region, i.e., the innermost dimension. This is demonstrated in Figure 3.11. If we look carefully at the iteration domain $\{i,j : 2 \leq i,j \leq 3\}$ in the figure, we cannot really say

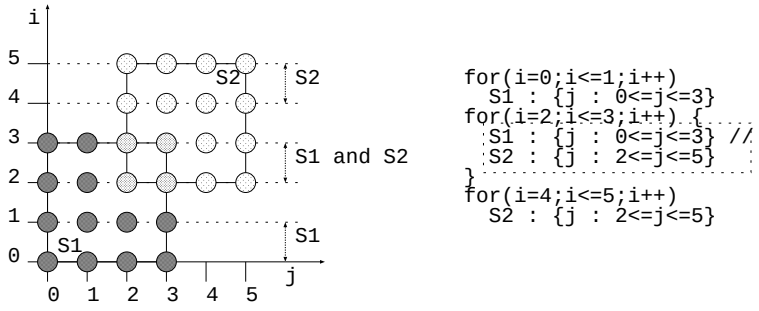


Figure 3.10: Quilleré et al. algorithm: Projecting the first dimension into disjoint polyhedra.

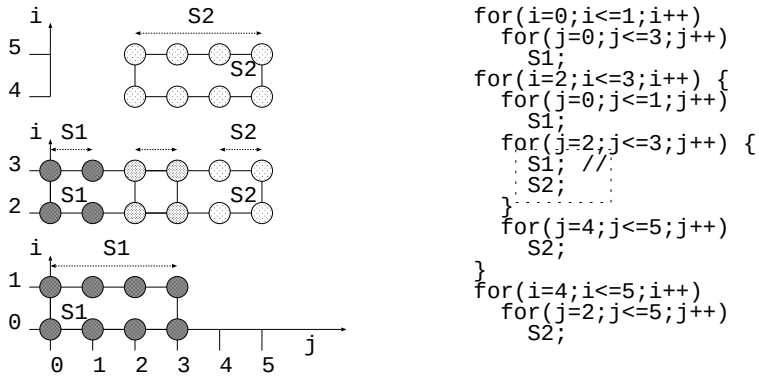
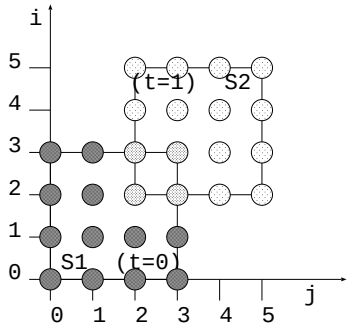


Figure 3.11: Quilleré et al. algorithm: Projecting the second dimension into disjoint polyhedra.

based on the geometrical information we have, which statement should be executed first during these iterations, if S1 or S2. To decide on that we introduced in Section 3.1 time dimensions in order to determine the scheduling of the statements in these cases. In the next subsection we show how time dimensions affect the scanning results.

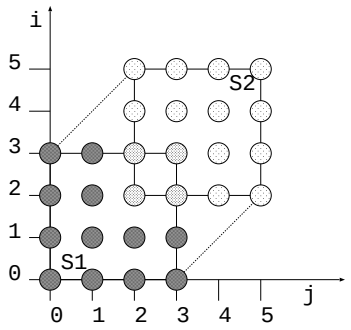
3.7.2 Scanning with the time dimensions

Time dimensions were introduced in Section 3.1. They decide on the scheduling of the iteration domains at a particular loop level when the ordering at that particular loop level cannot be determined. In Figure 3.11 we have implicitly put statement S1 before statement S2 in the disjoint polyhedra {[i,j] : 2≤i,j≤3} (see also the second 'i' loop and second 'j' loop within that 'i' loop in the corresponding code). However, in reality we cannot say if S1 is before S2 or vice versa. Thus we implicitly considered one innermost time loop 't' which was 0 for S1 and 1 for S2. After projection and splitting into disjoint polyhedra this corresponds to the code in Figure 3.11. If we do an interchange and put the time dimension as the outermost loop, first we split the projection to time loop into disjoint polyhedra S1: {[[i,j] : 0≤i,j≤3} and S2 : {[i,j]




```
for(t=0;t<=0;t++)
  for(i=0;i<=3;i++)
    for(j=0;j<=3;j++)
      S1;
for(t=1;t<=1;t++)
  for(i=2;i<=5;i++)
    for(j=2;j<=5;j++)
      S2;
```

Figure 3.12: Quilleré et al. algorithm: Projecting of the time dimensions.




```
for(i=0;i<=5;i++)
  for(j=max(0,i-3);
      j<=min(i+3,5);j++) {
    if(j<=3)
      S1;
    if(j>=2)
      S2;
  }
```

Figure 3.13: Quilleré et al. algorithm: Compact code.

: $2 \leq i,j \leq 5\}$. After the projection of the second 'i' and third 'j' dimension we get the code in Figure 3.12. Thus, with the time dimensions we have all the information to construct the corresponding ordering of the statements.

3.7.3 Scanning for compact code

The code in Figure 3.11 does not have any guards, however it consists of several loop nests and thus the code size is relatively large. If we would like to scan for compact code, we should use a perfectly nested loop to scan a convex superset (such as the bounding box) of union of all statements. Because the domain scanned by this loop is also a superset of each statement's domain, we cannot unconditionally execute the statements within the loop body. Rather, each statement must be guarded by conditions which test that the current loop index vector belongs to the iteration domain of that statement. The superset of the union scanned by the perfect loop may be simply the bounding box of the domain as done by the LooPo code generator [91] or the convex closure of the domain as done by the Omega code generator [119]. Figure 3.13 illustrates the scanning result of the same GM as in Figure 3.11 using convex closure of the domain resulting in the compact code.

Despite the compact code, the solutions based on the perfect loop do have severe limitations. In the code also the empty iterations are executed. E.g., in Figure 3.13 the

iterations in the triangular domains {[i,j] : 3≤i≤5 && i-3≤j≤2} and {[i,j] : 0≤i≤2 && 3≤j≤i+3} are empty and neither statement S1 nor statement S2 are executed during those iterations. There is also a control-flow overhead where each loop iteration must test the guards of each guarded statements. Also, the *min* and *max* function create the control-flow overhead. To obtain good trade-off between the control-flow overhead and size, the bounding box approach is preferred for the multimedia real-life applications. The disadvantage is the higher empty iteration count. However, it is not much higher in relative measures when compared to the convex closure approach. The bounding box approach also reflects the code layout when the loop transformation is done manually and is also preferred by the designers. The redundant control-flow can be optimized later using other techniques such as [94, 158, 159, 66]. There is also a possibility to choose the loop nest level from which on a convex superset is scanned resulting in the trade-off between the code size and control-flow overhead represented by empty iterations and guarded execution.

3.8 Postprocessing

Although beneficial for memory optimizations, the preprocessing usually introduces instruction and control-flow overhead, e.g., code duplicating after inlining and hiding scalars in the Layer 3 code, extending the loop bounds when considering the worst case, moving *if* conditions down in loop hierarchy and thus executing it more often when creating Layer 3 functions, function calls to Layer 3 functions, etc. To undo this overhead postprocessing is needed. After the transformations, scanning of geometrical model part and merging it with the non-geometrical model part the rewriting and hiding done in the preprocessing step has to be partially undone to ensure optimality of transformed application. In this step the application needs to be postprocessed (see Figure 3.4). Postprocessing (as well as preprocessing) are performed on the internal representation, e.g., on the AST.

To reduce the created overhead the postprocessing techniques and tools like the one of Falk et al. [66] for optimizing control flow or RACE like techniques such as code hoisting and Common Subexpression Elimination (CSE) techniques for elimination of code duplication [94] can be used to remove the negative preprocessing effects and further improve the code quality. Also Layer 3 code has to be expanded back to Layer 2 code to enable the global scope to the traditional compiler.

3.9 Open issues

The hierarchical rewriting is crucial for separation of the different layers of the code. To recap, the first layer contains process control flow, the second layer contains loop hierarchy and indexed signals and the third layer contains arithmetic, logic and data-dependent operations. The loop transformations are only applied to the second layer. To extract the second layer manually is not feasible for huge programs. Thus it is crucial to extract this layer automatically. In real-life code especially the Layer 2 and Layer 3 are intermingled together. To provide an automatic layering technique is a challenging task. We address this task in Chapter 4 where we propose

a systematic way of AST manipulation to hierarchically rewrite and hide data dependent conditions. The automatic rewriting is crucial to enable loop transformations based on the existing geometrical models and also use tools based on that model.

In the past the applications where mostly static with limited amount of data-dependent conditions and non-static constructs that are not supported by the GM. Thus, the GMs extracted were large and a lot of times the whole application corresponded to one big GM. However, current real-life applications have many more non-SCoP parts, i.e., parts that are not extractable to GM even after preprocessing. Also, many GMs have a very small size and contain only one particular kernel. To deal with this problem we propose an intra-task scenario approach in Chapter 5. The scenario approach specializes the code to several code versions with bigger SCoPs on which it can perform more optimizations at compile time. The decision which specialized (and optimized) code version will be used is postponed to the run-time.

Nowadays, the GLT step targets regularity and locality issues (or uses another particular cost function). In Chapter 6 we show that this does not have to be the optimal solution and that there exist trade-offs during the GLT phase. We discuss several trade-offs in this chapter. To our knowledge, so far nobody has focused on the trade-off issue for different costs during the loop transformations performed on the GM. The trade-offs between the energy cost and the performance were observed in the iterative compilation research. However, the authors propose to use a combined energy-performance factor [83] which then still results in a one-dimensional solution. Other authors in the same area [125, 2] trade-off the quality of the result vs. the time needed to obtain this result. However, the trade-offs among different properties of the program are still not explicitly considered in those approaches.

CHAPTER 4

Preprocessing for innermost conditions

C'est une grande habileté que de savoir cacher son habileté.
François de La Rochefoucauld
(1613-1680)

Every recent high level low-power design methodology contains a loop transformation stage. Loop transformations are one of the basic optimization techniques used at the high-level to improve source implementation quality towards low power. This technique either improves the parallelization opportunities at the instruction or data level or even task level, or it improves the locality of data accesses such that data can be stored in lower levels of the memory hierarchy, resulting in significant power gains.

Parallelization is beneficial for extracting more instruction and/or data level parallelism. Improved parallelism causes reduction in cycles. When lowering the clock speed or voltage it also has a positive effect on energy consumption.

Data locality is beneficial in two ways. First, by decreasing the distance between production and consumption of the same element, the life-time of that element is shortened. The memory allocated to this element is freed earlier for other data elements, typically reducing the total memory requirement [49]. Second, by decreasing the distance between multiple consumptions of the same element, the local copy with short life-time is typically placed in the smaller and faster memory [200]. This reduces the number of accesses to power hungry large memories that are further away from the processor in the memory hierarchy.

The loop transformations are nearly always performed on the GM which is very effective in dealing with complex transformations. However, the GM imposes strict limitations on the input code. We discussed the GM and its limitations in Section 3.1.

To overcome the limitations of the GM preprocessing/pruning is applied on the source code beforehand. The preprocessing/pruning step is essential, because it

reduces complexity, increases exploration freedom and hides undesired constructs of the original source code. This is true both for the designer and for tools. Another solution to overcome the limitations of the GM would be to extend the GM. E.g., parameterizable geometrical representation, that is data dependent can be used as an extension of traditional GM to support also data dependent constructs. However, this solution is not acceptable because the Integer Linear Programming (ILP) solvers used for the optimizations can not deal effectively with such an extended GM causing prohibitively long run-times of the solvers. Thus using preprocessing/pruning is preferred over such extension of the GM.

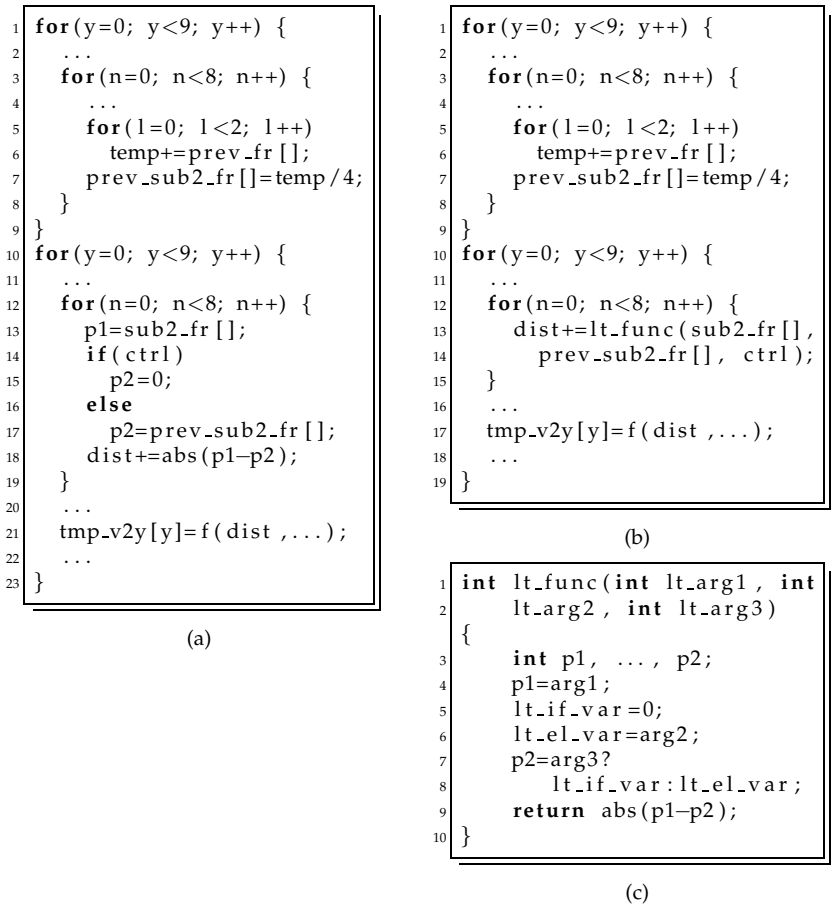
The preprocessing/pruning step consists of several substeps. They are hierarchical rewriting, hiding of undesired constructs, code expansion, array/pointer data-flow analysis, data-flow chain removal, weight-based removal and partitioning. For some of these substeps there exist or are under development systematic techniques. Absar et al. [1] proposed selective code inlining for code expansion. Vanbroekhoven et al. [209] are working on array/pointer data-flow analysis and data-flow chain removal which removes avoidable buffers in the program (see Figure 3.6a in Section 3.3). However, hierarchical rewriting and hiding of undesired constructs which are two basic and error prone substeps were left to the designer and are handled ad-hoc.

In this chapter we would like to handle these two important substeps in a systematic way. We propose a technique which manipulates Abstract Syntax Tree (AST) to hierarchically rewrite and hide data dependent conditions. The technique is based on automatic rewriting of an application in three layers. The first layer contains process control flow, the second layer contains loop hierarchy and indexed signals and the third layer contains arithmetic, logic and data-dependent operations. The loop transformations are only applied to the second layer. Afterwards layer three is inlined and propagated again into layer two. The automatic rewriting is crucial to enable loop transformations based on the existing geometrical models and also use tools based on those models. We contributed to the area of preprocessing with technique for hierarchical rewriting and hiding of undesired constructs and its implementation in prototype tool.

4.1 Problem definition

The kernels of modern applications contain the mixture of the second and third layer. The first layer, containing process control-flow, is usually very well separated by the designer. Also the exploration and optimization of the first layer is not the task of the traditional DTSE methodology, called processor level DTSE. The first layer is the target of the optimization of task level DTSE dealing with concurrent threads which is out of the focus of this dissertation and therefore we do not further discuss this issue. We will demonstrate the problem of separation of the second and third layer in processor level DTSE on an example in Figure 4.1.

Figure 4.1a shows a part of the QSDPCM video encoder [193]. The two nested loops represent two kernels, i.e., subsampling by 2 and motion estimation by 2, of the video encoder. The kernels are usually well optimized by the designer. One objective of a low-power design methodology is inter-kernel optimization where large



```
for(y=0; y<9; y++) {
  ...
  for(n=0; n<8; n++) {
    ...
    for(l=0; l<2; l++)
      temp+=prev_fr[];
    prev_sub2_fr[]=temp/4;
  }
}
for(y=0; y<9; y++) {
  ...
  for(n=0; n<8; n++) {
    p1=sub2_fr[];
    if(ctrl)
      p2=0;
    else
      p2=prev_sub2_fr[];
    dist+=abs(p1-p2);
  }
  ...
  tmp_v2y[y]=f(dist,...);
  ...
}
```

(a)

```
for(y=0; y<9; y++) {
  ...
  for(n=0; n<8; n++) {
    ...
    for(l=0; l<2; l++)
      temp+=prev_fr[];
    prev_sub2_fr[]=temp/4;
  }
}
for(y=0; y<9; y++) {
  ...
  for(n=0; n<8; n++) {
    dist+=lt_func(sub2_fr[],
      prev_sub2_fr[], ctrl);
  }
  ...
  tmp_v2y[y]=f(dist,...);
  ...
}
```

(b)

```
int lt_func(int lt_arg1, int
    lt_arg2, int lt_arg3)
{
    int p1, ..., p2;
    p1=arg1;
    lt_if_var=0;
    lt_el_var=arg2;
    p2=arg3?
       lt_if_var:lt_el_var;
    return abs(p1-p2);
}
```

(c)

Figure 4.1: Data dependent conditions as limiting factor for GLT: (a) original code segment that cannot be fused; (b) code segment after applying our technique that can be fused; (c) layer three code.

energy savings can be obtained. If we look at the second loop nest we can see a data dependent condition in the kernel. A GLT framework has a problem to analyze the code within data dependent conditions. E.g., it has problems to analyze the *prev_sub2_fr* (previous frame subsampled by 2) read access (the write access can be analyzed easily). To enable analyzing also this access inside the condition together with the scalars *p1* and *p2* (which are not targeted by high-level memory optimization) the data dependent condition (line 14) should be hidden in the third layer of the application (see Figure 4.1c). The second layer should contain only loop hierarchy and indexed signals as depicted in Figure 4.1b. Although *dist* is not an indexed signal yet, after conversion to Dynamic Single Assignment conversion (DSA) [209] (which is one of the preprocessing substeps) it will be. The *dist* scalar should also remain in second layer because its lifetime goes beyond the boundaries of the basic block (after if-conversion) that has been encapsulated into the *lt_func()* function. We assume that the basic blocks are already well optimized and do not need any further GLT. Thus they can be encapsulated into the *lt_func()* functions.

The code in Figure 4.1b can be extracted into the GM and thereafter GLT can be applied. Compared to the original code fragment in Figure 4.1a, the code in Figure 4.1b has the data dependent condition and the scalars *p1* and *p2* hidden (see Figure 4.1c). Also the code granularity has been raised to only loop and array signal level. This was achieved by elimination of "avoidable" scalars (scalars that are not input or output scalars of the basic block, like *p1* and *p2*) and hiding arithmetic, logic and bitwise operations. Section 4.2 explains how to go systematically from the code in Figure 4.1a to the code in Figure 4.1b, and it provides algorithms for this code transformation.

4.2 Hierarchical rewriting and hiding of data dependent conditions

In our approach, we assume that the potential exploration space is in one function. This can be achieved by systematically applying Selective Function Inlining [1]. We also consider pointer free code, which can be achieved by Pointer Analysis and Conversion techniques [184, 75]. Also, the data dependent addressing should be substituted by extreme cases (lower and upper bound). These bounds can be obtained either via dynamic profiling [117] or via analytic methods. To use extreme bounds usually over-constrains the dependencies computed later during loop transformations resulting in non-optimal application of these transformations. One solution to avoid this over-constraining of dependencies is using scenario methodology as is explained in Chapter 5.

Such a preprocessed application has to be rewritten in three layers. Our hierarchical rewriting and hiding of data dependent conditions technique which separates the second and the third layer consists of three main steps:

1. Moving the data dependent conditions to innermost loops.
2. Rewriting the innermost if conditions to ternary operators (if-conversion).
3. Encapsulation of if-converted basic block computations into functions.

Each of these steps will be discussed in detail in following subsections. After applying these steps we obtain code without data dependent conditions and with appropriate granularity for loop transformations.

4.2.1 Moving data dependent conditions to innermost loops

Moving data dependent conditions to the innermost level enlarges the exploration space. It looks very similar to ignoring the condition at the corresponding level as in the case of current approaches. However the condition is now kept at the innermost level and after the loop transformation it can be hoisted up again to the appropriate level. Other approaches transform the data dependent conditions to data independent conditions [114, 126]. However, these approaches use worst case situation and thus they loose some information about the code. In this respect they are similar to ignoring the condition.

In Figure 4.2 we show an example of condition moving. In Figure 4.2a we see the original code where the data-dependent condition is at the middle level. In Figure 4.2b the data dependent condition was moved to the innermost level, and thus it is prepared for two new steps, namely the rewriting of the condition to ternary operator and after that the encapsulation of the if-converted basic block computations into functions.

```
1 for(bx=0; bx<2; bx++) {
2   if(bl_last_ind[bx]>=0) {
3     for(cx=0; cx<8; cx++) {
4       bl_buff[bx][cx] = ...;
5     }
6   ...
7 }
```

(a)

```
1 for(bx=0; bx<2; bx++) {
2   for(cx=0; cx<8; cx++) {
3     if(bl_last_ind[bx]>=0) {
4       bl_buff[bx][cx] = ...;
5     }
6   ...
7 }
```

(b)

Figure 4.2: Moving data dependent conditions to innermost loop: (a) original code fragment code; (b) code fragment after condition moving.

At this point it is important to clarify about moving outermost conditions innermost. Those outermost conditions that are be part of the scenario approach discussed in Chapter 5 are not moved. The "scenario" conditions contain at least one of the relevant parameters for scenarios in the expression. The relevant parameters are discovered by scenario parameter discovery techniques [84, 86].

When moving a condition down in the loop hierarchy during preprocessing, the condition is executed more times compared to the original place. This is depicted on an example in Figure 4.3a where the condition is executed once and Figure 4.3b where the condition is executed six times. To avoid this redundant conditional execution, after the transformations we move data dependent conditions up again. If prohibited by the applied global loop transformations (e.g., loop merge as depicted in Figure 4.3b), we can still use the loop nest splitting principle [66] to move the data dependent condition up. However, this systematic optimization increases the code size. Thus it is sometimes beneficial not to perform loop transformations which blocks the condition to be moved up again without code size increase; so a

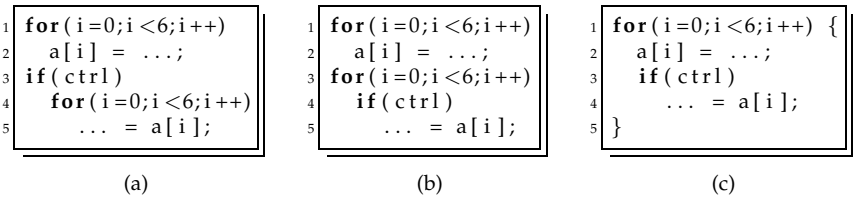


Figure 4.3: Example when moving back of data dependent condition is prohibited by LT: (a) original code fragment code; (b) code fragment after condition moving; (c) loop transformed code fragment where the if condition cannot be moved outside the for loop.

trade-off is involved. Future improvement to the existing loop transformation framework should use the control-flow complexity estimates during loop transformations. These estimates should avoid loop transformations that block the conditions which cause large overhead. This is one area for the future research.

Summarizing, we move the data dependent condition to innermost loop if:

1. it is not a scenario condition and either
2. (a) it can be moved up (back to its original position) after LT or
   (b) it is not too costly to put (and keep) it at the innermost level (costly in terms of number of additional evaluations of the condition)

The next two subsections explain how to hide such a moved data dependent condition at the innermost level by rewriting it to ternary operator and encapsulation of if-converted basic block computations into function.

4.2.2 Rewriting the innermost if conditions to ternary operators

Eliminating innermost data dependent conditions is represented by its transformation to a ternary operator. This transformation replaces the control-flow structure not supported by the geometrical model to an operator. This operator is an evaluation statement which is supported by current loop transformation flows. In the case of only assignments in the *if* condition body this one-to-one mapping is always possible. If two different variables or array elements are written in both branches of the *if* condition, two ternary operators are needed; each captures one variable/array element write.

Figure 4.4 shows an example of the rewriting of innermost *if* condition demonstrated on a part of the QSDPCM application [28]. We start following back the dependencies from list of array definitions to capture the data-dependent innermost conditions that contribute to the array data flow. We are not interested in other innermost conditions. The innermost condition detected during following back the flow dependencies is automatically rewritten to a ternary operator. The operator is then encapsulated into the particular function. In Figure 4.4a in the statement at Line 9 the use of variable *p2* is detected during following back the flow dependencies. If we ask for its definition we cross the control-flow boundary. We obtain two potential definitions depending on the *ctrl* control expression. If one potential definition,

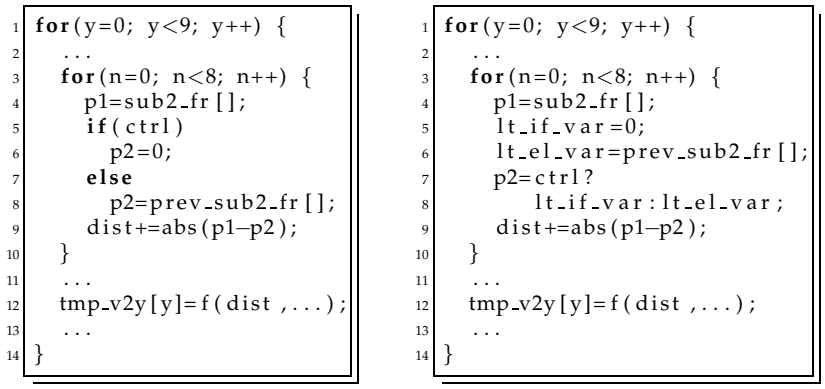


Figure 4.4: Rewriting the innermost data dependent if condition to ternary operator: (a) original code fragment code; (b) code fragment after condition rewriting.

i.e., one branch of the condition, is missing we add this branch into the Abstract Syntax Tree (AST). In the added branch we construct an identity copy statement, i.e., *p2=p2;*. The full control structure (both branches) is then rewritten to the ternary operator (see statement on Line 7 in Figure 4.4b). Note, that during the next step, i.e., the encapsulation of if-converted basic block computations into functions, all statements on Lines 4 – 9 will be collected to one function and the basic block will be replaced by one function call. After the rewriting of the condition to the ternary operator, the last step is easier implementable. Also, rewriting the condition to a ternary operator can help some compilers to interpret it as guarded execution.

The pseudocode of the algorithm for rewriting the innermost *if* conditions to ternary operators is listed in Figure 4.5. Before we describe the algorithm in detail, we explain the Factored User-Def chains (FUD) we use for scalar data-flow analysis. FUD chains are an improved form of use-def chains [219]. They have two important properties. The first is that each use of a variable is reached by a single definition. The second property is that special merge operators called pseudo-definitions are inserted into control-flow merge points when there exist multiple reaching definitions. These pseudo-definitions factor the multiple incoming reaching definitions, and are inserted to satisfy the first property for any variable use after the control-flow merge. E.g., in Figure 4.4a between the use of variable *p2* at Line 9 and two definitions of this variable in two branches of the if condition there exists a pseudo-definition which merges the two definitions of *p2* variable.

In the FUD chain scalar data-flow analysis, the definition corresponds to a write access and the use corresponds to a read access. For both the write and read accesses, we can get the statement which contain these accesses. We can also collect all uses (read accesses) in a statement. Further, we can follow back the flow dependency, i.e., the data dependency between a definition and any use of the same variable. Note that the variable has to be defined before it can be used. Following back the flow dependency is the same as searching for the given use (read) the corresponding definition (write). If during this searching the pseudo-definition is discovered, we identify that the real definition is within some control-flow construct. There exist

```
rewriting_innermost_if(
Input: AST with FUD analysis,
Output: Transformed AST (if-converted)) {
  unmark_all_data_dep_conditions();
  def_lifo = collect_all_array_defs();
  while(element = pop_element(def_lifo)) {
    statement = get_statement(element);
    use_list = collect_uses(statement);
    foreach use_item (use_list) {
      def_item = get_def(use_item);
      if (inside_data_dep_cond(def_item)
      && !marked_condition(def_item)) {
        replace_cond_by_ternary_operator(def_item);
        mark_cond(def_item);
      }
      if(!inside_data_dep_cond(def_item)
      && !array_def(def_item)) {
        push_element(def_item, def_lifo);
      }
    }
  }
  delete_marked_data_dep_cond();
}
```

Figure 4.5: Algorithm for rewriting the innermost if condition to ternary operators (if-conversion).

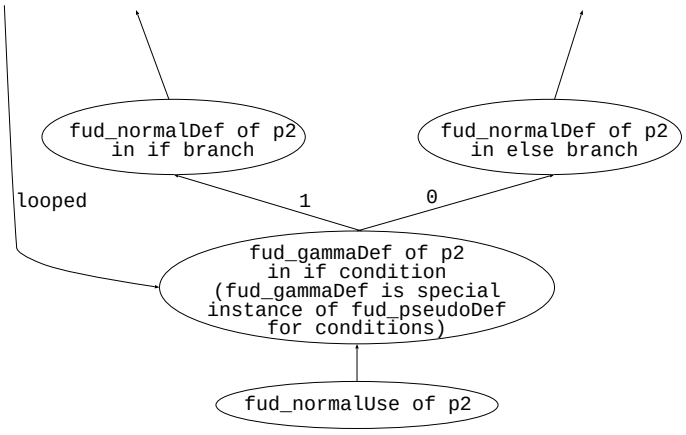


Figure 4.6: An example of a pseudo-definition for a condition. Between the use and the definitions in the branches of the if-condition the pseudo definition is present.

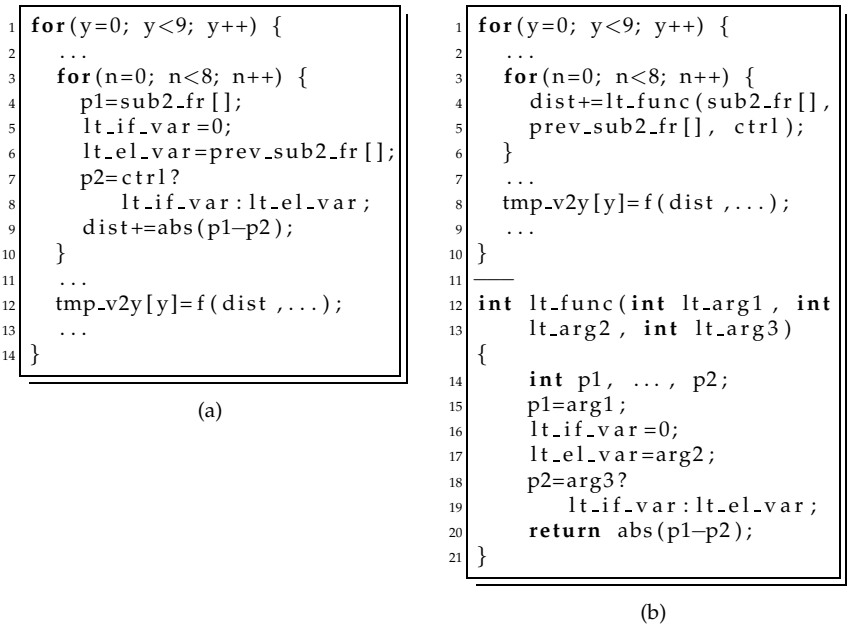
specialized pseudo-definitions for conditions and for loops in the FUD [219]. Thus, a write inside a condition can be easily identified when during following back the flow dependency from the use, the pseudo-definition for condition is detected. This is depicted in Figure 4.6 for variable *p2* from code in Figure 4.4a.

In the algorithm in Figure 4.5 we first collect all array definitions (writes) in the AST of the scope of our technique by traversing the corresponding part of the AST. The scope is usually the function we want to optimize with GLT. For each array definition we get the statement where the array is written and get all uses (reads) of the variables in that statement. Then we follow back the flow dependencies for each use and identify its definition till we do not reach the pseudo-definition that identifies the write inside the *if* condition or an array definition. Note that the pseudo-definition for *if* condition is special in the scalar data-flow FUD analysis we are using as was discussed in the previous paragraph. After identification of the control-flow boundary, the condition is replaced by the ternary operator by AST rewriting and this condition is marked as rewritten. At the end we delete all marked conditions which have been rewritten to the ternary operator.

The algorithm we presented in Figure 4.5 is simplified version of the real algorithm we implemented. In reality the search is not stopped after detecting the condition. When detecting the condition the algorithm is called recursively. With this implementation also nested innermost conditions can be handled. The only restriction is that the conditions have to be *if* conditions, thus not other conditional control structures such as *case* are allowed. However, these structures can be usually rewritten to the set of *if* conditions. Obviously, it is not possible to rewrite other then innermost conditions to the ternary operator. Thus the enabling Step 1, i.e., moving data dependent conditions down in the AST hierarchy (see Subsection 4.2.1) is performed before innermost condition rewriting.

4.2.3 Encapsulation of if-converted basic block computations into functions

After rewriting the innermost *if* condition to ternary operators, which actually represents if-conversion at the source code level, our code consists of if-converted basic blocks with a lot of scalars. We will demonstrate this on the part of the realistic example from the QSDPCM application [28]. In Figure 4.7a on Lines 4 – 9 there are 5 statements in the if-converted basic block. Those statements form one kernel which is usually very well optimized and should be transformed as one statement during GLT. Also, the if-converted basic block contains a lot of scalars that should be hidden for the GLT step. E.g., in Figure 4.7a, the scalars {*p1, lt_if_var, lt_el_var, p2*} are "avoidable" scalars. With "avoidable" scalars we mean intermediate variables that are not input variables or output variables of a particular if-converted basic block. Thus only the scalars whose lifetime does not span over the (if-converted) basic block boundaries are called "avoidable scalars" and can be encapsulated in Layer 3. Input and output scalars of a basic block remain in Layer 2 and are expanded to arrays using Dynamic Single Assignment conversion. The "avoidable" scalars can be eliminated in Layer 3 code via simple scalar copy propagation which is a well-known technique in the classic compiler literature [3, 154].



```
1  for(y=0; y<9; y++) {
2    ...
3    for(n=0; n<8; n++) {
4      p1=sub2_fr[];
5      lt_if_var=0;
6      lt_el_var=prev_sub2_fr[];
7      p2=ctrl?
8          lt_if_var:lt_el_var;
9      dist+=abs(p1-p2);
10   }
11   ...
12   tmp_v2y[y]=f(dist,...);
13   ...
14 }
```

(a)

```
1  for(y=0; y<9; y++) {
2    ...
3    for(n=0; n<8; n++) {
4      dist+=lt_func(sub2_fr[],
5      prev_sub2_fr[], ctrl);
6    }
7    ...
8    tmp_v2y[y]=f(dist,...);
9    ...
10 }
11 ---
12 int lt_func(int lt_arg1, int
13     lt_arg2, int lt_arg3)
   {
14     int p1, ..., p2;
15     p1=arg1;
16     lt_if_var=0;
17     lt_el_var=arg2;
18     p2=arg3?
19         lt_if_var:lt_el_var;
20     return abs(p1-p2);
21 }
```

(b)

Figure 4.7: Innermost data dependent condition hiding: (a) original code fragment code; (b) code fragment after condition hiding.

Figure 4.7 contains an example of the encapsulation of (if-converted) basic block computations into function. Every output variable of the basic block or the array in the basic block is detected and all the statements in the (if-converted) basic block this variable is depending on (directly or indirectly) are isolated. This group of statements is then encapsulated into a function. In the Layer 2 code only the function call to this function remains. The statements in the if-converted basic block are encapsulated into the *lt_func()* function and the basic block is replaced by this function as depicted in Figure 4.7b on Line 4. After the encapsulation, the code contains only Layer 2 constructs. Note, that in Figure 4.7a only one output variable *dist* is present and is depending on all the statements in the basic block, thus the isolation is not applied in this case.

In Figure 4.8 the algorithm for encapsulation of if-converted basic block computations into functions is listed. Actually, the algorithm works similar to simple scalar copy propagation which has been reimplemented to be able to collect all statements on the traversing path starting from a array definition. At the beginning we collect all array definitions (writes) in the corresponding part of the AST. For each array definition we get the statement where the array is written and get all uses (reads) of the variables in that statement. Till here, the algorithm resembles the algorithm in Figure 4.5. Here, the statement in the if-converted basic block is copied to the function and the following back of the scalar flow dependencies continues till the array definition or the boundary of the if-converted basic block is detected. The boundary is detected by identifying the use that is outside the (if-converted) basic block. This is possible with ATOMIUM FUD analysis.

If this boundary is detected the *def_lifo* has to be checked for the scalar that crosses

```
encapsulating_if_converted_basic_blocks(
Input: If-converted AST with FUD analysis
Output: Transformed AST with layer 2 and layer 3) {
  def_lifo = collect_all_array_defs();
  while(element = pop_element(def_lifo)) {
    statement = get_statement(element);
    if(in_if_converted_basic_block(statement) {
      copy_statement_to_new_function(statement);
    }
    use_list = collect_uses(statement);
    foreach use_item (use_list) {
      def_item = get_def(use_item);
      if (!array_def(def_item)) {
        def_lifo = push_element(def_item, def_lifo);
      }
      if(pseudo_def(def_item)) {
        check_and_reorder(def_item, def_lifo);
      }
    }
  }
  create_function_calls();
  eliminate_dead_code();
}
```

Figure 4.8: Algorithm for encapsulation of if-converted basic block computations into functions by scalar copy propagation.

the boundary. The *def_lifo* has to be reordered and this scalar has to be moved between the array definitions and the definitions that point within the if-converted basic block. This is needed, because before we go to the next if-converted basic block, all statements from the current basic block have to be collected. We will demonstrate it on example in Figure 4.7a. Let us assume we are at Line 7 and the *def_lifo* contains *...,ctrl,p1,p2*. If the *ctrl* could be before *p1* and *p2* we would pop *ctrl* before *p1* and *p2* and statements outside the basic block will be collected before this basic block is finished. Thus, this reordering is needed and not doing so will cause the intermingling of the statements from different if-converted basic blocks. At the end the function is created from the statements collected in the basic block. Note, that we do not follow loop carried dependencies like *dist*, otherwise we would unroll the loop around the basic block. Also, the dead code, i.e., the original statements that have been collected, are eliminated.

Note, that during copying of the statements code duplication can occur. E.g., if the statement on Line 7 in Figure 4.7a would be used also in another context, e.g., the *p2* scalar is used also somewhere else than on Line 9, this statement will be copied twice and encapsulated into two different functions. The final result of this step is encapsulation of all the functionality in the if-converted basic block between an array write and flow dependent array reads or (loop carried) scalars that cross the basic block boundary.

The introduced steps carry an overhead after loop transformations due to code duplication in Step 3, copying of the array content when only one branch is present in Step 2 and data dependent condition moving down in Step 1. Code duplication can be largely eliminated by Common Subexpression Elimination (CSE) [111, 94] after inlining Layer 3 back to Layer 2. However, sometimes we eliminate the opportunity

	Max. loop nest depth in a SCoP	Max. nr. of stat. in a SCoP
Original QSDPCM video encoder	6	6
QSDPCM video encoder after hierarchical rewriting	8	65

Table 4.1: Max. loop nest depth and max. nr. of statements in a SCoP before and after hierarchical rewriting.

	Nr. of main memory accesses	Improvement compared to previous row
Original QSDPCM video encoder	542.1×10^3	-
QSDPCM video encoder after GLT	445.5×10^3	17.8%
QSDPCM video encoder after hierarch. rewrit. and GLT	306.1×10^3	31.3%

Table 4.2: Comparison of original QSDPCM video encoder code, the code after GLT and the code after hierarchical rewriting and GLT.

for CSE by applied loop transformation, e.g., when a loop split is performed between two statements that are candidates for CSE. The expressions in these statements that are now in separate loop nests are of course no candidates for CSE any more. Similar estimates as we discussed at the end of Subsection 4.2.1 to avoid loop transformations which cause large duplication that cannot be solved by the CSE should be used here.

4.3 Results

The hierarchical rewriting and hiding of data dependent conditions was tested on the QSDPCM real-life application [28]. The application contains 26 innermost data dependent *if* conditions that prevent to use loop transformations on the global scope. In the original code, also some "avoidable" scalars are present.

The characteristics of the QSDPCM application w.r.t. maximal loop nest depth in a SCoP and maximal number of statements in a SCoP before and after hierarchical rewriting are depicted in Table 4.1. After applying our technique, the problematic conditions and "avoidable scalars" were hidden in Layer 3 allowing larger and deeper SCoPs.

For the QSDPCM application we also applied the complete DTSE methodology with and without our technique for preprocessing of innermost conditions. We observed the number of data memory accesses to the off-chip main memory which contribute significantly to the overall energy consumption.

In Table 4.2 we compare three different QSDPCM versions. The original QSDPCM

video encoder version corresponds to the version where GLT have not been applied. The QSDPCM video encoder after GLT version corresponds to the version where the fusion of the kernels that can be extracted to GM has been applied. The QSDPCM video encoder after hierarchical rewriting and GLT version corresponds to the version where the preprocessing of innermost conditions and fusion of all the kernels has been applied. We can observe 31.3% improvement when comparing our approach to the existing techniques without preprocessing.

4.4 Conclusions

In this chapter we have first motivated the need for hierarchical rewriting and data dependent conditions hiding for global loop transformations. This step separates the code in three layers. The first layer contains process control flow, the second layer contains loop hierarchy and indexed signals and the third layer contains arithmetic, logic and data-dependent operations. The first layer is usually well separated by the designer. However, the second and the third layer are intermingled. Such code is complex for the analysis and difficult to handle by the (DTSE) tools. Thus the separation of the second and third layer is an important step in the DTSE methodology.

However, till now the separation has been done manually, which is a tedious and error-prone task. In this chapter we proposed a technique for automatic separation of the second and third layer. We implemented our technique (except of moving the data-dependent conditions to innermost loops) using the AST library and the scalar data flow (FUD) analysis. Both are IMEC in-house products and are part of ATOMIUM [229] tool suite. We tested our technique on a real-life application. For this application we applied the DTSE methodology and we have seen that the technique allowed better utilization of this methodology.

CHAPTER 5

Preprocessing for outermost conditions

A scenario isn't a story that the gamemaster reads to the players, it's an outline for improvisational storytelling.
John M. Ford
(1957-2006)

Current multimedia streams contain the (en)coded sequence of audio/video frames. Before playing these multimedia contents the streams have to be decoded. The decoding consists of a sequence of multimedia kernels. Multimedia kernels are usually loop nests which include some processing (decoding) functionality, e.g., Viterbi, Fast Fourier Transformation (FFT), Requantization, Inverse Modified Discrete Cosine Transformation (IMDCT), Motion Estimation (ME) etc. Every kernel takes an input frame, processes it and produces an output frame. In modern multimedia streams different types of audio/video frames exist and the sequence of kernels used for the decoding depends on the type of the particular frame. In the application code the decision on kernels used for the decoding of the incoming frame is determined at run time by the outermost data dependent conditions in the frame decoding function. Mostly, the kernels fulfill the Static Control Part (SCoP) requirements (see Section 3.1) or can be preprocessed using e.g., the technique in Chapter 4, so that they fulfill these requirements. When the kernels are in the SCoP shape, the execution ordering within those is known at compile time. However, the execution ordering of the kernels, i.e., the sequence in which they occur, is known only at run time and it depends on the type of the incoming frame. This limits the scope of the design time optimizations such as loop transformations only at the kernels itself. However, we are not able to do cross-kernel optimizations, e.g., to merge two kernels and eliminate the intermediate buffer between them, at the design time. Thus, till now, the only option was to perform these optimizations at the run time, what can be cycle and energy consuming. Our approach presented in this chapter uses design time - run time approach to solve the problem. Multiple possible optimizations are performed at the design time, however at the run time the appropriate

optimization is selected.

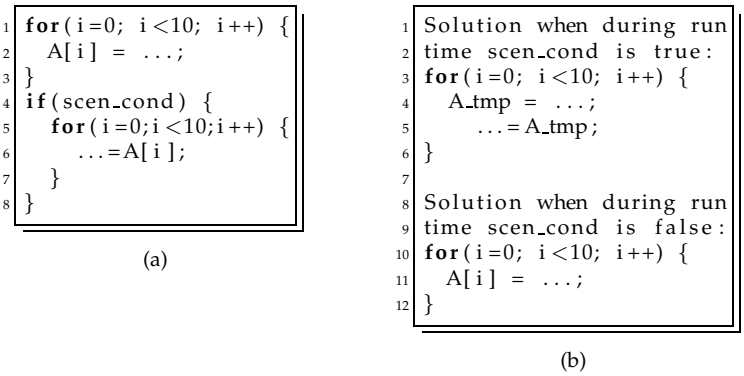
In this chapter after the problem definition in Section 5.1, in Section 5.2 we discuss how to model such applications consisting of kernels which ordering is know only at run-time. The model is needed to combine the Geometrical Model (GM) for SCoPs (kernels) with the data dependent control-flow of the application and is part of our solution to the problem. Section 5.3 explains how to construct easily a synthetic example of the proposed model and how to extract the model from a real-life application. Section 5.5 explains how to add profile information to the model using Ball-Larus profiling [15]. Section 5.6 proposes a technique using the model which goes beyond state-of-the-art loop transformations described in Section 3.6. Section 5.7 discusses several heuristics for our technique. Then the code generation phase is explained. In the next two sections the extension of our technique for loops with varying trip count and multi-dimensional exploration space are discussed. Finally a summary and conclusion section are provided.

5.1 Problem definition

Loop transformations are the crucial part of each state-of-the-art design methodology [28, 116, 145]. They improve the parallelization opportunities at the instruction or data level or improve the locality of data accesses, resulting in significant performance, area and power gains. The loop transformations are nearly always performed on a geometrical model [176, 217] which is very effective in dealing with generic loop transformations [218, 116, 46, 145]. Nowadays, the loop transformations performed on the geometrical model also start to take their place in popular modern compilers [19, 174].

The loop transformations performed on the geometrical model have to deal with strict limitations imposed by the model. Only those parts of the code which are "compile time" analyzable, called the SCoPs [19] can be parsed to the model and transformed. The SCoP is a maximal set of consecutive statements without *while* loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters. Intuitively, we can look at the SCoP as the geometrical model "basic block" on which the transformations can be applied. Note, that the SCoP has much coarser granularity than the basic block; it may contain multiple basic-blocks.

As a result of these limitations many optimizations opportunities are missed. We will demonstrate the problem on a example in Figure 5.1. The code in Figure 5.1a demonstrates two for loops where array A is produced in the first loop and the same array is conditionally consumed in the second loop. If the A array is consumed is determined only at run time. In the case, the second loop would not be in a condition, the two loops could be fused and array A could be in-placed, i.e., the memory locations of the array elements that are not used any more could be reused. Our (very simplified) solution is demonstrated in Figure 5.1b. We create two specialized cases where the condition is not present. The first case can be optimized at the design time. The decision which case is taken is postponed to the run time. Thus, we are able to optimize the code, however we have to pay in the code size increase for specialized



```
for(i=0; i<10; i++) {
  A[i] = ...;
}
if(scen_cond) {
  for(i=0;i<10;i++) {
    ...=A[i];
  }
}
```

(a)

```
Solution when during run
time scen_cond is true:
for(i=0; i<10; i++) {
  A_tmp = ...;
    ...=A_tmp;
}

Solution when during run
time scen_cond is false:
for(i=0; i<10; i++) {
  A[i] = ...;
}
```

(b)

Figure 5.1: Outermost data dependent conditions as limiting factor for GLT: (a) original code segment that cannot be fused; (b) specialized design time optimized solutions for different branches.

cases and also in the decision between those cases at run time. Still, this overhead is cheaper as to do the whole optimization at run time.

In the past, the multimedia and signal processing applications, i.e., applications from the application domain we target in this dissertation, where mostly fully statically analyzable and consisted of one big SCoP in the second layer and simple control-flow in the first layer of the application. Thus the code as depicted in Figure 5.1a was not present. To recap, the first layer contains the process control flow, the second layer contains the loop hierarchy and indexed signals and the third layer contains arithmetic, logic and data-dependent operations. Only the second layer containing memory management is the target for high-level low-power optimization. However, modern multimedia applications consist of smaller SCoPs in the second layer connected by rich control flow in the first layer. This control flow is usually data-dependent in current multimedia applications. Thus, in current applications very rich control flow (as simplified in Figure 5.1a) is present.

To go beyond the traditional scope of the SCoP, more focused techniques such as scenario creation [164, 163, 166] or optimization of the hottest path [35] have been proposed recently. These techniques combine the second "loop" layer and the first "control flow" layer and create larger SCoPs ("hyper"-SCoPs). The cost for the enlarged optimization space ("hyper"-SCoPs) is code duplication. The reasoning is similar to hyper-block [139, 140] and trace [73] creation, but on a much coarser level, i.e., on the level of SCoP instead of the basic block. However, to look at and optimize only the hottest path leads to suboptimal solution as we will see later in this dissertation. Our approach overcomes this limitation and provides better solution to the increased control-flow problem.

5.2 CFG+GM model

Current multimedia applications described above cannot be modeled only by the Geometrical Model (GM) described in Section 3.1 due to small SCoPs and rich control-

flow caused by a lot of outermost data dependent conditions as motivated in Section 5.1. Thus, we propose a model which combines the GM and the Control Flow Graph (CFG) constructed from the outermost data dependent conditions of the application. The GM models the SCoPs and the data flow array dependencies among them and the CFG models the outermost data dependent conditions in the main decoding function. The GM and its properties and limitations are described in Section 3.1. The technique demonstrated in this chapter cannot deal with arbitrary CFG and thus we do have several assumptions for the CFG in the main decoding function. Those restrictions are introduced to focus on certain class of CFG which occurs in our test applications and with respect to the simplified methods we are using for constructing CFG. More general solution for arbitrary (also cyclic) CFG is feasible when using advanced control-flow analysis and loop identification based on control-flow dominators [3] and path profiling of arbitrary control-flow graphs [15]. Those extensions require some engineering effort and are left for future work. Still, it would be desired that the CFG is reducible as defined in [3, 98] that the loops are unambiguously defined. CFGs of all real-life examples we have studied, fulfill our specific assumptions and they should not limit the generality of our technique for multimedia application domain. The assumptions for the CFG are:

- the CFG has to be a series-parallel Directed Acyclic Graph (DAG);
- the output degree of each node in the series-parallel graph is maximally two;
- the series chain (this term is explained below) in the series-parallel graph can have maximally one node.

We limit our technique only at the CFG which can be constructed when using only *if statements* as control structures. In the applications we have studied this is the case. Also a lot of applications can be rewritten to the form where only *if* statements are used as outermost control-flow structures. Such a graphs can be represented as series-parallel DAGs with the output degree of each node maximally two. Below we define the series-parallel graph more formally. We basically follow the definitions used in [64]:

Definition 5.1 *A* two-terminal graph (TTG) *is a graph with two distinguished vertices, s and t called source (start/initial node) and sink (end node), respectively.*

Definition 5.2 *The* parallel composition P = P(X,Y) *of two TTGs X and Y is a TTG created from the disjoint union of graphs X and Y by merging the sources of X and Y to create the source of P and merging the sinks of X and Y to create the sink of P.*

Definition 5.3 *The* series composition S = S(X,Y) *of two TTGs X and Y is a TTG created from the disjoint union of graphs X and Y by merging the sink of X with the source of Y. The source of X becomes source of P and the sink of Y becomes the sink of P.*

Definition 5.4 *A* two-terminal series-parallel graph (TTSPG) *is a graph that may be constructed by a sequence of series and parallel compositions starting from a set of copies of a single-edge complete graph K_2 with assigned terminals.*

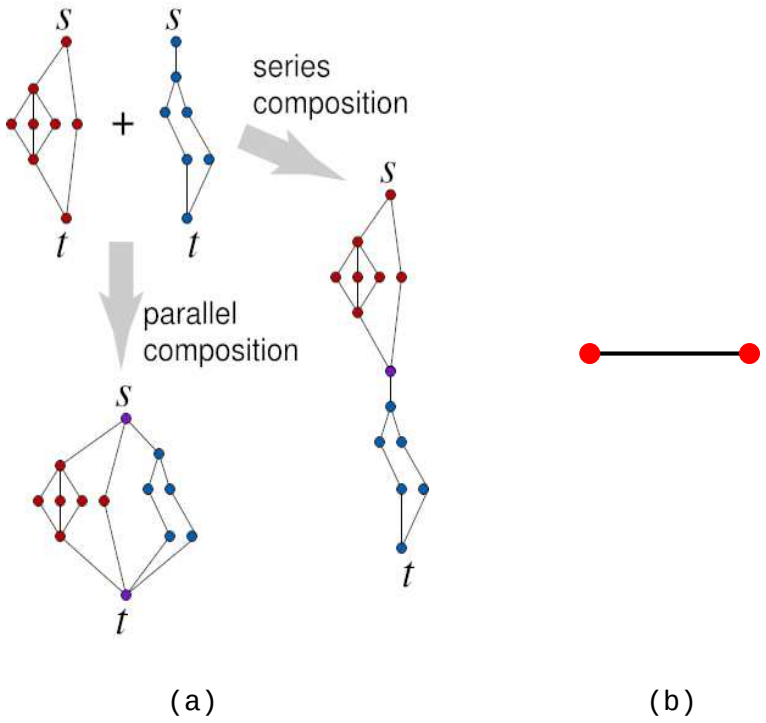


Figure 5.2: (a) The examples of series composition and parallel composition [240]. (b) K_2 graph [241].

Definition 5.5 *A* complete graph *is a graph where an edge connects every pair of vertices. The complete graph on n vertices has n vertices and n(n-1)/2 edges, and is denoted by* K_n.

Definition 5.6 *Finally, a graph is called* series-parallel (sp-graph), *if it is a TTSPG with some two its vertices being source and sink.*

The examples of series composition, parallel composition and K_2 graph are in Figure 5.2. Note, that for our case, the graphs are directed. Also the sp-graphs are more strict than the reducible CFG as defined in [3, 98].

In a sp-graph, a series chain of nodes is one or more nodes that are linked together in series to form a chain. If the series chain in the CFG would have 2 or more nodes, these CFG nodes representing the SCoPs of the application could be merged to one node. We assume this was done by the GM analysis.

5.3 Synthetic graphs

Preparing the application to a form where the CFG+GM model can be extracted is an tedious and error-prone task requiring preprocessing techniques such as selective function inlining [1], pointer analysis and conversion [184, 75], hierarchical rewrit-

ing [28] and dynamic single assignment conversion [209]. These techniques are systematic, however not automated yet. To be able to evaluate our techniques and heuristics much faster, we synthesize the CFG+GM model using the Task Graphs For Free (TGFF) pseudorandom graph generator and the Mersenne-Twister (MT) pseudorandom number generator algorithm.

TGFF is a tool which was originally developed in 1998 by R.P. Dick and D.L. Rhodes [57] to facilitate standardized random benchmarks for scheduling and allocation research, in general, and hardware-software co-synthesis research, in particular. TGFF is suitable for many applications that require generating pseudo-random graphs. The user can specify the number of task graphs to generate, the minimum number of task (nodes) per task graphs, the maximum number of input and output transmits (edges) per task, the seed for the pseudo-random number generator and some other parameters. K. Vallerio has subsequently updated and enhanced the code and documentation. The latest version [207] expands on TGFF features, providing a highly configurable random graph generator capable of generating several types of random graphs. The most significant improvement is the addition of a new algorithm which is capable of generating several types of random graphs including series-parallel chains. The number of series chains in generated graph and length of each chain are set by the TGFF commands *series_wid* and *series_len*. Another parameter, *series_must_rejoin*, will generate an extra (sink) node that will connect to the end of each chain. The TGFF generates only DAGs which suits our purpose better than more complex Synchronous Data Flow (SDF) graph generators which model cyclic dependencies, parallel and pipelined processing [194].

However, to use Task Graphs For Free (TGFF) directly does not produce graphs fulfilling the specification of the CFG in our model which was described in Section 5.2. The old TGFF generation algorithm [57] produces graphs which can have several sinks and if they have one sink they are not sp-graphs. Figure 5.3a shows a graph generated by the old TGFF generation algorithm. This graph has three sinks, namely nodes 9, 10 and 11. If we add an extra node 12 which will be the common sink for nodes 9, 10 and 11 the graph still does not fulfill the requirements of sp-graph in Section 5.2. The new TGFF generation algorithm [207] can produce sp-graphs. However, if the length of the series chain is set to one as required for a CFG the new algorithm is too constrained and produces only one type of graph. This type of graph always branches at the left node and rejoins in the last node (see Figure 5.3b). Thus, in this section except describing the synthetic graphs we propose also workaround for TGFF to produce wide range of synthetic CFG graphs with the desired properties.

To produce graphs according to the assumptions of the CFG in our model we use the new TGFF generation algorithm without the series chain length constraint and number of nodes $5\times - 10\times$ larger than required for our CFG (see Figure 5.4a). We process this graph and collapse the series chains with length larger than one to one node series. Figure 5.4a illustrates a graph generated by the new TGFF generation algorithm with relaxed series chain length constraint and in Figure 5.4b is the derived graph after series chain collapsing. Note that the node number in Figure 5.4b corresponds to the first node number in the series chain in Figure 5.4a. With this approach we have obtained synthetic CFGs fulfilling the properties of the CFG in our model (see Section 5.2) which are comparable with real-life CFGs. This would not be possible without subsequent modifications of TGFF generated graphs.

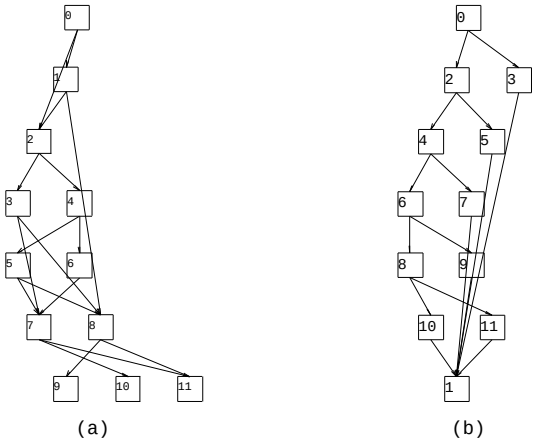


Figure 5.3: A graph generated by the TGFF generation algorithm. (a) output of old generation algorithm [task count (5,15), output task degree 2] (b) new generation algorithm [task count (5,15), output task degree 2, series must rejoin, series length 1].

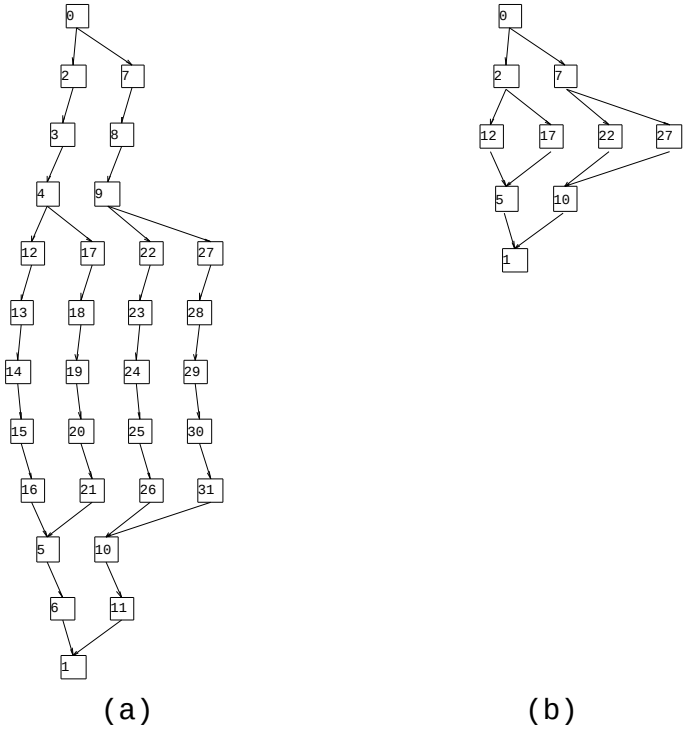


Figure 5.4: A graph generated by the TGFF generation - new generation algorithm [task count (5,15), output task degree 2, series must rejoin]. (a) before collapsing series chains (b) after collapsing series chains.

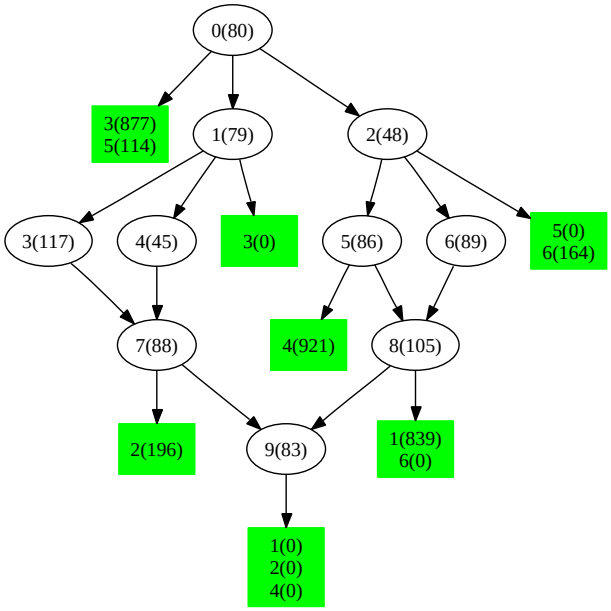


Figure 5.5: Example of synthesized CFG.

Each node in the CFG contains several properties representing the GM model for that node. These properties are generated pseudorandomly using the MT algorithm for generation uniform pseudorandom numbers [142]. The properties we generate are: *AST volume of the node*, *Dependency start nodes*, *Dependency lengths* and *Dependency sizes*.

The *AST volume of the node* represents the number of AST nodes in that particular CFG node. The *Dependency start node* represents the start node of an array dependency. I.e., it contains information in which node the array is produced. The *Dependency length* represents the number of nodes visited by a visitor during a breadth-first traversal when starting from *Dependency start node* till the dependency end node, i.e., the node where the array is consumed, is reached. Because using breadth-first traversal this is an atypical definition of dependency length (normally we would expect depth-first traversal). This definition is used, because the MT pseudorandom generator uses the uniform distribution when generating the numbers. Thus when using depth-first traversal the number of nodes between the *Dependency start node* till the dependency end node would be also uniform. However, we observed in the real-life applications, that the probability of small number of nodes between the *Dependency start node* till the dependency end node is higher. We achieve this property by using breadth-first traversal and number generator with uniform distribution. Of course, better solution would be to use appropriate random generator with non-uniform distribution with combination of uniform random generator and depth-first traversal. However, we did not find any that was suiting our distribution needs and thus we decided for the workaround with the breath-first search. The *Dependency size* represents the number of elements in the dependency.

Figure 5.5 shows an example generated using the parameters in Table 5.1 for the

Parameter	Interval for TGFF/MT generator
Number of TGFF nodes	(5,15)
AST volume of the node	(20,120)
Dependency start node	(1, num_vertices(TGFFcollaps))
Dependency length	(1, 5)
Dependency size	(100, 1000)

Table 5.1: An example of the parameters for synthesized CFG.

TGFF and the MT pseudorandom generator. The *AST volume of the node* is typed in the parentheses in the node after the node id number. The dependency generated with *Dependency start node*, *Dependency length* and *Dependency size* parameters can be identified with the dependency id number. These numbers are in the rectangles attached to the nodes. In the parentheses is the size of the dependency if the corresponding node is the start node, and 0 if the corresponding node is the end node of the dependency. E.g., in Figure 5.5 the dependency with id 3 and dependency with id 5 start at the first node. The dependency 3 has dependency size of 877 elements and ends in the next left node. The dependency 5 has dependency size of 114 elements and ends in the next right node. The AST volume of the first CFG node (node id 0) is 80 AST nodes, the volume of the next left CFG node (node id 1) is 79 and of the next right CFG node (node id 2) is 48 AST nodes.

5.4 Real-life graphs

The CFG+GM model can be extracted from real-life applications after applying all necessary preprocessing techniques such as selective function inlining [1], pointer analysis and conversion [184, 75], hierarchical rewriting [28] and dynamic single assignment conversion [209]. The code before extraction should contain only outermost data-dependent conditions whose branches contain the SCoPs. The extraction of the model is performed in three phases from the parsed AST of the application. In the first phase, the GM for each SCoP and the dependencies among/across SCoPs are extracted. Note that we extract all dependencies among/across SCoPs, i.e., in this phase we ignore the (outermost) data dependent control-flow of the code. In the second phase, the CFG where each node represents a SCoP and directed edges are used to represent outermost data dependent jumps in the control-flow, is extracted. The second phase also combines the GM and the CFG where the nodes of the CFG contain the GM information for the SCoP associated with that particular CFG node. This unique combination is one of the main contributions of the dissertation. Any code that consists of top-level control flow and the SCoPs can be parsed into our model. As mentioned above, our model assumes DAG as CFG within the time loop where the nodes of the CFG are SCoPs (containing conditions and loops with affine and manifest bounds). We have seen that the applications in the multimedia domain we target fulfill these requirements after using preprocessing techniques presented in Chapter 4. As mentioned in Section 5.2 more general solution for arbitrary (also cyclic) CFG is feasible. Note, that both the GM as well as the CFG still have links

```
1  if(header.block!=0) {
2    if(header.block==1)
3      for(i=0;i<576;i++)      //F kernel
4        sample[i]=f(in[i]);
5    else
6      for(i=0;i<576;i++)      //G kernel
7        out[i]=g(in[i]);
8  }
9  if(header.stereo==1)
10   for(i=0;i<576;i++)        //H kernel
11     out[i]=h(sample[i]);
```

Figure 5.6: Part of simplified MP3 source C code with 3 kernels connected by data dependent conditions.

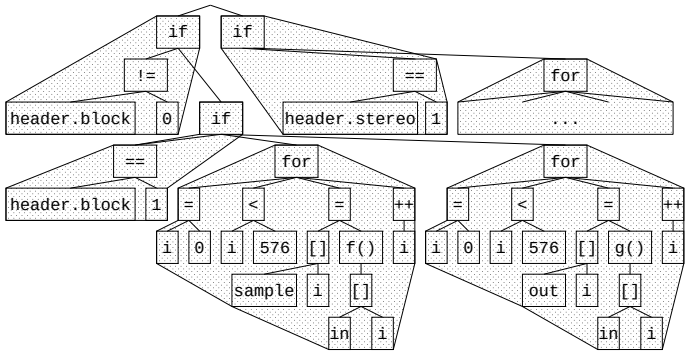


Figure 5.7: AST representation of the C code example in Figure 5.6.

to the original AST. The third phase counts the number of AST nodes in the SCoPs (*AST volume of the node*), identifies the *Dependency start nodes*, *Dependency end nodes* (thus also the *Dependency lengths*) and *Dependency sizes* from the parsed and extracted AST and GM+CFG representation.

The three phase extraction is demonstrated on the code in Figure 5.6. The code is a simplified version of the requantization and stereo kernels of the MP3 audio decoder application [132] for long and short blocks only. The whole application is described in detail in Section A.1. The code in Figure 5.6 consists of two kernels F and G, performing *f()* and *g()* requantization operations followed by kernel H, which performs a *h()* stereo operation on a one dimensional array (frame) with 576 elements. These kernels fulfill the requirement of a SCoP and thus they can be modeled using the GM. After the AST parsing the GM for each SCoP and the dependencies among/across SCoPs are extracted. For the AST parsing we use the ATOMIUM AST parser and for the GM extraction and representation we use ATOMIUM GM libraries. Both are parts of the in-house IMEC ATOMIUM framework [229]. The CFG is represented using the Boost Graph Library (BGL) [188].

The AST of the application in Figure 5.6 is in Figure 5.7. The GM model and the dependencies extracted are depicted on the left in Figure 5.8. In the kernel F the *sample* array is written for the iteration node from i=0 to i=575. This is the GM at the top left of the figure. In the middle left of the figure is the GM for kernel G where

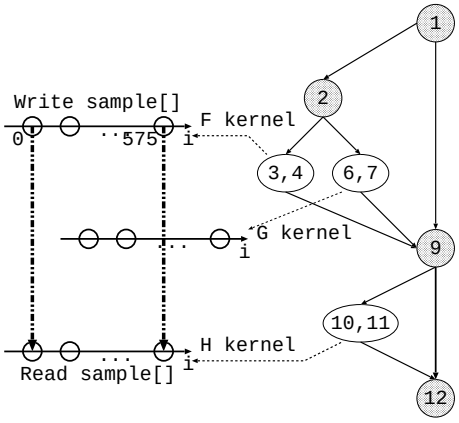


Figure 5.8: The GM model (iteration domain polytopes and dependency polytopes) of the kernels from Figure 5.6 (left side of the figure) and the CFG model from Figure 5.6 with the link to the GM model (right side of the figure). The nodes in the CFG contain line numbers.

the output is written for the iteration node from i=0 to i=575. The GM for kernel H is depicted at the bottom left of the figure. Here, the *sample* array is read and the output is written for the iteration node from i=0 to i=575. Between kernel F where *sample* array is written and kernel H where *sample* array is read exists a data flow array dependency of size 576. The data flow array dependency is depicted by the bold dash-dot-dot line between GM for kernel F and GM for kernel H.

Figure 5.8 (right side) shows the control dependencies between the kernels using the CFG. The CFG is constructed only from outermost data dependent conditions. The bodies of these conditions are the kernels. During the CFG construction the link to the GM discussed above is created. The numbers within the CFG nodes correspond to the line numbers in Figure 5.6. E.g., line 1 contains the outermost data dependent condition which is represented by a decision node with two output edges in the CFG.

After extracting the CFG+GM representation, the third phase of the extraction counts the number of AST nodes in the SCoPs (*AST volume of the node*), identifies the *Dependency start nodes*, *Dependency end nodes* and *Dependency sizes*. The resulting graph is in Figure 5.9. This graph combines quantitative information for analysis from Figure 5.7 and Figure 5.8.

In Figure 5.9 the *AST volume of the node* is typed in the parentheses in the node after the node id number. The dependency extracted from the GM can be identified with the dependency id number and the array access type and name in the square brackets. The array access type and name is additional compared to synthetic graphs (see Figure 5.5). As in the synthetic graphs, in the parentheses is the size of the dependency if the corresponding node is the start node, and 0 if the corresponding node is the end node of the dependency. E.g., in Figure 5.9 the dependency with id 1 starts at the node with id 2. This dependency has dependency size of 576 elements and ends at the node with id 5. The AST volume of the first CFG node (node id 0) is 4 AST nodes, the volume of the node with id 2 as well as node with id 5 is 17 AST nodes.

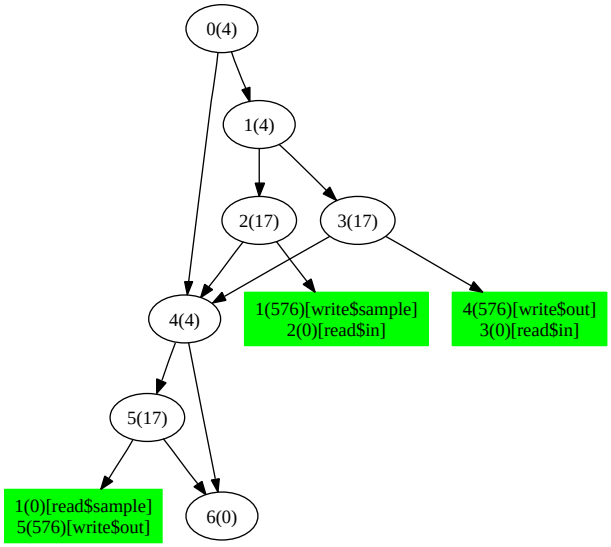


Figure 5.9: A CFG graph extracted from part of an MP3 audio decoder.

Note that the same basic representation is used for the real-life graph in Figure 5.9 and the synthetic graph in Figure 5.5.

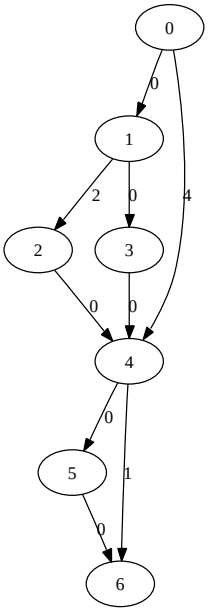
5.5 Ball-Larus path profiling in DAG

Any optimization technique should focus mainly on the bottlenecks in the important part of the application. Due to Amdahl's law [8] the overall speedup $S_{overall}$ achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S is

$$S_{overall} = \frac{1}{1 - P + \frac{P}{S}}.$$

Thus in our technique we should consider the frequency of the paths in the CFG, which determines the proportions P for paths of the whole program execution. To obtain this information in an efficient way we use Ball-Larus profiling [15].

Ball-Larus profiling is a simple and fast algorithm that selects and places profile instrumentation to minimize run-time overhead. The essential idea behind the path profiling algorithm is to identify all potential paths with states, which are encoded as integers. In Ball-Larus profiling the states are numbered from *0...n-1*, where n is the number of potential paths in the CFG. The profiling works only with CFGs that have been converted into a Directed Acyclic Graph (DAG) with a unique source vertex *ENTRY* and sink vertex *EXIT*. Our CFG is always a DAG, as specified in Section 5.2, thus the conversion is not necessary. The basic step in Ball-Larus profiling is to assign a non-negative constant value to each edge in a DAG, such that the sum value along any path from from *ENTRY* to *EXIT* is unique. We call the path sum



(a)

Path	BL Nr.
0 1 3 4 5 6	0
0 1 3 4 6	1
0 1 2 4 5 6	2
0 1 2 4 6	3
0 4 5 6	4
0 4 6	5

(b)

Figure 5.10: Assignment of non-negative constant values to edges for the CFG in Figure 5.9. The sum of these numbers along each path in the graph is a unique number from *0...5* identifying that path. (a) Annotated CFG; (b) Table with all possible paths.

the path number and it lies in the interval *0...n-1*, where *n* is the number of potential paths in the CFG.

Figure 5.10 shows an example of a DAG where to each edge a non-negative constant value is assigned. The algorithm for calculating those values is described in Figure 5.11.

After assigning values to the edges in a DAG, the instrumentation code is inserted. At the start of the program, we initialize all elements of an array *count* to 0. The array *count* has *n* elements where *n* is the number of paths in the program and at the end of the instrumentation it will contain the path histogram, i.e., how often were individual paths executed. At the edges with positive Val(e) value, at the *ENTRY* node and at the *EXIT* node we place instrumentation code as described in the algorithm in Figure 5.12.

During instrumentation the register *r* is set to 0 when we enter the *ENTRY* node and is updated along the edges with positive values till the *EXIT* node is reached. Then an array *count* indexed by the value of the register is incremented. At the end of the profiling the array *count* contains frequencies, i.e., number of times the path was executed, for each path.

Table 5.2 shows the results of Ball-Larus profiling of an MP3 audio decoder [132]. The CFG of the MP3 audio decoder contains 26 nodes and has 234 possible paths. Table 5.2 does not list the paths which where not executed during profiling. From the table we can see that only 6 paths are active for the majority of realistic bitstreams,

```
foreach vertex v in reverse topological order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    foreach edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) = NumPaths(v) + NumPaths(w);
    }
  }
}
```

Figure 5.11: Algorithm for assigning values to edges in a DAG [15].

```
Before program execution initialize count[]=0
At ENTRY vertex {
  instrument(v, 'r=0');
}
foreach edge e {
  if Val(e) != 0 {
    instrument(e, 'r+=Val(e)');
}
At EXIT vertex {
  instrument(v, 'count[r]++');
}
```

Figure 5.12: Simplified algorithm for placing instrumentation.

Path number	Number of times the path was executed
209	12079
221	2363
197	765
90	336
102	100
78	51
others	0

Table 5.2: Ball-Larus profiling for an MP3 audio decoder.

i.e., are executed one or more times as it was identified by the Ball-Larus profiling. This substantiates the 10%-90% principle, i.e., that less than 10% of paths are responsible for at least 90% of all executions.

5.6 Scenario technique

Our scenario technique creates clusters of paths in the application during design time. The clusters are created based on 2D objective function. First, the clusters should benefit from enabled loop transformations as it was demonstrated in Figure 5.1 in Section 5.1. Second, the clusters should be constructed with minimal code overhead. These two forces act in opposite directions resulting in the trade-off between enabled loop transformations and code overhead. The more detailed we cluster, the more loop transformations are enabled resulting in data memory footprint decrease. However, the more code overhead resulting in instruction memory footprint increase is present. The input for our scenario technique is the CFG+GM model explained in Section 5.2 which can be either synthesized (Section 5.3) or extracted from the real application (Section 5.4). The output is a Pareto curve [171] in a 2D exploration space data memory footprint vs. instruction memory footprint where each point on the curve represents a complete set of Control Flow subGraph (CFsG)s (a set of path clusters where all paths of the original CFG are present). Below are the definitions of a path, CFsG, complete set of CFsGs and Pareto set of CFsGs.

Definition 5.7 *A* path *in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. The first vertex is called the start vertex and the last vertex is called the end vertex.*

Definition 5.8 Control Flow subGraph (CFsG) *of a CFG is a series-parallel CFG which contains only some of the paths of the original CFG.*

Definition 5.9 *A set of CFsGs is a* complete set of CFsGs *if and only if it covers all paths in the original CFG from which it was derived.*

Definition 5.10 *A complete set of CFsGs is a* Pareto set of CFsGs *if and only if there is no other complete set of CFsGs which is better in all dimensions (smaller data memory footprint and smaller instruction memory footprint) of the exploration space.*

In this section, first we explain how we create the complete sets of CFsGs from the CFG+GM model. In the next two subsections we define our exploration space. At the end of this section we explain the trade-offs between the axes in the exploration space where only the Pareto sets of CFsGs occur.

5.6.1 Dividing the CFG into CFsGs

The CFG synthesized in Section 5.3 or extracted in Section 5.4 can be decomposed to individual paths. Such a decomposition is depicted in Figure 5.13. On the left side

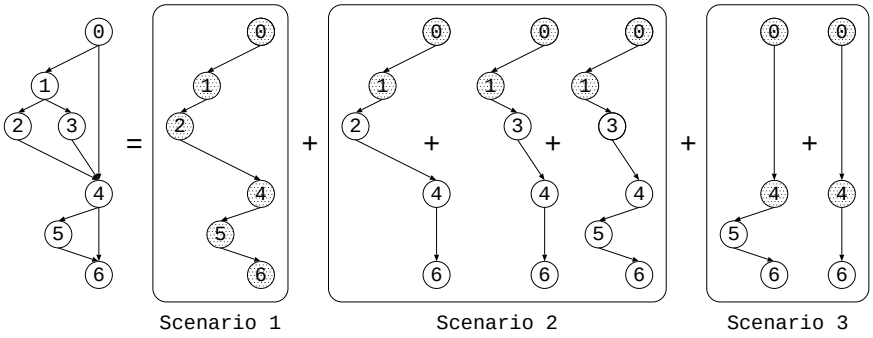


Figure 5.13: Decomposition of the CFG into individual paths and creating the CFsGs.

is the original CFG followed by the equal sign and the 6 individual paths occurring in this graph split by plus sign. For large graphs this may lead to an explosion of possible paths. As already motivated above, the decomposition enables more loop transformations. However, if the loop transformations would be applied separately on each path during the design time, this will cause large code overhead and also overhead during the run time when the actual (optimized) path has th be predicted/detected. Hence, some paths have to be grouped together. This is depicted by the boxes surrounding the paths which will be grouped together. Thus, on the right side of the figure (after the equal sign) we have 3 groups for our simple example. The first group is labeled *Scenario 1* and contains one path. The second group is labeled *Scenario 2* and contains three paths. The third group is labeled *Scenario 3* and contains two paths. This grouping solution is just an illustration randomly chosen here. The whole search space from which we can pick a solution contains B_n possibilities where n is the number of paths in the original CFG and B_n is the Bell number [21]. For our example in Figure 5.13 with n=6 the $B_6 = 203$. Thus our search space has 203 possibilities from which we have chosen one randomly for the example in Figure 5.13. We will discuss more in detail how to compute the Bell number later. The Bell number B_n is equal to the number of ways to partition a set (of all paths) into nonempty subsets (of scenarios) if the set (of all paths) has n elements [21]. Each nonempty subset, i.e., *Scenario 1-3*, builds the CFsG. The three scenarios form a complete set of CFsGs.

Till now, we did not explain how we cluster the paths. In this paragraph we provide general clustering principles and then we explain the simplification we made for our scenario technique. In Figure 5.14a is depicted a clustering example with two-dimensional objective cost function. Each path is represented by a (two-dimensional) Pareto curve which represents different path knob configurations (e.g., energy vs. performance trade-off). Normally, the paths with similar Pareto curves are clustered to one scenario at the design time. We expect that those paths exhibit similar properties, e.g., also with respect to the optimizations and represent similar code. Thus, the distance between two Pareto curves determines which paths to cluster. The cluster is then represented by its worst-case Pareto curve. Too large distance between the Pareto curves means too bad worst case for curve closest to origin. Based on this criterion Curve 1 and Curve 2 are clustered in Figure 5.14a. Thus, Curve 1 (representing path 1) and Curve 2 (representing path 2) form first scenario and Curve 3

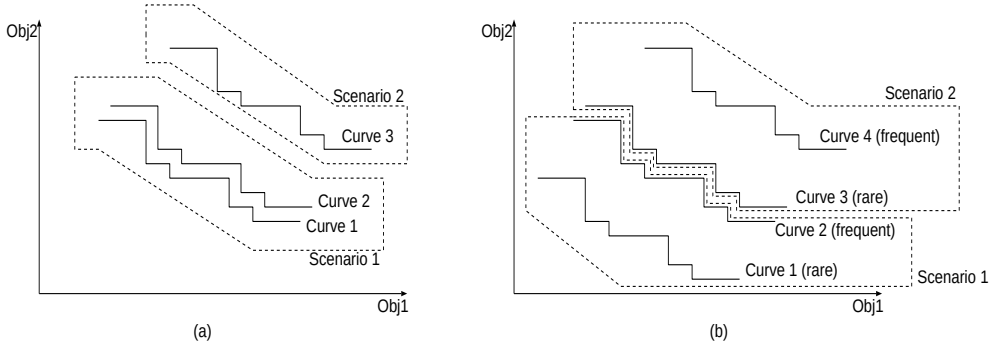


Figure 5.14: General clustering approach.

forms second scenario. Apart from the distance also the frequency of occurrence is important. If very frequent curve is clustered with very rare one which has worse Pareto curve, this scenario (and all paths in this scenario) inherits the worse (rare) Pareto curve. In such a case is better to cluster the frequent Pareto curves with better rare Pareto curves so that the frequent Pareto curve represents the created scenario. This is depicted in Figure 5.14b. This analysis and clustering is done at the design time. During the runtime, the appropriate scenario with its Pareto curve is identified and a concrete Pareto point is selected based on the required knob configuration. In the sequel we will use simplified Pareto curve "similarity" measure based on two metrics, the cost (in terms of instruction memory footprint) and the gain (in terms of data memory footprint) which are explained in Subsection 5.6.2 and 5.6.3. The "similarity" of the Pareto curves is also dependent on the cost of storing a Pareto curve which is platform dependent. Thus in our approach we provide several options for different platforms as explained in Subsection 5.6.4.

In Figure 5.13 we have a CFG where the nodes correspond to different SCoPs. This CFG contains 6 paths. One decision how to group these paths is depicted in the Figure 5.13. In the first group the path from node 0 to node 6 is without branching, so all the nodes in this group can create one big SCoP. This is emphasized by shadowing those nodes. In the second group the path from node 0 to node 1 is without branching, so SCoP node 0 and 1 can create a bigger SCoP. Again, those nodes that can be merged to create a bigger SCoP are shadowed. In the third group nodes 0 to 4 are without branching and can create a bigger SCoP. Bigger SCoPs enable more loop transformations compared to small SCoPs, because the loop transformations are not capable to go beyond the SCoP boundary. In each scenario represented by CFsG, the additional knowledge about the control-flow allows for stronger and more precise analysis and thus for better program optimization, i.e., the loop transformations for improved locality of the program. However, the additional knowledge requires also code replication resulting in code size increase. In the following subsections we will provide more details about the scenario creation technique, the estimation of code size increase in terms of AST nodes in the application and loop optimization potential.

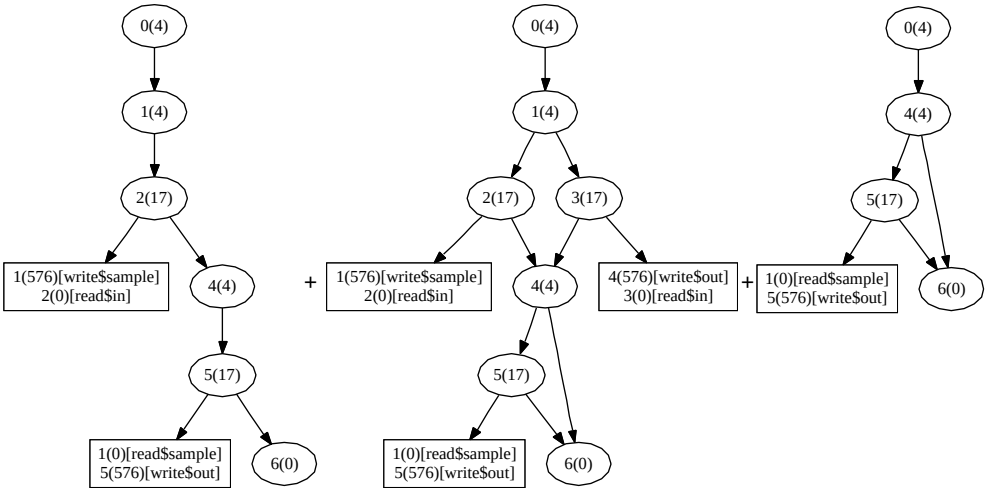


Figure 5.15: Decomposition of the CFG into individual paths and creating the CFsGs with GM and AST information annotation.

5.6.2 Cost: Instruction memory size increase

Each node in a CFsG has its AST subtree. This subtree has to be duplicated when the node occurs in other CFsG of the same (complete) CFsG set. *Note, that from now on with CFsG set we would understand complete CFsG set.* E.g., in Figure 5.13 node 0 of the original CFG has to be duplicated for *Scenario 1-3*, node 1 has to be duplicated for *Scenario 1-2* etc. This causes an increase of the code for the set of CFsG compared to original CFG. The absolute code size for the CFG and for the set of CFsGs can be estimated using the number of AST nodes in the original CFG and in the created set of CFsGs.

Figure 5.15 depicts the same set of CFsGs as Figure 5.13 with annotated informations about the (*AST volume of the node*), *Dependency start nodes*, *Dependency end nodes* and *Dependency sizes*. To determine the count of AST nodes of the original CFG we cumulatively add the *AST volume of the node* for each CFG node in the original CFG in Figure 5.9. This results in $4 \times 3 + 17 \times 3 + 0 = 63$ AST nodes (see Figure 5.9). The number of AST nodes in Figure 5.15 is $(4{\times}3{+}17{\times}3{+}0){+}(4{\times}3{+}17{\times}2{+}0){+}(4{\times}2{+}17{+}0) = 134$ AST nodes. The code size for set of CFsG in Figure 5.15 is more than double of the code size of the CFG in Figure 5.9. Thus more than double of the instruction memory size will be needed to store such a code. However, the code compaction techniques and code sharing between scenarios can be applied to certain extend after GLT.

5.6.3 Gain: Data memory size decrease

In the set of CFsGs some CFsGs may have array dependencies that do not cross any data dependent condition and that crossed a data dependent condition in the original CFG. E.g., see dependency with id 1 in the CFG in Figure 5.9 between write of *sample* array and read of *sample* array. In the original CFG this dependency crosses

the data dependent condition on line 9 in Figure 5.6. In the CFsG labeled as *Scenario 1* in Figure 5.15 it does not cross any data dependent condition. Thus it can be potentially optimized by the GLT step, e.g., by loop fusion. We look at the size of those dependencies, i.e., the number of elements that is written at one side of the dependency and read at the other side of the dependency (in our example 576). This corresponds to the number of dependency vectors in the dependency. To determine the optimization potential of creating a set of CFsGs we iterate over those array dependencies that can be potentially optimized now and whose could not be optimized in the original CFG. We cumulatively count the size of those array dependencies.

However, the improvement in the optimization potential depends also on the frequency of the CFsG where the optimization is possible. E.g., if the frequency of *Scenario 1* would be 10% of the whole execution the impact of the optimisation of dependency with id 1 would be only 10%. The dependency can occur also in other CFsG (see dependency with id 1 in *Scenario 2* in Figure 5.15). Those executions of the dependency are not going to be optimized and thus should not be counted. Thus the optimization potential of a CFsG is

$$Potential(CFsG) = \sum_{p_i \in CFsG} \left(f_{p_i} \times \sum_{Dep_k \in CFsG} V_{Dep_k}(p_i) \right)$$

where p_i is the *i*-th path in a CFsG and f_{p_i} is the frequency of this path obtained by Ball-Larus profiling (see Section 5.5). $V_{Dep_k}(p_i)$ is the dependency size of the *k*-th dependency in the CFsG along the path p_i. The dependency starts and ends at different nodes. It should cross only nodes with input and output degree one in the CFsG when going from the start node to the end node of the dependency. The start node should have output degree one and the end node should have input degree one. Then the optimization potential of the set of CFsGs is

$$Potential(set\ of\ CFsGs) = \sum_{CFsG \in set\ of\ CFsGs} Potential(CFsG)$$

We also define the the data memory size decrease and the data memory size increase as we will use them in the rest of this dissertation. The data memory size decrease reflects better the relation of the scenarios data memory size to the original application and intuitively represents better the gain. However, data memory size increase can be better depicted in the trade-off figures we are going to show in this chapter (for data memory size decrease the x-axis should be reversed to the opposite direction).

Definition 5.11 *The* data memory size decrease for a set of CFsGs *is the difference between the optimization potential for this set and the optimization potential for the original CFG (singleton set).*

Definition 5.12 *The* data memory size increase for a set of CFsGs *is the difference between the optimization potential for this set and the maximal optimization potential for the set of CFsGs*

where the latter is the set of CFsGs where each CFsG from the set is created by exactly one path in the original CFG. The set of CFsGs with best optimization potential has the same number of elements as the number of paths in the original CFG.

5.6.4 Trading-off instruction vs. data memory size

For each set of CFsGs we can determine the cost in terms of number of AST nodes for that set and the gain in terms of the optimization potential as defined in the previous subsection. The increase of the number of AST nodes causes instruction memory size increase and the increase of the optimization potential causes data memory size decrease (or the decrease of the optimization potential causes data memory size increase). Till now we have discussed only briefly how many sets of CFsGs exist if the original CFG has *n* paths. In the following paragraph we will explain more in detail these issues.

The number of ways of grouping *n* paths is called the Bell number B_n. It is equal to the number of ways to partition a set into nonempty subsets if the set has *n* elements [21]. For our example in Figure 5.13 with *n=6* the $B_6 = 203$. The simplest way to compute Bell number is using the recursive equation

$$B_{n+1} = \sum_{k=0}^{n} B_k \left(\begin{array}{c} n \\ k \end{array} \right), \; B_0 = 1$$

where $\left(\begin{array}{c} n \\ k \end{array} \right)$ is a binomial coefficient,

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k! \times (n-k)!}.$$

We can look at this equation as the dot product of the vector $(B_0, B_1, \ldots, B_n)$ and the vector $(\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n})$, i.e., the *n*-th row in the Pascal triangle [39]. For *n=5* we get $B_6 = (1, 1, 2, 5, 15, 52) \cdot (1, 5, 10, 10, 5, 1) = 203$. The B_n grows exponentially with the number of paths and to evaluate all solutions for larger number of paths is not feasible due to time and storage requirements. In Figure 5.16 we draw the full exploration space for 6 active paths of the full MP3 CFG. Note, that first the 6 active paths out of 234 paths have been selected and then grouped as shown in Figure 5.13. The remaining $234 - 6 = 228$ paths are not active, i.e., they do not occur in the activation trace (their frequency is 0) and they are not considered in our exploration. These paths are grouped to one scenario called backup scenario. The backup scenario is a piece of code that covers all paths that are not a target of the exploration (active paths). I.e., for the MP3 audio decoder it covers all 234-6=228 paths that have not been used during profiling. A valid solution will be selected for these paths but it will of course not be very optimized (worst case scenario).

Because the non-active paths which are part of the backup scenario are not a target of our exploration, the size of the backup scenario is not counted in the code metric we use. This is not a problem for our exploration, because the size of the backup scenario stays the same and represents the offset to the obtained code sizes for different groupings. Also, the backup scenario code should almost never occupy the L1

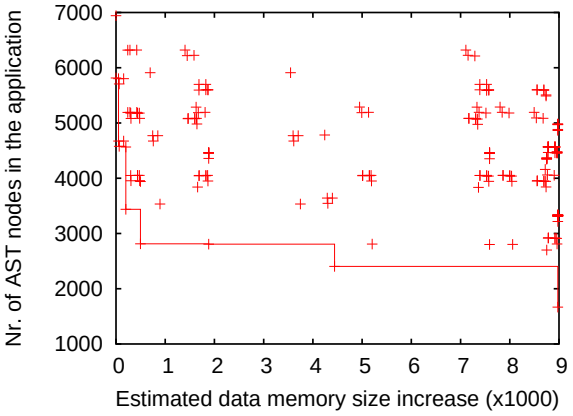


Figure 5.16: Full exploration space for 6 active paths of the MP3 audio decoder.

instruction memory (if it is not the case, we should consider to repartition our scenarios). It is located just for backup cases in the (huge) main memory where we do not have an instruction size limitation. This is different from the solution where we have the original source code without scenarios that does not exhibit yet the backup scenario. This code is always occupying L1 instruction memory.

The *Nr. of AST nodes in the application* is the total number of AST nodes after a grouping. Note, that some CFG nodes (corresponding to SCoPs as has been mentioned earlier) and thus AST nodes of that SCoPs have to be duplicated. The *Estimated data memory size increase* shows the loop transformation potential decrease when grouping more paths and thus discarding possibility for bigger SCoPs. From the 203 points obtained, only 8 are Pareto points (see the Pareto curve in Figure 5.16). Note, that the bottom rightmost point in our exploration in Figure 5.16 is also not representing the original CFG but the CFG constructed from the active paths only. We refer to this solution as to initial CFG. The backup scenario is still located in main memory additional to this initial solution. The original CFG may have larger code size and less optimization potential and thus it is not a Pareto optimal point in our exploration space. For an MP3 audio decoder we use as test driver in this chapter, the initial CFG and the original CFG have the same optimization potential (due to zero frequency non-active paths). However, the original CFG is slightly larger (has more nodes).

To evaluate all possible groupings can be time consuming especially for larger CFGs and a higher amount of active paths considered. Thus, in the following section we propose several heuristics that will speed up the grouping exploration and make it much more scalable.

5.7 Pruning the exploration space and heuristics

The number of possible sets of CFsGs grows exponentially with the number of paths in the original CFG. Thus also the time to evaluate all possible sets grows exponen-

tially with the number of paths in the original CFG. The parsing time requirements to the model grow with the number and complexity of CFG nodes. E.g., to evaluate all possible groupings of 6 paths in a 33 node original CFG takes ∼25s. To evaluate all possible groupings of 7 paths in a 100 node original CFG takes already ∼1000s. To evaluate all possible groupings of 14 paths in a 188 node original CFG is even not feasible due to limited memory resources as we will see later. Thus the brute force evaluation of scenario creation technique is not scalable w.r.t. increased size (in terms of nodes and explored paths) of the original CFG. To deal with this problem we propose in next subsections several heuristics for the scenario creation technique.

5.7.1 Selecting the most frequent paths

The CFG in our model represents all possible code paths among the kernels. However, due to the well-known 10%-90% principle (i.e., only 10% of the code is responsible for 90% of the execution time), only a few code paths are really hot. To identify these hot code paths we use Ball-Larus profiling [15]. This was already demonstrated for the MP3 audio decoder example where only 6 paths out of 234 are active (see Table 5.2). Thus we do not need to create the sets of CFsGs and perform the exploration from all paths available. It is sufficient to do the exploration only from the hot code paths that are usually only 10% of all possible paths. This will prune the exploration space significantly. The remaining 90% of paths form a backup scenario which is not a target of the exploration. The backup scenario is not considered in our optimization effort and it has to be activated each time we do not enter one of the hot code paths.

5.7.2 Coverage criterion heuristic

When grouping paths together, other paths can also occur in the resulting CFsG in addition to the paths the CFsG is composed of. In Figure 5.13 *Scenario 2* consists of 3 paths. Nevertheless, it also contains the path of *Scenario 1* because all graph edges that are in *Scenario 1* are also in *Scenario 2*. Thus, *Scenario 1* is fully covered by *Scenario 2*, i.e., the whole functionality of *Scenario 1* can be found in *Scenario 2*. In the coverage criterion heuristic, we first construct all the possible CFsG sets. Then we check in each CFsG set if it contains a scenario that can be covered by another scenario in the same set. The CFsG set, where a scenario is fully covered by another scenario from the same CFsG set, is not further evaluated and is pruned from the exploration space. Thus, because of full coverage of *Scenario 1* in *Scenario 2*, the whole clustering in Figure 5.13 is skipped and not evaluated further. Because we check the coverage of the scenarios we call this heuristic the "overage criterion heuristic".

The solution in Figure 5.13 can still be a Pareto point, if the optimization potential, i.e., the data memory size decrease, for *Scenario 1* is large. Then this value determines the overall data memory size decrease for that CFsG set. From this CFsG set we can derive another set by splitting *Scenario 2* in two subscenarios, one containing the path which contains node 2 and the other one containing the two remaining paths (which contain node 3). Because *Scenario 1* dominates the data memory size decrease (reduction), the new solution should not differ too much in data memory size. Also,

because only one scenario has been split further (*Scenario 2*), the code size should not change drastically. Thus, even when our heuristic has pruned the solution in Figure 5.13 out of the exploration space, if *Scenario 1* dominates the overall cost and gain, splitting *Scenario 2* should create a point which is not so far in the exploration space and is not pruned out.

5.7.3 Loss/Similarity heuristic

The previous heuristic was based on generating all possible CFsG sets and pruning them before evaluation. The sequel Loss/Similarity heuristic and Fruchterman-Reingold heuristic construct "optimal" CFsG sets directly. Our target is to construct a CFsG set where similar paths in terms of the common nodes in the paths are in one scenario and where also paths do keep their big SCoPs when grouped. Note, that when grouping the paths together, the SCoPs get smaller and we are interested in this difference. The reasons are the following: when grouping similar paths, the number of duplicated nodes will be much smaller than when grouping paths that are different. Also, we would like to group paths together which do not lose anything from the loop transformation potential when grouped, i.e., after grouping the big SCoPs do not get smaller.

We measure the similarity of two paths p_i and p_j as a ratio of nodes common to both paths, and all nodes of those two paths. This is an extension to the similarity measure presented in [165]. To count all nodes of two paths we count the nodes in the graph constructed by grouping those two paths. To count nodes common to both paths we use the inclusion-exclusion principle, i.e., the cardinality of the intersection of two sets is equal to the sum of cardinalities of those two sets minus the cardinality of the union of the two sets

$$Similarity(p_i, p_j) = \frac{\|p_i \cap p_j\|}{\|p_i \cup p_j\|} = \frac{\|p_i\| + \|p_j\| - \|p_i \cup p_j\|}{\|p_i \cup p_j\|}$$

The Similarity is a value in the interval $\langle 0, 1\rangle$, 0 means that the two paths are completely disjunct, 1 means that they are equal. Note, that we count the AST nodes (N_{AST}), not the CFG (SCoP) nodes, thus

$$Similarity(p_i, p_j) = \frac{N_{AST_{p_i}} + N_{AST_{p_j}} - N_{AST_{p_i \cup p_j}}}{N_{AST_{p_i \cup p_j}}}$$

We define the loss between two paths as loss of the loop transformation potential when these two paths are grouped together. Then the SCoPs are not so large as before and this decreases the optimization potential. We compute this loss as

$$\begin{aligned} Loss(p_i, p_j) &= f_{p_i} \times \sum_{Dep_k \in p_i} V_{Dep_k}(p_i) + f_{p_j} \times \sum_{Dep_k \in p_j} V_{Dep_k}(p_j) \\ &- \left(f_{p_i} \times \sum_{Dep_k \in p_i \cup p_j} V_{Dep_k}(p_i) + f_{p_j} \times \sum_{Dep_k \in p_i \cup p_j} V_{Dep_k}(p_j) \right) \end{aligned}$$

Paths	0	1	2	3	4
0		1.41	4.47	5.10	4.37
1			5.10	5.66	3.25
2				3.16	4.07
3					5.48
4					

Figure 5.17: Example of Loss/Similarity heuristic grouping (table).

where $V_{Dep_k}(p_i)$ is the dependency size of the *k*-th dependency along the path p_i in the CFsG $\{p_i\}$ (resp. $\{p_i \cup p_j\}$ in the second line). The dependency starts and ends at different nodes. It should cross only nodes with input and output degree one in the CFsG when going from the start node to the end node of the dependency. The start node should have output degree one and the end node should have input degree one. Similar definition is valid for $V_{Dep_k}(p_j)$. The *f* is the frequency of the path or of the union of two paths.

After obtaining those two measures, we compute the Loss/Similarity ratio for each pair of paths. This ratio defines the "distance" between the paths. Note that the similarity is a unit-less metric and the loss is not. We consider the similarity as a relative adaptation of the loss, i.e., if the similarity is 1, the loss is not increased. However, if the similarity is smaller, this penalty is transfered to the loss. With this approach, we obtained better results as when using two unit-less metrics. We believe it is because of the larger importance of the loss in our technique. Then we sort the pairs of paths according to distance and start to group paths with the closest distance. The closer the distance, the larger similarity and/or smaller loss.

An example of such an approach is in Figure 5.17. The items in the table determine the distance between the paths as computed by Loss/Similarity measure. We start from the grouping when each path is separate. Then we group two paths with closest distance, i.e., 0 and 1 (distance 1.41). That is another grouping possibility. The third grouping possibility we obtain when grouping also paths 2 and 3 together (distance 3.16). The next possibility is to add path 4 to the group of 0 and 1, etc. This is also depicted in Figure 5.18 where the nodes represent the paths and the edge labels show the distance between two paths. This heuristic is extremely fast and gives good results as we will demonstrate in Subsection 5.7.5.

5.7.4 Heuristic based on Fruchterman-Reingold layout

This heuristic is an extension of the previous one. Here, the inverse of the Loss/Similarity ratio defines an attractive force between two paths (not the distance). Then we also define the repulsive force that is equal to the product of the loop transformation potentials of the two paths, i.e.,

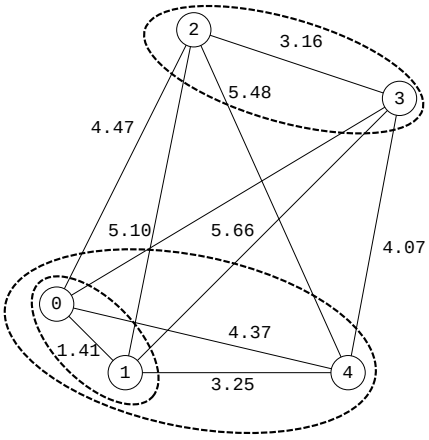


Figure 5.18: Example of Loss/Similarity heuristic grouping (graph).

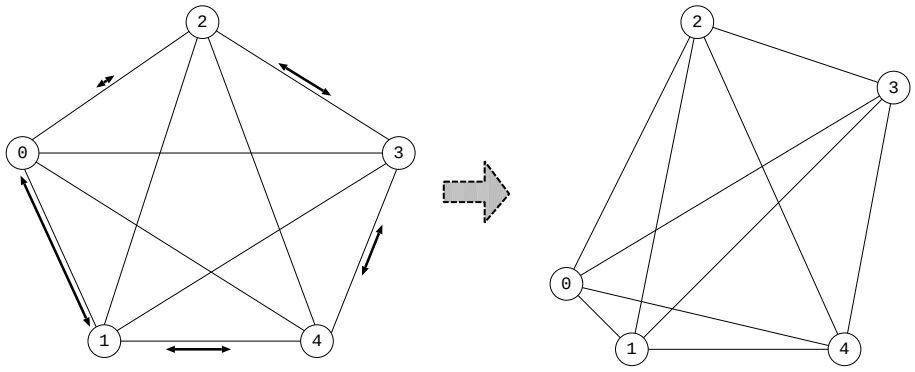


Figure 5.19: Principle of Fruchterman-Reingold layout heuristic. Note, that just some attractive forces are depicted in the figure.

$$Attractive\ force(p_i, p_j) = Similarity(p_i, p_j)/Loss(p_i, p_j)$$

$$Repulsive\ force(p_i, p_j) = \prod_{p \in \{p_i, p_j\}} \left(f_p \times \sum_{Dep_k \in p} V_{Dep_k}(p) \right)$$

We use those forces for force directed layout [78]. We construct a graph with random layout where the nodes represent the paths and we define the attractive and repulsive forces in the graph. Then we apply the forces under the defined cool down function [78]. We have used the default linear cooling function, where the cooling schedule begins with some initial temperature ($100°$ in our case) and gradually (linearly) in time reduces the temperature to zero. Compared to the more general simulated annealing [48] approach, the force directed layout, being a physical simulation involving forces, does not search directly for the energy minimum and is less

computationally demanding. The result is a new layout that is used as a start for our grouping technique, as shown in Figure 5.19. The grouping technique is the same as in the Loss/Similarity heuristic (see Figure 5.18). This approach is a bit slower compared to the previous one because of the new layout computation as depicted in Figure 5.19.

5.7.5 Results

The quality of the heuristics will be shown on a real-life MP3 audio decoder and on a set of synthetic examples generated with an adapted TGFF pseudorandom graph generator [57].

The CFG of MP3 audio decoder has 26 nodes and we focus on grouping exploration of the 6 most active paths (see Table 5.2). Note, that not all paths have to be grouped in the scenario creation. The paths that are not grouped form the backup scenario. The backup scenario is a piece of code that covers all paths that are not a target of the exploration (non-active paths). The profiling we used for active path identification is context sensitive. I.e., our test streams have been music test streams for which this is a representative distribution of paths. In a different context, e.g., for speech decoding other paths will pop up as important and some important paths from the music context will disappear. This will lead to different scenario creation and it is the reason, why our scenario technique is context sensitive.

For the synthetic examples, we use three sets: a small set, a middle set and a large set. We define small/middle/large w.r.t. the complexity and size of the graphs generated for the set, not w.r.t. the number of graphs in the set. The small set has in average 17 CFG nodes and in average 5 selected paths. It contains 20 graphs. The middle set has in average 72 CFG nodes and 7 selected paths. It contains 5 graphs. The large set has in average 188 CFG nodes and 14 selected paths. It contains 3 graphs. The information about the real-life example and the sets can be found also in the Table 5.3 after each set in parentheses in the form (*average CFG nodes in a graph*, *average active paths in a graph*, *nr. of graphs in the set*). We compare the Pareto curve of the full exploration space (see Figure 5.16 for MP3 audio decoder) to the curves obtained with the proposed heuristic. We define the accuracy of the heuristic as ratio between the areas below the two curves (the brute force *bf* and the heuristic *heur*). Our assumption is that we integrate (compute the area under the curve) starting from the solution where each scenario is an individual path (leftmost point) and ending by the solution where all the active paths create one scenario (rightmost point). The resulting accuracy ratio can be in the interval $(0, 1\rangle$. The closer is the ratio to 1 the better is the heuristic. Thus

$$Accuracy(bf, heur) = \int bf / \int heur$$

The results for the real-life MP3 audio decoder and for the synthetic examples are in Table 5.3. In the first column is the name of the testbench, in the second column are the heuristics applied for that testbench, the third column depicts how many solutions has been pruned from the whole exploration space, the forth and the fifth column depict accuracy of the heuristic and time required.

Benchmark	Heuristic	Pruned [%]	Accuracy	Time [s]
MP3 audio	Brute force	0	1	375
decoder[2]	Coverage criterion	80.0	0.91	131
(26,6,1)	Loss/Similarity	97.0	0.93	23.3
	F-R layout	97.0	0.85	23.2
Small	Brute force	0	1	16.6
synthetic	Coverage criterion	16.6	0.99	13.1
examples	Loss/Similarity	90.4	0.93	2.05
(17,5,20)	F-R layout	90.4	0.87	2.01
Middle	Brute force	0	1	666
synthetic	Coverage criterion	17.3	0.99	617
examples	Loss/Similarity	99.2	0.90	3.56
(72,7,5)	F-R layout	99.2	0.83	3.59
Large	Brute force	0	NA	NA
synthetic	Coverage criterion	NA	NA	NA
examples	Loss/Similarity	$99.\bar{9}$	NA	100
(188,14,3)	F-R layout	$99.\bar{9}$	NA	99.1

Table 5.3: Comparing the accuracy and CPU time requirement for different heuristics. The information after each set in parentheses gives an idea about the size of the graphs and number of graphs in the set. It is in the form (*average CFG nodes in a graph*, *average active paths in a graph*, *nr. of graphs in the set*).

Figure 5.20 and first part of Table 5.3 show the results for different heuristics for the real-life MP3 audio decoder. In this paragraph we discuss the results for this real life application. In next paragraph we focus on synthetic examples results. The Loss/Similarity (L/S) heuristic is fast and also accurate, because the ratio between this heuristic and the brute force curve is 0.93 (see Table 5.3). That means the brute force curve is only 7% better (in terms of area) than the L/S heuristic curve. The L/S heuristic prunes 97% of the search space. This number was computed as a ratio of number of pruned complete CFsG sets to all the possible complete CFsG sets. The L/S heuristic has missed only one Pareto point from the brute force Pareto curve. The coverage criterion heuristic has also good accuracy (0.91, i.e., 91%), but is has also high time requirement. This heuristic prunes 80% of the solutions (complete CFsG sets) in the exploration space for our real life example. The Fruchterman-Reingold (F-R) layout heuristic does not fulfill our expectations, despite the good speed, almost the same as the speed of the L/S heuristic. Also, the pruning factor is equivalent to L/S heuristic. However, the F-R layout heuristic is less accurate. Thus for an MP3 the L/S heuristic seems to be the best one.

Second part of Table 5.3 shows the results for three sets of synthetic examples. The results for small and middle set of synthetic examples confirm the previous statements, i.e., the coverage criterion is a slow but accurate heuristic. Here the coverage criterion heuristic outperforms the L/S heuristic in terms of accuracy. This heuristic prunes on average only 16.6% of the exploration space for the small set and 17.3% of the exploration space for the middle set compared to fast L/S and F-R layout

[2]The parsing of the application was not excluded from the time results.

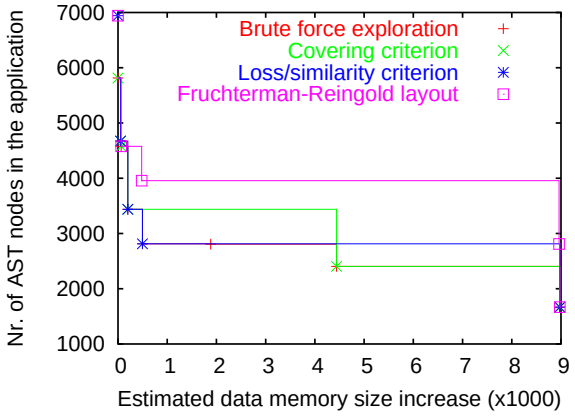


Figure 5.20: Comparison of different heuristics on MP3 audio decoder example.

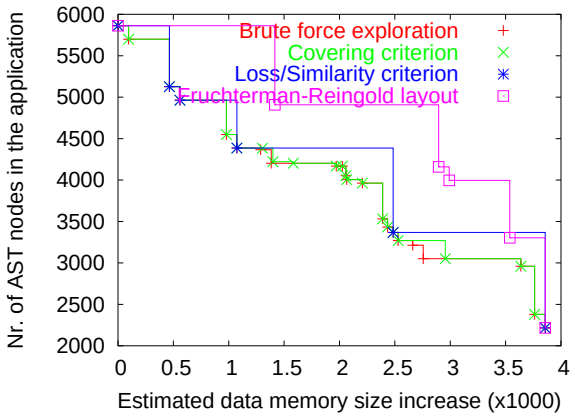


Figure 5.21: Comparison of different heuristics on a synthetic example (N_{CFG}=34 and N_{path}=6) from small set (TGFF seed = 6).

heuristics which prune 90.4% of the exploration space for small set and 99.2% of the exploration space for middle set. Thus, the L/S and F-R heuristics are much faster compared to coverage criterion heuristic. Also here, the F-R heuristic was a disappointment. The reason for the bad performance of the F-R heuristic is probably the strong repulsive force that is also partly involved in the definition of the attractive force (the loss) and should be better calibrated.

The large set of synthetic examples deserves special attention. Here, it was not possible obtain the results for brute force and coverage criterion heuristic due to limited memory resources (out of memory after 1 day of computing). Thus, the L/S and F-R heuristics are the only option from the proposed heuristics, we have for large graphs with a lot of paths. The large synthetic set shows the importance of the applied fast heuristics. If not using those, we would not be able to obtain any results for the large set. Due to the fact we could not obtain brute force results, also the accuracy is not available for the large synthetic set.

The fast heuristics (L/S and F-R) can be still improved by trading-off their accuracy for speed. Now, they provide only one solution for given number of scenarios (CFsGs). After analyzing the Pareto points obtained by the brute force, we have seen that several Pareto points can be obtained for given number of scenarios. We believe, taking more than the best heuristic solution for given number of scenarios for fast heuristics, will result into additional improvement in those heuristics. However, the time requirement will then also increase. The proposed heuristics belong to the class of hierarchical clustering approaches [113]. For the scenario creation, also other clustering approaches such as exclusive clustering (e.g., K-means clustering [138]), overlapping clustering (e.g., Fuzzy C-means clustering [59]) or probabilistic clustering (e.g., Mixture of Gaussians [54]) should be evaluated. Still, those techniques only propose clustering methods itself but do not provide the metrics which are important part of the clustering algorithm and are specific for given problem. We provide those metrics in our clustering heuristics. To evaluate those metrics with other clustering algorithms is part of the future work.

Note the time difference for the small synthetic examples and the real-life test-vehicle which is order of magnitude higher despite similar structure (number of nodes and active paths) of the CFGs. The large time difference is caused mainly by including of the C code parsing (constructing the CFG and GM out of the C code) to the real-example time and also by keeping both the C code and the model in the processor memory during computation for the code generation purpose.

5.8 Code generation and results

After obtaining the Pareto curve, each point corresponds to a set of CFsGs where the set of CFsGs jointly covers all active code paths (see Figure 5.22). The code generation phase takes a set of CFsGs for the Pareto point which is picked by the designer. The selected point depends on the trade-off between code size and data size increase the designer would like to make. From this set of CFsGs the set of C code functions is generated where one CFsG corresponds to one C code function. The C code functions do contain fewer blocking data dependent conditions compared to the original code (original CFG) and thus can be better optimized by the following steps of the DTSE methodology (e.g., the GLT).

Figure 5.22 shows the resulting Pareto curve for the MP3 audio decoder using the coverage criterion heuristic for scenario creation. For the code generation and optimization we have to pick one Pareto point. The choice will determine which scenario grouping (set of CFsGs) we choose for further optimizations. Let us assume we do not have enough L1 instruction memory so we go for cheap option w.r.t. the code size and select the second Pareto point from the right. This point corresponds to two scenarios (two CFsGs), namely {78,90,102} and {197,209,221}. The numbers in the curly brackets are the path numbers of paths which form the CFsG. Because we did the exploration only on the active paths we still need to have one backup scenario (see Subsection 5.7.1) which groups all non-active paths. This will be located in main memory and should be hardly used.

After selecting the Pareto point and generating the code functions, the optimizations (GLT) are applied on these functions. The GLT optimized codes are passed to the

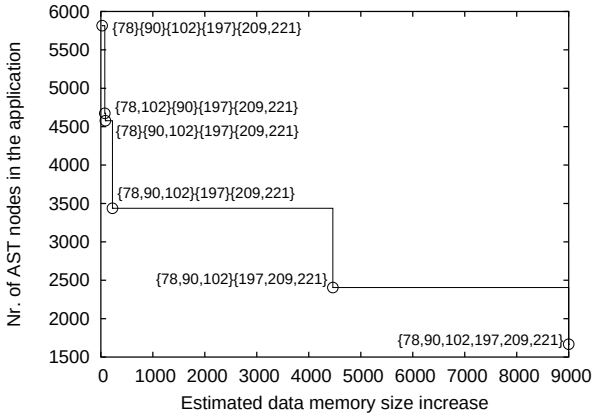


Figure 5.22: Results for MP3 audio decoder using coverage criterion heuristic for scenario creation. Each point corresponds to a set of CFsGs. In the curly brackets there are paths which form the CFsG.

Memory Compaction (MC) tool (see Section 2.2 in Chapter 2), which applies the in-place optimizations [50], and MHLA [26] tool. The MHLA tool decides on the optimal placement of the arrays and their copy candidates to the memory hierarchy layers. A copy candidate is part of an array which is stored in a lower level than the original array. Thus, the lower memory layer serves as a software controlled cache. Obviously, after the GLT transformation local memories can be better utilized. This results to fewer main memory accesses and thus to energy savings. The accesses to the different layers have been profiled using the ATOMIUM data memory profiling tool. We have measured the traffic from/to main memory and have determined the number of main memory accesses in a two-layer memory hierarchy where the size of L1 on-chip SPM is 2kB.

The 2kB L1 on-chip SPM size was selected to show the gains of our technique compared to the approach without scenarios. If we select a too large SPM (e.g., 4kB), all important arrays will be placed in the SPM and there will not be any difference when using GLT with or without scenarios, or not using GLT at all in terms of L1 accesses and L1 misses. Of course, 4kB SPM will consume slightly more energy compared to 2kB SPM. If we do not use SPM at all and use only the Main Memory (MM), all the arrays will be placed in the MM and again, there will not be any difference between using our technique or not. However, this solution will consume a significant amount of energy, because of the costly accesses to the MM. The trade-offs between those two extremes have been studied in [26, 106]. The first paper studies only the impact of MHLA decisions for different memory configurations, the second one studies the impact of GLT+MHLA for different configurations. Such a detailed study as in [106] has to be performed for each point of the trade-off in Figure 5.22 and it is out of the scope of this dissertation. Also, in this dissertation we do not deal with the detailed energy studies of SPM, cache or SRAM depending on size. These can be found in [16, 65].

Table 5.4 shows the results in terms of the number of main memory accesses for the MP3 audio decoder code, the code after GLT and the code after Scenarios and GLT.

	Nr. of main memory accesses	Improvement related to previous row
Original MP3 audio decoder	714.2×10^6	-
MP3 audio decoder after GLT	126.9×10^6	82.2%
MP3 audio decoder after Scenarios and GLT	68.8×10^6	45.8%

Table 5.4: Comparison of original MP3 audio decoder code, the code after GLT and the code after Scenarios and GLT.

We measured, that by using state-of-the-art techniques only (i.e., applying only GLT without scenario technique), we were able to optimize 25% lines of source code of the application, namely frequency inversion and polyphase synthesis filterbank kernels (for the details about the application see Appendix A). This leads to 82.2% reduction in the number of main memory accesses compared to the original version (see column 3 in Table 5.4). The main memory accesses correspond to a significant part of the platform energy cost and they will also result in a performance loss due to the "memory wall" already indicated earlier. This reduction is mainly due to transformations in the synthesis polyphase filterbank kernel (see Appendix A). The remaining kernels contain the data dependent conditions and thus they cannot be optimized by the state-of-the-art techniques. However, after applying the scenario technique we are able to optimize the whole decoding functionality of the application. This leads to a further decrease of 45.8% (see column 3 in Table 5.4) in the number of main memory accesses.

Thus, the reduction of data memory size by 49.6%, when going from the rightmost point to the second rightmost point as estimated in Figure 5.22, reduces the number of main memory accesses by 45.8%. The code size increases by factor of 2. Based on Table 5.7 we have $\sim$2kB of compiled code per scenario (when compiled with optimization for code size) for a set of 3 scenarios. To go to a 2 scenario solution, we have to combine 2 out of 3 scenarios. Even when the 2 scenarios we combine have totally different nodes, we can end up in 6kB per scenario set. This is still not a factor of 2 compared to the original code (4.65kB). There exist two reasons for this which we will discuss in sequel. As mentioned earlier, the rightmost point in our exploration is not the original code, but rather the initial code which corresponds to the initial CFG constructed from active paths only. Thus the initial code size would be smaller than 4.65kB. Also, creating more scenarios enables more optimizations resulting in a code size decrease which is not captured by our estimation. Thus, the code size estimator has to be better calibrated if we would want more accurate estimates. This is however not so obvious to obtain at compile-time especially when we would like the code size estimation of L1 instruction memory because of the dynamic and data dependent nature of our target application codes. This could be useful to estimate e.g., instruction cache misses as depicted in Table 5.7. This estimation if difficult if we do not know the exact activation trace. However, the ultimate accuracy of the estimators is not really needed in our approach. In practice we apply infrequent calibration at run-time to accommodate for inaccuracies of the estimates. If we discover that the shape of the real Pareto-curve moves too much

```
1 while(i<header.count) {
2   body;
3 }
```

a

```
1 //576 is the worst case
2 for (i=0;i<576;i++) {
3   if(i<header.count) {
4     body;
5   }
6 }
```

b

Figure 5.23: Rewriting of while loop: (a) original code segment; (b) code segment after rewriting while loop to a for loop and a condition.

from the original working points, we have to recalibrate and if needed even regroup the scenarios. Also the latter will happen only very rarely so the runtime overhead will remain negligible.

In the scenario approach we create at compile time the set of CFsGs which covers the whole functionality of the application. We call the CFsGs scenarios. During run time we have to select the appropriate scenario. How to select this scenario has not been discussed yet. In front of data information in each frame in the MP3 audio decoder is the header information which tells us which type of frame we are going to decode. Based on this information we also can choose the appropriate scenario for decoding the incoming frame. The decision mechanism is just large switch statement which redirects the control-flow to the appropriate scenario based on the header of the incoming frame. This decision mechanism is generated automatically by a perl script in our technique. If the header information would not be available, we would have had to use accurate scenario predictors similar to state-of-the-art branch predictors or specific scenario predictors, e.g., [55, 84, 86]. If the predictors would be wrong, we would have had to recover the input frame which can be time consuming.

5.9 Dealing with while loops

Till now we have considered that the kernels in the applications are SCoPs and thus can be GM modeled. This requires statically analyzable code without *while* loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters. Thus the kernels cannot contain data dependent conditions. If there is data dependent condition in the kernel it has to be hidden using technique in Chapter 4.

For the *while* loops there exist several solutions. In [90] the authors use irregular non-dense execution domains with run-time checks which could lead to large run-time overhead. In [164, 163] the authors do not deal with the while loops at all. So the nodes that contain while loop(s) cannot be GM modeled. This reduces the opportunity for GLT. The other option is to use the technique in [165]. Here, the while loop is rewritten to a for loop with worst case upper bound and a condition within the body. When the condition is true, the original body is executed. Otherwise, an empty iteration is performed (see Figure 5.23). The condition can be hidden by the technique described in Chapter 4. When using this approach we create extra over-

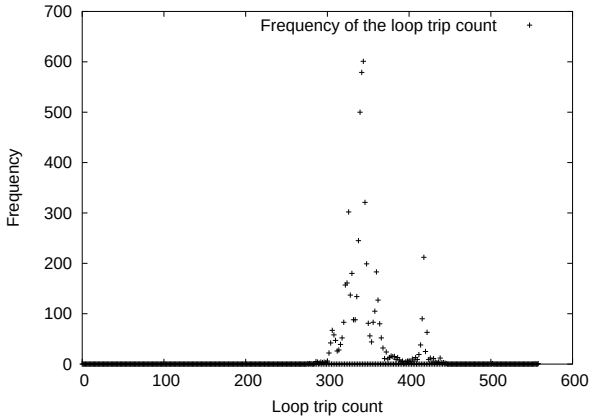


Figure 5.24: Trip count distribution for a while loop in the MP3 audio decoder.

head in empty iterations and possible redundant accesses to the memory subsystem when the condition is hidden. In that sense, this approach is similar to [90]. In this section we propose combination of preprocessing and scenario approach using loop profiling information. Scenario approach for loops was also used in [195] in the context of scenario-aware data flow models. Because of the preprocessing aspect, our approach has a cross-level nature with previous chapter.

To use always the worst case is not necessary if we know in advance the loop trip count profiling information, i.e., the frequency of different number of iterations of the loop and the upper bound of the trip count during run time [167]. Such information is depicted in Figure 5.24. From the figure we can see that in most cases only half of the worst case trip count is executed. Thus we can create a special case which will be used in most of the cases and the worst case which will be used otherwise. In general multiple frequent occurring special cases, having different trip counts, can be exploited.

In the following two subsections we explain how we obtain loop trip count information using profiling, and how we use this information to deal with loops with varying trip count.

5.9.1 Profiling of the loops with varying trip count

In this subsection we discuss how we can profile the loops in the application and obtain the trip count distribution of individual loops. Then we explain how we select only the loops that have varying trip count and that are the main target of our technique.

To prepare the code for trip count profiling we automatically insert function calls to *start_prof()*, *call_prof()* and *finish_prof()* functions when traversing the application Abstract Syntax Tree (AST) (for an example of the code after annotation see Figure 5.25). The *start_prof(i)* function call is inserted immediately before the loop, the *finish_prof(i)* immediately after the loop and the *call_prof(i)* as the first statement of the loop; *i* refers

```
if(header.block!=0) {
  if(header.block==1) {
    start_prof(5);
    for(i=0;i<header.count1;i++) {
      call_prof(5);
      ...
    }
    finish_prof(5)
  }
  ...
}
```

Figure 5.25: Example of the code annotation before trip count profiling.

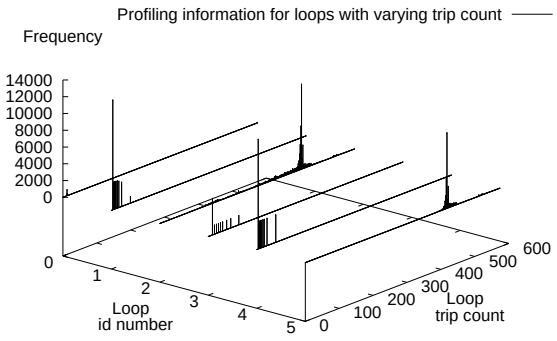


Figure 5.26: Trip count distribution for all while loops (with varying trip count) in the MP3 audio decoder.

to the loop number. After inserting the profiling code in the application we run the application using realistic input data. During the profiling, the trip count information is stored in a 2D dynamic array, where the first dimension is the loop number, the second dimension is the trip count of the loop. The content of the array is the frequency for given loop and given trip count.

We would like to profile only loops that have varying trip count. The loops which have constant trip count are modeled using the geometrical model as described in [19, 163]. Thus after profiling we prune all loops which have only one trip count value during the whole profiling. We still have to check if those loops are really *for* loops in the AST. If not, we can work with these loops as with the *for* loops where the obtained trip count will be the upper bound of the *for* loop. However, the worst case of those loops has to be considered in the backup scenario. After pruning we can plot the trip count histogram of all varying trip count loops in the application.

Figure 5.26 depicts the distribution of trip count in 3D space for an MP3 audio decoder. The x-axis contains the loop id numbers, the y-axis contains the trip count values and the z-axis contains the frequencies for given trip count and given loop. We observe that loops 0,1,3,4 have trip counts up to 100 and loops 2,5 have trip counts up to 400 with a high peak there. This is already interesting information, be-

cause if we had directly used the technique in [162] we would need to consider the worst case for all loops, which is 576 for all mentioned loops. However, they occur in different paths of the code and are used for decoding different type of blocks. This is why there is such a big difference in the profiling information of the *while* loops in the code. We will use the trip count histogram information in the next subsection for creating scenarios out of the original CFG.

Creation of scenarios for loops with varying trip count will be demonstrated on loop 5. This has several reasons. In our approach we can construct scenarios from directed acyclic CFGs only. As mentioned in Section 5.2, this is rather an implementation issue. Only the loop 5 is the outermost loop, i.e., there is not any other loop between the direct acyclic control-flow within the time loop and this loop. The other loops are nested deeper in the loop hierarchy and thus are not so interesting for scenario approach. Besides that, loops 0 and 3 are on infrequent paths. Also, this part of the exploration (scenarios with varying trip count) is still manual. After scenario approach would be extended for arbitrary CFGs and the scenarios with varying trip count would be automated, different combinations of multiple loops should be evaluated. Still, we do not expect that this will outperform our decision for the MP3 application.

5.9.2 Scenario technique for loops with varying trip count

The scenario technique explained in Section 5.6 requires that the CFG nodes fulfill the requirement of SCoP. Thus, when *while* loops are present in the kernel, the worst case number of iterations is used. However, when we define several special cases we can reduce the number of empty iterations. We assume that the number of iterations is data-dependent and thus not analyzable at compile-time. However, for the MP3 application it is known at run-time before entering the decoding frame functionality as depicted in Figure 5.25. Here, the number of iterations is dependent on the header information *count1*.

The best case is when we create a special case (separate *for* loop) for each loop iteration count. Then the number of iterations is

$$N_{iter} = \sum_{TC=0}^{WC} (f_{TC} \times TC)$$

where N_{iter} is the number of iterations, *WC* is the worst case trip count, *TC* is the trip count and f_{TC} is the frequency of this trip count. The N_{iter} corresponds to the number of iterations executed when the while loop is present. However, creating a special case for each trip count will create a large instruction memory footprint overhead similar to creating separate scenario from each path. If we decide to create only one special case (to have one special case and one worst case) all the occurred trip counts below the special case (and the special case) can use this implementation and the remaining trip counts have to use worst case implementation. Thus, the number of iterations is

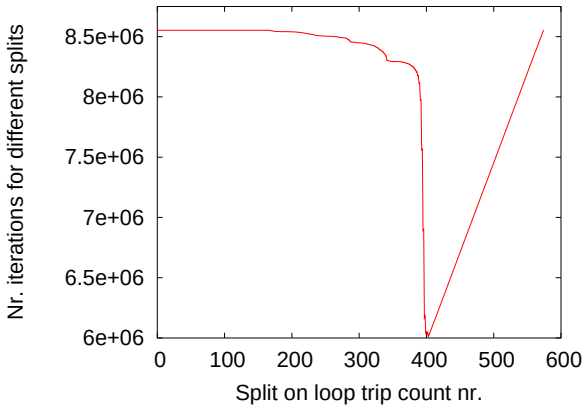


Figure 5.27: Number of iterations depending on the position of the split for 1D split for loop id number 5 from Figure 5.26.

$$N_{iter}(X_1) = \sum_{TC=0}^{X_1} f_{TC} \times X_1 + \sum_{TC=X_1+1}^{WC} f_{TC} \times WC$$

where N_{iter} is the number of iterations, X_1 is the split trip count (till this trip count the special case will be used), and *WC* is the worst case trip count. Depending on the position of the X_1, the $N_{iter}(X_1)$ will vary (see Figure 5.27). From this graph we can identify the X_1 where the $N_{iter}(X_1)$ is minimal which will be the best split for one special case.

Of course, we can do more splits than only one and create several special cases (till each trip count has its special case as explained above). For creating N splits we can define the number of iterations achieved with those splits as

$$N_{iter}(X_1, ..., X_N) = \sum_{i=1}^{N+1} \left(\sum_{TC=X_{i-1}+1}^{X_i} f_{TC} \times X_i \right)$$

where N_{iter} is the number of iterations, $(X_1, ..., X_N)$ are the positions of N splits, $X_0 = -1$ and $X_{N+1} = WC$ (WC is the worst case iteration count of the *while* loop). Note that this is an N dimensional function. Its minimum defines the N splits with the minimal number of iterations.

As explained already above, Figure 5.27 depicts the function of number of iterations depending on the position of the special case (the split) for N=1. The trip counts have been obtained based on Figure 5.26, loop id number 5. We can observe that the best split is after the peak of the trip count frequency. Thus the special case should have 400 number of iterations. Note, that the worst case is 576 iterations. Using number of iterations as cost function for the split assumes that the operations performed during each iteration are the same. If it is not so, this cost function has to be adapted and weighted appropriately. We will explain this with an example in the next paragraph.

```
    i=0;
    while(i<header.count) {
      if(i=<100)
        y=x<<3;
      else
        y=A[i]*7;
      i++;
    }
    ...
}
```

Figure 5.28: Example of the code when loop structure executed has changed after the 100th iteration.

The trip count itself and derived N_{iter} function as shown in Figure 5.27 is not necessarily sufficient as a cost function. In Figure 5.28 we see an example where the first 100 iterations are cheap compared to the remaining iterations. In the iteration interval $\langle 0, 100 \rangle$ the simple shift operation is performed in the loop. However, after the 100th iteration, the more complex multiplication is performed. Also, those iterations contain costly memory access we are concerned with. This difference has to be taken in the account in our cost function and thus each iteration has to be weighed based on the code complexity and memory accesses which are executed in that iteration. This is true for one dimensional instance of our cost function as well as for multi-dimensional general cost function as defined above. However, this is part of the future research.

As mentioned above, the special case causes the reduction in the number of empty iterations. However it requires also duplication of code, where two CFG nodes, the worst case and the special case have to be considered during CFG creation. This is depicted in Figure 5.29; Figure 5.29a shows the CFG without considering trip count profiling information (only the worst case node is present). Figure 5.29b depicts the CFG where both the worst case and the special case are considered. After obtaining the header information which is at the beginning of the frame, we know the iteration count for that frame and thus the right decision if we are in the special case or not can be taken. However, if this information would not be present, the predictors as mentioned already at the end of Section 5.8 would have to be used.

After creating the new CFG with the worst case and special case the Ball-Larus profiling is applied. The comparison of Ball-Larus profiling when only worst case is used and when both the worst case and the special case are used, is shown in Table 5.5. Note that the paths which have zero frequency do not occur in the table. However, they will be grouped together to form a backup scenario. As we can observe in Figure 5.29 two paths going from start node to the end node through the worst case node in the original CFG are refined to four paths in the new CFG. In Table 5.5 we see the individual path frequency information for the original CFG and the new CFG for an MP3 audio decoder application after the refinement. The path label is in the form XD_{nr} where X is the letter identifying the path in the old and new CFG, D is the number identifying more detailed path information in the new CFG and *nr* is the path number in the corresponding CFG. Note that only one *while* loop was split in the application, resulting in splits of three paths going through this loop into six paths. Our new approach where we consider also *while* loops, gives us a more de-

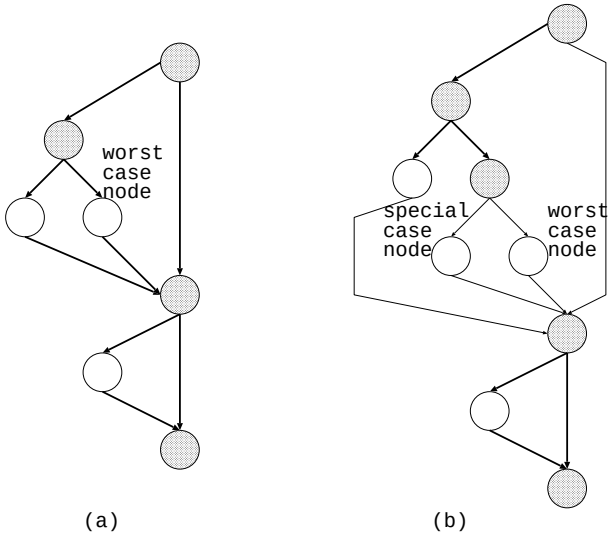


Figure 5.29: Original CFG without considering trip count profiling information and new CFG considering trip count profiling information and derived cost in Figure 5.27.

tailed view, as we see in Table 5.5. Thus it enables a better exploration of the search space. After the profiling we apply the scenario technique (see Section 5.6) which groups the individual paths together in the set of CFsGs. The heuristics proposed in Section 5.7 can be used to prune the exploration space.

We have applied the proposed technique on the MP3 audio decoder application and have compared it to the approach when only worst case for the *while* loops is used. When only the worst case has been used, we have identified for the MP3 audio decoder 234 paths. After considering both worst case and special case for loop number 5 we have identified 378 paths. For the worst case, only 6 paths are active and for the worst and special case 9 paths are active as depicted in Table 5.5. The total number of different groupings is then $B_6 = 203$ w/o while loop support (i.e., when using the worst case) and $B_9 = 21147$ with while loop support (when using both, special and worst case). B_n is the Bell number defined as the number of ways a set of n elements can be partitioned into non-empty subsets [21].

We applied the heuristic described in Subsection 5.7.2 to compare the results when using only worst case and both worst case and special case for the MP3 audio decoder. Using the coverage criterion heuristic the number of relevant groupings decreased from 203 to 61 w/o while loop support (worst case) and from 21147 to 2313 with while loop support (special case and worst case) respectively. Finally, 6 Pareto points were obtained for scenario technique w/o while loop support and 13 Pareto points were obtained for scenario technique with while loop support (see Figure 5.30). The non-Pareto points have been discarded using techniques in [227].

In Figure 5.30 we see the Pareto curve obtained when using only the worst case approximation of while loop, and the Pareto curve obtained when using both worst case and special case. On the x-axis is the estimated data memory size increase which corresponds to the missed opportunity for GLT expressed in number of dependen-

W/o while loop support		With while loop support	
Path nr	Freq	Path nr	Freq
A_{78}	51	A_{126}	51
B_{90}	336	$B1_{138}$	19
		$B2_{150}$	317
C_{102}	100	C_{174}	100
D_{197}	765	D_{317}	765
E_{209}	12079	$E1_{329}$	3398
		$E2_{341}$	8681
F_{221}	2363	$F1_{353}$	784
		$F2_{365}$	1576

Table 5.5: Comparing the characteristics of Ball-Larus profiling [15] for the CFG used in Section 5.5 and the new CFG created utilizing the information from trip count profiling.

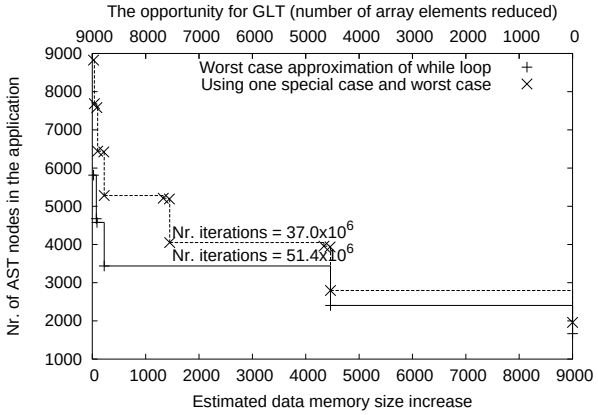


Figure 5.30: Comparing the results of scenario creation for scenarios without splitting the while loops (considering worst case) and with splitting the while loops (only 1 while loop considered) based on the trip count profiling. The result shows that the split has reduced the number of iterations compared to approach that has considered only the worst case situation.

cies that cannot be modeled due to data dependent control flow. We can look at this value also as the number of possible array elements that could still be reduced in GLT if we perform the GLT on individual paths. On the y-axis is the code size increase in number of AST nodes that we have to pay for increased GLT opportunity.

Based on Figure 5.30 we see that our technique requires more duplication and has the same amount of GLT potential as the technique w/o while loop support when considering worst case. This is obvious, because by creating scenarios for while loops we duplicate additional code and we do not gain any additional GLT potential compared to worst case approximation. However, we reduce the number of iterations when compared with the worst case. In the while loop number 5 we have focused on, we have gained 14.4×10^6 iterations compared to the technique when only worst case is used. The original number of iterations when applying the previous technique was 51.4×10^6 for this loop. After our technique we reduced the number of iterations to 37.0×10^6. That means we have obtained 39% improvement for that loop in number of iterations. Thus, even when splitting while loops always gives a dominated solution when compared to non splitting loop in the two-dimensional data memory size vs. code memory size exploration space, this is not true any more in the three dimensional data memory size vs. code memory size vs. number of iterations exploration space. This is also depicted in Figure 5.30 where the projection to 2D space data memory size vs. code memory size with umber of iterations above the curves is present. Except of splitting the while loops, the other approach is not to use worst case and leave while loops in the application code. This approach requires fewer iterations compared to our worst and special case approach. However, it has also fewer GLT opportunities and gives yet another Pareto working point in the 3D exploration space.

As we have said above, our approach requires larger code size increase because of the duplication of the while loop bodies. The code size increase happens only in the main program memory however, and not in the active part of the code that is loaded to the L1 instruction memory and the loop buffer. Also the code size increase drawn in Figure 5.30 happens only for the critical parts of the program. Thus the overall program size increase is still small. The whole MP3 audio decoder has 4262 lines of code. However, the kernels are described only in 529 lines of code. I.e., we are working with 12.5% of code which is responsible for 99% of execution time.

5.10 Switching cost

The scenario technique produces several C code functions where each C code function covers certain code paths (see Section 5.8). Depending on the incoming frame, the appropriate function which was generated from a CFsG covering some code paths is selected. Thus this function has to be loaded to the foreground instruction memory and executed. When the size of the foreground instruction memory is limited, the previous function residing in the memory is flushed because of switching to the new scenario. The switching activity depends on how the partitioning of the original CFG to the set of CFsGs is done. A small example - if we have paths 1,2 and 3 and the activation trace is 1, 2, 1, 2, 1, 2, 3, 3, 3, 3, 1, 2, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, ... obviously we would like to group path 1 and 2 together and keep path

```
At the beginning initialize histogram, time, last_occurence
and AvgT to 0. Then each time during entering the EXIT node
in Ball-Larus profiling call update_distances():

update_distances(
Input: path occured,
Output: updated average distance matrix AvgT) {
  //increment the counter for path (Ball-Larus profiling)
  histo[path]++;
  //increment the global time
  time++;
  for(i=0; i<NR_PATHS; i++) {
    if(last_occurence[i]) {
      x = (path<=i) ? path : i;
      y = (path<=i) ? i : path;
      //time between path and i
      tmpT = time-last_occurence[i];
      //average time: path and i
      AvgT[x][y] = AvgT[x][y]*(histo[path]-1)/histo[path]
                   + tmpT/histo[path];
    }
  }
  last_occurence[path]=time;
}
```

Figure 5.31: The update_distances() algorithm for determining the time distance between two paths.

3 separate. If we would group path 1 and 3 together (first CFsG) and keep path 2 separate (second CFsG) this will lead to constant switching between the first CFsG and the second CFsG. The importance of this switching depends heavily on how fast the switching will occur in practice. Note, that in this section we assume a special case where the switching between paths is very frequent. This can happen in future very dynamic applications. If the scenario is stable for some time, the switching cost is not so relevant.

In this section we provide a technique how to determine the set of CFsGs with minimal switching activity using Ball-Larus profiling [15] information and the Fruchterman-Reingold layout [78]. I.e., the technique should, based on the activation trace, provide such a grouping of the individual paths to the set of CFsGs that the switching activity (flush and load of scenarios) is minimal. To do so, first we propose to collect some extra information on top of Ball-Larus profiling and then use that information in the Fruchterman-Reingold layout.

5.10.1 Description of the profiling algorithm

Ball-Larus profiling (see Section 5.5) collects the histogram of paths during execution of the program. When executing a path in the CFG, the Ball-Larus profiling increments the counter for that path. We extend the simple updating of the path counter by adding extra information about average time between two paths. I.e., we compute how long ago in time units, where the time unit is equal to the execution of one path, from the current path were other paths taken. A similar idea was presented in [24] for determining the backward reuse distance. The algorithm for determining

Path	1	2	3
1	2.5	2	1.5
2	-	0	1
3	-	-	1

Table 5.6: Example of AvgT matrix defining the average distances between the paths for the activation trace 1, 2, 3, 3, 1, 1.

the average time between two paths is listed in Figure 5.31.

After this algorithm which is running together with Ball-Larus profiling the two dimensional AvgT array contains the average time between each pair of paths in the program. An example of the activation trace and obtained AvgT array is in Table 5.6. The AvgT array information can be used in two different ways; to evaluate the switching activity of the already selected set of CFsGs or to optimize the scenarios for minimal switching activity. Here we assume a special case when the switching cost is approximately uniform and does not depend on the source of switching and the destination of switching. The detailed study of the switching activity has been left for future work in the Multi-Processor System on Chip (MP-SoC) context where the impact of the switching is large [226]. As mentioned above, the AvgT array information can be used also to find the optimal grouping (set of CFsGs) w.r.t. switching activity and loading of scenarios based on Fruchterman-Reingold layout [78] as we will discuss next.

5.10.2 Using Fruchterman-Reingold layout for minimal switching activity scenarios

The AvgT array defined in the Subsection 5.10.1 contains the average time between each pair of paths in the program. When considering optimal switching activity between the CFsGs in the CFsG set, we would like to group together paths that are close in the average time. If we do not do so, the switching activity between the different CFsGs will be high. Table 5.6 contains also information how far in time, on average, are two consecutive runs of the same path. This information is on the diagonal of the AvgT two dimensional array. If that average time is small, it means that the corresponding path is executed in short time intervals again and again. This makes it a good candidate for a separate CFsG.

When using Fruchterman-Reingold layout [78] to determine minimal switching activity scenarios, we can look at AvgT array as at a matrix defining the attractive/repulsive forces between the vertices where the vertices represent the individual paths. The attractive force between two paths is the inverse of the value in AvgT matrix. The closer the paths are in average time the more we would like to group those paths and thus the bigger the attractive force should be. The repulsive force between two vertices, i.e., paths, is the product of the inverses of the corresponding values on the diagonal of AvgT matrix. The closer the executions of the same paths are the more this path would like to form the individual CFsG and repulses other nodes. The inverse function we use for deriving the forces is

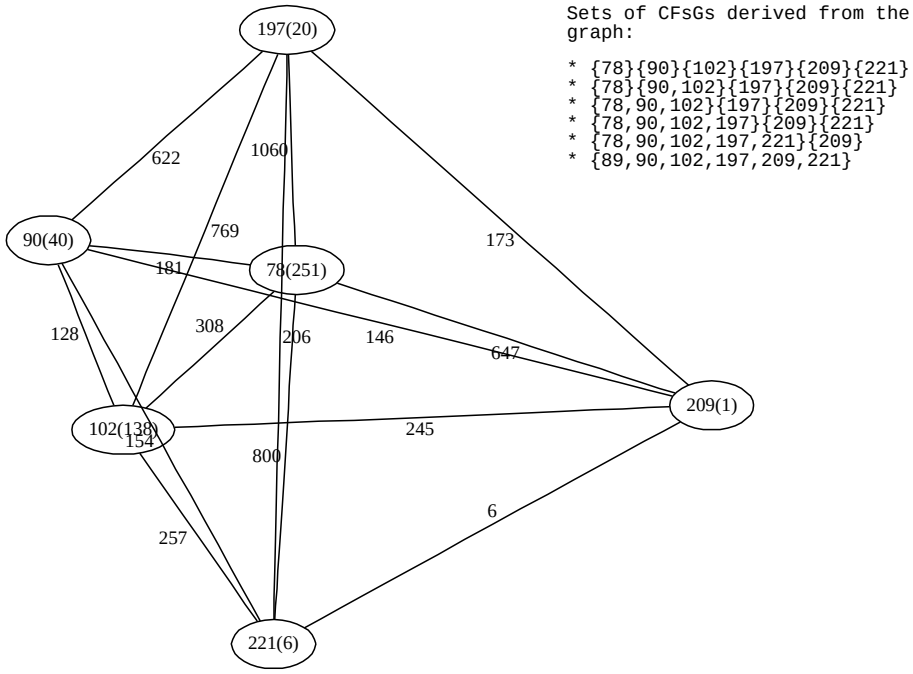


Figure 5.32: The graph after Fruchterman-Reingold layout where the nodes represent the paths and the edges the attractive forces between the nodes. The number in the node is the path number and the number in the parentheses in the node is the $AvgT_{i,i}$ value where *i* is the path number and contributes to repulsive force. The numbers on the edges are $AvgT_{i,j}$ values where *i,j* are the path numbers and define attractive forces. The closest nodes are grouped forming sets of CFsGs (sets of scenarios) with minimal switching activity. The different groupings are listed on the top-right of the figure.

$$Inversefunction_{i,j} = \frac{1}{1 + \ln AvgT_{i,j}}$$

The logarithm in the function reduces the rich spread of AvgT values we have observed.

After defining the vertices and attractive/repulsive forces we use a random layout for the first layout of the vertices. Then we apply the forces under the linear cool down function. However, Fruchterman-Reingold layout [78] considers constant attractive/repulsive forces between each two nodes connected with an edge. Fortunately, the Boost Graph Library (BGL), which we used for implementation, allows to redefine these forces.

After running Fruchterman-Reingold layout the graph layout should be improved with respect to the minimal switching activity. I.e., the vertices (paths) that would result in large switching activity are close together and can be clustered to one CFsG. We sort the edges of the graph based on minimal length. We start iterate over such a sorted collection of edges and cluster the vertices at the ends of the edge together. The more we cluster the smaller the code size and the bigger the data size will be.

	Size	Size (compiled with gcc -Os)	Nr. I1 reads (total)	Nr. I1 reads (Scen. code funcs only)	Nr. I1 misses (Scen. code funcs only)
Original	8k	4.65k	2.83×10^9	1.32×10^9	11.0×10^6
Switching cost	∼3k per scenario	∼2k per scenario	2.83×10^9	1.32×10^9	6.0×10^6
No switch cost	∼3k per scenario	∼2k per scenario	2.83×10^9	1.32×10^9	13.7×10^6

Table 5.7: Comparing code size, code size when compiled for this objective (-Os switch), Nr. I1 reads (total and within Scenario code) and Nr. I1 misses for 3 codes; original code w/o scenarios, scenario code (set of 3 CFsGs) with considering the switching cost and scenario code (set of 3 CFsGs) w/o considering the switching cost.

However, the obtained points (sets of CFsGs) should have smaller switching activity compared to other possible sets of CFsGs.

In Figure 5.32 is an example of a graph for the active paths of the MP3 audio decoder after Fruchterman-Reingold layout. The node id numbers correspond to the path numbers in the MP3 audio decoder. The number in the parentheses inside the node corresponds to the diagonal values of the AvgT matrix and the edge labels correspond to the non diagonal values of the AvgT matrix. If we start to cluster the paths we get six sets of CFsGs representing six points in the exploration space that are also listed in Figure 5.32. We have implement the technique using the C++ libraries - BGL [230], ATOMIUM (and BACKBONE (BB)) (which are part of in-house IMEC ATOMIUM framework [229]) and STL [234]. We tested one MP3 audio decoder scenario set (set of CFsGs) against the original code and scenario code where switching cost is not considered using cachegrid [231]. Cachegrind is a cache profiler which performs detailed simulation of the I1, D1 and L2 caches in the CPU. It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries. Cachegrid allows manually specify one, two or all three levels (I1/D1/L2) of the cache from the command line. For our purposes, we specified I1 as 2048 byte, direct mapped cache with line size 32 bytes.

The results of the experiment are in Table 5.7. When switching cost is not considered, the number of I1 misses can be even worse (by 24.5%) than the original code. However, when the switching cost is considered, we were able to reduce number of I1 misses by 45.5% compared to the original code. Note that we counted only I1 misses related to the scenario code, not the total number of I1 misses. We have observed, that the number of I1 misses to the functions called within scenario code, e.g., the "power" function or Discrete Cosine Transformation (DCT) function, which have not been considered in the last two columns in Table 5.7, can even increase for solution where switching cost is considered. Note that those functions have not been taken in the account during our exploration. W.r.t. the Figure 5.22, the "Switching cost" solution {78,90,102,197}{209}{221} has 3941 AST nodes and 495 estimated data memory size increase compared to Pareto solution {78,90,102}{197}{209,221}

which has 3437 AST nodes and 200 estimated data memory size increase. However, it has fewer I1 misses compared to this solution. Thus, if the switching cost is relevant, we should introduce it as an additional axis in our exploration space, similar as we did for the number of iterations in Section 5.9. But actually working this out in detail is left for future work.

5.11 Conclusions

Current real-life applications cannot be fully analyzed at compile time due to a lot of data dependent conditions at different loop nesting levels. Similar to any branching for instruction scheduling, these conditions limit the exploration space for high-level memory optimizations. Thus, the static "compile-time" models like the GM are not sufficient any more. Therefore, these models have to be extended and combined with more dynamic approaches such as the scenario approach we proposed in this chapter. The scenario creation technique allows more optimizations at compile time, where the decision on the particular scenario use is postponed to run-time.

After the problem definition, our model that combines the current GM with the CFG of the application has been explained. We have shown how to extract this model from a real-life application and how to synthesize our model using TGFF pseudo-random graph generator. Because of the limitations of the generated TGFF graphs, we have proposed a workaround resulting in wide range of synthetic CFG graphs with the desired properties. It has been clarified how to obtain the profiling information and how it is integrated into our model. We have proposed the scenario technique which is one of the main contributions of this dissertation and have suggested several heuristics to deal with the exponential complexity of the problem. In the last two sections, extensions of the technique towards *while* loop specialization and considering switching cost overhead have been provided.

CHAPTER 6

Trade-offs in the GLT

There are no solutions. There are only trade-offs.
Thomas Sowell
(1930-)

Nowadays, multimedia systems deal with huge amounts of memory accesses and large memory footprints. To alleviate the impact of these accesses and reduce the memory footprint high-level memory exploration and optimization techniques have been proposed. These techniques try to more efficiently utilize the memory hierarchy. An important step in these optimization techniques is the application of GLTs (see Chapter 3). They have a crucial effect on later data memory footprint optimization steps and code generation (see Chapter 2). However, most state-of-the-art work has focused only on individual objectives. The main objective studied in the literature involves improving the locality of data accesses and thus reducing the data memory footprint. Usually this work does not consider the trade-offs in the GLT step in relation to successive optimization steps. Therefore it is not globally efficient in mapping the application on the target platform.

This chapter discusses several trade-offs during the GLT and makes the GLT trade-off oriented. It shows that best locality does not automatically ensure the optimal solution for the used platform instance and that trade-offs should be involved during loop transformations when the used platform instance is unknown. This chapter first explains the problem of current design flows. Then it explains the different trade-offs on small educative examples followed by a formalization of the interactions among the different trade-off components. It also provides a real case study of the QSDPCM [193] application mapped to the ARM platform. In the case study we target the RISC and Very Long Instruction Word (VLIW) processor family where the exploitable ILP is increased by loop unrolling in the later compilation phase after our high-level exploration. We will explain how high-level estimation techniques should help us to capture these trade-offs and how important it is to steer the research towards this direction. A short overview of the joint research work in this area will

be given. At the end of this chapter we will show, using the MP3 audio decoder example, one common pitfall. Using the TI cl6x compiler we will show how GLTs can conflict with the kernel optimization and how the computation-storage trade-off at the algorithmic level can help us solve this issue.

6.1 Problem definition

Recent advanced multimedia systems typically use large amounts of data storage and transfers. Memory and bus consume a major part of the energy in the embedded systems [53, 148]. This is due to initial bad data locality. Improving the data locality by loop transformations has positive effects on both speed and energy consumption. The two immediate benefits of data locality, i.e., enabling the in-place optimization and enabling the data reuse analysis and memory hierarchy layer assignment [28], have been already discussed in Chapter 4.

The benefit of the overall data locality on the speed and power was shown by many groups in the past. However, most previous work has focused only on one optimal solution for loop transformations for a particular cost function which was mostly minimal lifetime of the individual array elements [28]. A few exceptions have been published going beyond this single focus [210, 178, 18]. The first paper studies overhead of data memory optimizations on the instruction memories in embedded processors and proposes appropriate countermeasures which keep both, instruction and data energy, low. The second paper provides an algorithm for code generation from Geometrical Model which enables some control over the trade-off between code size and control overhead. Still, the primary goal of the paper is to handle any non-convex union of domains with multiple statements, and allowing to systematically eliminate inner conditionals. The tuning parameters it offers for code generation are quite limited. Basically, the only parameter which can be specified for code generation is the loop depth where to apply domain separations. [18] extends this approach and offers several code generation parameters such as first and last loop depth to optimize in control. However, nobody systematically studied and identified the particular cost components contributing to the data locality (which is crucial for global energy reduction) and the possible negative effects of data locality improvements.

Figure 6.1 depicts the large group of loop transformations and the different cost components they affect (depicted by arrows). The Global Loop Transformations (GLT)s we target mainly in this dissertation are an important subgroup of all loop transformations applied. These transformations are applied mostly over loop nests in the program and usually have the overall regularity and locality as the main objective. Except of these transformations, also other loop transformations exist. These transformations are applied after the GLTs and are constrained by those global transformations. An example are the transformations targeting innermost loop nests for better performance and bandwidth. An important part of all loop transformations is the Geometrical Model (GM) scanning which generates the code from the model used in those transformations. The different transformations influence different cost components like inter in-place, intra in-place, data reuse, control flow complexity (including instruction locality), code size and parallelization for Instruction Level Parallelism (ILP), etc. These cost components determine the final area vs. perfor-

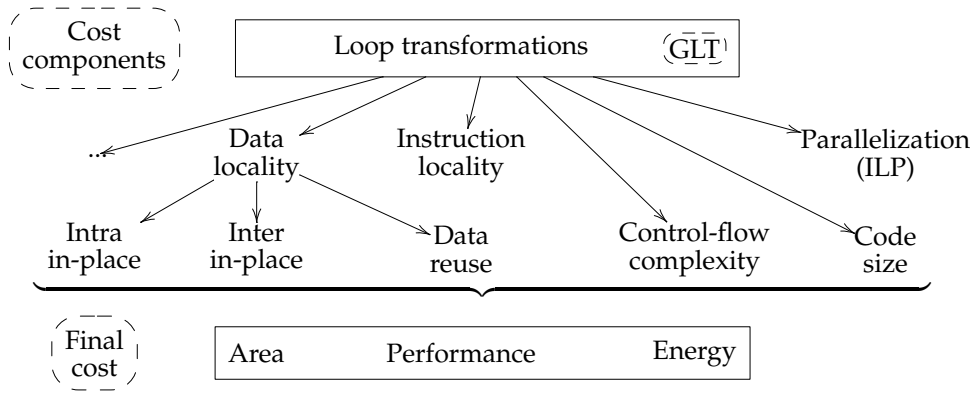


Figure 6.1: Relation between loop transformations (GLT = Global Loop Transformations), cost components and final cost.

mance vs. energy trade-off as depicted in Figure 6.1.

This chapter identifies the effects of the GLT on the different cost components such as inter in-place, intra in-place, data reuse and control flow complexity. To identify and evaluate these cost components is important. Only then, the loop transformations can be properly steered for a particular platform with particular data-path and memory hierarchy. This can be achieved by using high-level estimators which can give us estimations on the different axes we trade-off.

6.2 Trade-offs demonstrated on educative examples

Loop transformations improve the source code implementation in different aspects, e.g., data memory footprint, instruction memory footprint, control flow complexity, parallelism, etc. Nowadays, researchers focus only on one particular aspect. However, an optimum for one particular aspect may not be an optimum for another one.

We would like to provide the designer with multiple source code implementations after the loop transformation step. The implementations will differ in the amount of optimization for different aspects we mentioned in the previous paragraph and they will have different platform requirements. However, all of these implementations will be Pareto optimal solutions in our exploration space. A Pareto optimal solution cannot be improved upon without hurting at least one of the criteria.

To provide Pareto optimal solutions is crucial when mapping the application onto the platform. Only then can we select a (loop transformed) implementation that is tailored to the platform requirements. The Pareto optimal points can also be used during run-time selection [224] for an approach that exploits the Pareto point curve opportunities in a combined design/run-time method. When the platform resources change, we are able to switch to another implementation, again tailored to the platform requirements.

In this section we provide an overview of trade-offs we have identified during loop

Subsection	Cost components					
	Intra in-place	Inter in-place	Data reuse	Control-flow complexity	Code size	Parallel. (ILP)
6.2.1 and 6.4.1	X	X				
6.2.2 and 6.4.2	X		X			
6.2.3 and 6.4.4	X			X		
6.2.4	X					X
6.2.5				X	X	
6.4.3			X	X		

Table 6.1: Overview of the different trade-offs discussed in this chapter. In each row, the cost components involved in the trade-off are indicated with (X) in the corresponding column.

transformations. Each subsection explains one particular trade-off on an educative example. In our global approach in Section 6.4 we will demonstrate the trade-offs in a case study and combine them in three dimensional exploration space. The full overview of the trade-offs discussed in this chapter is in Table 6.1.

6.2.1 Intra in-place vs. inter in-place

In-place is an optimization technique aiming at reducing the required memory size of data structures. This is achieved by reusing the same memory locations for different data elements or data structures. This is possible if we can analyze the lifetimes of data values. We distinguish two types of in-place optimization, intra and inter. Inter in-place optimization reduces the required memory size by reusing the same memory location by two different arrays. Intra in-place optimization reduces the required memory size by reusing the same memory location by different array elements of the same array. The transformation thus exploits the limited lifetime of the data during program execution.

We will demonstrate the intra in-place vs. inter in-place trade-off on the educative code example in Figure 6.2. The implementation in Figure 6.2a consists of 4 separate loops. In the first loop array A is produced. In the second loop array A is consumed and array B is produced. In the third loop array B is consumed and array C is produced. In the fourth loop array C is consumed. After fusion (merging) of the loops we obtain the implementation in Figure 6.2b.

To achieve maximal inter in-place the lifetime of the whole array, i.e., between the write of the first element and read of the last element, has to be small. The maximal inter in-place is often achieved in the non-localized code with a lot of separate loops/loop nests, see code in Figure 6.2a, where the lifetime of the whole array spans over two loops. E.g., if we assume 1 time unit per iteration per statement where an array is written or read, the lifetime of the array A in Figure 6.2a will be 2*N time units. Note, that in Figure 6.2b the lifetime of the whole array is 4*N time units.

To achieve maximal intra in-place, the lifetime of each individual array element, i.e., time between its write and read, has to be small. This is usually achieved in fully

```
1  for(i=0; i<N; i++)
2   A[i] = ...;
3  for(i=0; i<N; i++)
4    if(i>=d)
5      B[i] = f(A[i-d], A[i]);
6    else
7      B[i] = 0;
8  for(i=0; i<N; i++)
9    C[i] = B[i];
10 for(i=0; i<N; i++)
11   if(i>=d)
12     ... = f(C[i-d], C[i]);
13   else
14     ...
```

(a) Code before loop fusion with good inter in-place possibility.

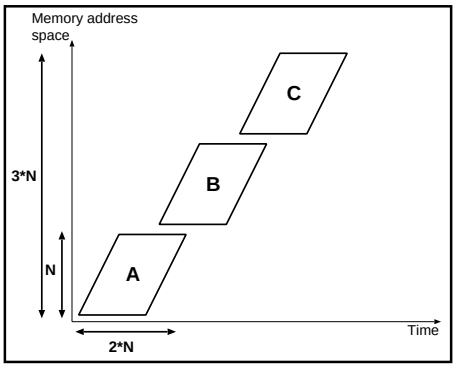
```
1  for(i=0; i<N; i++) {
2    A[i] = ...;
3    if(i>=d)
4      B[i] = f(A[i-d], A[i]);
5    else
6      B[i] = 0;
7    C[i] = B[i];
8    if(i>=d)
9      ... = f(C[i-d], C[i]);
10   else
11     ...
12 }
```

(b) Code after loop fusion with good intra in-place possibility.

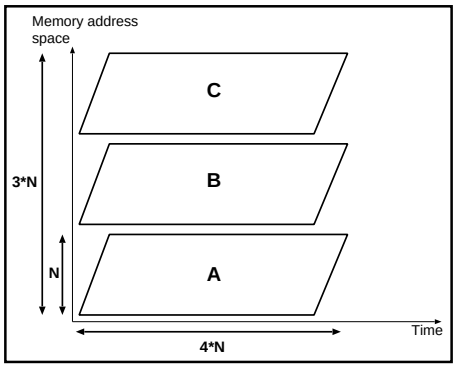
Figure 6.2: Inter in-place and intra in-place trade-off example.

merged and localized code. E.g., in Figure 6.2b the code from Figure 6.2a has been merged. This significantly reduces the lifetimes of the individual array elements. E.g., if we again consider 1 time unit per iteration per statement, the lifetime of one element (e.g. *A[0]*) in Figure 6.2b will be 4*d+1 time units where *d* is the inter-iteration dependency distance. Note, that in Figure 6.2a it is N+d+1 time units. The localized code in Figure 6.2b has small lifetime for individual array elements and the non-localized code in Figure 6.2a has small lifetime for whole arrays. Thus, the opportunity for inter in-place is better for non-localized code and the opportunity for intra in-place is better for localized code.

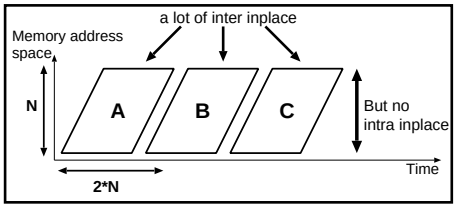
The difference between the non-localized code with maximal inter in-place and localized code with maximal intra in-place is demonstrated also in Figure 6.3. The three arrays from Figure 6.2 are represented in the time-address space graph. In Figure 6.3a and Figure 6.3b the original memory layout without in-place for the non-localized and localized code is shown. In Figure 6.3c and Figure 6.3d we see the memory layout after in-place (inter for Figure 6.3c and intra for Figure 6.3d). In Figure 6.3c the arrays are inter-in-placed because their respective lifetimes are short (2*N) and non-overlapping. However, they cannot be intra-in-placed because all elements are first produced and then consumed, so we need all N elements in the memory. In Figure 6.3b the array lifetime spans the whole execution 4*N. Thus the arrays cannot be inter-in-placed since their lifetimes are overlapping. However not all elements in the array have to be kept alive and thus intra-in-place can be applied. This is due to the good locality of the individual array elements. Consumption follows d iterations, i.e., 4*d+1 time units, after production and thus reuse of element locations resulting in smaller memory address space for arrays is possible. Furthermore, by improving the locality some intermediate buffers are not needed anymore. E.g., in the localized code array B is not present in Figure 6.3d because it was eliminated using advanced copy propagation [209].



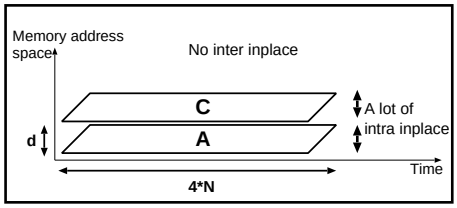
(a) Memory layout of the non-localized code in Figure 6.2a.



(b) Memory layout of the localized code in Figure 6.2b.



(c) Memory layout of the non-localized code in Figure 6.2a after inter in-place.



(d) Memory layout of the localized code in Figure 6.2b after intra in-place.

Figure 6.3: The trade-off between inter in-place and intra in-place.

```
1 for(i=0;i<N;i++) {
2   sum+=B[i]; //R1, e.g. read of B[0] in T0
3 }
4 for(i=0;i<N;i++) {
5   A[i]=g(B[i]); //W3(A[0]) and R2(B[0]) in T1
6   if(i>=d)
7     ... =f(A[i-d],sum,A[i]); //d is const.
8 }
```

Figure 6.4: Data size vs. nr. data transfers trade-off example (simple example).

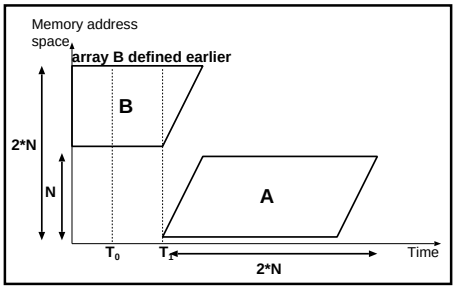
For our example, we can give a break-even point depending on the parameters d and N, i.e., the inter-iteration distance between production and consumption and the size of the array, when is the final memory size equal for both the solution, the local one and the non-local one. The break-even point is when $N = 2 \times d$ as can be derived from Figure 6.3.

6.2.2 Intra in-place vs. data reuse

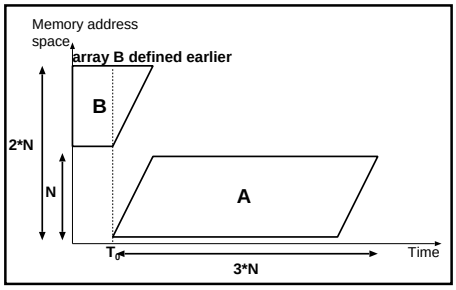
Intra in-place optimization reuses memory space of a single data element that is not needed any more as we explained in the previous subsection. To achieve a good intra in-place the lifetime of elements, i.e., the time between the production and consumption, has to be minimal. Thus the production and consumption of elements have to be placed as close as possible.

Data reuse optimization means to place a local copy of the part of the array which will be used (read) several times closer to the data path. This copy can also be a single array element placed in the foreground memory, i.e., register file. By placing the copy closer to the data path the number of accesses to higher levels of memory is reduced, because if the element is read again, it is read from the copy in the local memory and not from the main array in the main memory. To have a good reuse and to be able to place a local copy to the local memory, two conditions have to be fulfilled. First there has to be some reuse, i.e., the copy has to be read several times. Second, the reads have to be close in execution time so that the local copy is not occupying the local memory unnecessarily long.

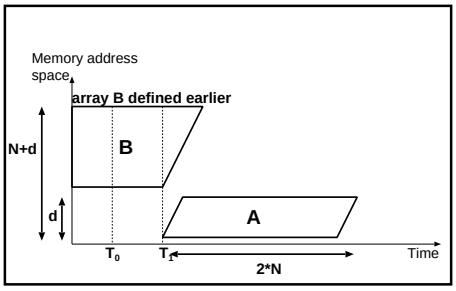
An educative example of the intra in-place vs. data reuse trade-off is illustrated in Figure 6.4. It shows two loops. The first loop reads elements of array *B[i]* which are cumulatively summed. The second loop (first statement) also reads elements of array *B[i]*, performs function *g()* on them and assigns them to array *A[i]*. The second statement of this loop reads array elements *A[i]* and *A[i-d]*, where *d* is a constant representing inter-iteration dependency distance, reads *sum* and performs function *f()*. Because the write of array *A* and the read of array *A* are in the same loop nest and we cannot put the statements in the second loop closer together, array *A* has optimal locality. After applying intra in-place only *d* elements have to be stored in the memory. However, the distance between the read of array *B* in the first loop and in the second loop is huge. We can improve this by moving the first statement of the second loop to the first loop. This will improve the data reuse for array *B*. However,



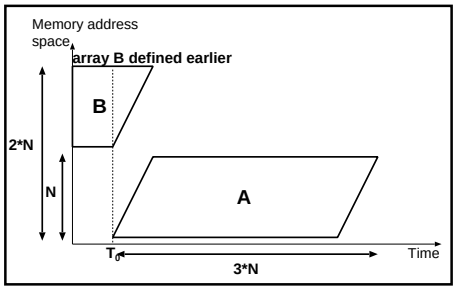
(a) Memory layout of the code in Figure 6.4.



(b) Memory layout after moving the statement *A[i]=g(B[i]);* to the first loop.



(c) Memory layout of the code in Figure 6.4 after in-place.



(d) Memory layout after moving the statement *A[i]=g(B[i]);* to the first loop after in-place is the same as in the figure above.

Figure 6.5: Data size vs. nr. data transfers trade-off: Explanation of the data transfer reduce vs. data storage size increase effect on time-address space axis.

it will also destroy good locality of array A. To put all three statements in one loop nest is not possible because of the loop carried dependency (writing of *sum*) in the first loop nest (first statement) followed by the dependency (*sum*) between the first and third statement. So the only two options are good locality of array A or good reuse of array B, but not both. This results in the trade-off between intra in-place and data reuse.

This example corresponds to the time-address space graph in Figure 6.5. Figure 6.5a corresponds to the situation in Figure 6.4 when the two consumptions of the same B memory location are in two different loop nests and are performed at two different time stamps T_0 and T_1 (for *A[0]*) which are far apart. The situation in Figure 6.5b corresponds to moving the second statement in Figure 6.4 to the first loop. Then, the two consumptions of the same B memory location are close together (time T_0 for array element *A[0]*). However, the intra in-place possibility of array A decreased. This is clear from the intra in-placed memory layouts in Figure 6.5c (the array A can be in-placed) and Figure 6.5d (the array A cannot be in-placed).

```
1  for(i=0;i<16;i++)
2    for(j=0;j<16;j++)
3      A[i][j] = in();
4  for(i=0;i<14;i++)
5    for(j=0;j<14;j++)
6      out =
7        A[i+2][j+2];
8  for(i=0;i<16;i++)
9    for(j=0;j<16;j++)
10     if(i<2||j<2)
11       ... = A[i][j];
```

(a) Original code with low control-flow complexity and high memory footprint requirement.

```
1  for(i=0;i<16;i++)
2    for(j=0;j<16;j++)
3    {
4      A[i][j] = in();
5      if(i>1 && j>1)
6        out = A[i][j];
7    }
8  for(i=0;i<16;i++)
9    for(j=0;j<16;j++)
10     if(i<2||j<2)
11       ... = A[i][j];
```

(b) Code with high control-flow complexity and low memory footprint requirement.

```
1  for(i=0;i<16;i++) {
2    for(j=0;j<16;j++)
3      A[i][j] = in();
4    if(i>1)
5      for(j=0;j<14;j++)
6        out = A[i][j+2];
7  }
8  for(i=0;i<16;i++)
9    for(j=0;j<16;j++)
10     if(i<2||j<2)
11       ... = A[i][j];
```

(c) Intermediate solution.

Figure 6.6: Initial code to demonstrate intra in-place vs. control-flow complexity example.

6.2.3 Intra in-place vs. control flow complexity

Trade-offs mentioned above have targeted the data part of the application. However, a trade-off also exists between the data part (intra in-place) and the control part (control flow complexity) of the application. This is illustrated by an educative example in Figure 6.6.

In the code in Figure 6.6a array A is produced in the first loop nest and part of the array is consumed in the second loop nest. After applying the loop transformations for improving locality the loop nests may be fused and the consumption of array A is shifted to satisfy the flow dependency (see Figure 6.6b). In the code in Figure 6.6b the lifetime of A[2][2], written and read in the iteration *(i,j) = (2,2)*, is not overlapping with the lifetime of A[2][3], written and read in the iteration *(i,j) = (2,3)*. So the A[2][3] can be mapped to the same memory location as A[2][2]. Similarly, the lifetime of A[2][3] is not overlapping with the lifetime of A[2][4], etc. Thus we do not need the declared size of A to store the whole array A in the memory. Instead of the 256 memory locations needed for the code in Figure 6.6a we need 61 memory locations in Figure 6.6b after intra in-place of the A array. Note, that we need to keep the A[0][j], A[1][j], A[i][0] and A[i][1] in the memory because they are assumed to be consumed later and we need one memory location for the array element processed in the loop in Figure 6.6b. That means we still need (2*16+2*16-4)+1=61 memory locations.

However, in the code in Figure 6.6b the *if* condition was introduced. This can be a problem for processors that do not support guarded execution and branch prediction. It may also block the loop from being software pipelined. Even for processors that deal well with conditions we still need to calculate the condition itself. Thus we say that the code in Figure 6.6b has higher control-flow complexity compared to the code in Figure 6.6a. This may severely slow down the application and results in the trade-off between intra in-place and control-flow complexity in the application.

The code in Figure 6.6a requires 256 memory locations, however executes 0 explicit *if* statements. Note, that still many branches caused by the *for* loops are present. The code in Figure 6.6b executes 256 *if* statements, however it needs only 61 memory locations.

Note, that also an intermediate solution exists (see Figure 6.6c). Here we decided to fuse and shift to satisfy the flow dependency only in the outermost loop. This eliminates the dependency on the innermost iterator in the *if* condition. However, this costs us 13 extra memory locations because we did not merge the innermost loop and thus we need to keep all elements A[i][2]-A[i][15] in the memory. Together with the A[0][j], A[1][j], A[i][0] and A[i][1] this makes (2*16+2*16-4)+14=74 memory locations. In this code we execute 16 *if* statements. This solution is between the two extreme solutions we presented in two previous paragraphs, i.e., the code with minimal footprint (see Figure 6.6b) but with high control-flow complexity and the code with footprint equal to declared size but with small control flow complexity. These 3 points define a Pareto curve which trades-off the intra in-place vs. control flow complexity.

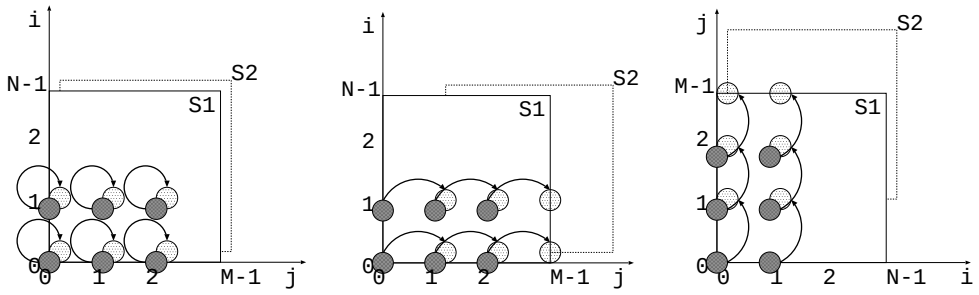
6.2.4 Intra in-place vs. ILP trade-off (for parallelization)

To achieve maximal speed-up of the application, Instruction Level Parallelism (ILP) is exploited by current compilers. Software pipelining, unrolling, function inlining, tail duplication, if-conversion, etc. can increase the amount of exploitable ILP. However, implementations using these techniques require larger memory footprint than original implementation. This results in a trade-off between performance and storage size.

Figure 6.7 illustrates a simple example of this trade-off. The initial implementation in Figure 6.7a produces in statement S1 array A. This array is consumed in statement S2. If each (production or consumption) operation takes 1 cycle the total number of required cycles would be $2 \times N \times M$. The required memory size is 1, because each element of array A is consumed immediately after production.

To increase the amount of parallelism, software pipelining can be used. In the geometrical model, this means to shift (i.e., translate the iteration domain polytope) S2 by +1 in the dimension we want to software pipeline as depicted in Figure 6.7b. Now, the production of an element in S1' and consumption of a previous element in S2' can be performed in parallel. This takes $2 \times N + N \times M \approx N \times M$ cycles so a reduction with a factor of 2 compared to the code in Figure 6.7a. However, the required storage size is 2, because we do not immediately consume the same element, but a previous element and thus we need two elements to store.

To further increase the amount of ILP parallelism, vector processing techniques can be used. By software pipelining we have created an inter-iteration dependency between the production and consumption of array A in the innermost dimension (see Figure 6.7b). However, there is no dependency in the outermost dimension. Thus after loop body split and loop interchange (see Figure 6.7c) we can unroll the (now) innermost dimension and process the whole array A in this dimension as a vector. This results in $2 + M \approx M$ cycles. However, the memory footprint increases to $2 \times N$. The vector processing techniques can be also used directly in Figure 6.7a where one



(a) (b) (c)

```
1 for(i=0;i<N;i++)
2   for(j=0;j<M;j++) {
3     A[i][j] = ...;   //S1
4     ... = A[i][j];   //S2
5   }
```

(a) Original code.

```
1 for(i=0;i<N;i++) {
2   A[i][0] = ...  //S1'
3   for(j=1;j<M;j++) {
4     ... = A[i][j-1];
  //S2'
5     A[i][j] = ...;   //S1''
6   }
7   ... = A[i][M];   //S2''
8 }
```

(b) Code with software pipelining.

```
1 for(i=0;i<N;i++) {
2   A[i][0] = ...  //S1'
3 }
4 for(j=1;j<M;j++)
5   for(i=0;i<N;i++) {
6     ... = A[i][j-1];
  //S2'
7     A[i][j] = ...;   //S1''
8   }
9 for(i=0;i<N;i++) {
10  ... = A[i][M];   //S2''
11 }
```

(c) Code with software pipelining and vector processing.

Figure 6.7: The trade-off between performance and area using more instruction level parallelism.

or both loops can be parallelized. This results to different trade-offs between number of cycles and the required memory footprint.

6.2.5 Code size vs. code complexity trade-off (during code generation)

Until now we have discussed trade-offs that affect in-place (inter and intra), data reuse, control flow complexity and ILP of the code during loop transformations. Also interesting trade-offs exist during code generation phase when the code is generated from the geometrical model. The geometrical model represents each iteration instance of the statement as a separate point in a multi-dimensional space. We explain this model directly using the example in Figure 6.8.

Figure 6.8a shows a simple code fragment. Array A is written in the first statement S1 and read in the second statement S2. The corresponding geometrical model is in Figure 6.8b. The iteration domains are represented by the rectangular boxes. The depicted arrow represents the dependency between the write and the read of the element, i.e., from the iteration where it is written to the iteration where it is read.

When extracting the code from the model in Figure 6.8b we can obtain (original) code in the Figure 6.8a, the code in Figure 6.8c or the code in Figure 6.8d depending on the algorithm we use for generating the code. A detailed explanation about the code extraction from geometrical model can be found in Section 3.7 in Chapter 3. The model keeps only information about execution ordering of the application and not about the structure of the code. The codes in Figure 6.8a,c,d have the same execution ordering, thus their geometric model is the same and thus all of the codes can be generated from the same geometric model. The differences among the codes are in empty iterations, control-flow overhead, and code size.

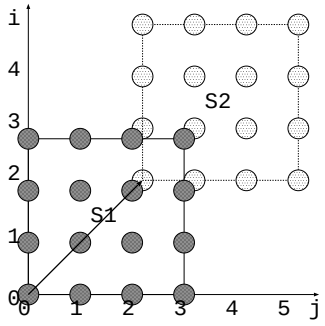
The code in Figure 6.8c is without control flow overhead (no *if* conditions). However, it is almost fully unrolled code consisting only of small loop nests which will be most probably unrolled by the compiler. Note also that this code does not have any empty iterations, i.e., all iterations are used for computation. The code in Figure 6.8a is compact, however it has some control flow overhead due to the *if* conditions. Also, it requires eight empty iterations where no element is written or read. The code in Figure 6.8d is compact as well, however it has the very large control flow overhead due to the *if* conditions and *min* and *max* functions. Nevertheless, it requires only 2 empty iterations.

These trade-offs have been already partly discussed by the authors working on code generators from the geometrical model [20, 178]. We mention them to have a more complete overview about the trade-offs in the loop transformation step where we also include the code generation step. Except of the techniques in code generation there is a post-generation technique proposed by Falk et al. [66]. By using the advanced loop nest splitting technique at the source code level a large part of control-flow overhead can be avoided. We consider this technique as a code generation technique, because it can be integrated into the code generation phase and it does not change the execution ordering of the application.

The code of Figure 6.8a contains two *if* conditions. We can apply here the intra in-

```
1  for(i=0;i<6;i++)
2    for(j=0;j<6;j++) {
3      if(i<=3 && j<=3)
4        A[i][j] = f(); //S1
5      if(i>=2 && j>=2)
6        B[i][j] = g(
7          A[i-2][j-2]); //S2
8    }
```

(a) Original code (scanning using bounding boxes).



(b) Geometrical model for the codes in Figure 6.8.

```
1  for(i=0;i<=1;i++)
2    for(j=0;j<=3;j++)
3      A[i][j] = f(); //S1
4  for(i=2;i<=3;i++) {
5    for(j=0;j<=1;j++)
6      A[i][j] = f(); //S1
7    for(j=2;j<=3;j++) {
8      A[i][j] = f(); //S1
9      B[i][j] = g(
10       A[i-2][j-2]); //S2
11   }
12   for(j=4;j<=5;j++)
13     B[i][j] = g(
14       A[i-2][j-2]); //S2
15 }
16 for(i=4;i<=5;i++)
17   for(j=2;j<=5;j++)
18     B[i][j] = g(
19       A[i-2][j-2]); //S2
```

(c) Scanned code for maximal unrolling and no empty iterations.

```
1  for(i=0;i<=5;i++)
2    for(j=max(0,i-3);
3        j<=min(i+3,5);j++) {
4      if(i<=3 && j<=3)
5        A[i][j] = f(); //S1
6      if(i>=2 && j>=2)
7        B[i][j] = g(
8          A[i-2][j-2]); //S2
9    }
```

(d) Intermediate solution.

Figure 6.8: The trade-off during code generation phase.

place vs. control flow complexity trade-off when shifting S2 and splitting the loops, as already discussed in Subsection 6.2.3. But, this transformation changes the execution order.

6.3 GLT trade-off cost components

In this section we identify the different cost components from Table 6.1 that mainly participate in the GLT trade-offs. GLTs target platform independent issues, such as locality and regularity, and they change the execution ordering of the statements. The other Loop Transformations (LT)s target other issues than locality and regularity, e.g., more platform specific issues such as bandwidth and performance. Also the impact on the code is different, the GLT have overall impact on the code (mostly on multiple loop nests) and the other LT have limited impact on the code (mostly one perfect loop nest or innermost loop). The position of GLT in the DTSE design flow is compliant with the importance in these transformations in the flow. Decisions on improving the data locality and regularity in the GLT are early in the DTSE design flow. Later in the flow, these decisions are refined using other LT. These improve e.g., the Storage Budget Optimization (SBO) opportunities or the ILP. The GLT are orthogonal [29] to other LT, i.e., the constraints created during the decisions in GLT are propagated to the other LT, but not the other way around. That means, the decisions made earlier in the flow do not have to be reconsidered, they can be only refined using additional (e.g., platform dependent) knowledge. In the past, some solutions that should be propagated to the lower levels have been pruned away, because of not considering trade-offs at the higher levels, i.e., GLT. To point at this problem is the main contribution of this chapter.

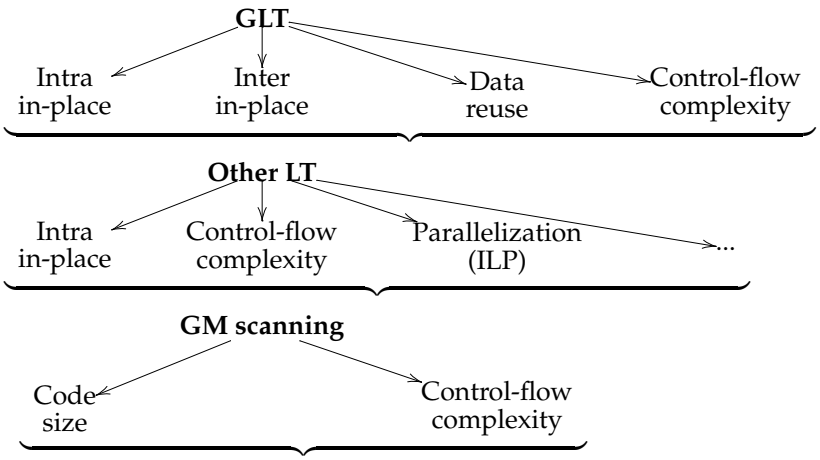


Figure 6.9: The cost components participating at GLT from Table 6.1 which are orthogonal to the other LT and GM scanning decisions.

During the GLT the execution ordering of the application changes significantly. Decisions at this level influence the data locality (intra in-place, inter in-place and data

reuse), the control-flow complexity and related instruction locality [210] (see Figure 6.9). As discussed in Subsection 6.2.1 the non-fused (non-localized) code is beneficial for inter in-place and also for the instruction locality. The fused (localized) code is however better for intra in-place and has worse instruction locality. The GLT related trade-offs are the trade-offs we mainly target in this chapter. The work of Vander Aa et al. [210] has already shown the importance of the instruction locality in these trade-offs. This is the reason why we do not discuss the instruction locality part of control-flow complexity further. However, the remaining GLT trade-offs are discussed in the educative examples in Subsection 6.2.1- 6.2.3 and on a case study in Subsection 6.4.1- 6.4.4.

After GLT, performing other LT can still improve different performance and bandwidth related aspects like control-flow complexity or ILP parallelism as depicted in Figure 6.9. An example of the trade-off in other LT is presented in Subsection 6.2.4. As already mentioned at the beginning of this chapter we target the RISC and VLIW processor family, where the ILP is enhanced by loop unrolling and software pipelining in the compiler. We consider such an ILP parallelization as orthogonal to the exploration space we target in the GLT. The decisions in our GLT exploration space are propagated and later used by the compiler. However, the compiler unrolling possibility does not constrain the GLT decisions.

During the GM scanning the execution ordering, thus the memory and locality related issues are fixed. However, here we can still decide if we go for more compact code with high code complexity or for a flattened code with low code complexity as depicted in Figure 6.9. An educative example of this trade-off has been shown in Subsection 6.2.5. We do not discuss this trade-off in our case study, because this was already discussed in detail in the geometrical model scanning papers [178, 20] and in Section 3.7 in Chapter 3. Further studying of fine grain control on code generation is out of the scope of this dissertation.

Thus, in the following case study we do not consider other LTs or GM scanning and we only focus on the exploration of the GLT trade-offs.

6.4 Case study and results

In this section we demonstrate GLT-related trade-offs on a real-life example, namely a QSDPCM video encoder [193]. Then we provide several Pareto points in the three dimensional exploration space. The dimensions of this space are Level 1 (L1) data storage size, number of data transfers to/from main memory, and the control flow complexity. Note, that the L1 data storage size is the overall size obtained by the combination of inter in-place and intra in-place trade-off. As mentioned above, we do not discuss the trade-offs in the code generation phase (Subsection 6.2.5). For the QSDPCM application with loop nests depth of eight the code size is exploding when using the scanning technique like in Figure 6.8c. We do not consider trade-offs for improving the parallelization potential (see Subsection 6.2.4). The target ARM processor we have mapped the application onto does not have multiple functional units and thus improving the parallelization potential would not help for the target platform with this processor.

6.4.1 Intra in-place vs. inter in-place

The trade-off between intra in-place and inter in-place discussed in Subsection 6.2.1 can be found in the real-life QSDPCM application [193]. We compared 3 versions of QSDPCM, namely non-localized, localized and partially localized, where only part of the loop nests is fused (merged). To evaluate the code we used the MHLA mapping tool [26] with a 2 layer memory hierarchy. The memory hierarchy is composed from Layer 1 1kB Scratchpad Memory (SPM) and Main Memory (MM). The tool considers inter and intra in-place and decides on the placement of the arrays and their copy candidates to different memory layers. Note, that the copy candidate is a part of the array which is stored in a lower level than the original array. The tool provides us with the size assigned to SPM and MM as well as the number of memory accesses and energy number computed based on this information and the memory model [26].

	Non-localized	Fully localized	Partially localized
Data L1+MM energy [uJ]	71.97	47.17	46.47
Data L1 data energy [uJ]	13.79	14.32	14.29
Data MM data energy [uJ]	58.18	32.85	32.17
Assigned size to L1 [bytes]	742	748	802
Assigned size to MM [bytes]	114048	66352	63360
Nr. L1 data mem.acc.	1.14×10^6	1.19×10^6	1.19×10^6
Nr. MM data mem.acc.	542×10^3	306×10^3	300×10^3

Table 6.2: Results for different QSDPCM implementations when exploring inter vs. intra in-place trade-off.

Table 6.2 gives an overview of the results obtained on the three code versions described above. We see that the best solution for energy is the partially-localized one. In all cases, the energy contributions from the L1 are very comparable which is somehow coincidental because the arrays and copies assigned in L1 are very different in the three cases. However the L2 energy contribution is much smaller in the fully and partially localized codes. This is due to the big reduction of the number of accesses to this layer. The non-localized solution is 55% worse than the partially localized solution. The fully localized solution is only 1.5% worse compared to non-localized solution. An important message is that the most local code is not always the best one for energy as most people would expect.

For the rest of the section we will not consider the intra in-place and inter in-place axes separately, but combine them into one data size axis. At the end, we are interested in the overall data size resulting from intra and inter in-place and not from the particular components of the data size.

6.4.2 Intra in-place vs. data reuse

The trade-off between intra in-place and data reuse discussed in Subsection 6.2.2 is demonstrated in the code in Figure 6.10 and 6.11 for a real life example taken

from QSDPCM. In this code we can observe a similar effect as explained in Subsection 6.2.2, i.e., that the (time) distance between the production and consumption of an array element (for intra in-place) and the time distance between several consumptions of other array elements (for data reuse) can conflict, but now on a much more complex example. In Figure 6.10 on Line 29 there is an array *prev_sub4_frame* which depends (via scalar variable *temp4*) on an input array *prev_frame* on Line 25. This input array is read two times in the application (Line 25 and 10). The two consumptions are too far apart to put the value in a register and use it for both reads. If we consider 1 time unit per iteration per statement with array read/write, the two reads are (2*2+1)*8*8+(4*4+1)*4+(4*4+1) = 405 time units far apart. You can check it by computing the time you need from the first read of *A[20*176+20]*, i.e., for y=x=m=n=k=l=0, to the second read of the same element.

After strip mining, fusion (merging) and shifting (bumping) of the inner loops it is possible to bring the two consumptions of the input array *prev_frame* so close together that they access the same memory element in the same iteration. This is shown in the code in Figure 6.11. Thus we can store this element in the foreground memory and save costly off-chip memory accesses by reusing the value from the foreground memory (register) (see Line 11 in the transformed code in Figure 6.11). However, this data reuse improving transformation will disrupt the good locality and good intra in-place optimization opportunity of the *prev_sub4_frame* array. The production of this array has to be shifted together with the whole loop nest further from its consumption (see the changed address expression for this array). We do not discuss the inter in-place optimization in this section, because the inter in-place opportunity remains the same and thus the inter in-place decisions are not affected.

	prev_sub4_fr. storage size	Assigned data size to L1	Nr. MM mem accesses	Total energy [uJ]
Best intra in-place	496	2544	307×10^3	32.9
Improved data reuse	541	2589	259×10^3	27.5

Table 6.3: Results for different QSDPCM codes when exploring intra in-place vs. data reuse trade-off.

We compared 2 versions of QSDPCM, namely the code with best intra in-place optimization (for a code fragment see Figure 6.10) and our adapted code (see Figure 6.11) with improved data reuse. To evaluate the codes we again used the MHLA mapping tool [26] with a 2 layer memory hierarchy[3]. The results are in Table 6.3. We have observed an almost 15% decrease in the number of main memory accesses compared to the best intra in-place code. However, we paid 45 memory locations extra for array *prev_sub4_frame* resulting in 1.7% increase of the data size stored in L1. This is not a problem as long as the *prev_sub4_frame* array still fits into L1. If it would have to be moved to main memory, the improved data reuse would not be beneficial because of increased main memory accesses due to accesses to *prev_sub4_frame* array.

[3]Note, that from now on other version of QSDPCM with a lot of bug fixes has been used compared to the Table 6.2. Therefore the results of Subsection 6.4.1 are not completely consistent with the remaining text.

```
for(y=-2; y<9; y++) {
  for(x=-2; x<11; x++) {
    if((y>=-2) && (y<9-1) && (x>=-2) && (x<11-1)) {
      for(m=0; m<8; m++) {
        for(n=0; n<8; n++) {
          if((8*(x+1)+(n+2)>=0) && (8*(x+1)+(n+2)<88) &&
            (8*(y+1)+(m+2)>=0) && (8*(y+1)+(m+2)<72)) {
            for(k=0; k<2; k++) {
              for(l=0; l<2; l++) {
                temp2+=prev_frame[(16*(y+1)+2*(m+2)+k
                )*176+16*(x+1)+2*(n+2)+l];
              }
            }
            prev_sub2_frame[(8*(y+1)+(m+2))*88+8*(x+1)
            +(n+2)]=temp2/4;
          }
        }
      }
    }
    if((y>=-1) && (y<9-1) && (x>=-1) && (x<11-1)) {
      for(m=0; m<4; m++) {
        for(n=0; n<4; n++) {
          for(k=0; k<4; k++) {
            for(l=0; l<4; l++) {
              temp4+=prev_frame[(16*(y+1)+4*m+k)*176
              +16*(x+1)+4*n+l];
            }
          }
          prev_sub4_frame[(4*(y+1)+m)*44+4*(x+1)+n]=
          temp4/16;
        }
      }
    }
    if((y>=0) && (y<9) && (x>=0) && (x<11)) {
      for(vy=0; vy<9; vy++) {
        for(vx=0; vx<9; vx++) {
          for(m=0; m<4; m++) {
            for(n=0; n<4; n++) {
              if((4*y+vy-4+m)>=0 && (4*y+vy+4+m)=<
                (36-1) && (4*x+vx-4+n)>=0 && (4*x+vx+4+n)=<
                (44-1)) {
                p2=prev_sub4_frame[(4*y+vy-4+m)*44+4*x+vx
                -4+n]];
              }
              ...
```

Figure 6.10: Data size vs. Nr. of data transfers trade-off example (QSDPCM). Code with good in-place and bad reuse.

```
for(y=-2; y<9; y++) {
  for(x=-2; x<11; x++) {
    for(m=0; m<4; m++) {
      for(n=0; n<4; n++) {
        if(((4*(x+1)+n+1)>=0) && ((4*(x+1)+n+1)<44) &&
          (4*(y+1)+m+1)>=0) && ((4*(y+1)+m+1)<36)) {
          for(i=0; i<2; i++) {
            for(j=0; j<2; j++) {
              for(k=0; k<2; k++) {
                for(l=0; l<2; l++) {
                  temp2+=prev_frame[(16*(y+1)+4*(m+1)+
                  2*i+k)*176+16*(x+1)+4*(n+1)+2*j+l];
                }
              }
              prev_sub2_frame[(8*(y+1)+2*(m+1)+i)*88+
              8*(x+1)+2*(n+1)+j]=temp2/4;
              temp4+=temp2;
            }
          }
          prev_sub4_frame[(4*(y+1)+m+1)*44+4*(x+1)+n+1]
          =temp4/16;
        }
      }
    }
    if((y>=0) && (y<9) && (x>=0) && (x<11)) {
      for(vy=0; vy<9; vy++) {
        for(vx=0; vx<9; vx++) {
          for(m=0; m<4; m++) {
            for(n=0; n<4; n++) {
              if((4*y+vy-4+m)>=0 && (4*y+vy+4+m)=<(36-1)
                && (4*x+vx-4+n)>=0 && (4*x+vx+4+n)=<
                (44-1)) {
                p2=prev_sub4_frame[(4*y+vy-4+m)*44+4*x+
                vx-4+n]];
              }
              ...
```

Figure 6.11: Data size vs. Nr. of data transfers trade-off example (QSDPCM). Code with bad in-place and good reuse.

6.4.3 Data reuse vs. control flow complexity

In Section 6.3 we have depicted the strong interaction between intra in-place and data reuse because both are part of the GLT group. If we look back at the code example in Subsection 6.4.2 several loop transformations have been applied to go from code in Figure 6.10 to Figure 6.11. First the strip mining has been performed to enable the fusion of the first two loop nests in Figure 6.10. This strip mining does neither change the intra in-place nor the data reuse. However, it allows hoisting the condition in the first loop nest two levels up resulting in 2206 *if* evaluations in the first two loop nests. Note, that we counted all three *if* conditions in these loop nests for Best intra in-place strip mined version. After the fusion of the first two loop nests and shifting to satisfy the dependency relations the intra in-place opportunity decreased, however the number of main memory accesses was reduced from 307×10^3 to 259×10^3 as has been shown in previous subsection. The number of *if* evaluations increased from 2206 to 2288 as will be shown in the next subsection. This results in a trade-off data reuse vs. control flow complexity when projected on data reuse vs. control flow complexity plane (see also Subsection 6.4.5). This trade-off is depicted in Table 6.4.

	Nr. MM mem accesses	Number of *if* eval.
Best intra in-place strip mined (SM)	307×10^3	2206
Improved data reuse	259×10^3	2288

Table 6.4: The trade-off between the data reuse and control flow complexity.

6.4.4 Intra in-place vs. control flow complexity

The trade-off between intra in-place and control flow complexity as discussed in Subsection 6.2.3 is demonstrated in the code in Figure 6.12 and 6.13 for the real life example. The extra control flow is present because of fusing and shifting the loops to obtain good intra in-place with improved data reuse while still satisfying the flow dependencies (see Figure 6.12). However, this control flow can be reduced when shifting the loops a little bit more than required for optimal in-place as shown in Figure 6.13. Here the Lines 10,14 and 19 were shifted a bit further as in the code of Figure 6.12. This caused a disruption of the locality for array *prev_sub4_frame* compared to the code in Figure 6.12. However, the first condition in the code in Figure 6.12 is hoisted and simplified in the code in Figure 6.13.

We compared three code versions of QSDPCM application, the one with improved data reuse in Figure 6.12, the one with improved code complexity in Figure 6.13 (Improved complexity 2) and an intermediate version (Improved complexity 1). In the Improved complexity 1 version, the condition was hoisted from the inner loop, sacrificing 25.5% L1 data memory space. In the Improved complexity 2 version, the condition was hoisted from the two inner loops, sacrificing 25.8% L1 data memory

```
for(y=-2; y<9; y++) {
  for(x=-2; x<11; x++) {
    for(m=0; m<4; m++) {
      for(n=0; n<4; n++) {
        if(((4*(x+1)+n+1)>=0) && ((4*(x+1)+n+1)<44) &&
          (4*(y+1)+m+1)>=0) && ((4*(y+1)+m+1)<36)) {
          for(i=0; i<2; i++) {
            for(j=0; j<2; j++) {
              for(k=0; k<2; k++) {
                for(l=0; l<2; l++) {
                  temp2+=prev_frame[(16*(y+1)+4*(m+1)+
                  2*i+k)*176+16*(x+1)+4*(n+1)+2*j+l];
                }
              }
              prev_sub2_frame[(8*(y+1)+2*(m+1)+i)*88+
              8*(x+1)+2*(n+1)+j]=temp2/4;
              temp4+=temp2;
            }
          }
          prev_sub4_frame[(4*(y+1)+m+1)*44+4*(x+1)+n+
          1]=temp4/16;
        }
      }
    }
    if((y>=0) && (y<9) && (x>=0) && (x<11)) {
      for(vy=0; vy<9; vy++) {
        for(vx=0; vx<9; vx++) {
          for(m=0; m<4; m++) {
            for(n=0; n<4; n++) {
              if((4*y+vy-4+m)>=0 && (4*y+vy+4+m)=<
              (36-1) && (4*x+vx-4+n)>=0 && (4*x+vx
              +4+n)=<(44-1)) {
                p2=prev_sub4_frame[(4*y+vy-4+m)*44
                +4*x+vx-4+n]];
              }
              ...
```

Figure 6.12: Performance and data size trade-off example (QSDPCM). Code with good in-place and bad code complexity.

	Assigned data size to L1	Control-flow expression	Number of *if* eval.
Improved data reuse	2589	if(f(y,m,x,n)) //no-hoisting	2288
Improved code complex. 1	3249	if(f(y,m,x)) //hoisting 1 level	572
Improved code complex. 2	3258	if(f(y,x)) //hoisting 2 levels	143

Table 6.5: Results for different QSDPCM codes when exploring intra in-place vs. control-flow complexity trade-off.

```
for(y=-2; y<9; y++) {
  for(x=-2; x<11; x++) {
    if(y+2>9 && x+2>11) {
      for(m=0; m<4; m++) {
        for(n=0; n<4; n++) {
          for(i=0; i<2; i++) {
            for(j=0; j<2; j++) {
              for(k=0; k<2; k++) {
                for(l=0; l<2; l++) {
                  temp2+=prev_frame[(16*(y+2)+4*m+2*i
                  +k)*176+16*(x+2)+4*n+2*j+l];
                }
              }
              prev_sub2_frame[(8*(y+2)+2*m+i)*88+8*(x
              +2)+2*n+j]=temp2/4;
              temp4+=temp2;
            }
          }
          prev_sub4_frame[(4*(y+2)+m)*44+4*(x+2)+n]=
          temp4/16;
        }
      }
    }
    if((y>=0) && (y<9) && (x>=0) && (x<11)) {
      for(vy=0; vy<9; vy++) {
        for(vx=0; vx<9; vx++) {
          for(m=0; m<4; m++) {
            for(n=0; n<4; n++) {
              if((4*y+vy-4+m)>=0 && (4*y+vy+4+m)=<
                (36-1) &&(4*x+vx-4+n)>=0 && (4*x+vx
                +4+n)=<(44-1)) {
                p2=prev_sub4_frame[(4*y+vy-4+m)*44+
                4*x+vx-4+n]];
              }
              ...
```

Figure 6.13: Performance and data size trade-off example (QSDPCM). Code with bad in-place and good code complexity.

space. The results are in Table 6.5. Note, that except fewer evaluations of the condition expression due to hoisting, also the expression itself has been simplified.

6.4.5 Combination of trade-offs

We combined the points discussed in Subsections 6.4.2– 6.4.3 to a three dimensional exploration space in Figure 6.14. The exploration has been done manually for the QSDPCM application and 4 Pareto points have been identified. The Best intra in-place version we started from (see Subsection 6.4.2) was a non-Pareto point from which we derived the Best intra in-place version after strip mining (SM) (see Subsection 6.4.3). The final results containing the initial point and all Pareto points for all three dimensions are depicted in Table 6.6.

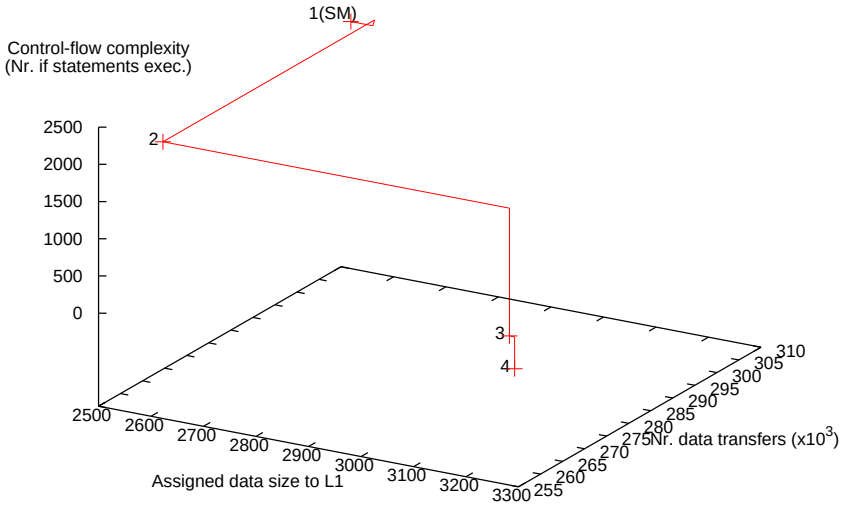


Figure 6.14: The 3D exploration space example (Nr. mem. accesses×data storage size×control flow complexity) for real life multimedia application (QSDPCM).

If we look at the 4 points obtained in Figure 6.14 we observe (going from point 1(SM) to 2, i.e., $1(SM) \mapsto 2$) a decrease of data transfers from 307×10^3 to 259×10^3. However, we have to sacrifice a data storage size increase from 2544 elements to 2589 elements. Thus, for a 15.6% decrease in number of memory accesses we have to pay 1.8% increase in data storage size. Note, that also the number of times the *if* statement is evaluated increased from 2206 to 2288. Further, we observe (going $2 \mapsto 3$) a decrease of number of times the *if* statement is evaluated from 2288 to 572 for an increase in memory size from 2589 to 3249. Thus, for a factor of 4 reduction in condition evaluation we have to pay 25.5% increase in storage size. If we decide to shift and hoist further (going $3 \mapsto 4$), we decrease the *if* condition evaluation again factor of 4. This is for a further increase in memory size by 0.03%. Most probably

	Assigned data size to L1	Number of data transf.	Number of *if* eval.
Best intra in-place	2544	307×10^3	7680
Best intra in-place strip mined (SM)	2544	307×10^3	2206
Improved data reuse	2589	259×10^3	2288
Impr. code complexity 1	3249	259×10^3	572
Impr. code complexity 2	3258	259×10^3	143

Table 6.6: Results for different QSDPCM codes when exploring intra in-place vs. data reuse vs. control-flow complexity trade-off.

when choosing between Improved code complexity 1 and Improved code complexity 2 the better solution is to go for Improved code complexity 2, because we have 4x fewer *if* statement evaluations compared to Improved code 1 for 0.03% memory size increase. This would not be the case, if the 9 elements cause the move of the affected array from L1 memory to main memory. Such a move will significantly increase the number of data transfers to main memory. Also, we would prefer Improved code complexity 1 code if the assembly code of the innermost basic block drastically changes, causing more cycles for this basic block as we will see in the next subsection.

6.4.6 Evaluation on ARM platform

We have compiled the best intra in-place version and the 4 Pareto optimal code versions we explored in the previous subsections with the ARM gcc cross compiler and have ran it on the SimIt ARM simulator [177]. We focus on the parts of the code where transformations have been applied (see code fragments in previous subsections) and measure the number of cycles, the execution time, and the binary size.

Code version	QSDPCM (part)		
	Cycles [$\times10^6$]	Time [ms]	Binary size [bytes]
Best intra in-place	14.336	69.4	13212
Best intra in-place strip mined (SM)	14.467	70.1	13340
Impr. data reuse	13.039	63.2	12988
Impr. code complexity 1	12.936	62.7	12956
Impr. code complexity 2	12.947	62.7	12892

Table 6.7: The number of cycles, the execution time and binary size for different QSDPCM codes when running on SimIt ARM simulator (206.4MHz host).

The results obtained are listed in Table 6.7. Although we have sacrificed the good data locality of the application to a certain extent, the application is running faster

due to improvement in other aspects such as data reuse and code complexity. Note, that in Table 6.7 the Best intra in-place strip mined (SM) version that is a Pareto point in Figure 6.14 has worse cycle count and binary size than the non-Pareto point Best intra in-place. The Best intra in-place strip mined (SM) version has the same assigned data size as the Best intra in-place version. The worse cycle count and binary size is due to more complex index expressions in the strip mined version causing more and complex code. The complexity of the index expression code is not taken into account yet. However, we plan to do it in our future work. The traditional code complexity metrics such as Halstead or McCabe metrics [96, 143] are too coarse to evaluate index expressions. The solution here could be to weight different operators in the index expression and sum those weights resulting in the index expression complexity. The combination with the above mentioned traditional code metrics could be beneficial, when the index expression contains control-flow introduced by ternary operators. Address optimizations performed by the source-to-source tools such as RACE (see Chapter 2) or by the compiler should be also considered in these index expression metrics triggering the need for address optimization estimators. As a result, the overhead in cycles will usually be relatively small [152, 80].

Code version	QSDPCM (part)		QSDPCM (w/o inner BB1)		QSDPCM (inner BB1)	
	Cycles [$\times 10^6$]	Time [ms]	Cycles [$\times 10^6$]	Time [ms]	Cycles [$\times 10^3$]	Time [ms]
Impr. data reuse	13.039	63.2	11.626	56.3	37.554	0.2
Impr. code complex. 1	12.936	62.7	11.546	55.9	37.432	0.2
Impr. code complex. 2	12.947	62.7	11.533	55.9	37.554	0.2

Table 6.8: The comparison of cycles and the execution time for the different QSDPCM codes in the intra in-place vs. control flow complexity trade-off when running on SimIt ARM simulator (206.4MHz host) (whole code, loop structure without the inner basic block and the inner basic block only).

If we focus in detail on results for Improved code complexity 1 and Improved complexity 2, we can observe that Improved code complexity 2 requires slightly more cycles although the condition is hoisted and it is much simpler compared to Improved code complexity 1 (see Table 6.5). Thus we analyzed the application without the innermost basic block and that basic block separately by measuring the number of cycles without the innermost basic block and the number of cycles for that basic block. The results are depicted in Table 6.8. For Improved complexity 2 the control-flow complexity is indeed simpler resulting in fewer cycles compared to Improved code complexity 1. However, the shifting caused more complex address computation (more assembly code) (similar to the Best intra in-place strip mined version above) resulting in more cycles for the basic block itself. However, if we look at the binary size metric, we see that the Improved code complexity 2 has 64 bytes less code compared to the Improved code complexity 1. Thus it is still a Pareto point in the three dimensional exploration space cycles vs. the data size vs. the binary size.

Figure 6.15 shows the relation between data size and machine cycles for the Best intra in-place version and 4 Pareto points from Figure 6.14. We observe that improvement in the other aspects than ultimate locality can bring significant gains for the applica-

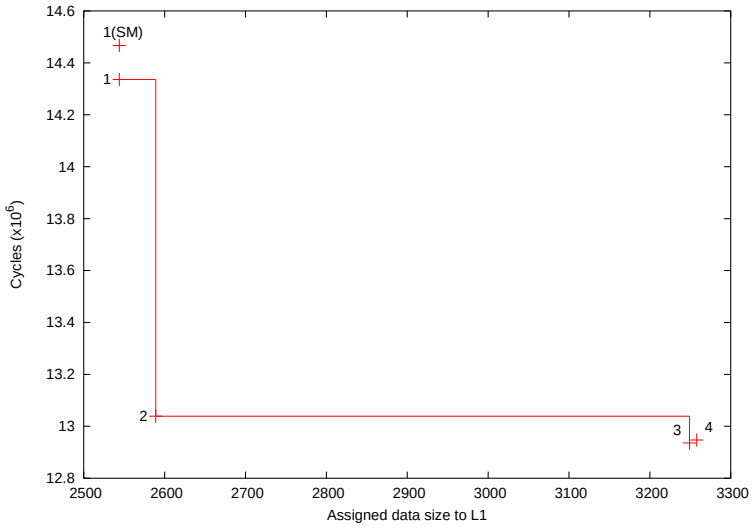


Figure 6.15: The 2D exploration space example (data storage size×cycles) for real life multimedia application (QSDPCM) on ARM 206.4MHz host.

tion. For an ARM 206.4MHz host it brought 10% reduction in the execution time for an 28% increase in the data size.

6.5 High-level estimators and their interaction with the GLT engine

In Sections 6.2 and 6.4 we have shown the importance of the different trade-offs in GLT. We have also explained that these trade-offs result from the different effects of the same GLT on the different underlaying steps of the DTSE methodology such as in-place or data reuse. These effects can be rapidly evaluated using high-level estimators. Thus, the goal of this section is to highlight the work in high-level estimators and the relation with the trade-off oriented GLT.

Note, that this section is an overview section with focus on the interaction between high-level estimators and GLT. More details on particular high-level estimators as well as the estimation error for the two estimators we briefly discuss in following subsections can be found in [122, 109]. In general, these estimators assume rectangular iteration domains and uniform dependency vectors, i.e. dependency vectors with same length and direction. This assumptions mostly hold for the application domain we are targeting resulting in accurate estimation. If the iteration domains are triangular or skewed, it can result to a factor of 2 overestimation. Then we can use more accurate approaches as discussed in [109] resulting in the trade-off between accuracy and computation time.

The DTSE reduces data transfers and storage size in the application. However, the resulting storage size is known only after the in-place step which is one of the last

steps in the DTSE design flow (see Chapter 2). But, the designer would like to know as soon as possible the required storage size going from specification to the implementation. Thus the storage size estimation tools are an important part of the early design flow. In the context of the DTSE optimization methodology [32] the storage size estimation is essential to get a global view during the transformation phase of DTSE design trajectory and guide the designer with relevant feedback toward the optimal decisions. Even if the DTSE design trajectory would be fully automated, it would require order of magnitude larger time intervals to evaluate the final impact of the transformations. Because the amount of transformations can be huge, the high-level estimators are crucial in the DTSE design trajectory to evaluate fast the impact of transformations during the early stages in the trajectory.

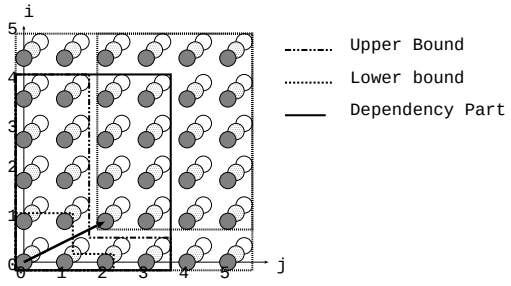
For storage size estimation two different approaches have been used in the past. The first one considers fully fixed execution ordering [214, 93, 228]. This is not possible in the first phase of design trajectory. The other approach estimates without execution ordering [13]. However, not taking execution ordering into account at all brings big disproportion between maximal and minimal estimated value. Often, a particular execution ordering is known at the beginning of the design flow and becomes more fixed when traversing the design trajectory. Thus it should be interesting to take the partial execution ordering into account. This is presented in the technique proposed by Kjeldsberg [122], where (partial) ordering constraints can be given by the designer. Moreover, this technique gives useful hints for execution ordering. This technique and our coupling to the GLT framework will be explained in Subsection 6.5.1.

The technique by Rydland [180] extends the technique of Kjeldsberg for inter in-place estimation. However, it deals only with fixed execution ordering. Hu [104] has proposed a memory requirement estimation technique for the translation substep of the GLT. In [106] he has proposed a hierarchical memory estimation technique using simplified Data Reuse Analysis (DRA) heuristic and platform independent MHLA heuristic. This approach is extended also for the linear transformation substep and is estimating also intra and inter in-place in [107, 108]. Hu's approach is incremental, i.e., it recomputes the data reuse only for parts of the code affected by the loop transformation, requiring much less estimation time compared to non-incremental approach. This technique and its link to the GLT trade-offs will be demonstrated in Subsection 6.5.2. A good overview about the in-place optimization and estimation can be found in [45, 12].

In Subsections 6.2.3, 6.2.5 and 6.4.4 we have seen that GLT affects also the control flow complexity of the application. Thus the high-level control-flow estimation is also required to interact with the GLT framework. However, this part was left for the future work. One solution would be using key microarchitecture-independent characteristics as proposed by Hoste [102] in another context. As we will see in Section 6.6 those have to be combined also with microarchitecture-dependent characteristics.

```
1 for(i=0;i<6;i++)
2   for(j=0;j<6;j++)
3     for(k=0;k<3;k++) {
4     A[i][j][k] = ...; //S1
5     if(i>0 && j>1)
6       B[i][j][k] =
7         A[i-1][j-2][k]; //S2
8   }
```

(a) Code example for STOREQ concept definitions.



(b) Upper and Lower bound for the example in Figure 6.16a.

Figure 6.16: STOREQ principles (thanks to Per Gunnar Kjeldsberg).

6.5.1 STOREQ high-level estimator and the GLT engine

The Storage Requirement estimation (STOREQ) methodology presented by Kjeldsberg [122] targets the intra and inter in-place estimation without execution ordering. The estimation requires that all the iteration domains are placed in CIS (see Chapter 3). Then it estimates upper and lower bounds on the size of individual dependencies based on the partially fixed execution ordering, thus it is estimating the hypothetically best and worst intra in-place. The designer can choose which dimensions in the CIS are fixed and which are not fixed to constrain the estimation. Finally, simultaneously alive dependencies and their maximal combined size is computed, thus inter in-place estimation is added. However, this part of the estimation is still improving and is under development.

The transformation estimated is the interchange of the loops forming the CIS. The effects of the translation part or of other GLT linear transformations than interchange in CIS dimensions cannot be estimated. Before discussing in detail our coupling to the GLT framework we will demonstrate the basic estimation principles on an example in Figure 6.16.

Figure 6.16a shows a simple code with one nested loop within which the array A is written in statement *S1*. The array element written in statement *S1* is read two j and one i iterations later. The corresponding GM with the dependency is depicted in Figure 6.16b. The dotted boxes bound the iteration domains of *S1* and *S2*. The solid line box bounds the Dependency Part (DP). DP is part of the production iteration domain which is participating at the dependency. E.g., for our example the iteration domains of *S1* is $\{(i,j,k) \in \mathbb{Z}^3 \mid 0 \leq i,j \leq 5 \wedge 0 \leq k \leq 2\}$, however only elements written in $\{(i,j,k) \in \mathbb{Z}^3 \mid 0 \leq i \leq 4 \wedge 0 \leq j \leq 3 \wedge 0 \leq k \leq 2\}$ are read in iteration domain *S2* and thus participate at the dependency. Thus, the DP contains the iteration domain nodes where array elements are produced that are read within the depending iteration domain.

The current order of the loop in Figure 6.16a determines that we iterate first in the

k dimension, then in the *j* dimension and only then in the *i* dimension. Changing execution order means for STOREQ that this sequence is changed. This corresponds to loop interchange. E.g., when interchanging *i* and *k* the new sequence will be; first iterating in the *i* dimension, then *j* and finally *k*. For the 3 dimensional loop nest we have $3! = 3 \times 2 \times 1 = 6$ possibilities of execution order. For each execution order different iteration nodes are visited when going from the iteration where an array element is written (e.g., (0,0,0) for *A[0][0][0]*) to the iteration node where the array element is read (e.g., (1,2,0) for *A[0][0][0]*). For each execution order those iteration nodes form a set and so we have 3! sets for our example with 3 dimensional loop nest. The intersection of those sets intersected with the set of iteration nodes which are within the DP creates other set of iteration nodes (see dashed line in Figure 6.16b). The cardinality of such a created set is the estimated Upper Bound. The union of those sets intersected with set of iteration nodes which are within the DP creates yet another set of iteration nodes (see dash-dot-dot line in Figure 6.16b). The cardinality of such a created set is the estimated Lower Bound. Thus we can see that none of those estimation is realistic, i.e., the execution ordering that can achieve the upper or the lower bound does not exist. The estimation gives only the interval within which the real value will be and it can be refined by fixing certain dimensions in the CIS. If all the dimensions are fixed it converges to one value. In this case there will be only one set of visited nodes and the intersection and union with the iteration nodes which are within the DP will be the same. The STOREQ methodology can also give hints which ordering is optimal for given dependency.

The STOREQ methodology presented by Kjeldsberg [122] is implemented in a prototype tool [123]. This tool is written in MATLAB and works with its own geometrical model. The PER tool mentioned in Chapter 3 has an option to generate this special model. Use of this option requires that the input C code must be written in "CIS form", i.e., all loops should be nested and all array accesses should happen in the body of the inner loop as depicted in Figure 6.16a. This is not a limitation of the PER tool, but rather a limitation of the STOREQ input format.

To integrate the STOREQ estimation tool to the GLT framework we have coupled the two frameworks via a PERL script which transforms our geometrical model (see Chapter 3) to the STOREQ format. This has allowed us to use the STOREQ as an integral part of our GLT framework.

However, STOREQ did estimate only in-place and neglected the data reuse and the memory hierarchy layer assignment step. Also it did not estimate other transformations except of interchange. Thus together with the Norway University of Science and Technology (NTNU) in Trondheim, namely Qubo Hu and his advisor Per Gunnar Kjeldsberg, we have launched the research in hierarchical memory storage estimation to close this gap.

6.5.2 Hierarchical memory storage estimation interaction with GLT engine

The STOREQ approach was developed as a separate tool without considering its later integration into the GLT framework as a steering component. It did estimate only intra in-place, despite the fact that inter in-place estimation is being developed

as well, and it did not estimate the data reuse. The new estimation framework called Hierarchical Memory Storage Estimation (HMSE) eliminates those restrictions.

HMSE [105, 106] can be easily integrated to the GLT framework. It estimates intra in-place, inter in-place, MHLA and data reuse steps of the DTSE methodology for a given loop transformation. The estimation is incremental, i.e., if another incremental transformation is applied, only the estimation for the parts of the code affected by the transformation is updated. The effect of the incremental transformations is usually limited to a certain code region [87] allowing to use the incremental approach also for the estimation.

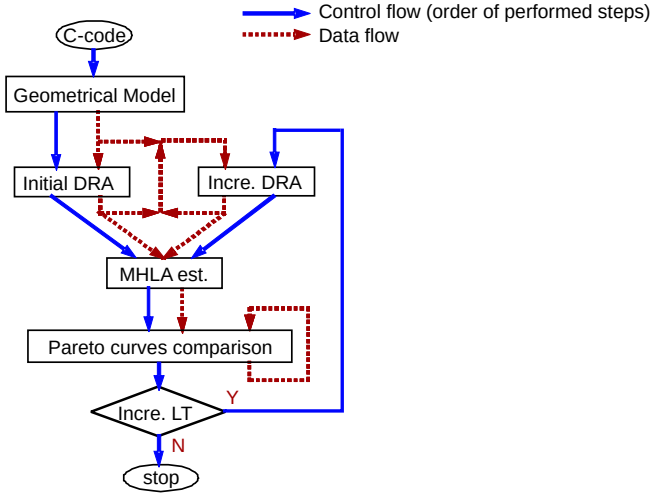


Figure 6.17: The flowgraph of HMSE framework (thanks to Qubo Hu).

Figure 6.17 depicts the flowgraph of the HMSE framework. After extracting the GM from the C code (which can be common with GLT framework) the initial Data Reuse Analysis (DRA) and MHLA estimation is used. The MHLA estimation takes into account a range of platform instances where the size of L1 on-chip memory is varying from zero to the size required for all data in the application. For each instance the mapping of arrays and their copy candidates computed in DRA is estimated. The complexity of this approach for two memory layers is $O(n \log n)$ where n is the number of arrays and copy candidates for all platform instances. This is very low; e.g., [26] analyzes only one instance of a two layer memory hierarchy with the complexity $O(2^n n^2 \log n)$. Thus, the MHLA estimation is very fast and results in the 2D Pareto curve trading-off the size of the L1 on-chip memory instance and the number of misses to the L1 on-chip memory (the dashed line in Figure 6.18). After the initial DRA and MHLA estimation the incremental loop transformation (e.g., fusion) is applied in the GLT framework. Then the incremental DRA is applied. That means the reconstruction of the copy candidates and data reuse trees for the affected code regions in GM. The incremental DRA is much faster compared to the initial DRA because it is performed locally. After that the MHLA is rerun and a new Pareto curve is generated (the solid line in Figure 6.18). Then another incremental loop transformation can be applied resulting in another Pareto curve (the dotted line in Figure 6.18 for interchange), etc. Each time the new Pareto curve is generated from MHLA esti-

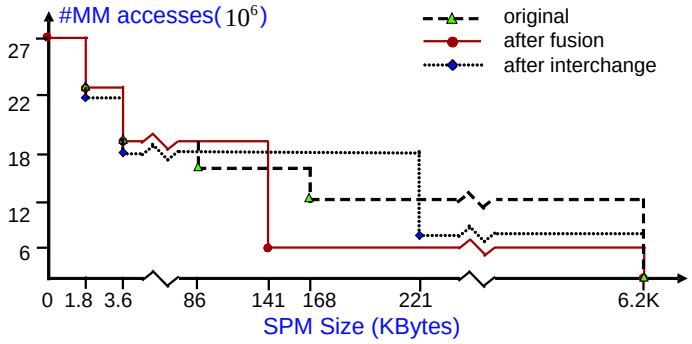


Figure 6.18: Pareto curve comparison in HMSE framework (thanks to Qubo Hu).

mation it is compared with the global Pareto curve and a new global Pareto curve is calculated. This global Pareto curve keeps information on which set of incremental transformations is optimal for the given L1 (SPM) size to achieve minimal number of L1 on-chip memory misses. The description of DRA and MHLA estimation is out of the scope of this dissertation and can be found in the dissertation of Qubo Hu [109].

The HMSE approach provides real trade-off curves between the size of the L1 memory instance and the number of misses to the L1 instance for the given set of incremental transformations. It keeps the global Pareto curve which determines the optimal incremental transformation from the set for the given L1 memory instance. The HMSE framework can be integrated into our GLT framework and the estimation can be used for steering the trade-off oriented GLTs. However, the code complexity estimation is not yet present in this work.

6.6 Effect of algorithmic kernel optimisations and computation-storage trade-off on the GLT

In Subsection 6.4.6 we have seen that the applied GLT have complicated the index expressions in the kernel. The generated assembly code has been more complex and has caused more instruction cycles despite the fact that the high-level code complexity (see Subsection 6.4.4) decreased. In this subsection we show a similar effect for software pipelining using the MP3 audio decoder application [132] mapped to the TI fixed point C64x processor. Software pipelining schedules instructions from a loop so that multiple iterations of the loop execute in parallel. This is beneficial for the performance of the application on Digital Signal Processor (DSP)s which have multiple Functional Unit (FU)s.

In this section we first focus on algorithmic kernel optimizations which make the kernels software pipelined. After that we will present how fusion (merging) of the kernels can affect software pipelining and how software pipelining interacts with algorithmic computation vs. storage trade-off.

We have focused on the long block requantization and joint middle-side stereo decoding kernels of the MP3 audio decoding application [132]; they are called requan-

```
for(...) {
  scale = scalefac_scale + 1;
  C = global_gain - 210 - (scalefac_l << scale);
  if (preflag)
    C -= (pretab << scale);
  //tab=[2^0, 2^(1/4), 2^(1/2), 2^(3/4)]
  y = tab[C&3]*(1 << (C >> 2));
  ...
}
```

Figure 6.19: The optimized requantization kernel of MP3 audio decoder; it shows how to efficiently compute C and $2^{\frac{1}{4}\times C} = 2^{\frac{C\%MOD_4}{4}} \times 2^{C/INT_4}$.

tization and stereo decoding from now on. These kernels are part of the most frequent path of the MP3 application and also part of one scenario as we identified earlier in this dissertation and thus is beneficial to optimize those using GLT. Scenario creation enabled this optimization which was not feasible before applying scenario technique. The requantization kernel computes the output sample *os* from the input sample *is* using the equation

$$os = sign(is) \times |is|^{\frac{4}{3}} \times 2^{\frac{1}{4}\times A} \times 2^{-B}$$

where A and B are given by:

$$A = global\_gain - 210$$

$$B = scalefac\_multiplier \times (scalefac\_l[ch][sfb] + preflag \times pretab[sfb])$$

The original implementation performs many calls to the "power" run-time support function. Furthermore, it has many multiplications and frequent calls to static Look-Up Table (LUT)s. It has also many floating point operations executed with the help of floating point libraries. Using these libraries similar to using run-time support functions and using statics prevents from applying software pipelining. Thus, our first goal has been to remove bottlenecks mentioned above to enable software pipelining. The static LUTs in function calls have been made global. The function calls have been inlined using the keyword *inline*. The floating point operations have been converted to fix point to avoid usage of floating point libraries. The biggest problem have been the "power" run-time-support function calls that cannot be inlined. To eliminate these calls an algorithmic change has been needed.

We can rewrite the equation computing output sample *os* as

$$os = sign(is) \times |is|^{\frac{4}{3}} \times 2^{\frac{1}{4}\times C}$$

where $C = A - 4 \times B$. The $|is|^{\frac{4}{3}}$ was implemented as a LUT. For computing the $C = A - 4 \times B$ we use the knowledge that *scalefac_multiplier* is 1 or $\frac{1}{2}$ depending on the *scalefac_scale* bit in the frame header and that preflag is just a bit in the frame

header. Then we can rewrite C as is listed in the code in Figure 6.19 in Lines 2–5. The expression $2^{\frac{1}{4}\times C}$ can be rewrite as

$$2^{\frac{1}{4}\times C} = 2^{\frac{4\times\left(C/^{INT}4\right)+C\%^{MOD}4}{4}} = 2^{C/^{INT}4} \times 2^{\frac{C\%^{MOD}4}{4}}$$

where $C/^{INT}4$ is the result of integer division by 4 of C and $C\%^{MOD}4$ is the modulo of C after integer division by 4.

The first part of the expression, namely $2^{C/^{INT}4}$ can be easily implemented using two shifts `1 << (C >> 2)`. The second part of the expression has 4 values only and can be tabulated `tab[C&3]`. For the 4 value LUT see Line 6 in Figure 6.19. The whole expression $y = 2^{\frac{1}{4}\times C}$ is implemented in Line 7 of the Figure 6.19.

The other implementation options are to use more or to fewer LUT entries. We can tabulate the whole $2^{\frac{1}{4}\times C}$ expression using an 384-entry LUT instead of our 4-entry LUT. We can also reduce the LUT for $|is|^{\frac{4}{3}}$ using the expression

$$|is|^{\frac{4}{3}} = 16 \times \left|\frac{is}{8}\right|^{\frac{4}{3}}$$

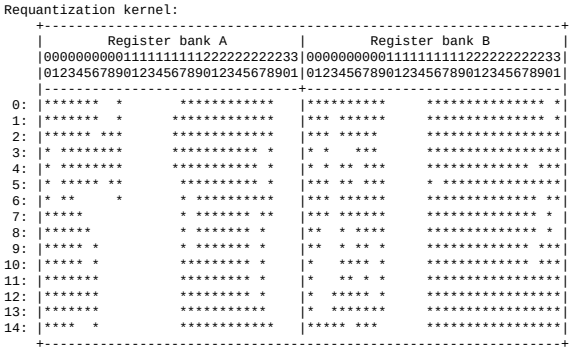
This causes reduction from 16kB LUT to 2kB LUT. Note, that the *is* value varies from 0 to 8206 and we need 2 bytes to store this value. We can also totally eliminate the LUT in $|is|^{\frac{4}{3}}$ using the Newton approximation method. However, this will require much more computational effort.

The stereo decoding kernel is very simple and is implemented straightforwardly from the algorithm description

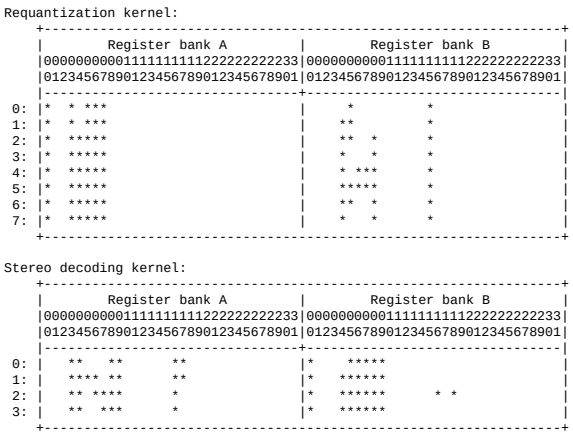
$$L_i = \frac{M_i + S_i}{\sqrt{2}}$$
$$R_i = \frac{M_i - S_i}{\sqrt{2}},$$

where M_i and S_i are middle and side value inputs which correspond to requantized *os* values in the two decoded channels, and R_i and L_i are right and left channel outputs.

The two kernels are now software pipelined. The initiation interval of the requantization kernel is 15 and the initiation interval of the stereo decoder kernel is 4. The initiation interval is an important value which determines the number of cycles between the initiation of successive iterations of the loop. The smaller the initiation interval, the fewer cycles it takes to execute a loop [236]. We can observe the initiation intervals and the TI C64x register usage from the register usage table for both kernels in Figure 6.20a. On the top we have the register usage table for the requantization kernel and on the bottom for the stereo decoder kernel. From the table we can observe that quite a lot of registers keeping intermediate values during computations are occupied in the requantization kernel. This is confirmed by the resource partition table of the critical resources in Table 6.9a. From the Table 6.9a we can see that for the requantization kernel the initiation interval is constrained by the number



(a) High register pressure in the MP3 audio decoder for computation intensive implementation when using 4-entry LUT for $2^{\frac{1}{4}\times C}$ computation in the requantization kernel.



(b) Low register pressure in the MP3 audio decoder for storage size intensive implementation when using 384-entry LUT for $2^{\frac{1}{4}\times C}$ computation in the requantization kernel.

Figure 6.20: The trade-off between computation and storage in the MP3 audio decoder and its effect on the register pressure.

	Requantization[4]		Stereo decoding	
	A-side	B-side	A-side	B-side
.S units	14*	14*	0	1
.X cross	14*	11	3	3
.D units	6	7	4*	4*
.T address	5	6	4*	4*

(a) Computation intensive implementation critical resource table.

	Requantization		Stereo decoding	
	A-side	B-side	A-side	B-side
.D units	8*	8*	4*	4*
.T address	4	8*	4*	4*

(b) Storage intensive implementation critical resource table.

Table 6.9: Critical resource tables.

of shift/Arithmetic-Logical Unit (ALU)/branch/fiels operation units (.S units) and the crossbars (.X cross) on the A-side due to the computationally intensive kernel. The problems of stereo decoding kernel are the data/addition/subtraction operations units (.D units) and the address paths (.T address).

The analysis had already shown that there is too much register pressure and some resources are critical and thus the loop fusion of those two kernels will not enable more ILP exploitation. We have tested it and after the necessary interchange and 1 time unrolling of the requantization kernel the initiation interval of the fused kernel becomes 34. This has confirmed our concerns that the register spilling in the fused kernel and the critical resources would not allow shortening of the initiation interval. Thus we have decided for another implementation in the computation vs. storage trade-off and have implemented the whole $2^{\frac{1}{4}\times C}$ expression by a 384-entry LUT table. The register usage table for this implementation is in Figure 6.20b and the resource partition table of the critical resources in Table 6.9b. From those tables we see that there is no register pressure problem any more in the requantization kernel and thus unrolling and fusion could enable more ILP exploitation. Also the critical resources are not computational resources any more but the data/addition/subtraction operations units (.D units) and the address paths (.T address). After interchanging and 1 time unrolling the requantization kernel, the kernels can be fused. The fusion caused a drop of the initiation interval to 10. The fusion enabled in-place optimizations which caused a further drop of initiation interval to 8 for the fused kernel.

[4]The values in the critical resource table are for iteration interval (II) 14, where no solution was found. The solution was found only by II 15. Unfortunatelly, the tables are constructed for the first II the compiler is trying.

The example of the QSDPCM video encoder mapped on the architecture with the ARM processor in Subsection 6.4.6 and the example of the MP3 audio decoder mapped on the architecture with the TI C64x processor have shown the relevance to consider the kernel implementation and optimization in relation to GLT. Thus, except high level estimations of storage size requirement (hierarchical or non-hierarchical), data reuse and high-level control-flow complexity, it would be beneficial to have also estimations of kernel implementation and optimization as part of the global control-flow complexity estimation. In contrast to to the high-level control-flow estimation this estimation cannot be platform independent, so the target platform description has to be one input for the estimation. Yet, it can be based on relative differences and comparisons of the platforms rather than on the absolute numbers. Similar to high-level estimation, this part is left for future work.

6.7 Conclusions

Multimedia applications require good performance for reasonable energy consumption. New methodologies reducing the number of data transfers and memory footprints, which are the crucial bottleneck of current multimedia applications, have been proposed. An important step of these methodologies are the GLT which enable the exploitation of following steps such as data reuse, in-place or efficient code generation.

However, most state-of-the-art loop transformation techniques aim at a particular goal and have a particular cost function. They do not consider that they affect all the subsequent steps. This requires to study important trade-offs that have not been analyzed yet.

In this chapter we have discussed and explained different trade-offs that loop transformations offer. We have explained these trade-offs on educative examples and demonstrated them on a real-life application, namely on the QSDPCM video encoder [193]. We have provided different Pareto points (versions of the application) in the 3D platform independent exploration space and mapped them in the 2D platform dependent exploration space.

To capture all these trade-offs among high level measures such as in-place, data reuse or control-flow complexity, high-level estimators, that can be used in early steps of our design flow, have to be used. We have discussed the development and future work in this area where we have also contributed.

At the end of this chapter we have discussed the GLT as enabler for exploiting more ILP. We have shown that GLT itself are not sufficient due to register pressure in the register banks and critical resources in the platform. To overcome this limitation we have to choose a different point in the algorithmic computation vs. storage trade-off at the kernel level. Thus we have to consider also those effects and come with future high-level estimators to capture these issues.

CHAPTER 7

Related work

Zu verlangen, daß er immer alles, was er je gelesen, behalten haben sollte, ist wie verlangen, daß er alles, was er je gegessen hat, noch in sich trage. Er hat von diesem leiblich, von jenem geistig gelebt und ist geworden, was er ist.
Arthur Schopenhauer
(1788-1860)

In this chapter we survey the related work. We split the chapter into three sections. The first section discusses the related work in hierarchical rewriting and condition hiding. We contributed to this area in Chapter 4. The second section gives an overview on the work performed in scenario related area. Scenarios were the main focus of Chapter 5. The third section summarizes the work in the loop transformation area with the link to trade-offs in GLT which was the title of Chapter 6.

7.1 Hierarchical rewriting and condition hiding

As stated in Chapter 4 in the DTSE optimizations we would like to focus only on the parts of the code that contain loops with large bounds and array signals which are data-dominant and thus relevant for the DTSE. The original code is usually intermingled; i.e., 1) constructs that are target of our optimization are spread across different functions and 2) constructs that should be hidden for the DTSE optimizations are in one function together with constructs we would like to optimize. In Chapter 4 we contributed to solving the second mentioned problem and to layering optimization scope at the design time. We created separate functions which encapsulate the constructs that should not be the target of DTSE optimizations. In this section we mention related work for both problems above.

To move the code from functions to a common scope, the inlining techniques are

used. In most of modern programming languages the designer can specify the functions that should be inlined by the keyword *inline* [192]. This inlining is the declaration based inlining; i.e., the inlining specification is used at the function declaration point to hint that all calls to that particular function have to be replaced with the corresponding body of the function. This approach does not allow to specify exactly which particular calls to the function are most essential for inlining. The function will be inlined at all call points where it is possible. This is not desired in the optimization context where only some call instances of the function should be inlined. In call based inlining approach the inlining specification is used at the function call point to hint that only at this particular point the inlining has to be done. However, this call point will be inlined independent of the followed control flow path, i.e. each invocation of this cal point will be inlined (e.g., if this call point is in the loop, it will be inlined for each iteration or if this call point is called from the different parts of the program, both will see this call point as inlined). The most selective inlining approach is the call instance based inlining approach, where different instances (in different control-flow paths) of the same function are distinguished and thus different invocations of the function holding the particular call point will see this call point as different call points. In the DTSE context to use the call instance based inlining was proposed in [1] to enlarge the exploration space for the DTSE optimizations. In [161] the call based and call instance based inlining was used to enlarge the exploration space for ADOPT optimizations resulting in the trade-off between the code size and speed-up achieved by the optimizations.

However, in the hierarchical rewriting and condition hiding it is also necessary to separate the code that is not the target of our optimizations. In [205, 206] the authors propose the function exlining technique which solves the inverse problem of function inlining. The authors provide two techniques, one for finding similar sequences of statements that can be replaced by calls to one function, and another for dividing a large set of statements into several functions, where each function performs a distinct computation. However, none of those two techniques targets specialized separation of the code as needed in the DTSE optimization context. In [150] an automated framework for code and data partitioning for the needs of data management is proposed. However, this framework is targeting high-level C++ specification only. A lot of work similar to exlining is in the synthesis and compilation area, where the sequence of statements or operators is grouped and replaced by powerful specialized block or instruction [95, 112, 4, 79]. The principles of hierarchical rewriting, i.e., separation of the code, in the optimization context have been worked out in the DTSE books [32, 28].

7.2 Scenarios

Scenarios create additional knowledge about the application during the design time that can be used and exploited. The actual scenario is then selected during the run-time. In Chapter 5 we have contributed to this research with providing the complete methodology for applying scenarios in the GLT context. The scenario knowledge is utilized for global optimizations during the design time. The decision which optimized scenario to use is postponed to run-time. Creating large optimization scopes

has already been proposed in the past in VLIW compiler optimization and scheduling techniques. In this section we first discuss those. Next, we make a link between our scenario instance for GLT and the general scenario approach and provide an overview about state-of-the-art in scenario work itself.

As the development of the architectures moved to the out-of-order execution and VLIW pipelined architectures (see Section 1.2), larger code blocks have been required for transformation and scheduling purposes to efficiently utilize the concurrency offered by those architectures. However, the applications were getting less analyzable at compile time due to more complex control-flow inside and among loop nests and different modes the applications can contain, resulting in the mapping problem described in Section 1.3. Thus the compiler and methodology designers have proposed several ways how to form larger blocks and go beyond the traditional block boundaries.

The first approach was to eliminate conditional branches from a program utilizing predicated execution support. This approach is the well known if-conversion approach [5, 154]. If-conversion was first proposed in automatic vectorization techniques for loops with conditional branches. It replaces conditional branches in the code with comparison instructions which set a predicate. Instructions which are control dependent on the branch are then converted to guarded instructions dependent on the value of the corresponding predicate. In this manner, control dependences are converted to data dependences in the code. If-conversion can eliminate all non-loop backward branches from a program [139]. However, it combines all execution paths in a region into a single block. Therefore, every instruction from that region has to be examined. So even instructions that would not be accessed so frequently due to different frequencies of the paths in the program, have to be executed [41]. Also, execution paths with subroutine calls or unresolvable memory accesses can restrict optimization and scheduling within the predicated block.

Well known techniques for the VLIW compiler optimization and scheduling which go beyond the traditional if-conversion technique are trace scheduling [73, 62], superblock creation [110], hyperblock creation [139, 140] and decision tree scheduling [103, 101]. They all have a similar goal - to enlarge the exploration space and allow global optimization or instruction scheduling without the negative effects of global if-conversion by focusing on certain execution paths only. In trace scheduling the code is divided into a set of traces that represent the frequently executed paths. There may be conditional branches out of the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances). Instructions are scheduled within each trace ignoring these control flow transitions. After scheduling, bookkeeping is required to ensure the correct execution of off-trace code by inserting compensation code. A superblock is a block of instructions such that control may enter from the top only (thus no side entrances are allowed), but may exit from one or more locations. The instructions within the superblock are not predicated, thus the superblock contains only instructions from one path of the program. A hyperblock is a superblock after if-conversion of set of paths and thus it contains instructions from different paths of the program. The blocks or paths which are part of the hyperblock are selected based on the execution frequency, size and instruction characteristics. If there are side entrances in the selected superblock/hyperblock region, the tail of the superblock/hyperblock is duplicated. Decision tree scheduling

is similar to superblock scheduling due to absence of the side entries. To perform compiler optimizations, predication can be employed in decision trees similar to hyperblock scheduling.

Inserting compensation code and tail duplication are approaches to duplicate the necessary code, used by trace scheduling and superblock scheduling. Compensation code needs much more engineering effort but has the potential advantage of requiring less code copying. However, [88] reports that trace scheduling does not always create less code growth and often creates more compared to superblock scheduling. In [36] the authors propose to combine the superblock scheduling and software pipelining to accelerate effectively a variety of programs.

Current multimedia systems use many programming constructs whose evaluation depends on the input of the program. This creates a gap between analysis and exploration requirements of current applications and the fully automated system level exploration tools. Examples such as tools for global memory optimization [28] or program-wide parallelization of loop nests [17] can only deal with programs that are analyzable at compile time and that are not dependent on program input.

To alleviate this gap and to make the applications more predictable requires a more advanced design time analysis. Thus, a scenario concept has been introduced that is based on partitioning the actual behavior of the application into distinct classes (scenarios) of typical behavior. Scenarios are selected at run-time, but they are exploited already at design time. The actual behavior is determined using specific profiling information [81, 208].

A scenario is defined as a set of Run-Time Situation (RTS)s which we choose to exploit in the same way. RTS is a piece of execution we treat as a unit, e.g., one path in the CFG. The actual RTS (e.g., path) is known only at the moment it occurs. The application execution is a sequence of RTSs. Examples of RTSs are an execution phase, an operating mode, one frame, temperature and variability situation, the wireless channel conditions, the set of active applications etc. In this dissertation the RTS is equivalent to one path in the directed acyclic CFG within the time loop as defined in Chapter 5. For real codes in our real target domain, by the time the RTS occurs, there is no time anymore to perform the exploitation. In the scenario approach, we do the exploitation at compile time for different RTSs. I.e., for the purpose of this dissertation the exploitation means the utilization of GLT within one RTS. Since the number of possible RTSs is large, it is necessary to cluster some RTSs in a scenario and use the same compile time exploitation for all of them. During the run-time phase, only the appropriate scenario has to be selected and the system switches to use this scenario by loading the appropriate code from the main or L1 cache memory into the L0 buffers. In order to distinguish from the use-case scenarios that are used when designing the system functionality [74], the full name is the application scenario [85]. Note, that other scenario instances we are going to discuss in the sequel are quite different from our approach which utilizes scenarios to create larger exploration space. Still, the basic principles, such as RTS characterization and identification, RTS clustering, scenario exploitation, scenario prediction and detection and scenario switching remain the same for all instances.

In [36] the authors use the information about periodic property of the multimedia application to present a new concept of Dynamic Voltage Scaling (DVS). Each period

in the application shows a large variation in terms of its execution time. The authors propose to supply the information of the execution time variations in addition to the content itself. This makes it possible to perform better DVS, resulting in lower energy consumption, as compared to the DVS approach which relies on Worst Case Execution Time (WCET) estimation. However, the authors do not specify how the periods should be identified.

The scenario concept was first used in [225] to capture the data dependent dynamic behavior inside a thread, in order to better schedule a multi-threaded application on a heterogeneous multi-processor architecture. In [81] scenarios have been used to refine the estimation of the WCET. This approach based purely on static analysis of application source code was extended towards profiling driven scenario detection and prediction for DVS aware scheduling in [84, 86]. In [35], the authors propose a mapping technique and compiler which identifies the hot path(s) and merges or duplicates the kernels, called Packet Processing Functions (PPF), in order to maximize system throughput focusing on the networking target domain. In [82] scenarios have been used for DVS similar to the work in [36]. Compared to [36], in [82] also a systematic methodology on how to detect the scenarios is proposed. However, the approach in [82] is purely static. It was extended towards profiling driven approach in [84, 86]. In [208] a number of phases is extracted in which each phase exhibits similar behavior. These phases are then exploited for hardware adaptation for energy efficiency.

Hyperblock or superblock formation, as enlargement of the scope of optimization and scheduling by considering multiple blocks of instructions, is considered as a fine grain approach. The basis for forming the hyper/superblock are the selected basic blocks that will participate in the hyper/superblock. Compared to this approach, we consider scenario creation as a coarse grain approach where the whole RTSs are identified and combined in the most beneficial way for the given target (DVS for energy, GLT, reconfiguration, etc.).

7.3 Loop transformations

In Chapter 6 we provided the different GLT trade-offs demonstrated on educational and real-life examples. GLT have been studied by many research groups in the past. In this section we try to survey an the state-of-the-art in loop transformations and the trade-offs studied during these transformations.

Most of the program execution is preformed in the loops [238, 133]. Thus, to improve the program performance, loops are the target of many optimization techniques already from the early age of optimizing or parallelizing compiler [3, 133]. Loop transformations aim at improving the cache performance or an effective use of parallel processing capabilities. In this section we first recap the work in loop transformations for parallelization. Then we survey the loop transformations for improving the locality and thus cache or software controlled Scratchpad Memory (SPM). We also briefly mention the most used GM libraries because the GM is mostly used as basis model for loop transformations. Finally we discuss the work in trade-off related loop transformations.

We can categorize the loop transformation work based on 1) the objective of loop transformations and 2) the model they use. The two main objectives of loop transformations are parallelization and improving locality. Based on the model we can distinguish work that uses Geometrical Model (GM) and work that does not use this model. We first start with an overview of the work that uses GM followed by an overview of the work that does not use GM. Note, however, that there has been a wealth of research in loop transformations. Thus this overview cannot be extensive and we rather introduce some pioneering work with respect to the different categories we just introduced. Good overview of loop transformations for locality can be found in [28] and for parallelization in [17].

Pioneering work in using GM for loop transformations aimed at improving the parallelism. In [133] the hyperplane method is used to identify a set of iteration points executed at a certain time stamp. The parallelization on GM has further been developed in the systolic array synthesis world [153, 151, 179]. The formulation is slightly different from the hyperplane method: loop transformations are performed by transforming the source polytope into a target polytope, and by then scanning that target polytope. For a parallel execution, the polytope is transformed (using an affine mapping) into a new polytope in a new coordinate system. In this coordinate system, certain dimensions correspond to space and others to time. Therefore the mapping is called a space-time mapping. Later on, the polytope model has also been used for parallelizing loop nests for massively parallel architectures [134, 71, 46, 72].

Initial research in loop transformation for locality focused on optimizing locality in a single perfectly nested loop by applying a single loop transformation [218]. The algorithm improves the locality of a loop nest by transforming the code via interchange, reversal, skewing and tiling. The algorithm has been implemented in the SUIF compiler. This approach did not allow to optimize large buffers between loop nest. In [116] perfectly nested loop nests are not required, however the authors consider only self reuse and it is reported that the algorithm does not obtain very good results for imperfectly nested loop nests. Another method to improve data locality is to perform data transformations [37] or use fine-grain scheduling techniques [10]. However, the fine-grain scheduling techniques do not really perform source code level loop transformations, since they do not lead to transformed code afterwards.

Most of the research in this field was based on the use of geometrical models where the set of transformations is limited to affine matrices. In [136] the authors discuss a loop transformation framework that is based on integer non-singular matrices. The framework includes transformations such as permutation, skewing and reversal, as well as a transformation called loop scaling. The authors claim that the framework is more general than existing ones; however, it is also more difficult to generate code. In [56] the authors specify the program using the Alpha language which uses a system of parameterized linear recurrence equations to represent the program. The authors propose a framework to transform the initial specifications into a parallel algorithm, i.e., to another system of recurrence equations, in which the time and the space index are separated. [71] proposes to use multidimensional affine schedules with lexicographic ordering for problems whose parallel complexity is polynomial. In [118] a framework for unifying iteration reordering transformations such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering is presented. In [141] the loop fusion and shifting techniques have been

proposed to maintain parallelism and allow the parallel execution of fused loops with minimal synchronization and to eliminate cache conflicts in fused loops. In [46] the authors deal with affine by statement scheduling, a high-level technique for the parallelization of loop nests with uniform dependences. In [38] the authors present an automatic method for computing the number of integer points contained in a convex polytope or in a union of convex polytopes. This can be used e.g., to compute the maximum available parallelism in the loop nest. Compared to the previous approaches which target parallelization, [77] is targeting the minimizing of memory accesses in loop nests by improving data temporal locality. The intermediate buffers are eliminated applying moving, merging and loop alignment transformations on the GM. This work was extended in [191].

Other researchers did not use Geometrical Model (GM) at all. Thus, they can usually apply only a very limited set of linear transformations. A good overview of those techniques can be found in [17]. In [6] the transformations are performed on the code itself to parallelize the FORTRAN programs. Therefore, different kinds of loop transformations (permutation, reversal, ...) usually have to be considered (and evaluated) individually. In [173] the authors propose two compiler schemes, namely cycle shrinking and run-time dependence checking, that can be used to automatically transform serial loops to a parallel form. Cycle shrinking performs loop parallelization at compile time. Run-time dependence checking, however, prepares the loop for run-time parallelization. This is achieved by inserting appropriate code in the source program, which automatically performs dependence checking and bookkeeping during program execution. In this case, parallelism is exploited at run time. The authors have also proposed a hardware solution for barrier synchronization, which greatly reduces the overhead associated with nested serial and DOALL loops. In [156] the authors use loop reverse and the permutation of two loops to fuse the loops and process them in the pipelined manner. They have improved the collective analysis technique which uses the graph coloring proposed in [182] to determine whether a cluster can be pipelined. In [117] the authors investigated dynamic loop analysis techniques to expose huge potential of the coarse-grain parallelism in programs.

At IMEC and the Computer Science Department of the K.U. Leuven we have addressed many of the problems of loop transformations for locality in our past research. Van Swaaij et al. [203] have proposed to work in two phases to limit the complexity and to improve scalability: a placement step and an ordering step. The placement step determines particular affine mapping functions for loop transformations to obtain improved overall locality. The ordering step defines the valid execution ordering. Danckaert [43] has split the placement step in a linear transformation step and a translation step. To steer these two steps, he also has provided several heuristics. The linear transformation step deals with the linear part and the translation with the constant part of the affine mapping function. This split further reduces overall algorithm complexity. Verdoolaege [211] has shown that it is possible to avoid the ordering step introduced by Van Swaaij in the past. He also further improved the heuristic steering techniques. However, up till now the loop transformations targeted one optimal solution. In reality, multiple optimal solutions leading to the trade-off exist, depending on processor architecture and the memory subsystem instance [107].

The Geometrical Model (GM) (a.k.a. polyhedral model) is the most preferred model to use for loop transformations for locality as well as for parallelization. The theory of convex polytopes and linear and integer programing used in the model is described in [92, 183]. Several libraries manipulating polyhedra have been implemented in the past. The four most famous are the PolyLib library [217], the Omega library [176], the Parametric Integer Programming (PIP) library [70, 68] and the Parma Polyhedral Library (PPL) library [11]. The PolyLib library is the library for manipulating polyhedral domains which represent the integer points in unions of polyhedra. The Omega library manipulates integer tuple relations and sets which are described using Presburger formulas [127, 187]. Presburger formulas are a class of logical formulas which can be built using affine constraints over integer variables, the logical operators $\neg$, $\vee$, $\wedge$, and the quantifiers $\forall$ and $\exists$. PIP/PipLib is the parametric integer linear programming solver developed by Feautrier for finding the lexicographic minimum of the set of integer points lying inside a convex polyhedron. This polyhedron can depend linearly on one or more integral parameters. The PPL is a modern C++ library providing numerical abstractions especially targeted at applications in the field of analysis and verification of complex systems. A more extensive list of polyhedral libraries can be found in [213]. Besides the public libraries for GM manipulation, also the proprietary GM libraries such as GM library in ATOMIUM [229] exist.

All researches above focused on improving one particular property of the program, e.g., data locality or feasibility to parallelize the program, without considering other effects caused by their transformations. Only few exceptions have been published on this individual objective. Vander Aa et. al. [210] have shown the trade-off between data locality and the instruction locality. After fusing small loop nests, the fused large loop nest cannot fit in the loop buffer resulting in overhead in the instruction memory. In the code generation phase [178] the authors mention different possibilities for generating the code, however they do not study explicitly the trade-off between the code complexity and code size as we do it in our work. Thus, until now, nobody systematically studied and identified the particular cost components contributing to the data locality (which is crucial for global energy reduction) and the negative effects of data locality improvements. In our work in Chapter 6 we focus on those effects and show that clear trade-offs exist during the loop transformation phase. Compared to [210] we discuss much smaller changes in the code achieved by loop shifting that do not affect the size of the loop nests and list all effects on the components contributing to the data locality. To extend the exploration scope of the global loop transformations different preprocessing techniques like Selective Function Inlining (SFI) [1], Pointer Analysis and Conversion (PAaC) [184, 75], Dynamic Single Assignment conversion (DSA) [209], hierarchical rewriting [28] and scenario creation [165] have been proposed. These preprocessing techniques also often require trade-offs between the freedom they allow for loop transformations, and extra cost we have to pay (e.g., code size). These trade-offs are orthogonal to global loop transformation trade-offs, i.e., the constraints created during the preprocessing are propagated to the GLT, but not other way around. This means, the trade-offs made at the preprocessing level cannot be revised, they can be only refined at the GLT level.

An interesting and novel area w.r.t. loop transformations is the adaptive and iter-

ative compilation. In [40] the optimization passes are selected and tailored to the compiled application. In the iterative compilation [124, 125, 83, 97] many variants of the source program are generated and the best one is selected after profiling the different variants on the target hardware. The variants that are best for energy are not always best for the performance resulting in a trade-off. However, this trade-off is not considered explicitly in iterative compilation research. To gradually improve the optimization results some authors are using machine learning [2]. The work of Pouchet et al. goes also in this direction in the context of polyhedral transformations. The more time we spend in the iterative compilation, the better results we obtain, resulting also in certain type of trade-off. However, the trade-offs among different cost properties such as data memory energy versus instruction memory overhead of the program are not explicitly considered in those approaches.

CHAPTER 8

Conclusions and future work

What is written without effort is in general read without pleasure.
Samuel Johnson
(1709-1784)

The low power requirements of embedded systems resulting from limited battery lifetime and hard constraints for heat dissipation demand efficient implementation of those systems. This can be achieved using high-level optimization methodologies like Data Transfer and Storage Exploration (DTSE). In this chapter we first summarize the contributions of this dissertation to the DTSE design flow for low power embedded systems. We have proposed several improvements in the flow: systematic hierarchical rewriting in the preprocessing step, use of scenarios to enlarge the Global Loop Transformations (GLT) exploration space, and trade-off oriented GLT. After summarizing the contributions in Section 8.1, we sketch directions of future research in the preprocessing, the scenarios and trade-off oriented GLT in Section 8.2.

8.1 Summary and conclusions

Data transfers and storage of large arrays in background memories are dominating contributors to the area and power consumption for all modern multimedia embedded systems. Modern high-level memory optimizations such as DTSE contribute to the cost-efficient realization of these systems. In these optimizations an important step involves GLT which enables later data-reuse and in-place optimizations. To benefit fully from these transformations, the right optimization scope has to be exposed for this step. This is especially true for embedded system applications with complex control-flow which prohibits design time optimizations, like advanced global loop transformations. Chapters 4 and 5 contributed to the preprocessing which sets the right scope for DTSE optimizations. In this dissertation we also identified that

the GLT step of DTSE prunes potentially good solutions from the exploration space. Thus in Chapter 6 we presented research towards multi-objective decisions in the loop transformation stage of DTSE.

All of those issues occur during the platform independent stage of the DTSE. Thus, the primary goal of this dissertation has been filling the gaps and extending some steps in this stage towards future dynamic applications with complex control flow and towards multi-objective decisions. The dissertation contributes particularly in four areas:

- Formalization and implementation of the hierarchical rewriting in preprocessing (Chapter 4) [162]
- Proposing a systematic methodology of scenario usage for extending the applicability of GLT (Chapter 5) [163, 164, 165, 166, 167, 169]
- Systematic application case studies leading to trade-off oriented GLT (Chapter 6) [168]
- Coupling of high-level estimators to the GLT framework as steering mechanism for trade-off oriented GLT (Chapter 6) [123, 12, 180, 104, 105, 106, 107, 108, 196, 197]

The DTSE methodology requires a clear separation of top-level process control flow, loop hierarchy, indexed signals and arithmetic, logic and data-dependent operations. The top-level control flow is usually well separated by the designer. However, the loop hierarchy, indexed signals, arithmetic, logic and data-dependent operations are intermingled. In this dissertation we have formalized and implemented the hierarchical rewriting which separates the loop hierarchy and indexed signals from arithmetic, logic and data-dependent operations. The splitting is performed on an Abstract Syntax Tree (AST) where the undesired constructs are identified and hidden by function encapsulation. The remaining code consists of loop hierarchy and indexed signals only, which is the starting point for the DTSE steps. We have developed an algorithm for rewriting the innermost if-conditions to ternary operators and encapsulation of if-converted basic block computations into functions. Using these algorithms on a QSDPCM video encoder followed by the DTSE optimization steps reduces the number of main memory accesses by 31.3% when compared to direct DTSE optimization without our preprocessing.

The GLT step of the DTSE methodology can be applied only to the Static Control Part (SCoP)s of a program. An SCoP is a maximal set of consecutive statements without *while* loops, where loop bounds and conditionals may only depend linearly on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters. In the past, most programs were static and the whole program usually fulfilled the requirements for being one SCoP. Future multimedia applications are dynamic and have quite complex CFG. They consists typically of large amounts of small SCoPs. In this dissertation we have proposed using scenarios to create bigger SCoPs which exhibit more optimization freedom compared to the original application. Using scenarios on top of the DTSE methodology reduces the number of main memory accesses for an MP3

audio decoder by 45.8% when compared to only applying the DTSE methodology. Except for the basic scenario methodology for GLT, we have developed also several heuristics with different accuracy and execution time. We have provided also extensions of the scenario methodology for while loops and for identification of scenarios when the switching cost is dominant.

We believe, that scenario oriented techniques will play an important role in future applications which are only partly predictable at run-time. This area is the most challenging future research from the topics that we covered in this dissertation. From an implementation point of view, scenario techniques, as proposed in Chapter 5, have to be covered as soon as possible by modern compilers. Particularly, for scenarios in GLT, GLT optimizing compilers which support the GM will be required. Such compilers will be available in the near future [174].

The state of the art in GLT focuses on one particular cost function. In this dissertation we have shown that this can lead to suboptimal solutions and that the trade-off oriented GLT approach is crucial. We have demonstrated it on small educational examples and also on a real-life QSDPCM example. The different versions of the real-life example resulting from the trade-off oriented GLT approach have been compiled for the ARM processor, resulting in a trade-off between the execution time and Layer 1 data memory size. We have proposed to combine the trade-off oriented GLT with high-level cost estimators which can rapidly estimate the impact of the incremental GLT on the remaining DTSE steps. This work has been performed together with our colleagues from the Norway University of Science and Technology. We have also shown on a MP3 audio decoder, that algorithmic kernel optimizations can affect the GLT decisions.

Parts of this dissertation have been implemented in a set of prototype tools based on different external and internal C++ libraries such as the ATOMIUM library set, the Backbone (BB) library, the Boost library set, and STL. The hierarchical rewriting prototype tool (3859 C++ lines) covers rewriting of the innermost if conditions to ternary operators and encapsulation of if-converted basic block computations into functions as explained in Chapter 4. The scenario creation prototype tool (6908 C++ lines) covers the scenario creation technique described in Chapter 5 from parsing the C code till generation of the code for different scenarios using different heuristics. The output of those tools can be coupled with the GLT tool input. Dealing with the *while* loops is still a semi-automatic process where the special cases for particular loop trip counts (see Section 5.9 in Chapter 5) have to be created manually. For the switching cost, just the heuristic is implemented. The tools are supported by different *perl* helper scripts. The trade-offs in Chapter 6 are performed manually. The STOREQ and HMSE estimators are supported by Matlab and *python* scripts resulting from joined research (and written by) colleagues from the Norway University of Science and Technology.

8.2 Directions of future research

In this dissertation we have filled several gaps in the context of the GLT step of the DTSE methodology. We proposed a systematic hierarchical rewriting and condition hiding methodology, a scenario methodology for GLT and a trade-off oriented GLT.

The first two contributions extend the optimization space of GLT. The last contribution avoids discarding good points form the GLT exploration space for the following steps. Still there is plenty of room for future research directions. Below we propose several future research topics, grouped in four areas: hierarchical rewriting, scenarios, trade-offs in GLT and high-level estimators.

In the *hierarchical rewriting* related area, we see the main extensions in preprocessing more general code, e.g., MATLAB, C++, Java, where the identification of the layers is also needed.

In the *scenario* related area, we would desire an approach using the N-dimensional Pareto curve distance for grouping the Run-Time Situation (RTS)s and generalizing scenario flow. As we can see in Section 7.2, in state-of-the-art research, the scenarios are already used for different purposes. The applications are going to be more and more dynamic and the scenario approach can help to handle this dynamism for different aspects like GLT or DVS. Future research should aim at unifying and generalizing these scenario flows. In scenarios for GLT, we would like to handle more general code represented by an arbitrary CFG. This can be achieved by using advanced control-flow analysis and loop identification based on control-flow dominators [3] and path profiling of arbitrary control-flow graphs [15]. Also the heuristics used in this dissertation can be further improved, e.g., by considering multiple solutions for a given number of scenarios in case of Loss/Similarity and Fruchterman-Reingold heuristics. Note, that the current realization of those heuristics provides only one point for a given number of scenarios. Considering multiple solutions will increase the quality, but the execution time will increase. The proposed clustering heuristics go into the direction of hierarchical clustering [113]. Also other clustering algorithms such as exclusive clustering (e.g., K-means clustering [138]) should be evaluated using our proposed clustering metrics. It would be interesting and challenging to try overlapping clustering (e.g., Fuzzy C-means clustering [59]) or probabilistic clustering (e.g., Mixture of Gaussians [54]). Advanced scenario predictors and run time switching between scenarios are other important aspects that have not been worked out in this dissertation. Work is ongoing in the scenario PhD team [237]. Also, a detailed analysis of scenario recovery overhead when the scenario predictor has made a wrong decision is needed.

In the *trade-off oriented GLT*, the relation between the high-level metrics and the area, power and performance has to be studied. However, to properly identify this relation, the platform information and power models have to be analyzed together with the high-level metrics.

In the *high-level estimation* related area, the development of a control-flow complexity estimator is crucial as we have seen in Chapter 6. This estimator has to be coupled with the Hierarchical Memory Storage Estimation (HMSE) and later integrated into a trade-off oriented GLT framework.

Both the scenario concept and the trade-off oriented concept of GLT in processor level DTSE can also be reused inside the task level DTSE and data level DTSE stages [27]. Task level DTSE focuses on data transfer and storage issues within multiple dynamic threads mapped on multiple processing elements (where each processing element can consist of multiple processors in an array). Data level DTSE objectives are the data transfer and storage issues within a homogeneous (processor)

array (also including sub-word parallelism). Processor level DTSE deals with the memory issues inside one thread and one processor (and its memory hierarchy). In this dissertation we have focused only on the last issue; the instantiation for the other two flows requires also further research. Thus, some of the proposed techniques may have to be reevaluated in light of automatic parallelization where also other aspects (e.g., communication) have to be considered.

APPENDIX A

Functional description of the test-vehicle applications

In this appendix we describe the functionality of two multimedia algorithms, an audio decoder and a video encoder which we have used in most of our experiments.

A.1 MP3 audio decoder

The MPEG-1 Layer 3 decoder is a frame-based algorithm for decoding a bitstream from the perceptual audio encoder (see Figure A.1). A frame is an instance coding 1152 mono or stereo samples and is divided into two granules of 576 samples. Each granule consists of 32 subband blocks of 18 frequency lines. One channel in one granule is called a block.

After receiving a frame, the frame is Huffman decoded (see shaded part on Figure A.1). Then, on the Huffman decoded frame, several computational kernels are applied (see the rest of Figure A.1). The computational kernels contain large loop nests with mostly manifest affine loop bounds, conditions and array indices and deal with multidimensional arrays as stated in the characteristics of our target application domain in Chapter 1. For functional details about the different computational kernels, see e.g., [132].

Several MPEG-1 Layer 3 decoders exist, e.g., [100]. Their implementations include processor-specific low-level optimizations, focusing on performance, and trading-off computation for storage. This is definitely not good for energy. Thus, these decoder implementations are not suited as an application driver for low-power high-level techniques we are discussing in this dissertation. However, the reference code from Lagerström [132] is close to the initial specification. Also, high level algorithmic

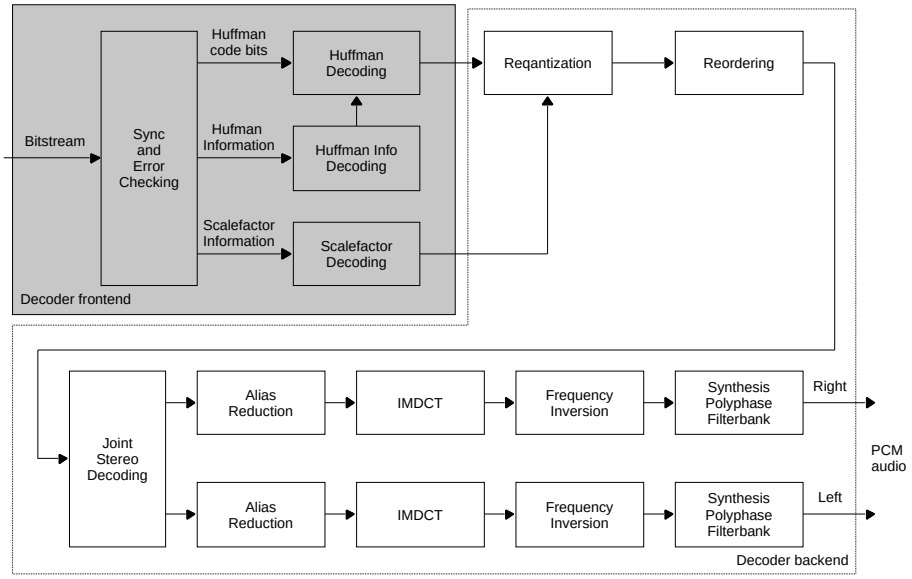


Figure A.1: MPEG-1 Layer 3 audio decoder structure.

optimizations, like using Newton's method in the requantization kernel, using fast IMDCT in IMDCT kernel and using fast DCT in matrixing operation of the synthesis polyphase filterbank [132], are already performed on the code.

For the decoding of the MPEG-1 Layer 3 blocks (one channel in one granule), different methods are used depending of the initial coding of the block on the encoder site. Each block can be coded using a short window, mixed window or one of 3 types of long windows. The type of window determines on how many samples the IMDCT is performed and what is the weight of the individual samples. Also, each block can be coded using stereo decoding or joint stereo decoding (middle side and/or intensity stereo decoding). The combination of these options introduces different modes of decoding which are represented with data dependent conditions in the driver code. However, the probability distribution of the different modes is not uniform. To identify the most probable modes the application needs to be profiled.

A.2 QSDPCM video encoder

The QSDPCM is a data-dominated multimedia application with many accesses and large arrays. The application has an inter frame compression technique for video images which can be used for image sequences at very low bit-rates (videophone, videoconference, etc.). It involves a three-stage hierarchical Motion Estimation (ME), a quadtree based encoding of the motion compensated frame-to-frame difference signal, and a quantization followed by Huffman compression. This section briefly explains the application; a more detailed description can be found in [193].

Figure A.2 gives an overview of the components in the application. A ME is based on

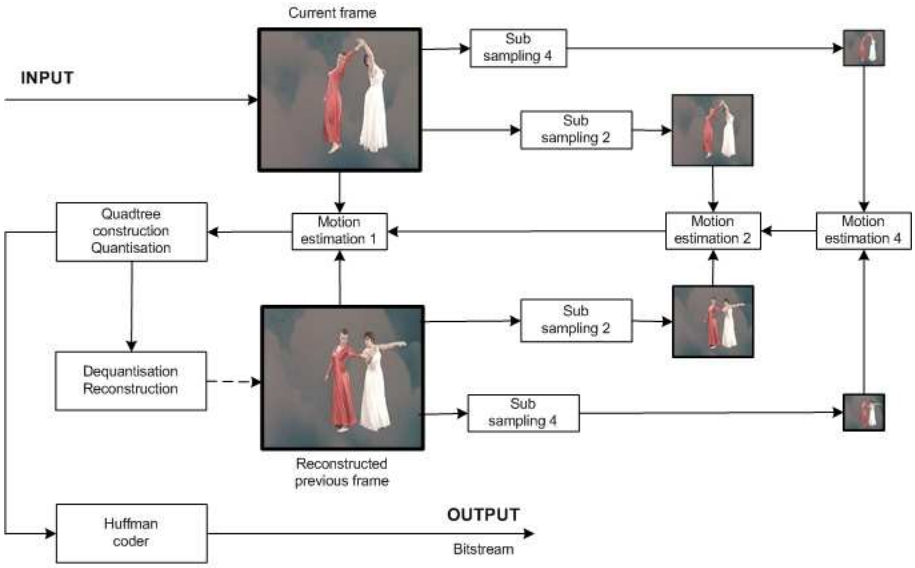


Figure A.2: QSDPCM video encoder structure.

the temporal correlation between successive frames. Therefore, the ME searches for the blocks that have the biggest similarity. A block in the current frame is matched with a block in the previous frame and a Motion Vector (MV) points to this matching block. An exhaustive search for matching blocks, however, can be very expensive. Therefore, in this application an initial search is made on a subsampled frame. The smaller data sizes reduce the effort needed to search in a large range by hiding details. The details are gradually added in 3 stages. First a motion estimation with a block of 4×4 in a 4 times subsampled frame (4 in both dimensions thus $16\times$ smaller) is searched in a range of ±4 subsampled pixels. Note that this is a range of ±16 pixels on the full resolution frame. Then the motion vector is refined in the sub-sampled by two resolution. A search of ±2 subsampled pixels is made, starting from the MV found in the previous stage. This method will extend the maximum range of searching, because the search is continued even when an extreme vector was found in the first stage. This brings the total search range to ±20 pixels in full resolution. Finally, the last (full) search of ±1 pixel is made on the full resolution frame and extends the range to a total search range of ±21 pixels. Thus a search is performed over 43×43 locations requiring a search area of 58×58 pixels in the previous frame.

All video codecs transform, except of the MV, also additional information to efficiently code the spatial correlation between the current image and motion compensated previous image represented by the coded error image. Mostly, the error image is decomposed into Macro Block (MB)s of size 16×16 that are transformed and coded individually. DCT is the most widely used transform for this purpose. Although the DCT is a very appropriate technique for coding of highly correlated natural images, the same transformation does not work efficiently when applied on motion compensation prediction error images.

As long as the motion estimator can track the moving object accurately, the resulting

motion compensation error images exhibit line drawing characteristics with significant amplitudes appearing mainly along the boundaries and in high-contrast regions of moving objects. These characteristics are the reason for the unsatisfactory behavior of the DCT.

Thus, the motion compensation error images with these characteristics are weakly correlated, and a DCT coding does not result in significant further compression of the motion compensation error signal. The author of the QSDPCM application [193] shows that the motion compensation error images can better be directly encoded without transformation to an intermediate frequency domain.

In the QSDPCM method, the Motion Compensation (MC) error image is adaptively decomposed into blocks of variable size, where in each block the MC error image is represented by the local sample means. Essentially, the algorithm first tries to code the entire MB of the error image using one single mean value. This mean value is subtracted from the pixel values to calculate the residual errors. In a second stage, the MB can be decomposed into 4 smaller blocks when the residual error is too large. These smaller blocks code the residual error using 4 means. This process is repeated hierarchically down to the pixel level if necessary. The variable block size structure is described by a quadtree. The local sample means are independently quantized and Huffman coded.

Nederlandse samenvatting

Nullum'st iam dictum quod non sit dictum prius.
Publius Afer Terentius
(185BC-159BC)

De gegevensoverdracht en -opslag van grote meerdimensionale rijen in achtergrondgeheugens hebben een dominante invloed wat betreft oppervlaktevereisten en vermogenverbruik voor alle moderne ingebedde multimediasystemen. Moderne hoog-niveau optimalisaties zoals DTSE dragen in grote mate bij tot de kosten-efficiënte verwezenlijking van deze systemen. Binnen deze optimalisaties is een grote rol weggelegd voor globale lustransformaties (GLT), die latere hergebruik- en in-plaats-geheugenbeheeroptimalisaties mogelijk maken. Opdat deze transformaties ten volle tot hun recht zouden kunnen komen, moet de juiste draagwijdte van deze stap blootgesteld worden. Dit is vooral het geval voor ingebedde systeemtoepassingen met complexe controlestromen die verhinderen dat optimalisaties, zoals geavanceerde globale lustransformaties, tijdens het ontwerp gebeuren.

Hoofdstukken 4 en 5 dragen bij tot de voorbewerking die de draagwijdte van de DTSE-optimalisaties maximaliseren. In deze dissertatie tonen we ook aan dat de GLT-stap van de DTSE-methodologie soms waardevolle alternatieven uit de exploratieruimte wegsnoeit. Daarom presenteren we in Hoofdstuk 6 onderzoek naar beslissingen met meervoudige doelstellingen in het transformatiestadium van de DTSE-methodologie.

Al deze kwesties komen aan bod tijdens het platformonafhankelijke stadium van de DTSE-methodologie. Daarom is de primaire doelstelling van deze dissertatie het opvullen van de gaten in, en het uitbreiden van, een aantal stappen in dit stadium naar toekomstige dynamische toepassingen met complexe controlestromen en naar beslissingen met meervoudige doelstellingen. Deze dissertatie draagt vooral bij op de volgende vier gebieden:

- De formalisatie en implementatie van hiërarchische herschrijving tijdens de voorbewerking (Hoofdstuk 4) [162])
- Het voorstellen van een systematische methodologie die gebruik maakt van

scenario's voor het uitbreiden van de toepasbaarheid van GLT (Hoofdstuk 5) [163, 164, 165, 166, 167, 169]

- Systematische gevalstudies van toepassingen, die leiden tot afwegingsgebaseerde GLT (Hoofdstuk 6) [168]
- Het koppelen van hoog-niveaus schatters in het GLT raamwerk als sturingsmechanisme voor afwegingsgebaseerde GLT (Hoofdstuk 6) [123, 12, 180, 104, 105, 106, 107, 108, 196, 197]

De DTSE-methodologie heeft nood aan een duidelijke scheiding tussen de topniveau process controlestromen, de lushiërarchie, de geïndexeerde signalen en wiskundige bewerkingen, de logica en de data-afhankelijke bewerkingen. De topniveau controlestromen zijn gewoonlijk reeds behoorlijk afgescheiden door de ontwerper. Echter, de lushiërarchie, de geïndexeerde signalen en wiskundige bewerkingen, de logica en de data-afhankelijke bewerkingen zijn meestal door elkaar gemengd. In deze dissertatie hebben we de hiërarchische herschrijving geformaliseerd die de lushiërarchie en geïndexeerde signalen van de overige componenten scheidt. Deze scheiding wordt uitgevoerd op een Abstract Syntax Tree (AST) waar de ongewenste constructies geïdentificeerd worden en daarna verborgen worden door ze te verpakken in functies. De overblijvende code bevat dan enkel de lushiërarchie en de geïndexeerde signalen, wat een ideaal startpunt is voor de DTSE-stappen. We hebben een algoritme ontwikkeld voor het herschrijven van de binnenste *if*-condities naar ternaire operatoren en het inpakken van de geconverteerde basisblokken in functies. Door gebruik te maken van deze algoritmes op de QSDPCM video-encoder, gevolgd door de DTSE-optimalisatiestappen, wordt het aantal toegangen tot het hoofdgeheugen met 31,3% gereduceerd in vergelijking met de directe toepassing van DTSE-optimalisaties zonder onze voorbewerkingen.

De GLT-stap van de DTSE-methodologie kan enkel toegepast worden op de Static Control Part (SCoP)s van een toepassing. Een SCoP is een maximale groep van opeenvolgende programma-instructies zonder *while*-lussen, waar de lusgrenzen en condities enkel linear afhangen van invarianten binnen deze groep van instructies. Deze invarianten omvatten symbolische constanten, formele functieparameters en tellers van omliggende lussen. In het verleden waren de meeste toepassingen vrij statisch qua gedrag en voldeed de toepassing in zijn geheel dikwijls aan de vereisten van een SCoP. Toekomstige multimediatoepassingen zijn echter zeer dynamisch en hebben vrij complexe controlestromen. Deze bevatten meestal een groot aantal kleine SCoPs. In deze dissertatie stellen we het gebruik van scenario's voor om grotere SCoPs te creëren die meer optimalisatievrijheid vertonen in vergelijking met de oorspronkelijke toepassing. Door het gebruik van scenario's bovenop de DTSE-methodologie kunnen we het aantal toegangen naar het hoofdgeheugen van een MP3 audiodecoder met 45,8% reduceren in vergelijking met het geval waarin enkel de DTSE-methodologie wordt toegepast. Naast de basis scenario-methodologie voor GLT, hebben we ook een aantal heuristieken ontwikkeld met verschillende nauwkeurigheden en uitvoeringstijden. We hebben ook uitbreidingen aan de scenario-methodologie voorgesteld voor *while*-lussen en voor de identificatie van scenario's wanneer de kosten voor het omschakelen dominant zijn.

We zijn ervan overtuigd dat de scenariogebaseerde technieken een belangrijke rol

gaan spelen in toekomstige toepassingen waarvan het gedrag slechts beperkt voorspelbaar is tijdens de ontwerpfase. Dit onderzoeksonderwerp is het meest uitdagende van de onderwerpen die we in deze dissertatie behandelen. Vanuit een implementatiestandpunt zouden scenariotechnieken, zoals voorgesteld in Hoofdstuk 5, zo snel mogelijk moeten geïntegreerd worden in moderne compilers. Voor het gebruik van scenario's voor GLT hebben deze compilers nood aan een geometrische modellering. Dit soort van compilers zal in de nabije toekomst beschikbaar zijn [174].

Het state-of-the-art onderzoek naar GLT concentreert zich op een specifieke kostenfunctie. In deze dissertatie tonen we aan dat dit kan leiden tot suboptimale oplossingen en dat een afwegingsgebaseerde GLT-benadering van cruciaal belang is. We tonen dit aan op kleine educatieve voorbeelden, maar ook op het realistische QSDPCM-voorbeeld. De verschillende versies van dit laatste voorbeeld die bekomen zijn met behulp van de afwegingsgebaseerde GLT-benadering hebben we gecompileerd op een ARM processor, en dit resulteerde in een afweging tussen de uitvoeringstijd van de toepassing en de benodigde grootte van de L1 geheugenlaag. We stellen voor om deze afwegingsgebaseerde GLT te combineren met hoog-niveau kostenschatters die snel de impact van de incrementele GLT kunnen afschatten voor de overblijvende DTSE-stappen. Dit onderzoek werd uitgevoerd samen met onze collega's van de Norway University of Science and Technology. We hebben ook aangetoond op een MP3 audiodecoder dat algoritmische kerneloptimalisaties de GLT-beslissingen kunnen beïnvloeden.

Delen van deze dissertatie werden geïmplementeerd in een aantal prototype-programma's, gebaseerd op zowel interne als externe C++ bibliotheken zoals ATOMIUM, de Boost bibliotheken en STL. Het hiërarchische herschrijvingsprototype (3859 lijnen C++) omvat het herschrijven van binnenste *if*-condities naar ternaire operatoren en het verpakken van de geconverteerde basisblokken in functies, zoals beschreven in Hoofdstuk 4. Het scenariocreatie-prototype (6908 lijnen C++) omvat de scenariecreatietechniek beschreven in Hoofdstuk 5 vanaf het inlezen van de C code tot de generatie van de code voor verschillende scenario's, gebruik makende van verschillende heuristieken. De uitvoer van deze prototypes kan als invoer gebruikt worden voor het GLT-prototype. Het behandelen van de lussen gebeurt vooralsnog semi-automatisch, waarbij de speciale gevallen manueel geconstrueerd moeten worden. Voor de omschakelingskostenschatting is enkel de heuristiek geïmplementeerd. Deze programma's worden bijgestaan door verschillende *perl* hulpscripts. De afwegingen uit Hoofdstuk 6 worden manueel uitgevoerd. De STOREQ en HMSE schatters worden ondersteund door Matlab en *python* hulpscripts, resulterende uit gemeenschappelijk onderzoek samen met (en geschreven door) collega's van de Norway University of Science and Technology.

Bibliography

[1] Absar, J., Catthoor, F. and Das, K., "Call-instance based function inlining for increasing data access related optimisation opportunities", *Technical report*, IMEC, Leuven, Belgium, 2003.

[2] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F.P., Thomson, J., Toussaint, M. and Williams, C.K.I., "Using Machine Learning to Focus Iterative Optimization", *The Fourth International Symposium on Code Generation and Optimization* (CGO 2006), pp. 295–305, New York, New York, March 2006.

[3] Aho, A.V., Sethi, R. and Ulmann, J.D., "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986.

[4] Alippi, C., Fornaciari, W., Pozzi, L. and Sami, M., "A DAG-Based Design Approach for Reconfigurable VLIW Processors", *Proceedings of 2nd ACM/IEEE Design and Test in Europe Conf.* (DATE), pp. 778–780, Munich, Germany, Mar. 1999.

[5] Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J., "Conversion of control dependence to data dependence", *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, Jan. 1983.

[6] Allen, R. and Kennedy, K., "Automatic translation of Fortran programs to vector form", *ACM Trans. on Programming Languages and Systems*, Vol.9, No.4, pp.491–542, Oct. 1987.

[7] Amarashinge, S.P., Anderson, J.M., Lam, M.S. and Tseng, C.W., "The SUIF Compiler for Scalable Parallel Machines", *In Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[8] Amdahl, G., "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings*, (30), pp.483–485, 1967.

[9] Ancourt, C. and Irigoin, F., "Scanning polyhedra with do loops", *In 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.39–50, June 1991.

[10] Ancourt, C., Barthou, D., Guettier, C., Irigoin, F., Jeannet, B., Jourdan, J. and Mattioli, J., "Automatic data mapping of signal processing applications", *Proc. Intnl. Conf. on Applic.-Spec. Array Processors*, Zurich, Switzerland, pp.350–362, July 1997.

[11] Bagnara, R., Ricci, E., Zaffanella, E. and Hill, P.M., "Possibly not closed convex polyhedra and the Parma Polyhedra Library", In M.V. Hermenegildo and G. Puebla (Eds.), *Static Analysis: Proceedings of the 9th International Symposium*, Vol. 2477 of Lecture Notes in Computer Science, Madrid, Spain, pp. 213–229, Springer-Verlag, Berlin, 2002.

[12] Balasa, F., Kjeldsberg, P.G., Palkovic, M., Vandecappelle, A. and Catthoor, F., "Loop Transformation Methodologies for Array-Oriented Memory Management", *invited paper at IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2006)*, Steamboat Springs, Colorado, USA, September 2006.

[13] Balasa, F., Catthoor, F. and De Man, H., "Background memory area estimation for multidimensional signal processing systems", *IEEE Transactions on VLSI Systems*, Vol.3, No.2, pp.157–172, 1995.

[14] Balasa, F., Catthoor, F. and De Man, H., "Practical Solutions for Counting Scalars and Dependences in ATOMIUM – a Memory Management System for Multi-dimensional Signal Processing", *IEEE Trans. on Comp.-aided Design*, Vol.CAD-16, No.2, pp.133–145, Feb. 1997.

[15] Ball, T. and Larus, J.R., "Efficient Path Profiling", *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.46–57, Paris, France, 1996.

[16] Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M. and Marwedel, P., "Scratchpad memory: design alternative for cache onchip memory in embedded systems", *In CODES '02: Proceedings of the 10th international symposium on Hardware/software codesign*, ACM Press, pp. 73–78, New York, NY, USA, 2002.

[17] Banerjee, U., Eigenmann, R., Nicolau, A. and Padua, D., "Automatic program parallelisation", *Proc. of the IEEE*, invited paper, Vol.81, No.2, pp.211–243, Feb. 1993.

[18] Bastoul, C., "Generating Loops for Scanning Polyhedra: CLooG's Users's Guide", *Technical report*, http://www.prism.uvsq.fr/čedb/bastools/cloog.html, September 2002.

[19] Bastoul, C., Cohen, A., Girbal, S., Sharma, S. and Temam, O., "Putting polyhedral loop transformations to work", *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers*, LNCS 2958, pp.209–225, College Station, Sept. 2003.

[20] Bastoul, C., "Code Generation in the Polyhedral Model Is Easier Than You Think", *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, pp.7–16, Sept. 2004.

[21] Bell, E.T., "Exponential Numbers", *Amer. Math. Monthly*, Vol. 41, pp. 411-419, 1934.

[22] Benini, L. and de Micheli, G., "System-level power optimization: Techniques and tools", *ACM Transations on Design Automation of Electronic Systems*, Vol. 5, Nr. 2, pp. 115–192, Apr. 2000.

[23] Berlin, D., Edelsohn, D. and Pop, S., "High-level loop optimizations for GCC", *In Proceedings of the 2004 GCC Developers Summit*, pp.37–54, June 2004.

[24] Beyls, K., "Software Methods to Improve Data Locality and Cache Behavior", *Doctoraatsproefschrift Faculteit Toegepaste Wetenschappen*, Universiteit Gent, 2004.

[25] Brockmeyer, E., Vandecappelle, A. and Catthoor, F., "Systematic Cycle budget versus System Power Trade-off: a New Perspective on System Exploration of Real-time Data-dominated Applications", *Proc. IEEE Intnl. Symp. on Low Power Design*, Rapallo, Italy, pp.137–142, Aug. 2000.

[26] Brockmeyer, E., Miranda, M., Catthoor, F. and Corporaal, H., "Layer Assignment Techniques for Low Energy in Multi-layered Memory Organisations", *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* (DATE), Munich, Germany, pp.1070–1075, March 2003.

[27] Catthoor, F. and Brockmeyer, E., "Unified meta-flow summary for low-power data-dominated applications", chapter in "Unified low-power design flow for data-dominated multi-media and telecom applications", F.Catthoor (ed.), Kluwer Acad. Publ., Boston, pp.7–23, 2000.

[28] Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P.G., Van Achteren, T. and Omnes, T., "Data access and storage management for embedded programmable processors", ISBN 0-7923-7689-7, Kluwer Acad. Publ., Boston, 2002.

[29] Catthoor, F., "Meta model for human interaction and decision-making: meta-concepts and balloonist vs road-builder metaphore", slide set, Jan-July 2005.

[30] Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L. and De Man, H., "Global communication and memory optimizing transformations for low power signal processing systems", *IEEE Wsh. on VLSI signal processing*, La Jolla CA, Oct. 1994. Also in *VLSI Signal Processing VII*, J.Rabaey, P.Chau, J.Eldon (eds.), IEEE Press, New York, pp.178–187, 1994.

[31] Catthoor, F., Janssen, M., Nachtergaele, L. and De Man, H., "System-level data-flow transformations for power reduction in image and video processing", *Proc. Intnl. Conf. on Electronic Circuits and Systems* (ICECS), Rhodos, Greece, pp.1025–1028, Oct. 1996.

[32] Catthoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L. and Vandecappelle, A., "Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.

[33] Catthoor, F., Janssen, M., Nachtergaele, L. and De Man, H., "System-level data-flow transformation exploration and power-area trade-offs demonstrated on video codecs", special issue on "Systematic trade-off analysis in signal processing systems design" (eds. M.Ibrahim, W.Wolf) in *J. of VLSI Signal Processing*, Vol.18, No.1, Kluwer, Boston, pp.39–50, 1998.

[34] Chandrakasan, A.P., Potkonjak, M., Mehra, R., Rabaey, J. and Brodersen, R.W., "Optimizing Power Using Transformations", *IEEE Transactions on CAD*, Vol. 14, No. 1, pp. 12–31, 1995.

[35] Chen, M.K., Li, X.F., Lian, R., Lin, J.H., Liu, L., Liu, T. and Ju, R., "Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease Programming", *Proc. of the SIGPLAN'05 Conf. on Programming Language Design and Implementation*, Chicago, IL, USA, pp.224–236, 2005.

[36] Chung, E.Y., De Micheli, G. and Benini, L., "Contents provider-assisted dynamic voltage scaling for low energy multimedia applications", *Proceedings of the IEEE Intnl. Symp. on Low Power Electornic and Design (ISLPED)*, pp. 42–47, 2002.

[37] Cierniak, M. and Li, W., "Unifying Data and Control Transformations for Distributed Shared-Memory Machines", *Proc. of the SIGPLAN'95 Conf. on Programming Language Design and Implementation*, La Jolla, pp.205–217, Feb. 1995.

[38] Clauss, P. and Loechner, V., "Parametric analysis of polyhedral iteration spaces", *J. of VLSI Signal Processing*, Vol.19, pp.179–194, Kluwer, 1998.

[39] Comtet, L., "Advanced Combinatorics: The Art of Finite and Infinite Expansions", *rev. enl. ed. Dordrecht*, Netherlands: Reidel, 1974.

[40] Cooper, K.D., Subramanian, D. and Torczon, L., "Adaptive Optimizing Compilers for the 21st Century", *Journal of Supercomputing*, Vol. 23(1), pp. 7–22, 2002.

[41] Corporaal, H., "Transport Triggered Architectures", Chapter 7, *PhD thesis*, Technische Universiteit Delft, 1995.

[42] Cupak, M., Catthoor, F. and De Man, H., "Verification of loop transformations for complex data dominated applications", *Proc. Intnl. High Level Design Validation and Test Wsh.*, La Jolla, CA, pp.72–79, Nov. 1998.

[43] Danckaert, K., Catthoor, F. and De Man, H., "A preprocessing step for global loop transformations for data transfer and storage optimization", *Proc. Intnl. Conf. on Compilers, Arch. and Synth. for Emb. Sys.*, San Jose CA, pp.34–40, Nov. 2000.

[44] Danckaert, K., "Loop transformations for data transfer and storage reduction on multiprocessor systems", *PhD thesis*, Department of Electrical Engineering, Katholieke Universiteit Leuven, 2001.

[45] Darte, A., Schreiber, R. and Villard, G., "Lattice-based memory allocation", *IEEE Transaction on Computers*, Vol. 54(10), pp. 1242–1257, ISSN:0018-9340, October 2005.

[46] Darte, A. and Robert, Y., "Affine-by-statement scheduling of uniform and affine loop nests over parametric domains", *Journal of Parallel and Distributed Computing*, Vol.29(1), pp.43–59, 1995.

[47] Dasygenis, M., Brockmeyer, E., Durinck, B., Catthoor, F., Soudris, D. and Thanailakis, A., " A Memory Hierarchical Layer Assigning and Prefetching Technique to Overcome the Memory Performance/Energy Bottleneck", *Proc. 8th ACM/IEEE Design and Test in Europe Conf.* (DATE), Munich, Germany, pp.946–947, March 2005.

[48] Davidson, R. and Harel, D., "Drawing Graphs Nicely Using Simulated Annealing", *ACM Transactions on Graphics*, Vol. 15(4), pp. 301–331, 1996.

[49] De Greef, E., Catthoor, F. and De Man, H., "Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications", special issue on "Parallel Processing and Multi-media" (ed. A.Krikelis), in *Parallel Computing* Elsevier, Vol.23, No.12, Dec. 1997.

[50] De Greef, E., Catthoor, F. and De Man, H., "Program transformation strategies for reduced power and memory size in pseudo–regular multimedia applications", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol.8, No.6, pp.719-733, Oct. 1998.

[51] De Greef, E., "Storage size reduction for multimedia applications", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, Jan. 1998.

[52] *http://www.date-conference.com/conference/2002/keynotes/deman/deman_slides.pdf*

[53] De Man, H., Catthoor, F., Goossens, G., Vanhoof, J., Van Meerbergen, J., Note, S. and Huisken, J., "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proc. of the IEEE*, special issue on "The future of computer-aided Design", Vol.78, No.2, pp.319–335, Feb. 1990.

[54] Dempster, A.P., Laird, N.M. and Rubin, D.B., "Maximum Likelihood from Incomplete Data via the EM algorithm", *Journal of the Royal Statistical Society*, Series B, Vol. 39(1), pp. 1–38, 1977.

[55] Desmet, V., Vandierendonck, H. and De Bosschere, K., "2FAR: A 2bcgskew Predictor Fused by an Alloyed Redundant History Skewed Perceptron Branch Predictor", *Journal of Instruction-Level Parallelism*, Vol. 7, pp. 1–11, 2005.

[56] Dezan, C., Le Verge, H., Quinton, P. and Saouter, Y., "The Alpha du CENTAUR experiment", in *Algorithms and parallel VLSI architectures II*, P.Quinton and Y.Robert (eds.), Elsevier, Amsterdam, pp.325–334, 1992.

[57] Dick, R.P., Rhodes, D.L. and Wolf, W., "TGFF Task Graphs for Free", *Proc. of the 6th International Workshop on Hardware/Software Co-design (CODES/CASHE)*, pp.97–101, March 1998.

[58] Diguet, J.P., Wuytack, S., Catthoor, F. and De Man, H., "Formalized methodology for data reuse exploration in hierarchical memory mappings", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey CA, pp.30–35, Aug. 1997.

[59] Dunn, J.C., "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters", *Journal of Cybernetics*, Vol. 3, pp. 32–57, 1973.

[60] Ellervee, P., Miranda, M., Catthoor, F. and Hemani, A., "System-level data format exploration for dynamically allocated data structures", *IEEE Trans. on Computer-aided design*, Vol.20, No.12, Dec. 2001.

[61] Ellmenreich, N., Faber, P., Griebl, M., Günz, R., Keimer, H., Meisl, W., Wetzel, S., Wieninger, Ch. and Wüst, A., "LooPo - Loop Parallelization in the Polytope Model", *Technical report*, Uni. Passau, December 2001.

[62] Ellis, J, "Bulldog: A Compiler for VLIW Architectures", The MIT Press, Cambridge, MA, 1985.

[63] van Engelen, R.A. and Gallivan, K.A., "An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications", *Proceedings of the 2001 International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA 2001)*, Maui, Hawaii, pp.80–89, Jan. 2001.

[64] Eppstein, D., "Parallel Recognition of Series-Parallel Graphs", *Information and Computation*, Vol. 98(1), pp. 41–55, May 1992.

[65] Evans, R. and Franzon, P., "Energy consumption modeling and optimization for SRAMs", *IEEE Journal of Solid-state Circuits*, Vol.SC-30, No.5, pp.571–579, May 1995.

[66] Falk, H. and Marwedel, P., "Control Flow Optimization by Loop Nest Splitting at the Source Code Level", *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* (DATE), Munich, Germany, pp.410–415, March 2003.

[67] Falk, H., "Source code optimisation techniques for data-flow dominated embedded software", *Doctoral dissertation*, Univ. of Dortmund, 2004.

[68] Feautrier, P., Collard, J.F. and Bastoul, C., "Solving systems of affine (in)equalities", *Technical Report*, PRiSM, Versailles University, 2002.

[69] Feautrier, P., "Array expansion", *Proceedings of the 2nd International Conference on Supercomputing (ICS'88)*, pp.429–441, ACM Press, 1988.

[70] Feautrier, P., "Parametric integer programming", *RAIRO Recherche Oprationnelle*, Vol. 22(3), pp. 243–268, 1988.

[71] Feautrier, P., "Some efficient solutions to the affine scheduling problem. Part II. multidimensional time", *International Journal of Parallel Programming*, Vol. 21(6), pp.389–420, 1992.

[72] Feautrier, P., "Automatic parallelization in the polytope model", *Lecture Notes in Computer Science*, Vol. 1132, pp. 79–103, 1996.

[73] Fisher, J.A., "Trace scheduling: a technique for global microcode compaction", *IEEE Trans. on Computers*, Vol.C-30, No.7, pp.478–490, July 1981.

[74] Fowler, M., "UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition", Chapter 9 (Use cases), pp.99–106, Addison Wesley Publishing Company, Reading, MA, 2003.

[75] Franke, B. and O'Boyle, M., "Array Recovery and High-Level Transformations for DSP Applications", *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 2, Issue 2, pp.132–162, May 2003.

[76] Franssen, F., Nachtergaele, L., Samsom, H., Catthoor, F. and De Man, H., "Control flow optimization for fast system simulation and storage minimization", *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, Paris, France, pp.20–24, Feb. 1994.

[77] Fraboulet, A., Huard, G. and Mignotte, A., "Loop alignment for memory access optimisation", *Proc. 12th ACM/IEEE Intnl. Symp. on System-Level Synthesis* (ISSS), San Jose CA, pp.71–70, Dec. 1999.

[78] Fruchterman, T. and Reingold, E., "Graph Drawing by Force-Directed Placement", *Software–Practice & Experience*, Vol. 21(11), pp. 1129–1164, 1991.

[79] Galuzzi, C., Panainte, E.M., Yankova, Y., Bertels, K., Vassiliadis, S., "Automatic selection of application-specific instruction-set extensions", *Proceedings of the 4th Intnl. Conf. on International Conference on Hardware Software Codesign (CODES'06)*, pp. 160–165, Seoul, Korea, 2006.

[80] Ghez, C., Miranda, M., Vandecappelle, A., Catthoor, F., Verkest, D., "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Lafayette LA, IEEE Press, pp.623–632, Oct. 2000.

[81] Gheorghita, V.S., Stuijk, S., Basten, T. and Corporaal, H., "Automatic Scenario Detection for Improved WCET Estimation", *42nd Design Automation Conference* (DAC), pp. 101–104, Anaheim, CA, June 2005.

[82] Gheorghita, S.V., Basten, T. and Corporaal, H., "Intra-task Scenario-aware Voltage Scheduling", *Proc. of the Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2005)*, pp. 177–184, San Francisco, CA, Sep. 2005.

[83] Gheorghita, S.V., Corporaal, H. and Basten, T., "Iterative Compilation for Energy Reduction", *Journal of Embedded Computing*, pp. 509–520, Vol. 1, Issue 4, Dec. 2005.

[84] Gheorghita, S.V., Basten, T. and Corporaal, H., "Profiling Driven Scenario Detection and Prediction for Multimedia Applications", *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (IC-SAMOS), pp. 63–70, Samos, Greece, July 2006.

[85] Gheorghita, S.V., Basten, T. and Corporaal, H., "Application Scenarios in Streaming-Oriented Embedded System Design", *Proc. of the International Symposium in System-on-Chip (SoC 2006)*, pp. 175–178, Tampere, Finland, Nov. 2006.

[86] Gheorghita, S.V., Basten, T. and Corporaal, H., "Scenario Selection and Prediction for DVS-Aware Scheduling of Multimedia Applications", *Journal of VLSI Signal Processing Systems*, To appear, http://dx.doi.org/10.1007/s11265-007-0086-1, 2007.

[87] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M. and Temam, O., "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies", Intl. J. of Parallel Programming, Vol. 34(3), pp. 261–317, Special issue on Microgrids. June 2006.

[88] Gregg, D., " Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling", *Proceeding of 10th International Conference on Compiler Construction (CC 2001)*, held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), pp. 200, Genova, Italy, Apr. 2001.

[89] Griebl, M. and Lengauer, C., "On the Space-time Mapping of WHILE-Loops", *Parallel Processing Letters Journal (PPL)*, Vol.4, Nr.3, pp.221–232, Sept. 1994.

[90] Griebl, M. and Collard, J.F., "Generation of Synchronous Code for Automatic Parallelization of while Loops", *European Conference on Parallel Processing (Euro-Par) 1995*, pp. 315–326, 1995.

[91] Griebl, M., Lengauer, C. and Wetzel, S., "Code Generation in the Polytope Model", *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pp.106–111, 1998.

[92] Grünbaum, B., *Convex Polytopes*, Interscience/Wiley, London, 1967.

[93] Grun, P., Balasa, F. and Dutt, N., "Memory Size Estimation for Multimedia Applications", *Proceedings of the 6th Int. Workshop on HW/SW Codesign (CODES/CACHE)*, pp. 145–149, 1998.

[94] Gupta, S., Miranda, M., Catthoor, F. and Gupta, R., "Analysis of high-level address code transformations for programmable processors", *Proc. 3rd ACM/IEEE Design and Test in Europe Conf.* (DATE), Paris, France, pp.9–13, April 2000.

[95] Gutberlet, P. and Rosentiel, W., "Specification of interface components for synchronous data paths", *Proceedings of the International Workshop on High-Level Synthesis*, pp. 134–139, 1993.

[96] Halstead, M.H., "Elements of Software Science", *Operating and Programming Systems Series*, Vol. 7, Elsevier, New York, 1977.

[97] Haneda, M., "Statistical compiler tuning", *PhD thesis*, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University, 2006.

[98] Hecht, M.S. and Ullman, J.D., "Characterization of Reducible Flow Graphs", *Journal of the Association for Computing Machinery*, Vol. 21, No. 3, pp.167–175, July 1974.

[99] Hennessy, J.L. and Patterson, D.A., "Computer-Architecture: A Quantitative Approach", *Morgan Kaufmann Publishers*, Third Edition, 2003.

[100] Hipp, M., et al., "MPG123 MP3 Decoder Source Code", http://www.mpg123.de, April 2001.

[101] Hoogerbrugge, J. and Augusteijn, A., "Instruction Scheduling for Trimedia", *In Journal of Instruction-Level Parallelism*, Vol. 1, Issue 1, Feb. 1999.

[102] Hoste, K. and Eeckhout, L., "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics", *Proceedings of IEEE International Symposium on Workload Characterization (IIWWC 2006)*, pp. 83–92, San Jose, CA, October 2006.

[103] Hsu, P.Y.T. and Davidson, E.S., "Highly Concurrent Scalar Processing", *In Proceedings of the 13th International Symposium on Computer Architecture*, 14(2), pp.386–395, Tokyo, Japan, June 1986.

[104] Hu, Q., Palkovic, M. and Kjeldsberg, P.G., "Memory Requirement Optimization with Loop Fusion and Loop Shifting", *Proceedings of 30th Euromicro conference*, pp. 272–278, Rennes, France, Aug. 2004.

[105] Hu, Q., Brockmeyer, E., Palkovic, M., Kjeldsberg, P.G. and Catthoor, F., "Memory Hierarchy Usage Estimation for Global Loop Transformations", *IEEE NORCHIP Conference (NORCHIP 2004)*, pp. 301–304, Oslo, Norway, Nov. 2004.

[106] Hu, Q., Vandecappelle, A., Palkovic, M., Kjeldsberg, P.G., Brockmeyer, E. and Catthoor, F., "Hierarchical Memory Size Estimation for Loop Fusion and Loop Shifting of Data Dominated Applications", *Proceedings of the 11th Asia and South Pacific Design Automation Conference (ASP-DAC 2006)*, pp. 606–611, Yokohama City, Japan, January 2006.

[107] Hu, Q., Kjeldsberg, P.G., Palkovic, M., Vandecappelle, A. and Catthoor, F., "Incremental Hierarchical Memory Size Estimation for Steering of Loop Transformations", *The ACM Transactions on Design Automation of Electronic Systems (TODAES)*, (accepted) 2007.

[108] Hu, Q., Vandecappelle, A., Kjeldsberg, P.G., Catthoor, F. and Palkovic, M., "Fast Memory Footprint Estimation based on Dependency Distance", *Proc. 10th ACM/IEEE Design and Test in Europe Conf.* (DATE), Nice, France, pp.1–6, April 2007.

[109] Hu, Q., "Hierarchical Memory Size Estimation for Loop Transformation and Data Memory Platform Exploration", *Doctoral dissertation*, Faculty of Information Technology, Mathematics and Electrical Engineering, Norway University of Science and Technology, Norway, March 2007.

[110] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Water, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G. and Lavery, D.M., "The superblock: An effective structure for VLIW and superscalar compilation", *Journal of Supercomputing*, pp. 229–248, Jan. 1993.

[111] Janssen, J.M., Catthoor, F. and De Man, H., "A Specification Invariant Technique for Operation Cost Minimisation in Flow-graphs". *Proc. 7th ACM/IEEE Intnl. Symp. on High-Level Synthesis*, Niagara-on-the-Lake, Canada, pp.146–151, May 1994.

[112] Jerraya, A., Park, I. and O'Brien, K., "Amical: An interactive high-level synthesis environment", *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 58–62, 1993.

[113] Johnson, S.C., "Hierarchical Clustering Schemes", *Psychometrika*, Vol. 2, pp. 241–254, 1967.

[114] Jung, B., Jeong, Y. and Burleson, W.P., "Distributed Control Synthesis for Data-Dependent Iterative Algorithms", *Proc. on IEEE Int. Conf. on Application Specific Array Processors*, San Francisco, pp. 57–68, August 1994.

[115] Kandemir, M.T., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I. and Parikh, A., "Dynamic Management of Scratch-Pad Memory Space", *In Proceedings of Design Automation Conference*, pp. 690-695, Las Vegas, Nevada, September 2001.

[116] Kandemir, M., Ramanujam, J., Choudhary, A. and Banerjee, P., "A layout-conscious iteration space transformation technique", *IEEE Transactions on Computers*, 50(12):1321–1335, 2001.

[117] Karkowski, I. and Corporaal, H., "Overcoming the Limitations of the Traditional Loop Parallelization", *Journal of Future Generation Computer Systems (FCGS)*, Vol. 13(4-5), pp. 407–416, Elsevier Science, 1998.

[118] Kelly, W. and Pugh, W., "A framework for unifying reordering transformations", *Technical Report CS-TR-3193*, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[119] Kelly, W., Pugh, W. and Rosser, E., "Code Generation for Multiple Mappings", *Frontiers'95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995.

[120] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T. and Wonnacott, D., "New user inteface for petit and other interfaces: user guide", *Technical report*, University of Maryland, December 1996.

[121] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T. and Wonnacott, D., "The Omega calculator and library", *Technical report*, University of Maryland, November 1996.

[122] Kjeldsberg, P.G., "Storage Requirement Estimation and Optimization for Data Intensive Applications", *PhD thesis*, Norwegian university of Science and Technology, 2001.

[123] Kjeldsberg, P.G., Palkovic, M., Catthoor, F. and Aas, E.J., "STOREQ tool for STOrage Requirement estimation and optimization of data intensive applications", Univ. booth demonstration, *5th ACM/IEEE Design and Test in Europe Conf.* (DATE), pp.256, Paris, France, April 2002.

[124] Knijnenburg, P.M.W., Kisuki, T. and O'Boyle, M.F.P., "Iterative Compilation", *Embedded Processor Design Challenges: System Architecture, Modeling and Simulation (SAMOS)*, Springer Lecture Notes in Computer Science, Vol. 2268, pp. 171–187, 2002.

[125] Knijnenburg, P.M.W., Kisuki, T. and O'Boyle, M.F.P., "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation", *Journal of Supercomputing*, Vol. 24(1), pp. 43–67, 2003.

[126] Kountouris, A.A. and Wolinski, Ch., "Hierarchical Conditional Dependency Graphs as a Unifying Design Representation in the CODESIS High-Level Synthesis System", *Proc. of 13th Intnl. Symposium on System Synthesis (ISSS'00)*, Madrid, Spain, pp.66–71, Sept. 2000.

[127] Kreisel, G. and Krivine, J.L., *Elements of mathematical logic*, North-Holland, Amsterdam, 1967.

[128] Kulkarni, P.A., Whalley, D.B., Tyson, G.S. and Davidson, J.W., "In search of near-optimal optimization phase orderings", *Proceedings of the 2006 LCTES Conference*, ACM SIGPLAN Notices, Vol. 41, Issue 7, pp. 83–92 , July 2006.

[129] Kulkarni, C., Moolenaar, D., Nachtergaele, L., Catthoor, F. and De Man, H., "System level energy-delay exploration for multimedia applications on embedded cores with hardware caches", *Journal of VLSI signal processing*, Vol. 22, No. 1, pp. 45–58, August 1999.

[130] Kulkarni, C., "Cache optimization for multimedia applications", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, Feb. 2001.

[131] *http://www.kurzweilai.net/articles/art0134.html?printable=1*

[132] Lagerström, K., "Design and Implementation of an MP3 Decoder", M.Sc. thesis, Chalmers University of Technology, Sweden, http://www.kmlager.com/mp3/, May 2001.

[133] Lamport, L., "The parallel execution of DO loops", *Communications of the ACM*, Vol. 17(2), pp. 83–93, Feb. 1974.

[134] Lengauer, C., "Loop Parallelization in the Polytope Model", *Proc. of the 4th International Conference on Concurrency Theory (CONCUR'93)*, in Lecture Notes in Computer Science 715, Springer-Verlag, pp.398–416, 1993.

[135] Leupers, R., "Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools", Kluwer Academic Publishers, ISBN 0-7923-7989-6, November 2000.

[136] Li, W. and Pingali, K., "A singular loop transformation framework based on non-singular matrices", *Proc. 5th Annual Wsh. on Languages and Compilers for Parallelism*, New Haven CN, Aug. 1992.

[137] Loveman, D.B., "Program improvement by source-to-source transformation", *Journal of the ACM*, Vol.24, No.1, pp.121-145, 1977.

[138] MacQueen, J.B., "Some Methods for classification and Analysis of Multivariate Observations", *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, Berkeley, University of California Press, 1967.

[139] Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A., "Effective Compiler Support for Predicated Execution Using the Hyperblock", *Proc. of the 25th International Symposium on Microarchitecture*, pp.45–54, Dec. 1992.

[140] Mahlke, S.A., "Exploiting Instruction Level Parallelism in the Presence of Conditional Branches", *PhD thesis*, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[141] Manjikian, N. and Abdelrahman, T., "Fusion of loops for parallelism and locality", Comp. Systems Res. Inst. Univ. of Toronto, technical report CSRI-315, Canada, Feb. 1995.

[142] Matsumoto, M. and Nishimura, T., "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", *ACM Trans. on Modeling and Computer Simulation (TOMACS)*, Vol. 8(1), pp.3–30, 1998.

[143] McCabe, T., "A Software Complexity Measure", *IEEE Trans. on Software Engineering*, Vol. 2(4), pp. 308–320, Dec. 1976.

[144] McKee, S.A., "Reflections on the Memory Wall", *Proc. Computing Frontiers (CF'04)* (invited paper), Ischia, IT, April 2004.

[145] McKinley, K., Carr, S. and Tseng, C-W., "Improving data locality with loop transformations", *ACM Trans. on Programming Languages and Systems*, Vol.18, No.4, pp.424–453, July 1996.

[146] Mei, B., Vernalde, S., Verkest, D., De Man, H. and Lauwereins, R., "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained configurable matrix", *Proc. IEEE Conf. on Field-Programmable Logic and its Applications (FPL)*, Lisbon, Portugal, pp. 61–70, Sep. 2003.

[147] Mei, B., "A Coarse-Grained Reconfigurable Architecture Template and Its Compilation Techniques", *PhD. thesis*, Department of Electrical Engineering, Katholieke Universiteit Leuven, 2005.

[148] Meng, T.H., Gordon, B., Tsern, E. and Hung, A., "Portable video-on-demand in wireless communication", special issue on "Low power electronics" of the *Proc. of the IEEE*, Vol.83, No.4, pp.659–680, April 1995.

[149] Merill, J., "GENERIC and GIMPLE: a new tree representation for entire functions", *In Proceedings of the 2003 GCC Developers Summit*, pp.171–180, May 2003.

[150] Milidonis, A., Dimitroulakos, G., Galanis, M.D., Theodoridis, G., Goutis, C. and Catthoor, F., "An Automated C++ Code and Data Partitioning Framework for Data Management of Data-Intensive Applications" *Proc. of 8th Intnl. Wsh. on Software and Compilers for Embedded Systems (SCOPES)*, pp.122–136, Oct. 2004.

[151] Miranker, W. and Winkler, A., "Space-time representation of computational structures", *Computing*, pp. 93–114, 1984.

[152] Miranda, M., Catthoor, F., Janssen, M. and De Man, H., "High-level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.677–686, Dec. 1998.

[153] Moldovan, D., "On the design of algorithms for VLSI systolic arrays", *Proc. of the IEEE*, Vol.71, No.1, pp. 113–120, Jan. 1983.

[154] Muchnick, S., "Advanced compiler design and implementation", *Morgan Kaufmann Publishers Inc.* , ISBN 1-55860-320-4, 1997.

[155] Novillo, D., "Tree SSA - a new optimization infrastructure for GCC", *In Proceedings of the 2003 GCC Developers Summit*, pp.181–193, May 2003.

[156] Olsen, R. and Gao, G., "Collective analysis and transformation of loop clusters", *technical report*, ACAPS Technical Memo 24, McGill University, April 1992.

[157] Omnes, T., "Acropolis: un precompilateur de specification pour l'exploration du transfert et du stockage des donnees en conception de systemes embarques a haut debit", *Doctoral Dissertation*, Ecole des Mines de Paris, May 2001.

[158] Palkovic, M., Miranda, M., Catthoor, F. and Verkest, D., "High-level condition expression transformations for design exploration and improved synthesis of an arithmetic coder", *3rd Workshop on System Design Automation (SDA 2000)*, Rathen, Germany, pp.53–59, March 2000.

[159] Palkovic, M., Miranda, M., Catthoor, F. and Verkest, D., "High-level condition expression transformations for design exploration", *SystemDesign Automation – Fundamentals, Principles, Methods, Examples*, R.Merker and W.Schwarz (eds.), Kluwer Acad. Publ., Boston, pp. 56–64, 2001.

[160] Palkovic, M., Miranda, M., Denolf, K., Vos, P. and Catthoor, F., "Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG4 Video Decoder", *IEEE 15th Intnl. Conf. on VLSI Design and 7th ASP-DAC*, Bangalore, India, pp.547–552, January 2002.

[161] Palkovic, M., Miranda, M. and Catthoor, F., "Systematic Power-Performance Trade-Off in MPEG4 by means of Selective Function Inlining steered by Address Optimisation Opportunities", *In Proc. 5th ACM/IEEE Design and Test in Europe Conf. (DATE)*, pp.1072–1077, Paris, France, March 2002.

[162] Palkovic, M., Brockmeyer, E., Corporaal, H., Catthoor, F. and Vounckx, J., "Hierarchical rewriting and hiding of data dependent conditions to enable global loop transformations", *Proc. 2nd Wsh. on Optim. for DSP and Embedded Systems (ODES)*, Palo Alto CA, March 2004.

[163] Palkovic, M., Corporaal, H. and Catthoor, F., "Global Memory Optimisation for Embedded Systems allowed by Code Duplication", *Proc. of 9th Intnl. Wsh. on Software and Compilers for Embedded Systems (SCOPES)*, pp.72–80, Dallas, TX, Sep. 2005.

[164] Palkovic, M., Brockmeyer, E., Vanbroekhoven, P., Corporaal, H. and Catthoor, F., "Systematic Preprocessing of Data Dependent Constructs for Embedded Systems", *Proc. IEEE Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Leuven, Belgium, Lecture Notes Comp. Sc., Springer-Verlag, Vol.3728, pp.89–90, Sep. 2005.

[165] Palkovic, M., Brockmeyer, E., Vanbroekhoven, P., Corporaal H. and Catthoor, F., "Systematic Preprocessing of Data Dependent Constructs for Embedded Systems", *Journal of Low Power Electronics (JOLPE)*, American Scientific Publ., Vol.2, No.1, pp.9–17, April 2006.

[166] Palkovic, M., Corporaal, H. and Catthoor, F., "Dealing with data dependent conditions to enable general global source code transformations", *International Journal of Embedded Systems (IJES)*, Inderscience Publ., in press, 2007.

[167] Palkovic, M., Catthoor, F. and Corporaal, H., "Dealing with Variable Trip Count Loops in System Level Exploration", *Proc. 4th Wsh. on Optim. for DSP and Embedded Systems (ODES)*, pp. 19–28, Manhattan NY, March 2006.

[168] Palkovic, M., Corporaal, H. and Catthoor, F., "Trade-offs in Global Loop Transformations", *The ACM Transactions on Design Automation of Electronic Systems (TODAES)*, (submitted) 2007.

[169] Palkovic, M., Corporaal, H. and Catthoor, F., "Heuristics for Scenario Creation to Enable General Loop Transformations", *Proc. IEEE International Symposium on System-on-Chip (SoC)*, (accepted) Tampere, Finland, pp.–, Nov. 2007.

[170] Panda, P.R., Dutt, N.D. and Nicolau, A., "Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration", Kluwer Academic Publishers, ISBN0792383621, 1998.

[171] Pareto, V., "Cours D'Economie Politique", volume I–II, Lausanne, 1896.

[172] Pedram, M., "Power Optimization and Management in Embedded Systems", *Proceedings of the Asia South Pacific Design Automation Conference*, pp. 239–244, Yokohama, Japan, January 2001.

[173] Polychronopoulos, C., "Compiler optimizations for enhancing parallelism and their impact on the architecture design", *IEEE Trans. on Computers*, Vol.37, No.8, pp.991–1004, Aug. 1988.

[174] Pop, S., Cohen, A., Bastoul, C., Girbal, S., Jouvelot, P., Silber, G.A. and Vasilache, N., "Graphite: Loop optimizations based on the polyhedral model for GCC", *4th GCC Developper's Summit*, Ottawa, Canada, June 2006.

[175] Pouchet, L.N., Bastoul, C., Cohen, A. and Vasilache, N., "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time", *The Fifth International Symposium on Code Generation and Optimization* (CGO 2007), pp. 144–156, San Jose, California, March 2007.

[176] Pugh, W., "The Omega Test: a fast and practical integer programming algorithm for dependence analysis", *Communications of the. ACM*, Vol.35, No.8, Aug. 1992.

[177] Qin, W. and Malik, S., "Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation", *Proceedings of 2003 Design Automation and Test in Europe Conference (DATE 03)*, pp.556–561, Mar. 2003.

[178] Quilleré, F., Rajopadhye, S. and Wilde, D., "Generation of Efficient Nested Loops from Polyhedra", *International Journal of Parallel Programming*, Vol. 28, No. 5, Oct. 2000.

[179] Quinton, P., "Automatic synthesis of systolic arrays from recurrent uniform equations", *11th Intnl. Symp. Computer Architecture*, Ann Arbor, pp. 208–214, June 1984.

[180] Rydland, P., Palkovic, M., Kjeldsberg, P.G., Brockmeyer, E. and Catthoor, F., "Inter in-place storage size requirement estimation", *Proceedings of IEEE NORCHIP Conference (NORCHIP 2003)*, pp. 240–243, Riga, Latvia, November 2003.

[181] Samsom, H., Franssen, F., Catthoor, F. and De Man, H., "System-level Verification of Video and Image Processing Specifications", *Proc. 8th ACM/IEEE Intnl. Symp. on System-Level Synthesis* (ISSS), Cannes, France, pp.144–149, Sep. 1995.

[182] Sarkar, V. and Gao, G.R., "Optimization of array accesses by collective loop transformations", *Proceedings of the 5th International Conference on Supercomputing*, Cologne, West Germany, pp. 194–205, 1991.

[183] Schrijver, A., *Theory of Linear and Integer Programming*, John Wiley & Sons, 1986.

[184] Semeria, L. and De Micheli, G., "SpC: synthesis of pointers in C", *Proc. IEEE Intnl. Conf. on Comp. Aided Design*, Santa Clara CA, pp.340–346, Nov. 1998.

[185] Shashidar, K.C., Vandecappelle, A. and Catthoor, F., "Low Power Design of Turbo Decoder Module with Exploration of Energy-Performance Trade-offs", *Wsh. on Compilers and Operating Systems for Low Power (COLP'01)* in conjunction with *Intnl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, Barcelona, Spain, pp. 10.1–10.6, Sep. 2001.

[186] Shashidar, K.C., Bruynooghe, M., Catthoor, F. and Janssens, G., "Verification of source code transformations by program equivalence checking", *Proc. 14th Intnl. Conf. on Compiler Construction (CC)*, Edinburgh, Scotland, *Lecture Notes in Computer Science*, Vol.3443, pp.221–236, April 2005.

[187] Shostak, R.E., "On the SUP-INF method for proving Presburger formulas", *Journal of the ACM*, Vol. 24(4), pp. 529–543, Oct. 1977.

[188] Siek, J.G., Lee, L.Q. and Lumsdaine, A., "The Boost Graph Library User Guide and Reference Manual", Addison-Wesley, ISBN-0201729148, 2001.

[189] Sjodin, J., Froderberg, B. and Lindgren, T., "Allocation of Global Data Objects in On-Chip RAM", *In ACM Proc. Workshop on Compiler and Architectural Support for Embedded Computer Systems*, Washington DC, Dec. 1998.

[190] Slock, P., Wuytack, S., Catthoor, F. and De Jong, G. "Fast and extensive system-level memory exploration for ATM applications", *Proc. 10th ACM/IEEE Intnl. Symp. on System-Level Synthesis* (ISSS), Antwerp, Belgium, pp.74–81, Sep. 1997.

[191] Song, Y., Xu, R., Wang, Ch. and Li, Z., "Data locality enhancement by memory reduction", *International Conference on Supercomputing*, pp. 50–64, 2001.

[192] Stroustrup, B., *The C++ Programming Language (3rd edition)*, Addison-Wesley Pub. Co., ISBN 0-201-70073-5, Feb. 2000.

[193] Strobach, P., "QSDPCM – A New Technique in Scene Adaptive Coding," *Proc. 4th Eur. Signal Processing Conf.*, EUSIPCO-88, Grenoble, France, Elsevier Publ., Amsterdam, pp.1141–1144, Sep. 1988.

[194] Stuijk, S., Geilen, M. and Basten, T., "SDF3: SDF For Free", *In Proceedings of the 6th IEEE International Conference on Application of Concurrency to System Design* (ACSD 2006), pp. 276–278, June 2006.

[195] Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V. and Stuijk, S., "A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis", *In Proceedings of the 4th ACM & IEEE Conference on Formal Methods and Models in CoDesign* (MEMOCODE 2006), pp. 185–194, Napa Valley, California, July 2006.

[196] Thörnberg, B., Palkovic, M., Hu, Q., Olsson, L., Kjeldsberg, P.G., O'Nils, M. and Catthoor, F., "Bit-Width Constrained Memory Hierarchy Optimization for Real-Time Video Systems", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 26, Issue 4, pp.781-800, April 2007.

[197] Thörnberg, B., Hu, Q., Palkovic, M., O'Nils, M. and Kjeldsberg, P.G., "Polyhedral space generation and memory estimation from interface and memory models of real-time video systems", *Journal of Systems and Software*, Vol. 79(2), pp. 231–245, 2006.

[198] Tiwari, V., Malik, S., Wolfe, A. and Lee, M., "Instruction level power analysis and optimization of software", *Journal of VLSI Signal Processing*, pp. 1–18, Kluwer Academic Publishing, Boston, 1996.

[199] Touati, S.A.A. and Barthou, D., "On the decidability of phase ordering problem in optimizing compilation", *Proc. Conf. on Computing Frontiers*, pp. 147–156, Ischia, Italy, May 2006.

[200] Van Achteren, T., Deconinck, G., Catthoor, F. and Lauwereins, R., "Data reuse exploration methodology for loop-dominated applications", *Proc. 5th ACM/IEEE Design and Test in Europe Conf.* (DATE), Paris, France, pp.428–435, April 2002.

[201] Van Achteren, T., "Data reuse exploration techniques for multimedia applications", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, Sep. 2004.

[202] van Meeuwen, T., "Data-cache Conflict-miss Reduction by High-level Data-layout Transformations", *M.Sc. Thesis*, TUE, Eindhoven, 27 August 2002.

[203] van Swaaij, M., Franssen, F., Catthoor, F. and De Man, H., "Modelling data and control flow for high-level memory management", *Proc. 3rd ACM/IEEE Europ. Design Automation Conf.*, Brussels, Belgium, pp.8–13, March 1992.

[204] van Swaaij, M., Franssen, F., Catthoor, F. and De Man, H., "High-level modelling of data and control flow for signal processing systems", in *Design Methodologies for VLSI DSP Architectures and Applications*, M.Bayoumi (ed.), Kluwer, Boston, pp.219–259, 1994.

[205] Vahid, F., "Procedure exlining: a transformation for improved system and behavioral synthesis", *Proceedings of the 8th International Symposium on Systems Synthesis (ISSS)*, pp. 84–89, Cannes, France, 1995.

[206] Vahid, F., "Procedure exlining: a new system-level specification transformation", *European Design Automation Conference with EURO-VHDL '95 (EURODAC)*, pp. 508, 1995.

[207] Vallerio, K., "Task Graphs for Free (TGFF v3.0)", *User's manual*, August 2003.

[208] Vandeputte, F., Eeckhout, L. and De Bosschere, K., "A Detailed Study on Phase Predictors", *EuroPar 2005 Conference (Europar'2005)*, Lecture Notes in Computer Science ,Springer-Verlag, pp.571–581, Lisbon, Portugal, 2005.

[209] Vanbroekhoven, P., Janssens, G., Bruynooghe, M., Corporaal, H. and Catthoor, F., "Advanced copy propagation for arrays", *Proc. of the SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems* (LCTES'03), San Diego CA, pp.24–33, June 2003.

[210] Vander Aa, T., Corporaal, H., Catthoor, F. and Deconinck, G., "Combining data and instruction memory energy optimizations for embedded applications", *Proc. of the 3rd workshop on Embedded Systems for Real-Time Multimedia*, New York, USA, pp.121–126, Sept. 2005.

[211] Verdoolaege, S., Catthoor, F., Bruynooghe, M. and Janssens, G., "Multi-dimensional Incremental Loop Fusion for Data Locality", *IEEE 14th Intnl. Conf. on Application-specific Systems, Architectures and Processors*, Hague, The Netherlands, June 2003.

[212] Verma, M., Steinke, S. and Marwedel, P., "Data Partitioning for Maximal Scratchpad Usage", *In Proceedings of Asia South Pacific Design Automated Conference*, pp. 77–83, Kitakyushu, Japan, January 2003.

[213] Verdoolaege, S., "Incremental loop transformations and enumeration of parametric sets" *PhD thesis*, Department of Computer Science, Katholieke Universiteit Leuven, 2005.

[214] Verbauwhede, I.M., Scheers, C.J. and Rabaey, J.M., "Memory Estimation for High Level Synthesis", *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 343–348, 1994.

[215] Wilkes, M., "The memory gap", *27th Annual Intnl. Symp. on Computer Architecture*, Keynote speech at Wsh. on "Solving the Memory Wall Problem", Vancouver BC, Canada, June 2000.

[216] Wilkes, M.V., "The best way to design an automatic calculating machine", *Manchester Univ. Computer Inaugural Conf.*, pp.16-18, 1951.

[217] Wilde, D., "A Library for Doing Polyhedral Operations", M.Sc. thesis, Oregon State Univ., Dec. 1993. In co-operation with IRISA/INRIA, Rennes, France.

[218] Wolf, M. and Lam, M., "A data locality optimizing algorithm", *Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation*, Toronto, Canada, pp.30–43, June 1991.

[219] Wolfe, M., "High Performance Compilers for Parallel Computing", Addison-Wesley Publishing Co., ISBN: 0-8053-2730-4, 2006.

[220] Wulf, W.A. and McKee, S.A., "Hitting the memory wall: Implications of the obvious", *ACM SIGARCH Computer Architecture News*, Vol. 23, pp. 20–24, March 1995.

[221] Wuytack, S., Catthoor, F., Nachtergaele, L. and De Man, H., "Power Exploration for Data Dominated Video Applications", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey CA, pp.359–364, Aug. 1996.

[222] Wuytack, S., Diguet, J.P., Catthoor, F. and De Man, H., "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.529–537, Dec. 1998.

[223] Wuytack, S., Catthoor, F., De Jong, G. and De Man, H., "Minimizing the Required Memory Bandwidth in VLSI System Realizations", *IEEE Trans. on VLSI Systems*, Vol.7, No.4, pp.433–441, Dec. 1999.

[224] Yang, P., Wong, C., Marchal, P., Catthoor, F., Desmet, D., Verkest, D. and Lauwereins, R., "Energy-Aware Runtime Scheduling for Embedded Multi-processor SOCs", *IEEE Design and Test of Computers*, special issue on "Application-specific multi-processor mapping", Vol.18, No.5, pp.46–58, Sep. 2001.

[225] Yang, P., Marchal, P., Wong, C., Himpe, S., Catthoor, F., David, P., Vounckx, J. and Lauwereins, R., "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems", invited paper in *Proc. 15th ACM/IEEE Intnl. Symp. on System-Level Synthesis* (ISSS), Kyoto, Japan, pp.112–119, Oct. 2002.

[226] Ykman-Couvreur, Ch., Nollet, V., Marescaux, Th., Brockmeyer, E., Catthoor, F. and Corporaal, H., "Pareto-Based Application Specification for MP-SoC Customized Run-Time Management", *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Springer-Verlag, Samos, Greece, pp.78–84, July 2006.

[227] Yukish, M., "Algorithms to Identify Pareto Points in Multi-Dimensional Data Sets", *PhD thesis*, Pennsylvania State University, Aug. 2004.

[228] Zhao, Y. and Malik, S., "Exact Memory Size Estimation for Array Computation without Loop Unrolling", *Proceedings of the 36st ACM/IEEE Design Automation Conference*, pp. 811–816, 1999.

[229] *http://www.imec.be/design/atomium/*

[230] *http://www.boost.org*

[231] *http://valgrind.org/info/tools.html*

[232] *http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html*

[233] CoWare Inc.: *http://www.coware.com*

[234] *http://www.sgi.com/tech/stl/*

[235] Target Compiler Technologies: *http://www.retarget.com*

[236] Texas Instruments, "TMS320C6000 Optimizing Compiler v. 6.0 Beta (User's Guide)", July 2005.

[237] *http://www.es.ele.tue.nl/scenarios*

[238] *http://en.wikipedia.org/wiki/Loop_transformation*

[239] *http://en.wikipedia.org/wiki/Moore's_Law*

[240] *http://en.wikipedia.org/wiki/Series-parallel_graph*

[241] *http://en.wikipedia.org/wiki/Complete_graph*

[242] *http://www.yaml.org*

List of publications

Palkovic, M., Miranda, M., Catthoor, F. and Verkest, D., "High-level condition expression transformations for design exploration and improved synthesis of an arithmetic coder", *3rd Workshop on System Design Automation (SDA 2000)*, Rathen, Germany, pp.53–59, March 2000.

Ac, V., Balaz, M., Fristacky, N., Gramatova, E., Janiszewski, L., Kaczmarczyk, J., Kobus, A., Konfal, M., Likavcan, S., Palkovic, M., Sekierska, K. and Weber, B., "Application of the ITE 80C51 core for development of the general purpose smart card reader", *4th VILAB USER FORUM*, Györ, Hungary, April 2001.

Palkovic, M., Miranda, M., Catthoor, F. and Verkest, D., "High-level condition expression transformations for design exploration", *SystemDesign Automation – Fundamentals, Principles, Methods, Examples*, R.Merker and W.Schwarz (eds.), Kluwer Acad. Publ., Boston, pp. 56-64, 2001.

Kjeldsberg, P.G., Palkovic, M., Catthoor, F. and Aas, E.J., "STOREQ tool for STOrage Requirement estimation and optimization of data intensive applications", Univ. booth demonstration, *5th ACM/IEEE Design and Test in Europe Conf.* (DATE), pp.256, Paris, France, April 2002.

Palkovic, M., Miranda, M., Denolf, K., Vos, P. and Catthoor, F., "Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG4 Video Decoder", *IEEE 15th Intnl. Conf. on VLSI Design and 7th ASP-DAC*, Bangalore, India, pp.547–552, January 2002.

Palkovic, M., Miranda, M. and Catthoor, F., "Systematic Power-Performance Trade-Off in MPEG4 by means of Selective Function Inlining steered by Address Optimisation Opportunities", *In Proc. 5th ACM/-IEEE Design and Test in Europe Conf. (DATE)*, pp.1072–1077, Paris, France, March 2002.

Rydland, P., Palkovic, M., Kjeldsberg, P.G., Brockmeyer, E. and Catthoor, F., "Inter in-place storage size requirement estimation", *Proceedings of IEEE NORCHIP Conference (NORCHIP 2003)*, pp. 240–243, Riga, Latvia, November 2003.

Hu, Q., Palkovic, M. and Kjeldsberg, P.G., "Memory Requirement Optimization with Loop Fusion and Loop Shifting", *Proceedings of 30th Euromicro conference*, pp. 272–278, Rennes, France, Aug. 2004.

Hu, Q., Brockmeyer, E., Palkovic, M., Kjeldsberg, P.G. and Catthoor, F., "Memory Hierarchy Usage Estimation for Global Loop Transformations", *IEEE NORCHIP Conference (NORCHIP 2004)*, pp. 301–304, Oslo, Norway, Nov. 2004.

Palkovic, M., Brockmeyer, E., Corporaal, H., Catthoor, F. and Vounckx, J., "Hierarchical rewriting and hiding of data dependent conditions to enable global loop transformations", *Proc. 2nd Wsh. on Optim. for DSP and Embedded Systems (ODES)*, Palo Alto CA, March 2004.

Palkovic, M., Corporaal, H. and Catthoor, F., "Global Memory Optimisation for Embedded Systems allowed by Code Duplication",*Proc. of 9th Intnl. Wsh. on Software and Compilers for Embedded Systems (SCOPES)*, pp.72–80, Dallas, TX, Sep. 2005.

Palkovic, M., Brockmeyer, E., Vanbroekhoven, P., Corporaal, H. and Catthoor, F.,"Systematic Preprocessing of Data Dependent Constructs for Embedded Systems", *Proc. IEEE Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Leuven, Belgium, Lecture Notes Comp. Sc., Springer-Verlag, Vol.3728, pp.89–90, Sep. 2005.

Balasa, F., Kjeldsberg, P.G., Palkovic, M., Vandecappelle, A. and Catthoor, F., "Loop Transformation Methodologies for Array-Oriented Memory Management", *invited paper at IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2006)*, Steamboat Springs, Colorado, USA, September 2006.

Hu, Q., Vandecappelle, A., Palkovic, M., Kjeldsberg, P.G., Brockmeyer, E. and Catthoor, F., "Hierarchical Memory Size Estimation for Loop Fusion and Loop Shifting of Data Dominated Applications", *Proceedings of the 11th Asia and South Pacific Design Automation Conference (ASP-DAC 2006)*, pp. 606–611, Yokohama City, Japan, January 2006.

Palkovic, M., Brockmeyer, E., Vanbroekhoven, P., Corporaal H. and Catthoor, F.,"Systematic Preprocessing of Data Dependent Constructs for Embedded Systems", *Journal of Low Power Electronics (JOLPE)*, American Scientific Publ., Vol.2, No.1, pp.9–17, April 2006.

Palkovic, M., Corporaal, H. and Catthoor, F., "Dealing with data dependent conditions to enable general global source code transformations", *International Journal of Embedded Systems (IJES)*, Inderscience Publ., in press, 2007.

Palkovic, M., Catthoor, F. and Corporaal, H., "Dealing with Variable Trip Count Loops in System Level Exploration", *Proc. 4th Wsh. on Optim. for DSP and Embedded Systems (ODES)*, pp. 19–28, Manhattan NY, March 2006.

Thörnberg, B., Hu, Q., Palkovic, M., O'Nils, M. and Kjeldsberg, P.G., "Polyhedral space generation and memory estimation from interface and memory models of real-time video systems", *Journal of Systems and Software*, Vol. 79(2), pp. 231–245, 2006.

Hu, Q., Kjeldsberg, P.G., Palkovic, M., Vandecappelle, A. and Catthoor, F., "Incremental Hierarchical Memory Size Estimation for Steering of Loop Transformations", *The ACM Transactions on Design Automation of Electronic Systems (TODAES)*, (accepted) 2007.

Hu, Q., Vandecappelle, A., Kjeldsberg, P.G., Catthoor, F. and Palkovic, M., "Fast Memory Footprint Estimation based on Dependency Distance", *Proc. 10th ACM/IEEE Design and Test in Europe Conf.* (DATE), Nice, France, pp.1–6, April 2007.

Palkovic, M., Corporaal, H. and Catthoor, F., "Trade-offs in Loop Transformations", *The ACM Transactions on Design Automation of Electronic Systems (TODAES)*, (submitted) 2007.

Palkovic, M., Corporaal, H. and Catthoor, F., "Heuristics for Scenario Creation to Enable General Loop Transformations", *Proc. IEEE International Symposium on System-on-Chip (SoC)*, (accepted) Tampere, Finland, pp.–, Nov. 2007.

Thörnberg, B., Palkovic, M., Hu, Q., Olsson, L., Kjeldsberg, P., O'Nils, M. and Catthoor, F., "Bit-Width Constrained Memory Hierarchy Optimization for Real-Time Video Systems", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 26, Issue 4, pp.781–800, April 2007.

List of Acronyms

ADOPT Address Optimization

AG AST-GM link

ALU Arithmetic-Logical Unit

ASIC Application Specific Integrated Circuit

AST Abstract Syntax Tree

ATOMIUM ATOMIUM is the framework developed at IMEC

BB BACKBONE is the IMEC internal library similar to STL

BGL Boost Graph Library

BG Basic Group

CFG Control Flow Graph

CFsG Control Flow subGraph

CISC Complex Instruction Set Computer

CIS Common Iteration Space

CLooG Chunky Loop Generator

CPU Central Processing Unit

CSE Common Subexpression Elimination

DAG Directed Acyclic Graph

DCT Discrete Cosine Transformation

DLP Data Level Parallelism

DP Dependency Part

DRA Data Reuse Analysis

DSA Dynamic Single Assignment conversion

DSP Digital Signal Processor

DTSE Data Transfer and Storage Exploration

DVS Dynamic Voltage Scaling

FFT Fast Fourier Transformation

FUD Factored User-Def chains

FU Functional Unit

GLT Global Loop Transformations

GM Geometrical Model

HMSE Hierarchical Memory Storage Estimation

HPF High Performance Fortran

IDCT Inverse Discrete Cosine Transformation

ILP Instruction Level Parallelism

IMDCT Inverse Modified Discrete Cosine Transformation

IMEC Inter-University Micro-Electronics Center

IR Internal Representation

LT Loop Transformations

LUT Look-Up Table

MAA Memory Allocation and Assignment

MA Memory Architect

MB Macro Block

MC Memory Compaction

ME Motion Estimation

MHLA Memory Hierarchy Layer Assignment tool

MH Memory Hierarchy

MM Main Memory

MP-SoC Multi-Processor System on Chip

MP3 MPEG-1 Layer 3 audio decoder

MT Mersenne-Twister

MV Motion Vector

ORC Open Research Compiler

PAaC Pointer Analysis and Conversion

PDG Polyhedral Dependency Graph

PER Polyhedral Extraction Routines

PIP Parametric Integer Programming

PPF Packet Processing Functions

PPL Parma Polyhedral Library

QSDPCM Quadtree Structured Difference Pulse Code Modulation

RACE Reduction of Arithmetic Cost of Expressions

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTS Run-Time Situation

RTL Register Transfer Language

SBO Storage Budget Optimization

SCBD Storage Cycle Budget Distribution

SCoP Static Control Part

SDF Synchronous Data Flow

SFI Selective Function Inlining

SPM Scratchpad Memory

SRAM Static Random Access Memory

STL Standard Template Library

STOREQ Storage Requirement estimation

SUIF Standford University Intermediate Format

TCM Task Concurrency Management

TGFF Task Graphs For Free

VLIW Very Long Instruction Word

WCET Worst Case Execution Time

WLooG WHIRL Loop Generator

YAML YAML Ain't Markup Language

ada.pl advanced dependency analysis in perl

da.pl dependency analysis in perl

pers Polyhedral Extraction Routines from SUIF

s2c SUIF to C

sda.pl simple dependency analysis in perl

sloog SUIF Loop Generator

w2p WHIRL to Polyhedra

Curriculum vitae

Martin Palkovič was born in Bratislava, Slovakia, on June 22, 1977. He received his B.Sc. and M.Sc. degrees in Electrical Engineering from the Slovak University of Technology, Bratislava, Slovakia, in 1999 and 2001 respectively, and his M.Sc. degree in Economics from the University of Economics, Bratislava, Slovakia, in 2000. He joined the Nomadic Embedded Systems division at the Inter-University Micro Electronics Center (IMEC), Leuven, Belgium, in March 2001, where he is currently a researcher in the Design Technology group. From 2002 to 2007 he was also working towards the PhD degree in the department of Electrical Engineering at the Technische Universiteit Eindhoven, The Netherlands. His research interests include high-level optimizations in data dominated multimedia applications and wireless systems, related aspects of system design automation, and platform architectures for low power.